Développement mobile sous Android

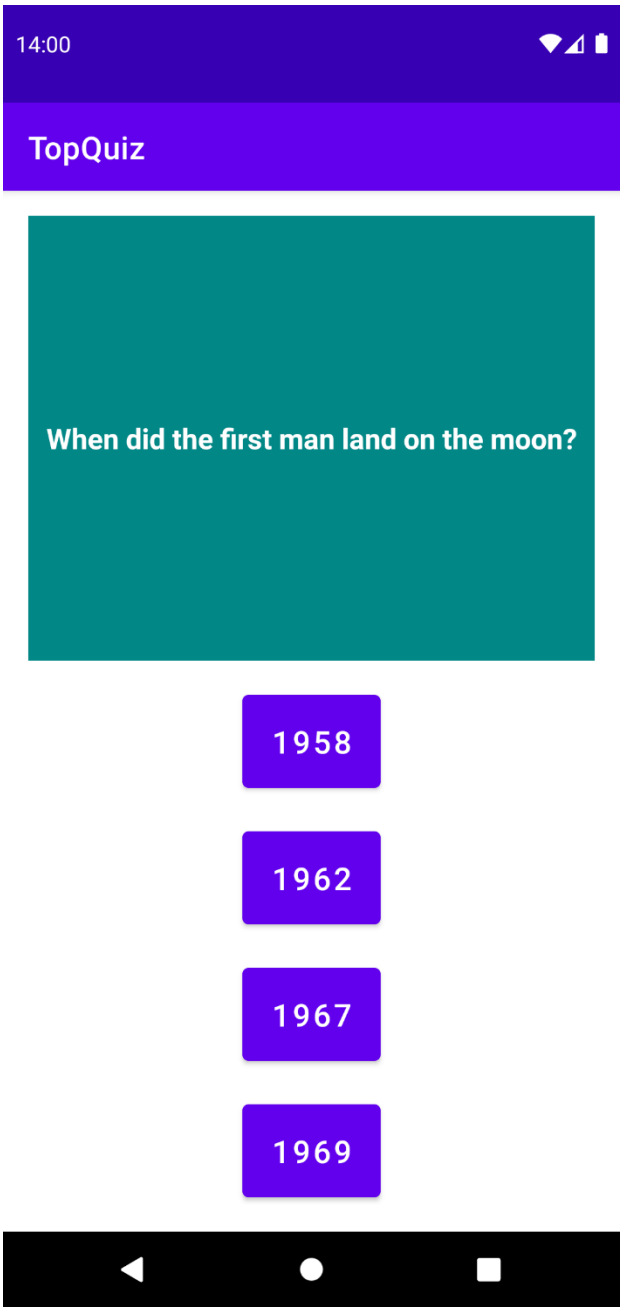
PARTIE II : ÉTENDEZ LES FONCTIONNALITÉS DE VOTRE APPLICATION

DEUXIEME PARTIE – ÉTENDEZ LES FONCTIONNALITÉS DE VOTRE APPLICATION

Chapitre 7. Créez une seconde activité

Dans la première partie de ce cours, nous avons créé une première activité, permettant de récupérer le prénom de l'utilisateur et de démarrer le jeu. Dans cette deuxième partie, nous allons créer une seconde activité. Cette activité permettra d'afficher le contenu du jeu. Elle sera démarrée par la première activité lorsque l'utilisateur cliquera sur le bouton de lancement du jeu.

Vous vous demandez sûrement comment fonctionne le jeu ! C'est une demande légitime. C'est très simple : TopQuiz va poser une série de quatre questions à l'utilisateur, choisies aléatoirement dans une banque de plusieurs questions. Pour chaque question, il aura le choix entre quatre réponses possibles. Par exemple :

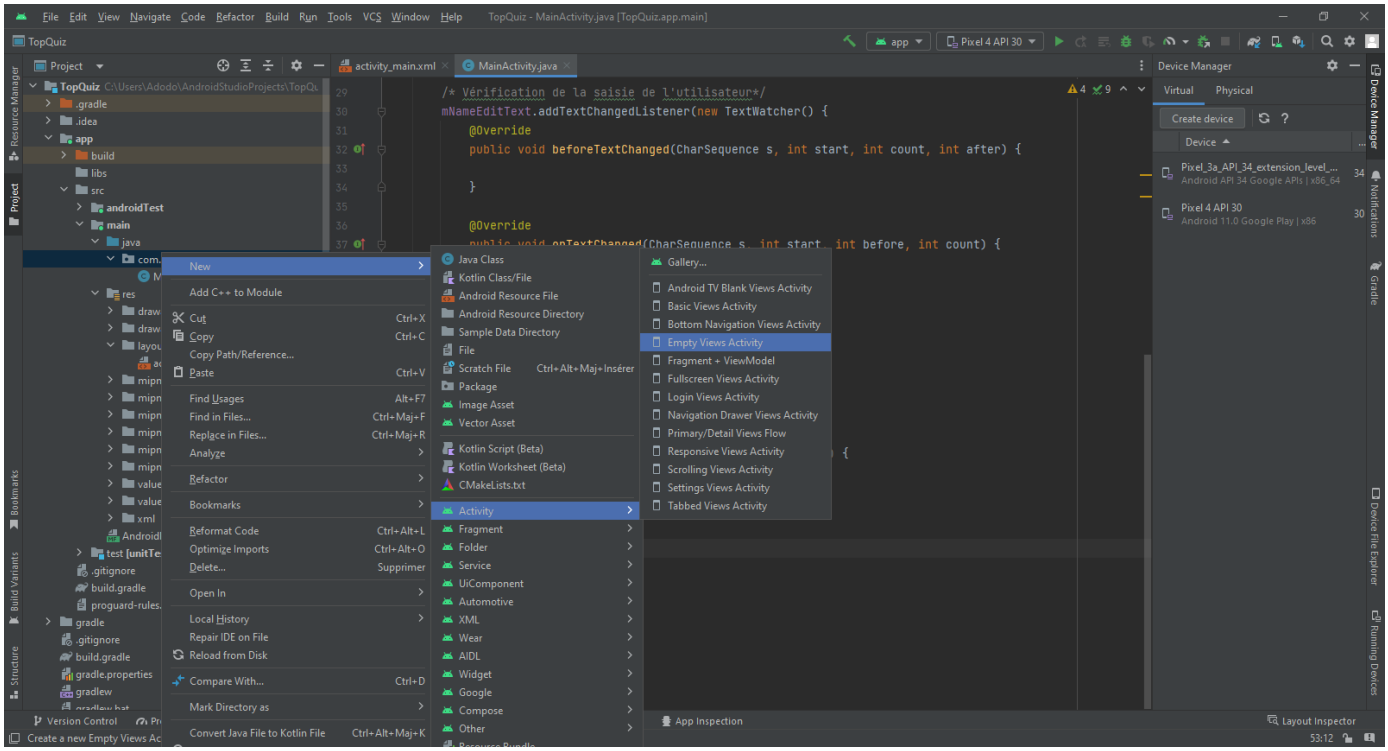


Visualisez l'aperçu de notre superbe application

7.1. Configuration de la nouvelle activité

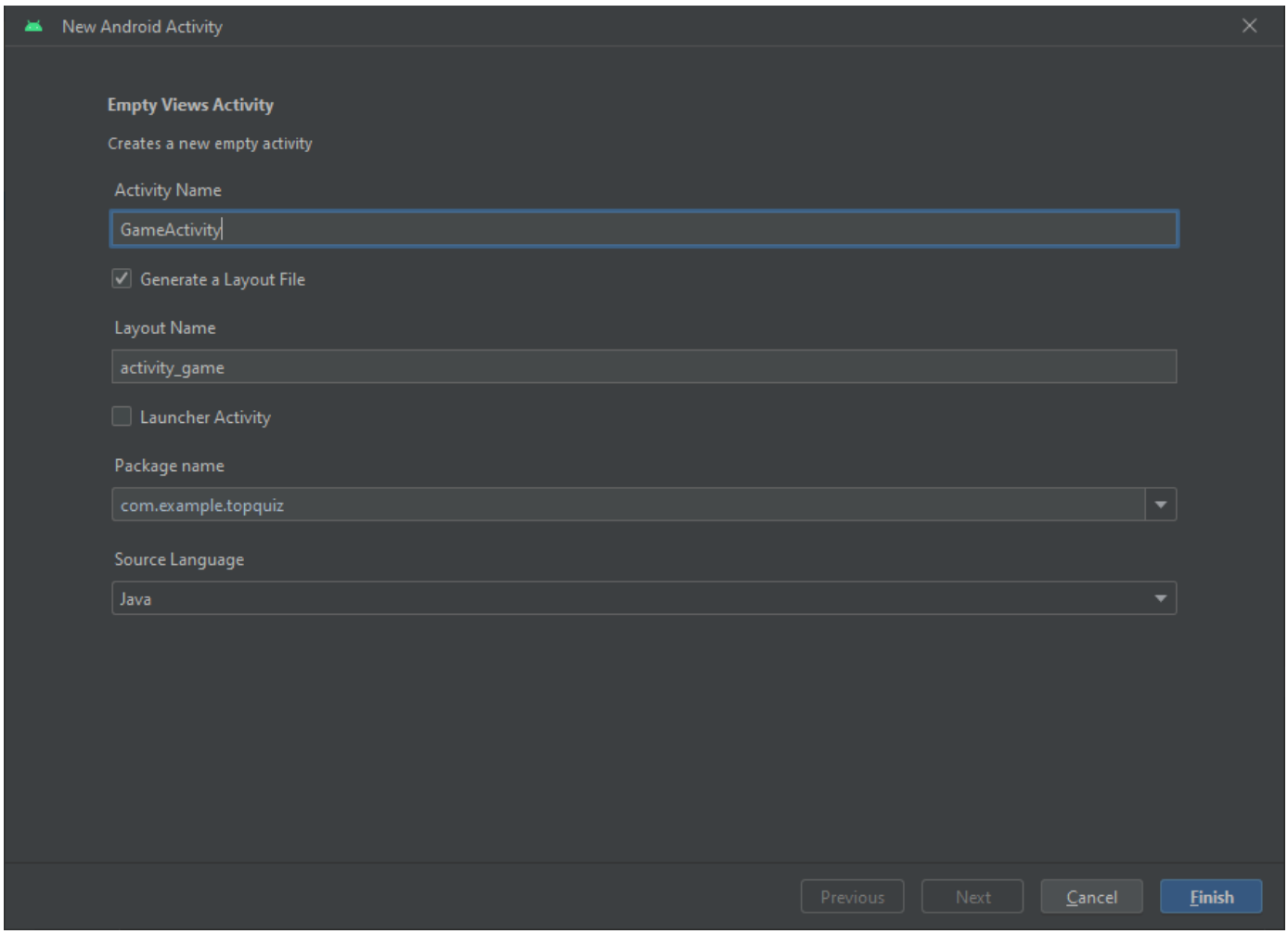
7.1.1. Création des fichiers

Pour rappel, une activité est généralement composée d'une classe Java et de son fichier layout associé. Positionnez-vous dans l'arborescence de votre projet, au niveau du répertoire dans lequel est contenue la classe *MainActivity* créée précédemment. Faites un clic droit dessus, puis sélectionnez l'option **New > Activity > Empty Views Activity**.



Dans l'arborescence, allez dans New > Activity > Empty Views Activity pour créer votre fichier layout

Une nouvelle fenêtre s'affiche, vous permettant de configurer le nom de l'activité et de son fichier layout. Sachant que cette activité va gérer le jeu, nous l'appellerons logiquement *GameActivity*. Laissez cochée la case **Generate Layout File** : le nom du fichier layout généré doit être *game_activity*. Cliquez sur le bouton **Finish**.



Configurez le nom de l'activité et de son fichier layout

7.1.2. *Création du layout*

L'interface de cette nouvelle activité va contenir les éléments suivants :

- l'intitulé de la question posée à l'utilisateur ;
- les quatre réponses possibles, chacune représentée par un bouton.

Pour ce faire, ouvrez le fichier *activity_game.xml* situé dans le répertoire **res/layout**, puis modifiez le code comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:padding="8dp">

  <TextView
    android:id="@+id/game_activity_textview_question"
```

```
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_margin="8dp"
    android:layout_weight="4"
    android:background="@color/teal_700"
    android:gravity="center"
    android:textColor="@color/white"
    android:textSize="18sp"
    android:textStyle="bold"
    tools:text="Question?" />

<Button
    android:id="@+id/game_activity_button_1"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:layout_gravity="center_horizontal"
    android:layout_margin="8dp"
    android:layout_weight="1"
    android:padding="16dp"
    android:textSize="20sp"
    tools:text="Answer1" />

<Button
    android:id="@+id/game_activity_button_2"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:layout_gravity="center_horizontal"
    android:layout_margin="8dp"
    android:layout_weight="1"
    android:padding="16dp"
    android:textSize="20sp"
    tools:text="Answer2" />

<Button
    android:id="@+id/game_activity_button_3"
    android:layout_width="wrap_content"
```

```
    android:layout_height="0dp"
    android:layout_gravity="center_horizontal"
    android:layout_margin="8dp"
    android:layout_weight="1"
    android:padding="16dp"
    android:textSize="20sp"
    tools:text="Answer3" />

  <Button
    android:id="@+id/game_activity_button_4"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:layout_gravity="center_horizontal"
    android:layout_margin="8dp"
    android:layout_weight="1"
    android:padding="16dp"
    android:textSize="20sp"
    tools:text="Answer4" />
</LinearLayout>
```

Vous devez déjà être familier avec certains attributs, tels que les marges, le texte à afficher ou les identifiants assignés aux éléments. Notez bien que les identifiants sont obligatoires, ce sont eux qui permettront de référencer les éléments depuis le code, et de les mettre à jour dynamiquement. Néanmoins, d'autres doivent vous interpeller.

L'attribut **android:textSize** permet d'indiquer une taille spécifique pour la police. L'unité à utiliser est le **sp** (alors que c'est le **dp** pour les distances, rappelez-vous). N'hésitez pas à vous référer à cette page du Material Design (https://material.io/design/layout/pixel-density.html#pixel-density-on-android) pour avoir les idées plus claires.

L'attribut **android:textColor** permet de spécifier quelle police de couleur utiliser, pour un champ texte ou un bouton.

Pour attribuer une couleur à un élément, vous avez le choix entre trois valeurs possibles :

- une valeur hexadécimale, par exemple *#990000FF* pour un bleu légèrement transparent ;
- une ressource définie au niveau d'Android, par exemple *@android:color/white* ;
- une ressource définie au niveau de votre projet, par exemple *@color/colorPrimary*. Vous trouverez la valeur correspondant à cette ressource dans le fichier *colors.xml* situé dans le répertoire **res/values**.

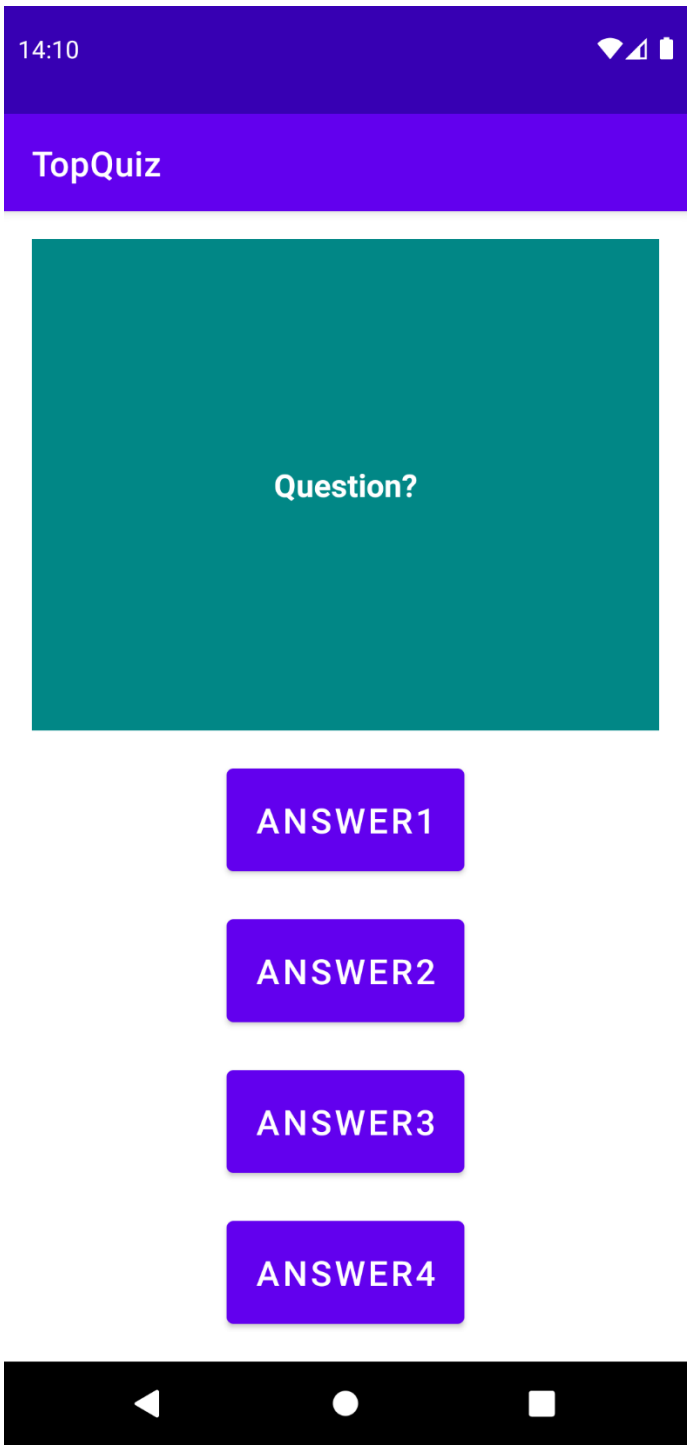
Plus de précisions **sur cette page :** https://developer.android.com/guide/topics/resources/more-resources.html#Color.

L'attribut **android:background** prend le même paramètre que **textColor**, mais permet pour sa part de préciser la couleur d'arrière-plan de la majorité des éléments (bouton, champ texte, layout, etc.).

L'attribut **android:textStyle** permet de préciser le style à utiliser pour une police. Vous avez le choix entre **bold** (gras), **italic** (italique) ou **normal**, la valeur par défaut.

Enfin, le dernier attribut très important est **android:layout_weight**. C'est un attribut spécifique aux enfants d'un LinearLayout. Il permet de préciser le poids d'un élément par rapport aux autres. Je m'explique. Si vous assignez un poids identique pour tous les éléments (par exemple 1), tous ces éléments auront la même taille. Si vous assignez un poids égal à 2 à l'un des éléments, alors sa taille sera deux fois plus importante que celle des autres. Essayez de faire varier le poids de l'élément TextView du code ci-dessus, afin de voir le résultat. N'hésitez pas à consulter cette page (https://developer.android.com/guide/topics/ui/layout/linear.html#Weight) pour avoir une explication plus détaillée.

Lorsque vous utilisez l'attribut **layout_weight** pour un élément donné, vous devez impérativement spécifier une hauteur ou une largeur égale à **0 dp**, suivant l'orientation. Plus précisément, si l'orientation est verticale, la hauteur doit être égale à 0 dp. Si l'orientation est horizontale, la largeur doit être égale à 0 dp.
Le résultat produit doit être identique à la capture ci-dessous. Amusez-vous à modifier les différents attributs pour donner un style qui vous est propre !



L'interface de la nouvelle activité que vous devriez obtenir

7.1.3. Branchement des widgets

Maintenant que les éléments sont définis dans l'interface, il est nécessaire de les référencer dans le code. Sachant que vous devenez des experts Android, cela ne devrait pas vous poser de problème ! Un petit coup de pouce pour les plus étourdis :

- déclarez une variable membre privée pour chaque élément que vous souhaitez référencer (vous devriez avoir une variable de type *TextView* et quatre de type *Button*) ;
- dans la méthode **onCreate**, branchez chaque widget en utilisant la méthode **findViewById()**.

7.2. Lancement de la nouvelle activité

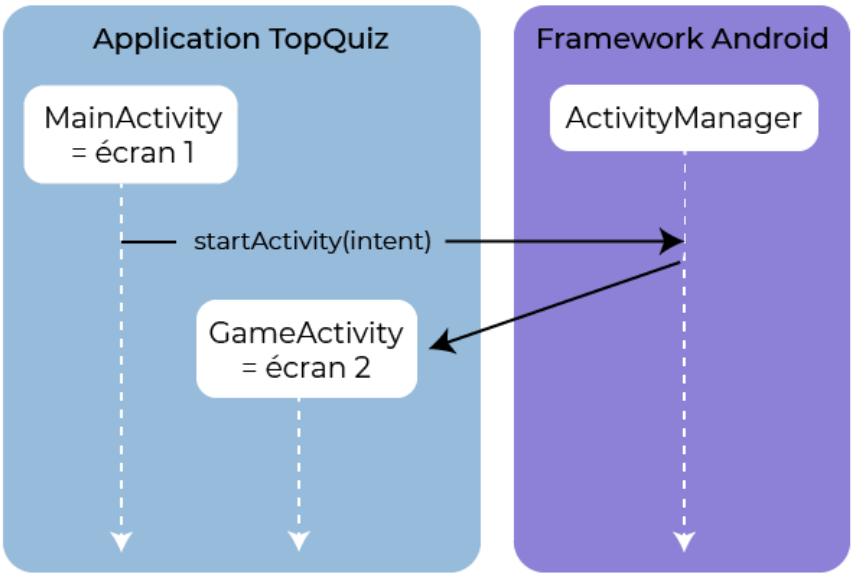
C'est bien beau tout ça, mais comment lancer cette nouvelle activité ? En cliquant sur le bouton *Let's Play* de l'écran d'accueil ! Certes, mais cela ne vous avance guère.

Rappelez-vous : nous avions commencé à implémenter la méthode **onClick** dans la classe *MainActivity*. Cette méthode est appelée à chaque fois que l'utilisateur appuie sur le bouton. Et c'est à cet endroit précis que nous allons démarrer notre nouvelle activité.

Pour démarrer une activité, Android propose la méthode **startActivity()**. Cette méthode propose plusieurs signatures. Celle qui nous intéresse est la suivante (c'est la plus simple) :

```
public void startActivity(Intent intent)
```

Cette méthode permet de communiquer avec le système d'exploitation Android en lui demandant de démarrer une activité donnée. Pour préciser quelle activité lancer, un objet spécifique est utilisé : **Intent** (https://developer.android.com/training/basics/firstapp/starting-activity.html#BuildIntent). Lorsque la méthode **startActivity()** est appelée, l'objet interne Android **ActivityManager** inspecte le contenu de l'objet **Intent** et démarre l'activité correspondante.



Lorsque la méthode startActivity() est appelée, l'objet interne Android ActivityManager inspecte le contenu de l'objet Intent et démarre l'activité correspondante.

L'objet **Intent**, quant à lui, peut être créé de différentes façons. Parmi les nombreux constructeurs de la classe **Intent**, celui qui nous intéresse est le suivant :

```
public Intent(Context packageContext, Class<?> cls)
```

Le premier paramètre correspond au contexte de l'application. Pour faire simple, cela correspond à l'activité appelante (car la classe *Activity* hérite de la classe *Context*). Le second paramètre correspond à la classe de l'activité à démarrer.

Je vous l'accorde, cela peut sembler bien compliqué pour démarrer une simple activité. Mais dans les faits, cela se traduit simplement par les deux lignes suivantes :

```
Intent gameActivityIntent = new Intent(MainActivity.this, GameActivity.class);
startActivity(gameActivityIntent);
```

Ajoutez ces deux lignes dans la méthode **onClick**, lancez l'application, cliquez sur le bouton, et vous devez voir apparaître la nouvelle activité. Pour revenir à l'activité précédente, utilisez le bouton de retour Android en bas à gauche de l'écran.

Lorsque vous avez créé l'activité *GameActivity*, Android Studio a automatiquement mis à jour le fichier *AndroidManifest.xml* situé dans le répertoire **app/src/main/res**, en ajoutant une entrée de type *activity*. Cette entrée est indispensable, car elle permet à Android de retrouver la classe correspondant à l'activité que vous souhaitez démarrer. Essayez de supprimer cette entrée puis de relancer l'application : plantage assuré !

7.3. En résumé

- Pour créer une nouvelle Activity dans un projet, il faut :
  - créer les fichiers ;
  - créer le layout en XML ;
  - modifier le Manifest pour y ajouter un élément “<activity>” portant son nom.
- Les Intents permettent de lancer de nouvelles Activity grâce à la méthode *startActivity()*.

Votre deuxième activité est maintenant créée. Dans le prochain chapitre, nous nous pencherons sur une notion d'architecture logicielle ; l'architecture Modèle-Vue-Contrôleur.

Chapitre 8. Découvrez l'architecture Modèle-Vue-Contrôleur

Dans ce chapitre, nous allons aborder une notion d'architecture logicielle, afin de l'appliquer à notre application TopQuiz.

Lorsque vous développez une application Android, vous êtes libre d'utiliser l'architecture logicielle de votre choix. Toutefois, votre code risque de ne pas être compris par le commun des mortels, vos collègues vous ignoreront et n'iront plus déjeuner avec vous. De votre côté, vous risquez également de peiner à maintenir le code d'un autre, voire de jeter votre clavier par la fenêtre (sauf si vous codez dans une cave).

De fait, les développeurs débutant Android (mais pas qu'eux !) utilisent une architecture très simple appelée **MVC** (https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller), pour **Model-View-Controller**, (ou *Modèle-Vue-Contrôleur*, en français). Cette architecture, bien que vieillissante, est très classique et facile à aborder. C'est donc celle que nous utiliserons pour notre application TopQuiz.

Retenez que l'architecture recommandée (https://developer.android.com/jetpack/guide#recommended-app-arch) par Google pour Android est le **MVVM** (pour **Model-View-ViewModel**). Pour le moment, concentrons-nous sur le MVC, plus simple à aborder !

À partir de maintenant, j'utiliserai le trigramme MVC au lieu d'écrire Model-View-Controller.

8.1. L'architecture MVC

8.1.1. Description

L'architecture MVC consiste à découper son code pour qu'il appartienne à l'une des trois composantes du MVC. Lorsque vous créez une nouvelle classe ou un nouveau fichier, vous devez donc savoir à quelle composante il appartient :

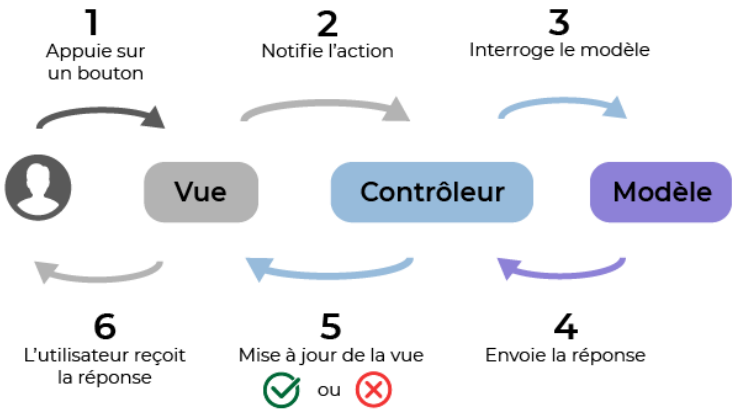
- **Modèle** : contient les données de l'application et la logique métier. Par exemple, les comptes des utilisateurs, les produits que vous vendez, un ensemble de photos, etc. La composante *modèle* n'a aucune connaissance de l'interface graphique. Dans notre application TopQuiz, elle regroupera l'ensemble des questions et des réponses associées.
- **Vue** : contient tout ce qui est visible à l'écran et qui propose une interaction avec l'utilisateur. Par exemple, les boutons, les images, les zones de saisie, etc. Dans notre application TopQuiz, cette composante est définie dans les fichiers *activity_main.xml* et *activity_game.xml*.
- **Contrôleur** : c'est la "colle" entre la vue et le modèle, qui gère également la logique de l'application. Le contrôleur permet de réagir aux interactions de l'utilisateur et de lui présenter les données qu'il demande. Et ces données, où les récupère-t-il ? Dans le modèle, bien entendu ! Dans notre application TopQuiz, cela correspond au code situé dans les fichiers *MainActivity.java* et *GameActivity.java*.

Afin de rendre votre code plus modulaire et maintenable (vos futurs collègues vous remercieront), n'hésitez pas à créer des packages complémentaires dans Android Studio.

Comment créer un sous-répertoire pour organiser votre code ? C'est très simple. Faites un clic droit sur le dossier dans lequel vous souhaitez créer un sous-répertoire, puis sélectionnez **New > Package**. Nommez-le (par exemple *controller*). Ensuite, déplacez les fichiers dans ce nouveau répertoire (par exemple les fichiers de type *Activity*), et validez en cliquant sur le bouton **Refactor**.

8.1.2. Application

Lorsque l'utilisateur interagit avec l'application, par exemple en répondant à une question, voilà ce qu'il se passe :



Les différentes phases d'application du modèle MVC lorsque l'utilisateur intéragit avec l'application

Dans le point 3, le contrôleur pourrait tout à fait mettre à jour le modèle si nécessaire : par exemple si l'utilisateur mettait à jour ses préférences.

8.1.3. Avantages

En appliquant l'architecture MVC, vous permettez à vos collègues de mieux comprendre votre code. Vous aurez également la garantie de pouvoir réutiliser votre code très facilement. Par exemple, vous pourriez décider d'exporter le modèle de votre application sous forme de librairie, afin qu'un autre développeur puisse l'utiliser dans son application. Ainsi, peut-être proposera-t-il une interface graphique complètement différente de la vôtre !

8.1.4. Inconvénients

Vous aurez à créer et à maintenir davantage de fichiers (il est tellement plus tentant de tout mettre au même endroit !). Mais c'est un mal pour un bien, croyez-moi.

8.2. En résumé

- L'architecture MVC est abordable et simple à intégrer, elle convient donc aux applications simples.
- L'Activity est le contrôleur : il fait le lien entre la vue (les Widgets du layout) et la logique métier (le modèle).
- En MVC, le modèle peut être réutilisé dans d'autres Activity pour éviter la duplication de code.

Voilà, vous venez de découvrir l'architecture MVC. Dans le prochain chapitre, nous allons nous intéresser à la composante modèle de l'architecture MVC, voir comment le définir et l'implémenter dans notre application.

Chapitre 9. Définissez votre premier modèle

Dans ce chapitre, nous allons nous intéresser à la composante *modèle* de l'architecture MVC, et voir comment le définir et l'implémenter dans notre application.

9.1. Créez le modèle

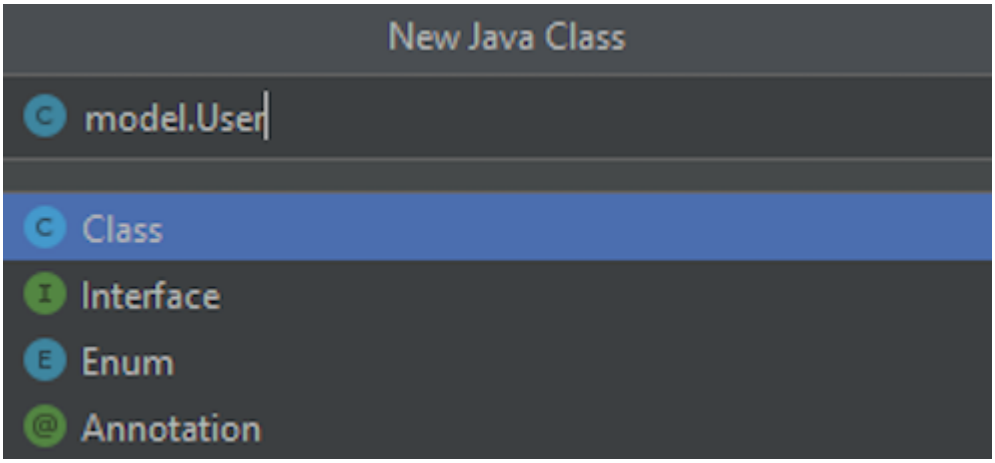
Dans notre super application TopQuiz, notre modèle va contenir les informations suivantes :

- le prénom de l'utilisateur (qu'il saisit sur le premier écran) ;
- le score de l'utilisateur ;
- la liste des questions à afficher ;
- la liste des réponses possibles pour chacune des questions ;
- la bonne réponse pour chacune des questions.

Le modèle n'est pas nécessairement un seul fichier : ce peut être un ensemble de classes utilisées pour gérer des données différentes. Dans notre cas, il est logique de créer au moins deux classes distinctes : l'une pour gérer les informations et le score de l'utilisateur, l'autre pour gérer l'ensemble des questions.

9.1.1. Gestion des données utilisateur

Commencez par créer une classe nommée **User**, stockée dans le package **model**. Pour ce faire, faites un clic droit sur le package **model** de votre arborescence (sous réserve que vous l'ayez au préalablement créé), puis sélectionnez **New > Java Class**.



Faites un clic droit sur le package model et sélectionnez New > Java Class pour créer une classe User

Indiquez le nom de la classe et laissez la ligne **Class** en surbrillance.

Dans cette nouvelle classe, ajoutez une variable nommée **mFirstName**, pour stocker le prénom de l'utilisateur. Rappelez-vous : c'est une bonne habitude de préfixer le nom d'une variable pour déterminer d'un coup d'œil quelle est sa portée. En l'occurrence, *m* signifie *member*. Le lecteur saura très facilement qu'il s'agit d'un attribut de classe.

Sachant que nous sommes entre gens sérieux, nous allons préciser que la variable **mFirstName** est privée (avec l'attribut ***private***), et ajouter deux méthodes : l'une pour accéder à sa valeur et l'autre pour la modifier (*getter* et *setter*, en anglais). Ce n'est pas la peine d'écrire ces méthodes à la main. Rappelez-vous : le développeur informatique est flemmard (c'est la raison pour laquelle beaucoup de personnes ont choisi ce métier). Nous allons donc gentiment demander à Android Studio de générer ces méthodes pour nous !

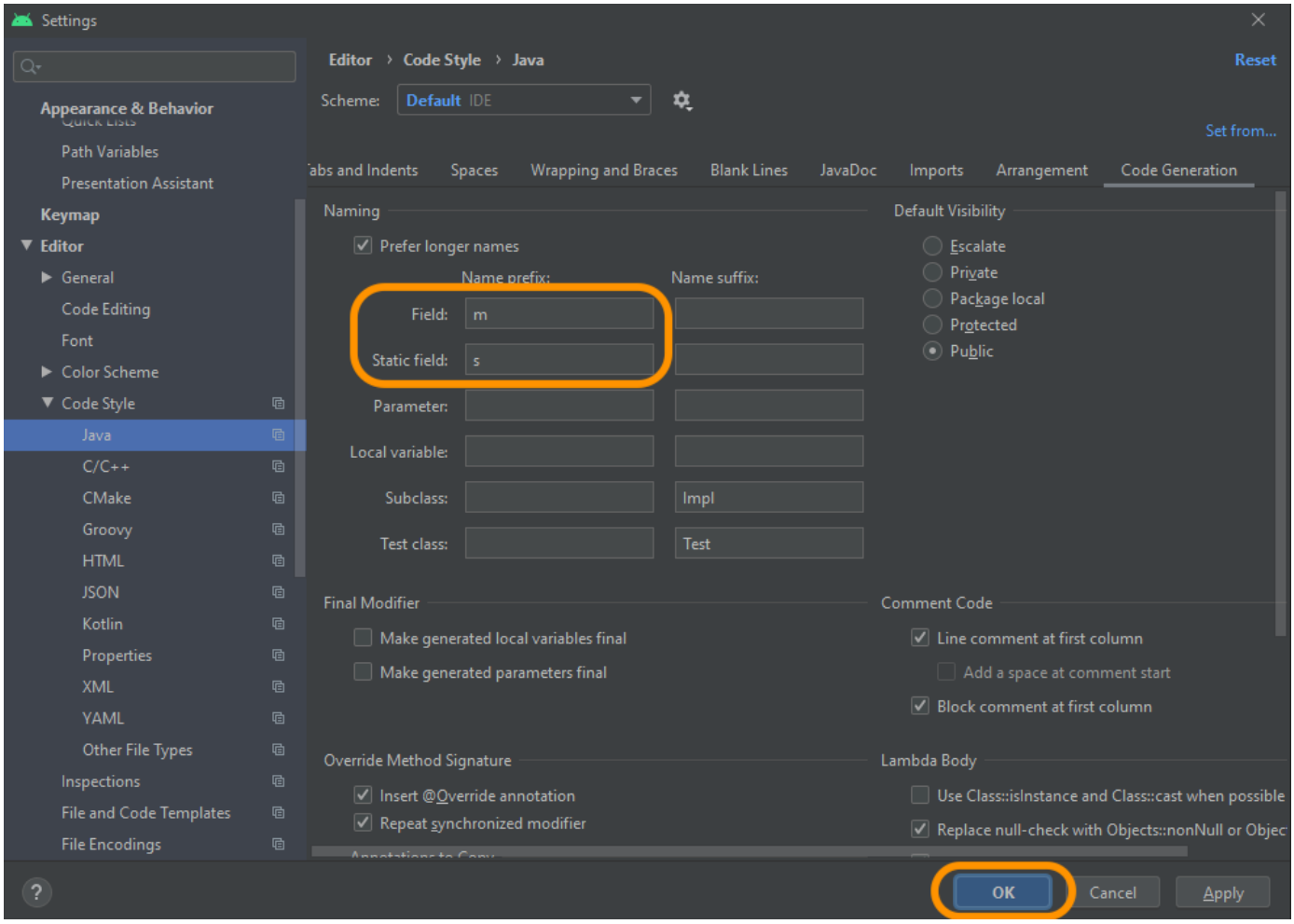
Pour ce faire, positionnez le curseur de votre souris sur la ligne en dessous de la variable **mFirstName**, appuyez sur ALT + Insert sur PC, ou ⌘ + N sur Mac, puis sélectionnez **Getter and Setter**. Sachant que

vous n'avez qu'une seule variable membre, Android Studio la sélectionne par défaut. Si vous en aviez plusieurs, vous pourriez sélectionner celle(s) qui vous intéresse(nt). Cliquez sur le bouton **OK**. Magie !

Android Studio génère les noms de méthodes en prenant mécaniquement le nom de la variable et en la suffixant par *get* ou *set*. Sachant que nous avons préfixé notre variable par *m*, cela ne nous arrange pas : Android Studio génère *getmFirstName* et *setmFirstName*.

Heureusement, il est tout à fait possible de pallier ce problème en indiquant à Android Studio d'ignorer cette lettre. Rendez-vous dans les préférences d'Android Studio, puis dans **Editor > Code Style > Java**, puis dans l'onglet **Code Generation**. Dans la colonne **Name Prefix**, saisissez *m* pour le champ **Field** et *s* pour le champ **Static Field**. Cliquez sur le bouton **Apply** puis **OK**.

Effacez le code généré précédemment puis générez-le de nouveau. Magie ! (bis)



Dans la colonne Name Prefix, saisissez m pour le champ Field et s pour le champ Static Field. Cliquez sur le bouton Apply puis OK

9.1.2. *Mémorisation du prénom du joueur*

Lorsque le joueur a terminé de saisir son prénom, il clique sur le bouton de démarrage de jeu. C'est probablement le moment le plus opportun pour mémoriser son prénom dans le modèle.

Pour ce faire, ajoutez un attribut du type **User** dans la classe *MainActivity* et initialisez-le dans la méthode **onCreate() comme ceci :**

```
mUser = new User();
```

Ensuite, mémorisez le prénom du joueur lorsqu'il clique sur le bouton, c'est-à-dire dans la méthode **onClick()** du bouton :

```
mUser.setFirstName(mNameEditText.getText().toString());
```

9.1.3. *Gestion des questions*

De la même façon, commencez par créer une classe Java nommée **Question**. L'instance de cette classe contiendra une question, avec les quatre réponses associées et la bonne réponse correspondante. De ce fait, la classe va contenir les trois attributs suivants :

```
private final String mQuestion;
private final List<String> mChoiceList;
private final int mAnswerIndex;
```

Le premier est le texte de la question. Le deuxième est la liste des réponses proposées. Le troisième est l'index de la réponse dans la liste précédente.

N'oubliez pas qu'en programmation, un index de tableau ou de liste **commence toujours à 0**. Ainsi, si vous avez quatre réponses possibles et que la bonne réponse est la dernière de la liste, l'index vaudra 3. Que se passerait-il si l'index valait 4 ?

Utilisez la fonction de génération d'Android Studio pour créer automatiquement le constructeur et les *getters* correspondants.

9.1.4. *Banque de questions*

Dans l'idéal, à chaque nouvelle partie, il faudrait que les questions soient différentes, et affichées dans un ordre aléatoire. Pour ce faire, nous allons créer une classe Java spécifique nommée **QuestionBank**, qui va gérer cette liste de questions pour nous. Voici l'implémentation de cette classe :

```
package com.example.topquiz;

import java.io.Serializable;
import java.util.Collections;
import java.util.List;

public class QuestionBank implements Serializable {

  private final List<Question> mQuestionList;
  private int mNextQuestionIndex;

  public QuestionBank(List<Question> questionList) {
     // Shuffle the question list before storing it
     mQuestionList = questionList;
     Collections.shuffle(mQuestionList);
  }

  public Question getCurrentQuestion() {
     return mQuestionList.get(mQuestionIndex);
  }
```

```
  public Question getNextQuestion() {
    // Loop over the questions and return a new one at each call
    mQuestionIndex++;
    return getCurrentQuestion();
  }
}
```

Vous êtes libre de la développer à votre façon : ma version n'étant qu'une variante parmi d'autres !

9.2. Mettez à jour le contrôleur

Maintenant que le modèle est disponible, nous allons pouvoir l'utiliser dans le contrôleur. Le code qui nous intéresse est situé dans la classe **GameActivity** : c'est elle qui aura la charge d'afficher les différentes questions à l'utilisateur.

Tout d'abord, nous allons créer l'ensemble des questions du jeu, et les ajouter dans la banque de questions. Si vous souhaitez ajouter vos propres questions pour épater vos amis et vos collègues, rendez-vous sur ce site : http://www.quiz-questions.net/.

La solution la plus propre consiste à créer une méthode permettant de générer les différentes questions. Pour ce faire, déclarez dans votre activité un attribut du type **QuestionBank**, et ajoutez une méthode permettant de renvoyer un objet de ce type. Dans cette méthode, créez autant de questions que vous le souhaitez, ajoutez-les à la banque de questions puis renvoyez l'objet valorisé. Par exemple :

```
private QuestionBank generateQuestions(){
  Question question1 = new Question(
    "Who is the creator of Android?",
    Arrays.asList(
        "Andy Rubin",
        "Steve Wozniak",
        "Jake Wharton",
        "Paul Smith"
    ),
    0
  );

  Question question2 = new Question(
    "When did the first man land on the moon?",
    Arrays.asList(
        "1958",
        "1962",
        "1967",
        "1969"
```

```
    ),
    3
  );

  Question question3 = new Question(
    "What is the house number of The Simpsons?",
    Arrays.asList(
        "42",
        "101",
        "666",
        "742"
    ),
    3
  );

  return new QuestionBank(Arrays.asList(question1, question2, question3));
}
```

Il vous suffit d'initialiser votre banque de questions dès la création de l'attribut à la suite des quatre boutons :

```
private final QuestionBank mQuestionBank = generateQuestions();
```

9.3. En résumé

- Un modèle permet de stocker des données et de les réutiliser plus tard.
- Il est possible de préfixer les variables de classe avec *m* et les variables de classes statiques avec *s* pour les identifier plus facilement dans le code.

Voilà, le modèle est implémenté et le contrôleur est à jour pour utiliser ce nouveau modèle, c'est parfait. Dans le prochain chapitre, nous implémenterons la logique de jeu dans la classe GameActivity (notre contrôleur).

Chapitre 10. Implémentez la logique de jeu dans le contrôleur

Dans ce chapitre, nous allons commencer à développer la logique de jeu dans la classe **GameActivity**, qui endosse le rôle de contrôleur dans l'architecture MVC. Ce contrôleur aura pour rôle :

- d'interroger le modèle pour récupérer une question, puis de la présenter au joueur avec quatre propositions de réponse ;
- de récupérer la réponse du joueur, de vérifier sa validité et d'afficher le statut (réponse correcte ou erronée) ;
- de comptabiliser le score du joueur ;
- en fin de partie, de présenter le score au joueur.

Nous allons nous intéresser aux deux premiers points : la présentation d'une question et le traitement de la réponse.

10.1. Implémentez la logique de jeu

10.1.1. Affichage d'une question

Tout d'abord, il est nécessaire de récupérer une question dans le modèle de données, grâce à la méthode **getQuestion()** de la classe **QuestionBank**. Pour mettre à jour l'affichage, sachant qu'il sera nécessaire de le faire plusieurs fois au cours du jeu, le plus simple consiste à créer une méthode dédiée. Cette méthode peut prendre en paramètre une instance de **Question** puis en afficher les valeurs. Voici son implémentation :

```
private void displayQuestion(final Question question) {
// Set the text for the question text view and the four buttons
    mTextView.setText(question.getQuestion());
    mGameButton1.setText(question.getChoiceList().get(0));
    mGameButton2.setText(question.getChoiceList().get(1));
    mGameButton3.setText(question.getChoiceList().get(2));
    mGameButton4.setText(question.getChoiceList().get(3));
}
```

10.1.2. Sélection de la réponse

Lorsque le joueur sélectionne une réponse, il faut intercepter son clic sur le bouton. Comme nous l'avons vu précédemment, il est nécessaire d'ajouter un *listener* sur chaque bouton, en implémentant l'interface **View.OnClickListener**. Nous pourrions naïvement écrire le code suivant pour chaque bouton :

```
mAnswerButton1.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    // Handle click on the first button
  }
});
```

Cela fonctionnerait très bien, mais aurait pour conséquence une duplication de code très nette. Or, dupliquer du code, c'est mal : davantage d'octets à gérer, plus de lignes, donc plus de bugs, davantage de traitement processeur, un réchauffement climatique plus rapide, et des dauphins qui meurent.

Au lieu de créer une classe anonyme qui implémente l'interface **View.OnClickListener**, nous allons plutôt implémenter cette interface au niveau de la classe **GameActivity** elle-même ! Ainsi, la méthode **onClick()** ne sera définie qu'une seule fois. Nous devons donc enrichir la déclaration de la classe **GameActivity** comme suit :

```
public class GameActivity extends AppCompatActivity implements View.OnClickListener
```

Vous constatez que la ligne est soulignée en rouge. En effet, Android Studio détecte que vous n'avez pas implémenté toutes les méthodes de l'interface. Pour récupérer automatiquement la liste des méthodes à implémenter et générer le squelette correspondant, appuyez sur ALT + Insert sur Windows, ou ⌘ + N sur Mac, et sélectionnez **Implement Methods...** Vous pouvez également taper CTRL + I directement.
Toujours dans la méthode **onCreate()**, n'oubliez pas de préciser que la classe elle-même sera désormais le *listener* de l'événement de clic :

```
// Get the id for the four buttons.
mGameButton1 = findViewById(R.id.game_activity_button_1);
mGameButton2 = findViewById(R.id.game_activity_button_2);
mGameButton3 = findViewById(R.id.game_activity_button_3);
mGameButton4 = findViewById(R.id.game_activity_button_4);

// Use the same listener for the four buttons.
// The view id value will be used to distinguish the button triggered
mGameButton1.setOnClickListener(this);
mGameButton2.setOnClickListener(this);
mGameButton3.setOnClickListener(this);
mGameButton4.setOnClickListener(this);

// Call the displayQuestion method with the current question
displayQuestion(mQuestionBank.getCurrentQuestion());
```

La méthode **onClick()** sera désormais appelée à chaque fois que l'utilisateur cliquera sur un des quatre boutons de réponse.

10.1.3. Vérification de la réponse

Dans la signature de la méthode **onClick()**, vous constatez que le paramètre passé en entrée est une vue. Cette vue correspond au bouton sur lequel l'utilisateur a cliqué (vous pourriez tout à fait récupérer le bouton associé en effectuant un *cast*). De ce fait, il vous suffit simplement de comparer les widgets précédemment référencés grâce à **findViewById()** et la vue passée en paramètre, pour déterminer le bouton sur lequel l'utilisateur a cliqué. Voici donc le corps de la méthode **onClick()** pour la vérification de la réponse :

```
int index;

if (view == mGameButton1) {
   index = 0;
} else if (view == mGameButton2) {
```

```
  index = 1;
} else if (view == mGameButton3) {
  index = 2;
} else if (view == mGameButton4) {
  index = 3;
} else {
  throw new IllegalStateException("Unknown clicked view : " + view);
}
```

Rappelez-vous, dans le modèle de donnée, nous avions défini l'index de la réponse correcte pour chaque question.

10.1.4. Affichage de la réponse

Pour l'affichage de la réponse, lorsque l'utilisateur sélectionne une réponse, un message est affiché en bas de l'écran. Ce message s'affiche pour une durée assez courte puis disparaît automatiquement. Il s'agit d'un message de type **Toast** (https://developer.android.com/guide/topics/ui/notifiers/toasts.html). Son utilisation est très simple, et sa construction s'effectue en appelant la méthode statique **makeText()**. Cette méthode demande trois paramètres :

- le contexte de l'application (généralement l'activité elle-même) ;
- le texte à afficher ;
- la durée d'affichage. La classe Toast fournit deux durées prédéfinies : **LENGTH_SHORT** et **LENGTH_LONG**, correspondant respectivement à 2 secondes et 3,5 secondes. Vous pouvez définir vous-même la durée d'affichage en utilisant la méthode **setDuration()** sur l'instance de Toast.

Une fois l'objet *Toast* construit, il suffit d'appeler la méthode **show()** sur l'instance pour l'afficher. Vous pouvez chaîner les appels pour gagner du temps comme suit :

```
Toast.makeText(this, "Correct!", Toast.LENGTH_SHORT).show();
```

Nous pouvons donc terminer l'implémentation de la méthode **onClick()** comme suit :

```
if (index == mQuestionBank.getCurrentQuestion().getAnswerIndex()) {
    Toast.makeText(this, "Correct!", Toast.LENGTH_SHORT).show();
} else {
    Toast.makeText(this, "Incorrect!", Toast.LENGTH_SHORT).show();
}
```

10.2. 4.3. En résumé

- Il faut toujours penser à réfactorer son code. Si on identifie du code similaire, il peut être bon de le découper.
- Pour afficher une petite pop-up temporaire en bas de l'écran, on peut utiliser la méthode *Toast.makeText().show()*.

Vous savez maintenant mettre en œuvre une architecture MVC, en interrogeant le modèle pour récupérer de l'information et la présenter à l'utilisateur. Vous commencez à avoir une application fonctionnelle ! Rendez-vous dans le chapitre suivant pour gérer le score du joueur.

Chapitre 11. Présentez le score au joueur

Dans ce court chapitre, nous allons continuer à développer la logique de jeu dans la classe **GameActivity**, et plus précisément la gestion du score du joueur : sa comptabilisation et son affichage.

11.1. Gestion du nombre de questions

Tout d'abord, c'est à vous de décider combien de questions vous souhaitez proposer à l'utilisateur. Le plus simple consiste à créer un attribut de type **int** :

```
private int mRemainingQuestionCount;
```

Puis d'initialiser sa valeur dans la méthode **onCreate()** :

```
mRemainingQuestionCount = 4;
```

Cette variable vous permettra de déterminer quand arrêter le jeu. Prévoyez au moins deux fois plus de questions dans votre banque de questions, afin d'éviter de toujours voir apparaître les mêmes !

Nous décidons par exemple qu'une partie comprend quatre questions. Il est donc nécessaire d'implémenter une logique permettant d'arrêter le jeu après la quatrième question posée.

La variable **mRemainingQuestionCount** sera utilisée en tant que compteur pour savoir quand arrêter le jeu. De fait, à chaque fois que l'utilisateur répondra à une question, cette variable sera décrémentée. Tant qu'elle ne sera pas égale à 0, la question suivante sera posée. Sinon, le jeu s'arrête.

Le résultat est le suivant (à implémenter dans la méthode **onClick()**, après vérification de la réponse) :

```
mRemainingQuestionCount--;

if (mRemainingQuestionCount > 0) {
   mCurrentQuestion = mQuestionBank.getNextQuestion();
   displayQuestion(mCurrentQuestion);
} else {
   // No question left, end the game
}
```

Remarque : la variable *mCurrentQuestion* doit d'abord être créée et sa valeur récupérée au-dessus de l'appel à la méthode displayQuestion dans la méthode **onCreate** pour récupérer la question courante comme ceci :

```
mCurrentQuestion = mQuestionBank.getCurrentQuestion();
```

Vous pouvez ensuite remplacer le contenu de la méthode displayQuestion par la variable *mCurrentQuestion* que nous venons de créer comme ceci :

```
displayQuestion(mCurrentQuestion);
```

11.2. Gestion du score

11.2.1. Comptabilisation du score

Cette partie-là est assez simple, et je suis sûr que vous n'auriez pas besoin de moi pour cette partie. Très simplement, il suffit de déclarer une variable de type entier, puis d'incrémenter sa valeur à chaque fois que le joueur saisit une bonne réponse.

```
private int mScore;
```

Ensuite nous initialisons sa valeur dans la méthode onCreate à zéro comme ceci :

```
mScore = 0;
```

Ainsi, à chaque fois que l'utilisateur donne la bonne réponse cette variable est incrémentée. Nous modifions donc notre code comme ceci :

```
if (index == mQuestionBank.getCurrentQuestion().getAnswerIndex()) {
    Toast.makeText(this, "Correct!", Toast.LENGTH_SHORT).show();
    mScore++;
} else {
    Toast.makeText(this, "Incorrect!", Toast.LENGTH_SHORT).show();
}
```

11.2.2. Affichage du score

Lorsque le joueur a répondu à toutes les questions, la phase finale de jeu est enclenchée :

1. Le score est affiché à l'utilisateur dans une boîte de dialogue.
2. L'écran d'accueil est automatiquement affiché lorsque l'utilisateur clique sur le bouton *OK*.

La boîte de dialogue utilisée est gérée par la classe AlertDialog (https://developer.android.com/guide/topics/ui/dialogs.html#AlertDialog). Cette boîte de dialogue est entièrement paramétrable comme vous pouvez le voir sur le lien ci-dessus mais nous allons l'utiliser de la manière la plus standard possible. À savoir : afficher un titre, un message et un bouton.

Avant d'aller plus loin... une précision : lorsque nous avons démarré la seconde activité (*GameActivity* en l'occurrence), la première activité (*MainActivity*, pour ne pas la nommer) n'a pas été détruite. Elle existe toujours ! En fait, l'affichage de *GameActivity* s'effectue au-dessus de *MainActivity*. En d'autres termes, les deux activités se superposent. Ceci ne devrait pas être fait ainsi. Pourquoi vous dis-je cela ? Vous allez comprendre !
Voici le code permettant de configurer et d'afficher la boîte de dialogue :

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);


builder.setTitle("Well done!")
   .setMessage("Your score is " + mScore)
   .setPositiveButton("OK", new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
         finish();
      }
```

```
})
.create()
.show();
```

Voici quelques explications :

- ligne 1 : il est nécessaire d'utiliser un objet spécifique pour "construire" la boîte de dialogue. C'est la sous-classe **Builder** qui s'en charge ;
- ligne 3 : nous définissons le titre de la boîte de dialogue ;
- ligne 4 : nous définissons le texte à afficher dans la boîte de dialogue ;
- ligne 5 : nous définissons le texte du bouton à afficher, et fournissons l'implémentation de l'interface permettant de gérer le clic sur le bouton ;
- ligne 11 : nous demandons à l'instance de **Builder** de construire la boîte de dialogue avec les paramètres que nous avons prédéfinis ;
- ligne 12 : nous affichons notre belle boîte de dialogue.

Et la ligne 8 dans tout cela ? Eh bien la méthode finish() (https://developer.android.com/reference/android/app/Activity.html#finish()) permet simplement de dire au système : *j'en ai terminé avec l'activité courante, arrêtez-la et ramenez-moi à l'activité précédente*. Pour faire simple, cela revient à cliquer sur le bouton *Back* du téléphone.

11.3. En résumé

- Pour gérer le nombre de questions de votre jeu, créez un attribut de type **int** et initialisez sa valeur dans la méthode **onCreate()**.
- Pour comptabiliser le score d'un joueur, déclarez une variable de type entier et incrémentez sa valeur quand le joueur saisit une bonne réponse.

Nous voici arrivés à la fin de la deuxième partie de ce cours ! Nous allons maintenant perfectionner notre application dans la troisième partie de ce cours.