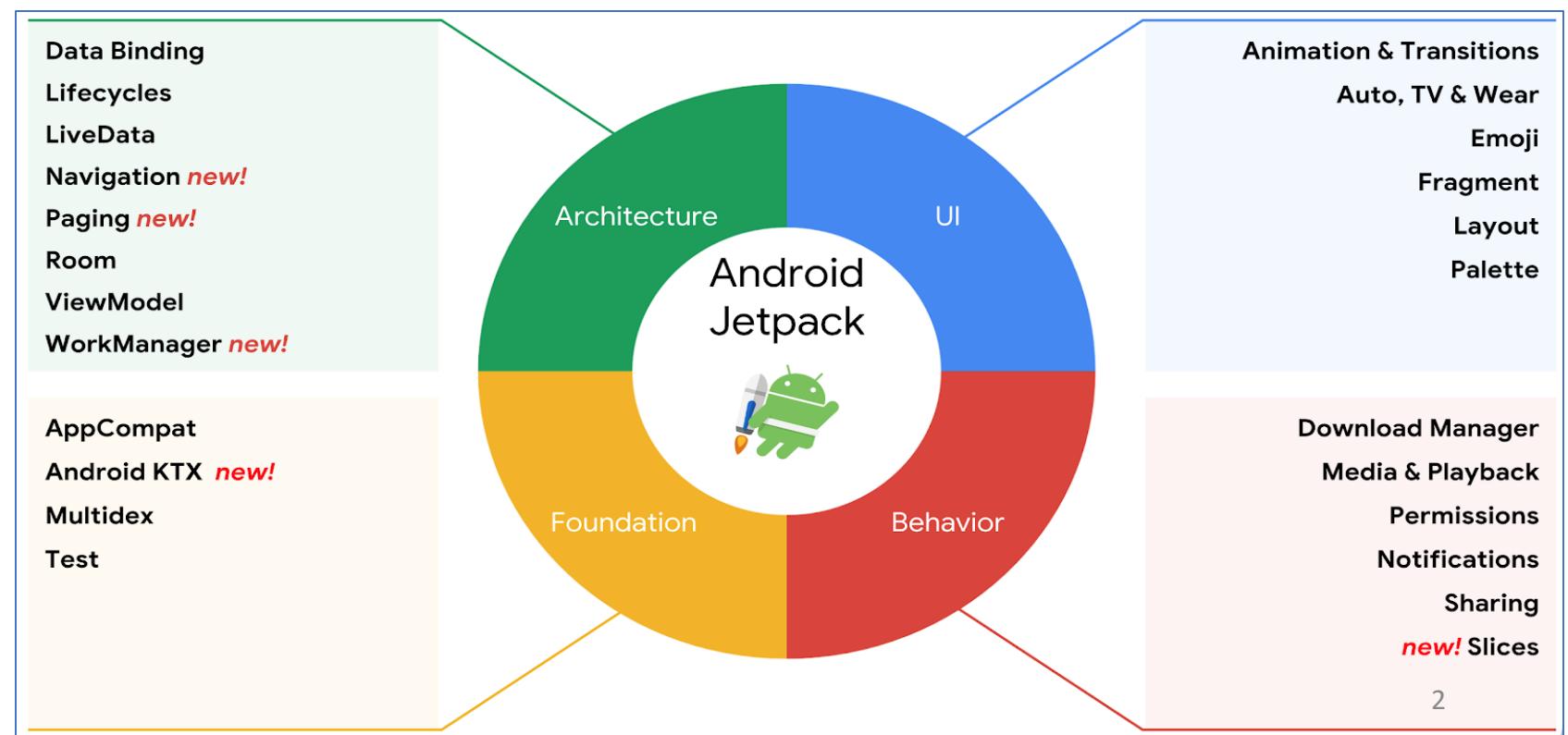


JetPack

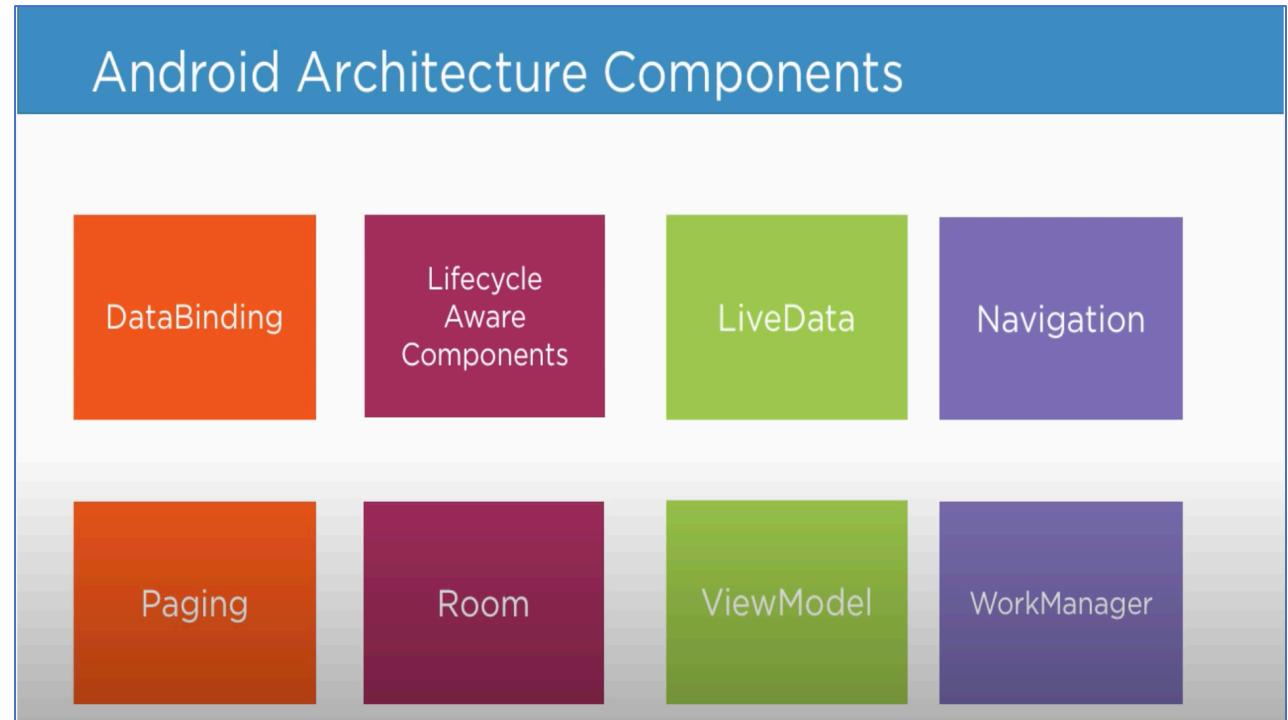


Qu'est ce que JetPack ?

- Jetpack est une suite de bibliothèques pour :
 - Aider les développeurs à suivre les meilleures pratiques.
 - Réduire le temps de développement.
- Les composants architecturaux se classent en 4 catégories :
 - Architecture
 - Foundation
 - UI
 - Behaviour



Les composants « Architecture »

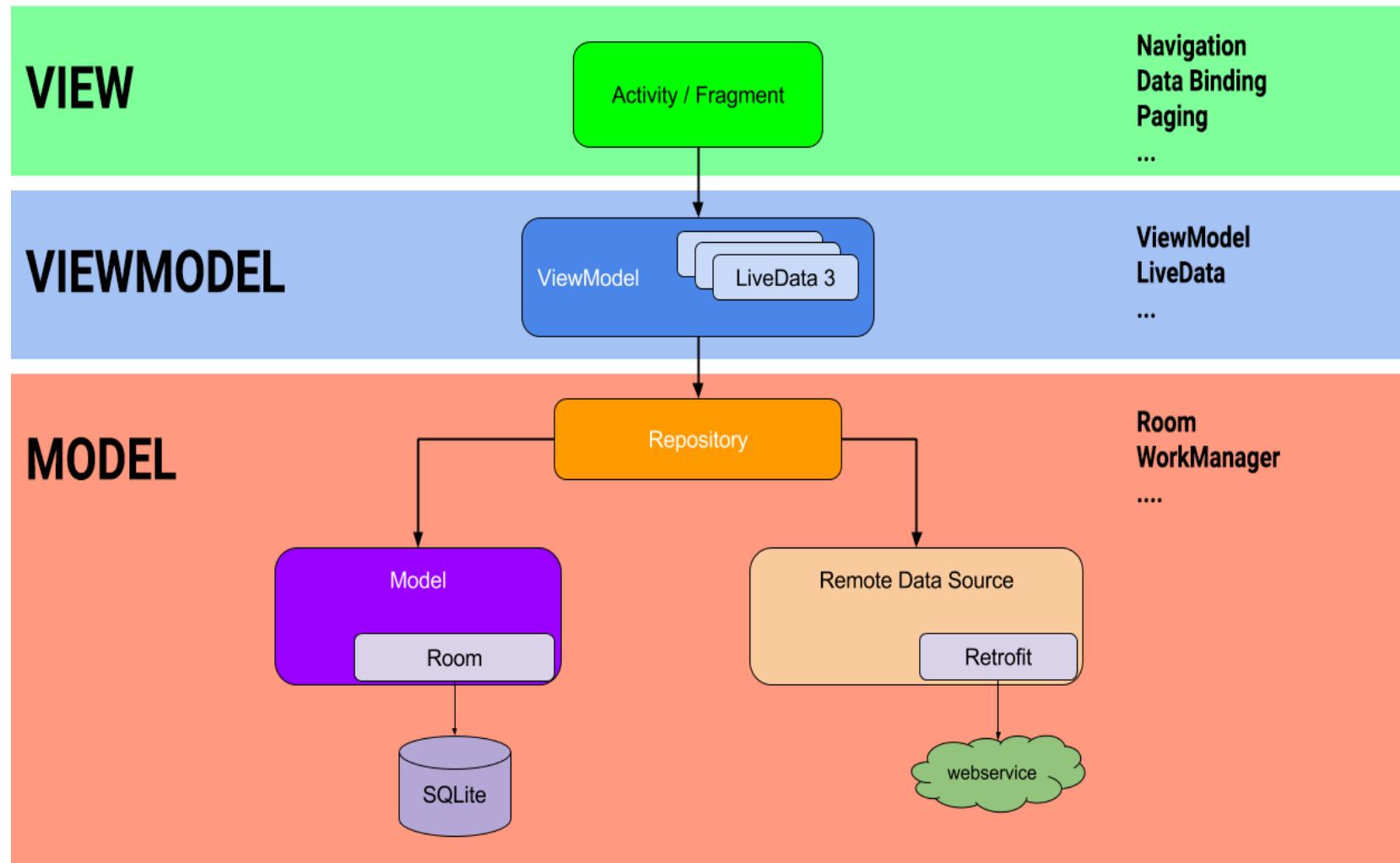


- Les composants de l'architecture Android:
 - Une partie d'Android Jetpack.
 - Sont un ensemble de bibliothèques qui aident à concevoir des applications robustes, testables et maintenables.

Les composants « Architecture » : La persistance de données

- Composants de l'architecture Android pour la gestion de la persistance:
 - **Composant pour Gérer le cycle de vie des composants d'interface utilisateur** et gérer la persistance des données et charger facilement les données dans l'interface utilisateur.
 - **LiveData** : composant pour créer des objets de données qui notifient les vues lorsque la base de données sous-jacente change.
 - **ViewModel** : composant pour stocker les données liées à l'interface utilisateur qui ne sont pas détruites lors des rotations d'applications.
 - **Room** : Une bibliothèque de mappage d'objets *SQLite*. Utilisée pour convertir facilement les données de table *SQLite* en objets Java.

Rôles des Composants d'Architecture



Composants sensibles au Cycle de vie

DataBinding

Lifecycle
Aware
Components

LiveData

Navigation

Paging

Room

ViewModel

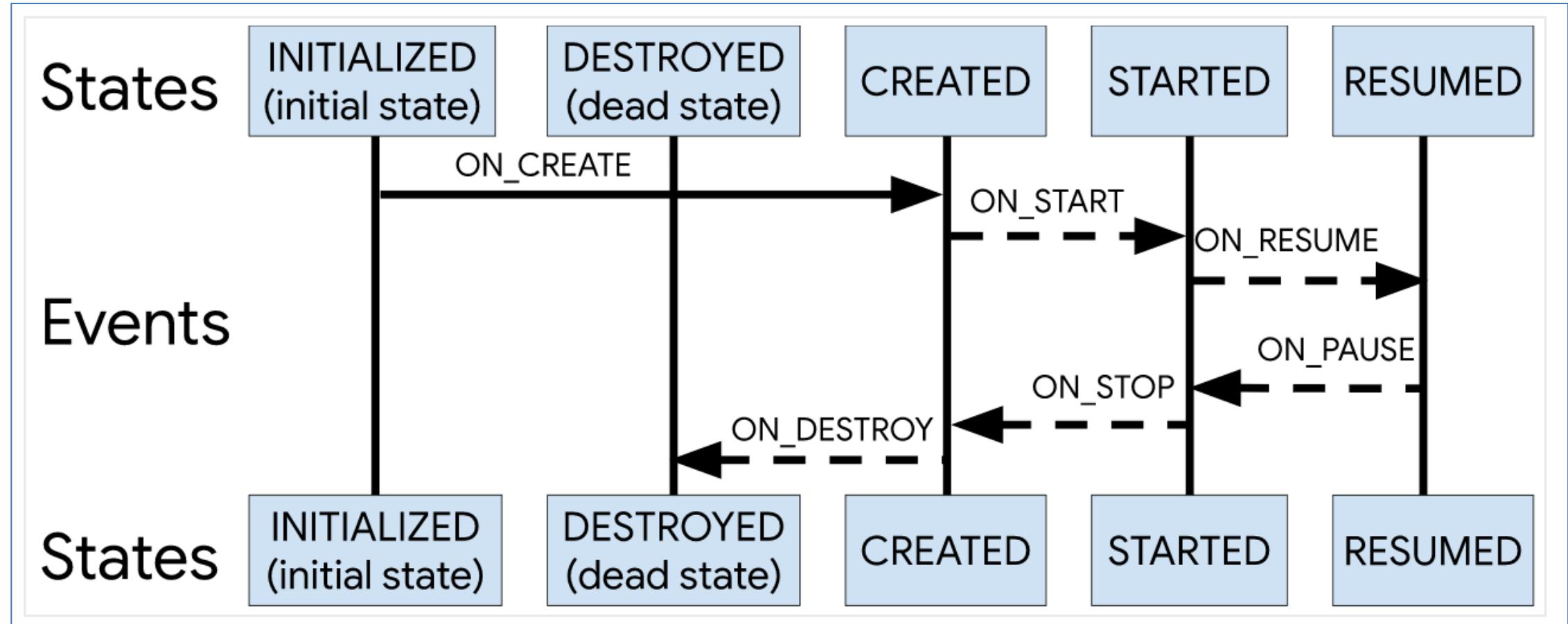
WorkManager

Gestion des composants sensibles au Cycle de Vie

Gestion des cycles de vie

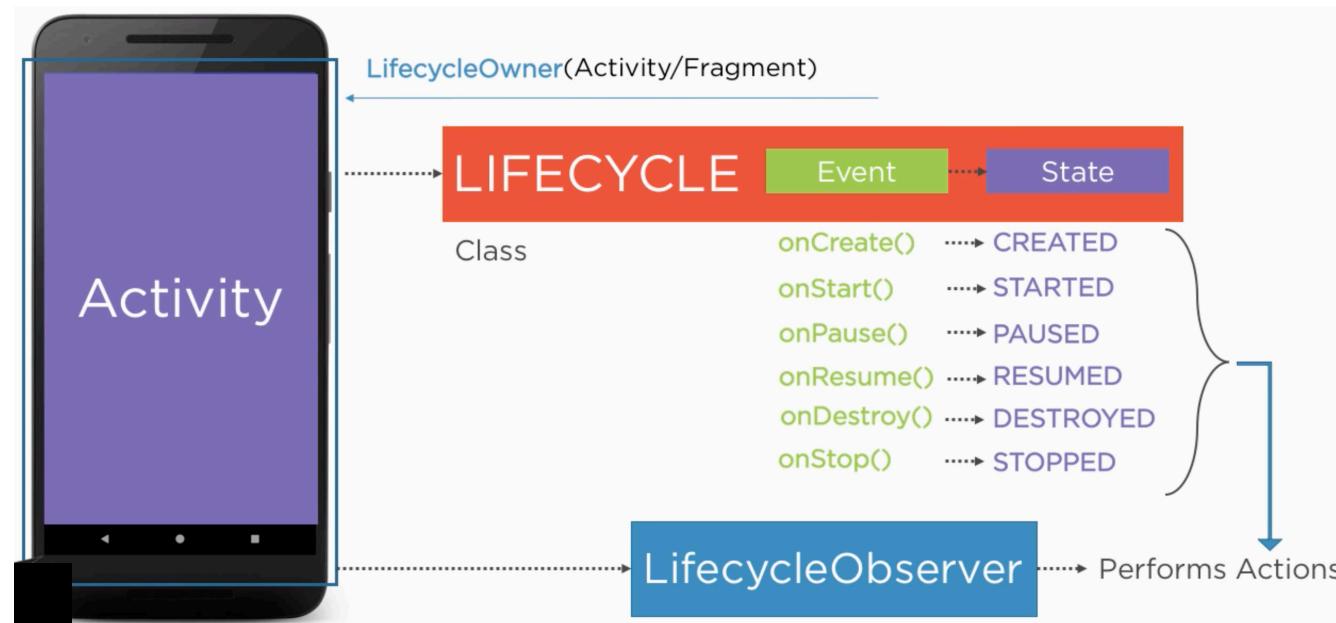
- Les composants prenant en charge le cycle de vie exécutent des actions en réponse à une modification de l'état du cycle de vie d'un autre composant, comme des activités et des fragments.
- Le package ***androidx.lifecycle*** fournit des classes et des interfaces qui permettent de créer des composants prenant en compte le cycle de vie:
 - Sont des composants qui peuvent automatiquement ajuster leur comportement en fonction de l'état actuel du cycle de vie d'une activité ou d'un fragment.

États et événements qui composent le cycle de vie de l'activité Android



Principes de fonctionnement

- Un composant sensible au **LifeCycle** permet de gérer les changements pendant le cycle de vie des composants UI.
- Le package associé à cette propriété : `android.arch.lifecycle`.
- La classe **Lifecycle** : fournit les informations liées au cycle de vie du propriétaire du cycle de vie (**LifecycleOwner**) pour permettre à d'autres objets d'observer cet état.
- Les objets instances de **Lifecycle** utilisent deux valeurs (énumération) pour réifier l'état du cycle de vie :
 - **Event** (Événement) : Les événements du cycle de vie qui sont distribués à partir de l'infrastructure et de la classe Lifecycle.
 - **State** (État) : L'état actuel du composant suivi par l'objet Lifecycle.
- **LifecycleOwner** fournit l'état du lifecycle (lifecycle status) aux composants sensibles au cycle de vie (**LifecycleObserver**).
- **LifecycleObserver** enregistre l'état du cycle de vie pour réaliser des actions spécifiques.



Propriétaire et observateur du cycle de vie

- Une classe peut surveiller l'état du cycle de vie du composant **en ajoutant des annotations** à ses méthodes.
- Ensuite, ajouter un observateur en appelant la méthode **addObserver ()** de la classe **Lifecycle** et en passant une instance de votre observateur.

```
// LifecycleOwner
public class MainActivity extends AppCompatActivity {

    private String TAG = this.getClass().getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Log.i(TAG, msg: "Owner ON_CREATE");
        getLifecycle().addObserver(new MainActivityObserver());
    }
}

// LifecycleObserver
public class MainActivityObserver implements LifecycleObserver {

    private String TAG = this.getClass().getSimpleName();

    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    public void onCreateEvent() {
        Log.i(TAG, msg: "Observer ON_CREATE");
    }
}
```

Déclarer les dépendances adéquates

- Pour ajouter une dépendance à **Lifecycle**, il faut ajouter le référentiel Google *Maven* au projet.
- Ajouter les dépendances pour les artefacts nécessaires dans le fichier *build.gradle* de l'application.
- Remarque
 - Les API des extensions de cycle de vie sont devenues obsolètes.
 - Ajouter plutôt des dépendances pour les artefacts de cycle de vie spécifiques.

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation 'com.android.support:appcompat-v7:27.1.1'  
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'com.android.support.test:runner:1.0.2'  
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'  
  
    def lifecycle_version = "1.1.1"  
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"  
    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version"  
}
```

```
dependencies {  
    def lifecycle_version = "2.3.0"  
    def arch_version = "2.1.0"  
  
    // ViewModel  
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"  
    // LiveData  
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"  
    // Lifecycles only (without ViewModel or LiveData)  
    implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"  
  
    // Saved state module for ViewModel  
    implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$lifecycle_version"  
  
    // Annotation processor  
    annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"  
    // alternately - if using Java8, use the following instead of lifecycle-compiler  
    implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"  
  
    // optional - helpers for implementing LifecycleOwner in a Service  
    implementation "androidx.lifecycle:lifecycle-service:$lifecycle_version"  
  
    // optional - ProcessLifecycleOwner provides a lifecycle for the whole application process  
    implementation "androidx.lifecycle:lifecycle-process:$lifecycle_version"  
  
    // optional - ReactiveStreams support for LiveData  
    implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"  
  
    // optional - Test helpers for LiveData  
    testImplementation "androidx.arch.core:core-testing:$arch_version"  
}
```

ViewModel

Qu'est ce que le composant ViewModel ?

- La classe **ViewModel** est conçue pour stocker et gérer les données liées à l'interface utilisateur en tenant compte du cycle de vie.
- La classe **ViewModel** permet aux données de survivre aux modifications de configuration telles que les rotations d'écran.

Pourquoi le composant ViewModel ?

- Couplage du cycle de vie UI et du cycle de vie contrôleur
 - Le framework Android gère les cycles de vie des contrôleurs d'interface utilisateur, tels que les activités et les fragments.
 - Le framework peut décider de détruire ou de recréer un contrôleur d'interface utilisateur en réponse à certaines actions utilisateur ou événements de périphérique qui sont complètement hors de votre contrôle.
 - Si le système détruit ou recrée un contrôleur d'interface utilisateur, toutes les données transitoires liées à l'interface utilisateur que vous y stockez sont perdues.
 - Exemple : l'application peut inclure une liste d'utilisateurs dans l'une de ses activités. Lorsque l'activité est recréée pour un changement de configuration, la nouvelle activité doit récupérer à nouveau la liste des utilisateurs.
 - Pour les données simples, l'activité peut utiliser la méthode **onSaveInstanceState ()** et restaurer ses données à partir du bundle dans **onCreate ()**, mais cette approche ne convient que pour de petites quantités de données qui peuvent être sérialisées puis déserialisées, pas pour des quantités potentiellement importantes de données comme une liste d'utilisateurs ou de bitmaps.



Pourquoi le composant ViewModel ?

- Séparation des préoccupations

- Les contrôleurs d'interface utilisateur doivent fréquemment effectuer des appels asynchrones dont le retour peut prendre un certain temps.
 - Le contrôleur d'interface utilisateur gère ces appels et s'assurer que le système les nettoie après sa destruction pour éviter les fuites de mémoire potentielles.
 - Cette gestion nécessite beaucoup de maintenance, et dans le cas où l'objet est recréé pour un changement de configuration, c'est un gaspillage de ressources puisque l'objet peut avoir à réémettre les appels qu'il a déjà effectués.
- Les contrôleurs d'interface utilisateur tels que les activités et les fragments sont principalement destinés à afficher des données d'interface utilisateur, à réagir aux actions de l'utilisateur ou à gérer les communications du système d'exploitation, telles que les demandes d'autorisation.
- Exiger des contrôleurs d'interface utilisateur qu'ils soient également responsables du chargement des données à partir d'une base de données ou d'un réseau ajoute des préoccupations supplémentaires à la classe.
- Attribuer une responsabilité excessive aux contrôleurs de l'interface utilisateur de cette manière rend également les tests beaucoup plus difficiles.
- Conclusion : Il est plus facile et plus efficace de séparer la propriété des données de vue de la logique du contrôleur d'interface utilisateur.



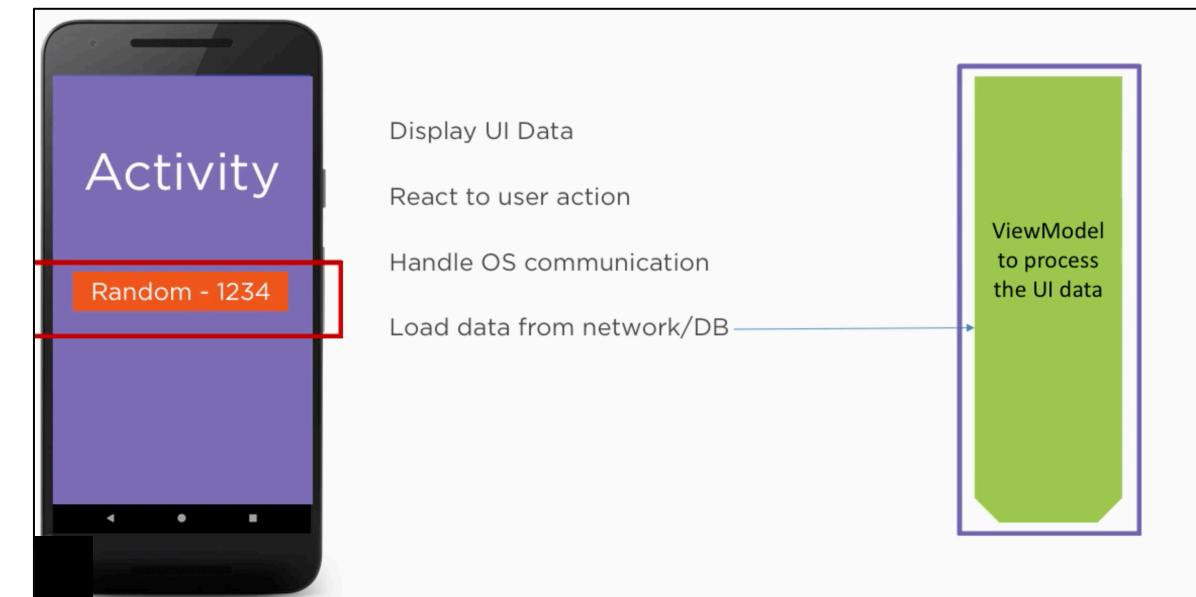
- Display UI Data
- React to user action
- Handle OS communication
- Load data from network/DB

ViewModel : Séparation des cycles de vie des préoccupations

- **ViewModel** est responsable de la préparation des données pour l'interface utilisateur.
- Les objets **ViewModel** sont automatiquement conservés lors des modifications de configuration afin que les données qu'ils contiennent soient immédiatement disponibles pour l'activité ou l'instance de fragment suivante.



Séparer les cycle de vie du contrôleur et celui de l'UI



Séparer les préoccupations

Implémenter ViewModel

- Exemple 1 : Afficher un nombre aléatoire
 - Classe qui gère les données de l'UI

```
public class MainActivityDataGenerator {  
  
    private String TAG = this.getClass().getSimpleName();  
    private String myRandomNumber;  
  
    public String getNumber() {  
        Log.i(TAG, msg: "Get number");  
        if (myRandomNumber == null) {  
            createNumber();  
        }  
        return myRandomNumber;  
    }  
  
    private void createNumber() {  
        Log.i(TAG, msg: "Create new number");  
        Random random = new Random();  
        myRandomNumber = "Number: " + (random.nextInt( bound: 10 - 1 ) + 1);  
    }  
}
```

Implémenter ViewModel

- Exemple 1 : Afficher un nombre aléatoire
 - Classe contrôleur : coupage entre données UI et Cycle vie Contrôleur

```
public class MainActivity extends AppCompatActivity {

    private String TAG = this.getClass().getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show();
            }
        });
    }

    TextView mTextView = findViewById(R.id.tvNumber);
    MainActivityDataGenerator myData = new MainActivityDataGenerator();
    String myRandomNumber = myData.getNumber();
    mTextView.setText(myRandomNumber);
}
```

Implémenter ViewModel

- Exemple 1 : Afficher un nombre aléatoire
 - Création d'une classe ViewModel

```
public class MainActivityDataGenerator extends ViewModel {  
  
    private String TAG = this.getClass().getSimpleName();  
    private String myRandomNumber;  
  
    public String getNumber() {  
        Log.i(TAG, msg: "Get number");  
        if (myRandomNumber == null) {  
            createNumber();  
        }  
        return myRandomNumber;  
    }  
  
    private void createNumber() {  
        Log.i(TAG, msg: "Create new number");  
        Random random = new Random();  
        myRandomNumber = "Number: " + (random.nextInt( bound: 10 - 1) + 1);  
    }  
  
    @Override  
    protected void onCleared() {  
        super.onCleared();  
        Log.i(TAG, msg: "ViewModel Destroyed");  
    }  
}
```

Implémenter ViewModel

- Exemple 1 : Afficher un nombre aléatoire
 - Récupérer une instance ViewModel

```
public class MainActivity extends AppCompatActivity {

    private String TAG = this.getClass().getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = findViewById(R.id.fab);
        fab.setOnClickListener((view) -> {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
        });
    }

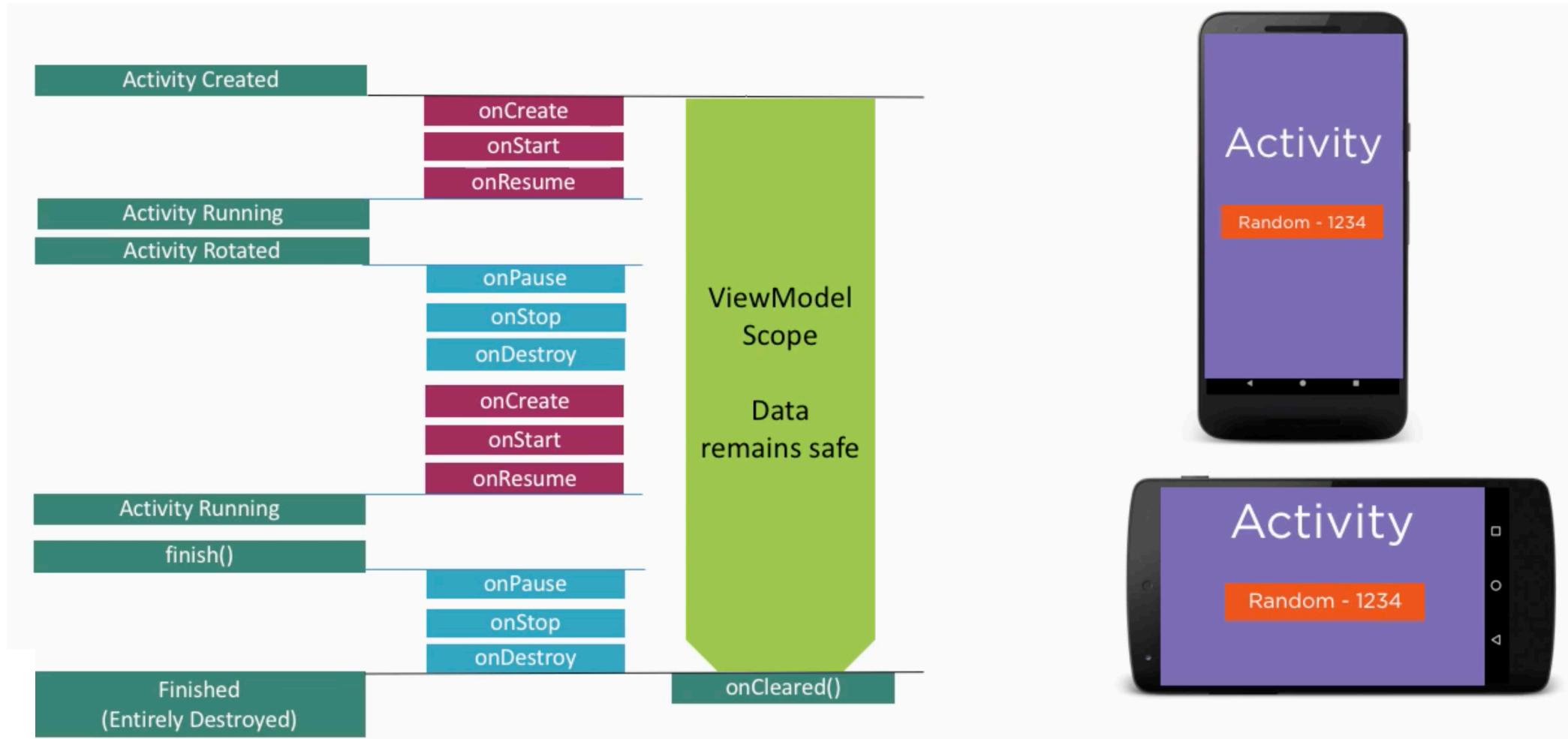
    // ...
    TextView mTextView = findViewById(R.id.tvNumber);
    MainActivityDataGenerator myData = new MainActivityDataGenerator();
    MainActivityDataGenerator model = ViewModelProviders.of(activity: this).get(MainActivityDataGenerator.class);
    String myRandomNumber = model.getNumber();
    mTextView.setText(myRandomNumber);

    Log.i(TAG, msg: "Random Number Set");
}
```

Implémenter ViewModel

- Un ViewModel est demandé la première fois que le système appelle la méthode **onCreate ()** d'un objet d'activité. Le système peut appeler **onCreate ()** plusieurs fois au cours de la vie d'une activité, par exemple lors de la rotation de l'écran d'un appareil.
- Le **ViewModel** existe à partir du moment où un **ViewModel** est demandé pour la première fois jusqu'à ce que l'activité soit terminée et détruite.
- Si l'activité est recréée, elle reçoit la même instance **ViewModel** que celle créée par la première activité.
- Lorsque l'activité du propriétaire est terminée, le framework appelle la méthode **onCleared ()** des objets **ViewModel** afin de pouvoir nettoyer les ressources.
- Remarque : Un **ViewModel** ne doit jamais faire référence :
 - A une vue
 - A un cycle de vie ou une classe pouvant contenir une référence au contexte d'activité.

Le cycle de vie d'un ViewModel



Implémenter un ViewModel

- Exemple 2 : Afficher une liste d'utilisateur

1. Attribuer la responsabilité d'acquérir et de conserver la liste des utilisateurs à un **ViewModel**, au lieu d'une activité ou d'un fragment.

2. Accéder à la liste à partir d'une activité

```
public class MyViewModel extends ViewModel {  
    private MutableLiveData<List<User>> users;  
    public LiveData<List<User>> getUsers() {  
        if (users == null) {  
            users = new MutableLiveData<List<User>>();  
            loadUsers();  
        }  
        return users;  
    }  
  
    private void loadUsers() {  
        // Do an asynchronous operation to fetch users.  
    }  
}
```

```
public class MyActivity extends AppCompatActivity {  
    public void onCreate(Bundle savedInstanceState) {  
        // Create a ViewModel the first time the system calls an activity's onCreate() method.  
        // Re-created activities receive the same MyViewModel instance created by the first activi-  
  
        MyViewModel model = new ViewModelProvider(this).get(MyViewModel.class);  
        model.getUsers().observe(this, users -> {  
            // update UI  
        });  
    }  
}
```

Implémenter un ViewModel

- Les objets **ViewModel** sont conçus pour survivre à des instantiations spécifiques de vues ou **LifecycleOwners**.
- Cette conception permet d'écrire des tests pour couvrir un **ViewModel** plus facilement car il ne connaît pas les objets de vue et de cycle de vie.
- Les objets **ViewModel** peuvent contenir des **LifecycleObservers**, tels que des objets **LiveData**.
- Cependant, les objets **ViewModel** ne doivent jamais observer les modifications des observables sensibles au cycle de vie, tels que les objets **LiveData**.
- Si le **ViewModel** a besoin du contexte **Application**, par exemple pour trouver un service système, il peut étendre la classe **AndroidViewModel** et avoir un constructeur qui reçoit l'Application dans le constructeur, car la classe Application étend Context.

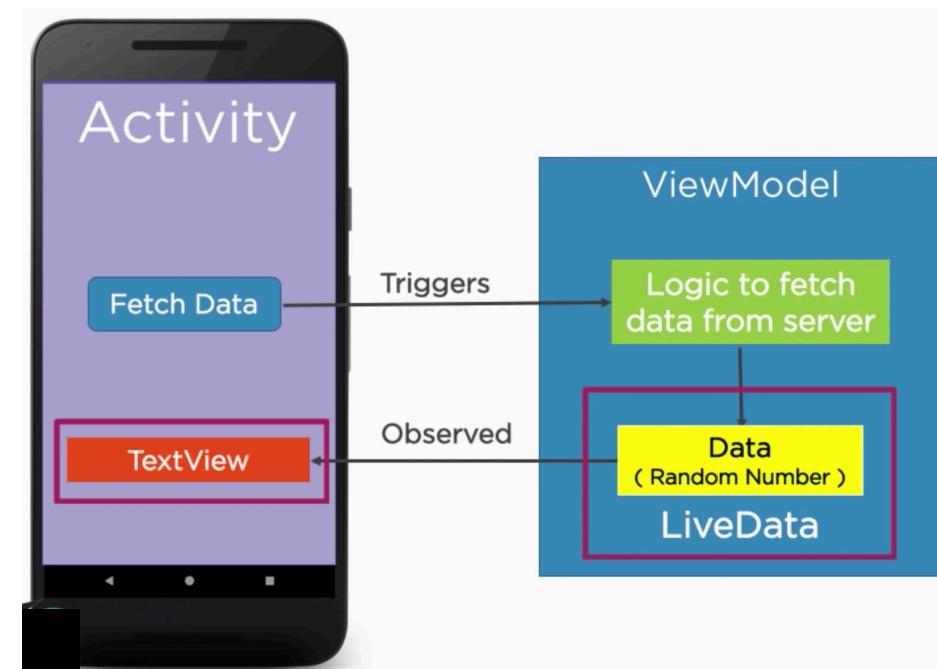
Implémenter un ViewModel

- Une instance de ViewModel
 - Survie après les changements de configurations, exemple : Rotation d'écran.
 - Pas la même chose que **onSaveInstanceState()**.
 - Utilisée pour des données complexes comme les listes et les bitmap.
 - Conserve et gère les données liées à l'UI.
 - Détruite uniquement si l'activité propriétaire est complètement détruite (par **onCleared()**).
 - Joue le rôle d'une couche de communication entre la base de données et l'UI.

LiveData

Qu'est ce que une LiveData ?

- **LiveData** est une classe détentrice de données observables.
- Contrairement à un observable ordinaire, **LiveData** est sensible au cycle de vie, ce qui signifie qu'il respecte le cycle de vie des autres composants de l'application, tels que les activités, les fragments ou les services.
 - Cette prise de conscience garantit que LiveData ne met à jour que les observateurs de composants d'application qui sont dans un état de cycle de vie actif.
- LiveData considère qu'un observateur, qui est représenté par la classe **Observer**, est dans un état actif si son cycle de vie est à l'état **STARTED** ou **RESUMED**.
- **LiveData** informe uniquement les observateurs actifs des mises à jour.
- Les observateurs inactifs enregistrés pour regarder les objets **LiveData** ne sont pas informés des modifications.
- Remarque: pour importer des composants **LiveData** dans un projet Android, il faut ajouter les dépendances nécessaires.



Les Avantages de l'Utilisation de LiveData

- Permet d'adapter l'interface utilisateur en fonction de l'état de vos données
 - **LiveData** suit le modèle d'observateur.
 - **LiveData** notifie les objets Observer lorsque les données sous-jacentes changent : pas besoin de mettre à jour l'interface utilisateur à chaque fois que les données de l'application changent, car l'observateur le fait pour vous.
- Aucune fuite de mémoire : Les observateurs sont liés aux objets **Lifecycle** et nettoient après eux-mêmes lorsque leur cycle de vie associé est détruit.
- Pas de plantages dus à des activités arrêtées : Si le cycle de vie de l'observateur est inactif, comme dans le cas d'une activité dans la pile arrière, alors il ne reçoit aucun événement **LiveData**.
- Pas de gestion manuelle du cycle de vie : Les composants de l'interface utilisateur observent simplement les données pertinentes et ne s'arrêtent ni ne reprennent l'observation.
 - **LiveData** gère automatiquement tout cela car il est conscient des changements d'état du cycle de vie lors de l'observation.
- Données toujours à jour : Si un cycle de vie devient inactif, il reçoit les dernières données lorsqu'il redevient actif. Par exemple, une activité qui était en arrière-plan reçoit les dernières données juste après son retour au premier plan.
- Changements de configuration appropriés : Si une activité ou un fragment est recréé en raison d'un changement de configuration, comme la rotation de l'appareil, il reçoit immédiatement les dernières données disponibles.
- Etc.

Utiliser des objets LiveData

- Les étapes pour utiliser des objets **LiveData**:
 - Créer une instance de **LiveData** pour contenir un certain type de données.
 - Cela se fait généralement dans votre classe ViewModel.
 - Créer un objet **Observer** qui définit la méthode **onChanged ()**, qui contrôle ce qui se passe lorsque les données conservées de l'objet **LiveData** changent.
 - Créer généralement un objet **Observer** dans un contrôleur d'interface utilisateur, tel qu'une activité ou un fragment.
 - Attacher l'objet **Observer** à l'objet **LiveData** à l'aide de la méthode **observe ()**.
 - La méthode **observe ()** prend un **objet LifecycleOwner**.
 - Cela abonne l'objet Observer à l'objet **LiveData** afin qu'il soit informé des modifications.
 - Attacher généralement l'objet **Observer** à un contrôleur d'interface utilisateur, tel qu'une activité ou un fragment.

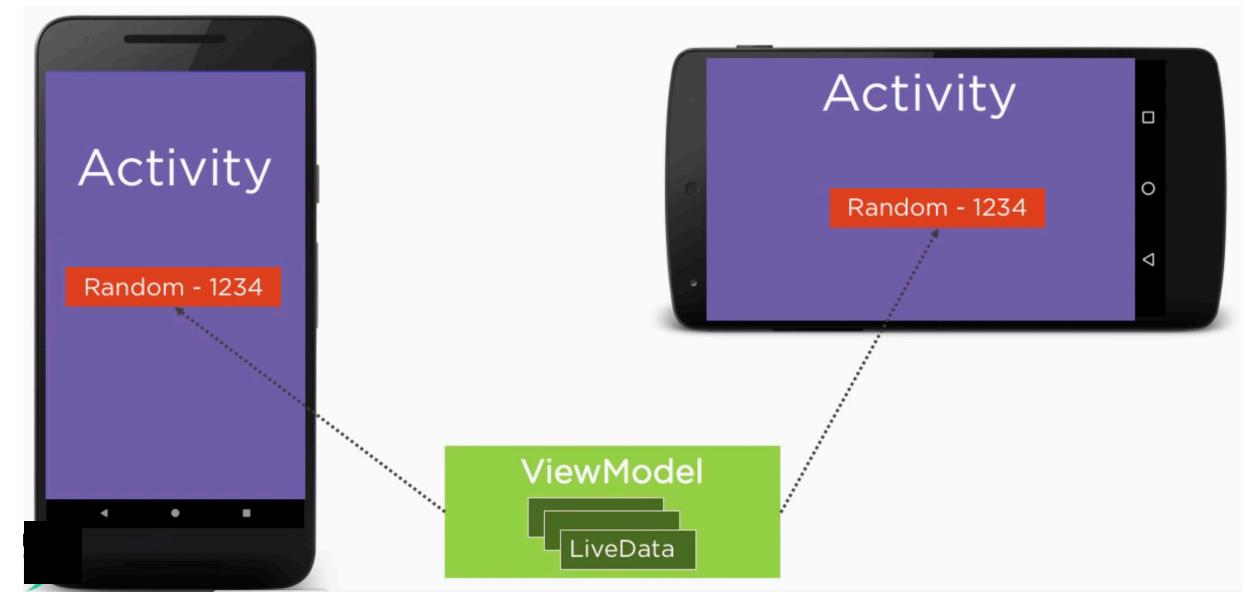
Utiliser des objets LiveData

- Remarques:
 - Il est possible d'enregistrer un observateur sans objet **LifecycleOwner** associé à l'aide de la méthode **observeForever** (**Observer**).
 - Dans ce cas, l'observateur est considéré comme toujours actif et est donc toujours informé des modifications. Vous pouvez supprimer ces observateurs en appelant la méthode **removeObserver** (**Observer**).
 - Lorsque la valeur stockée dans l'objet **LiveData** est mise à jour, cela déclenche tous les observateurs enregistrés tant que le **LifecycleOwner** attaché est à l'état actif.
 - **LiveData** permet aux observateurs du contrôleur de l'interface utilisateur de s'abonner aux mises à jour.
 - Lorsque les données détenues par l'objet **LiveData** changent, l'interface utilisateur se met automatiquement à jour en réponse.

Créer des objets LiveData

- **LiveData** est un wrapper qui peut être utilisé avec toutes les données, y compris les objets qui implémentent des collections, comme List.
- Un objet LiveData est généralement stocké dans un objet ViewModel et est accessible via une méthode getter, comme illustré dans l'exemple suivant:

```
public class NameViewModel extends ViewModel {  
  
    // Create a LiveData with a String  
    private MutableLiveData<String> currentName;  
  
    public MutableLiveData<String> getCurrentName() {  
        if (currentName == null) {  
            currentName = new MutableLiveData<String>();  
        }  
        return currentName;  
    }  
  
    // Rest of the ViewModel...  
}
```



Observer les objets LiveData

- Dans la plupart des cas, la méthode **onCreate ()** d'un composant d'application est le bon endroit pour commencer à observer un objet **LiveData** pour les raisons suivantes:
 - Pour garantir que le système n'effectue pas d'appels redondants à partir d'une activité ou de la méthode **onResume ()** d'un fragment.
 - Pour s'assurer que l'activité ou le fragment contient des données qu'il peut afficher dès qu'il devient actif.
- Dès qu'un composant d'application est à l'état **STARTED**, il reçoit la valeur la plus récente des objets **LiveData** qu'il observe. Cela ne se produit que si l'objet **LiveData** à observer a été défini.
- En règle générale, **LiveData** fournit des mises à jour uniquement lorsque les données changent, et uniquement aux observateurs actifs.
 - Une exception à ce comportement est que les observateurs reçoivent également une mise à jour lorsqu'ils passent d'un état inactif à un état actif.
- De plus, si l'observateur passe d'inactif à actif une seconde fois, il ne reçoit une mise à jour que si la valeur a changé depuis la dernière fois qu'il est devenu actif.

Observer les objets LiveData

- Exemple

- Après avoir appelé **observe ()** avec **nameObserver** passé en paramètre
- **onChanged ()** est immédiatement appelé en fournissant la valeur la plus récente stockée dans **mCurrentName**.
- Si l'objet **LiveData** n'a pas défini de valeur dans **mCurrentName**, **onChanged ()** n'est pas appelé.

```
public class NameActivity extends AppCompatActivity {  
  
    private NameViewModel model;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Other code to setup the activity...  
  
        // Get the ViewModel.  
        model = new ViewModelProvider(this).get(NameViewModel.class);  
  
        // Create the observer which updates the UI.  
        final Observer<String> nameObserver = new Observer<String>() {  
            @Override  
            public void onChanged(@Nullable final String newName) {  
                // Update the UI, in this case, a TextView.  
                nameTextView.setText(newName);  
            }  
        };  
  
        // Observe the LiveData, passing in this activity as the LifecycleOwner and the  
        // observer.  
        model.getCurrentName().observe(this, nameObserver);  
    }  
}
```

Mettre à jour les objets LiveData

- **LiveData** ne dispose d'aucune méthode accessible au public pour mettre à jour les données stockées.
- La classe **MutableLiveData** expose publiquement les méthodes **setValue (T)** et **postValue (T)** et sont utilisées pour modifier la valeur stockée dans un objet **LiveData**.
- Généralement, **MutableLiveData** est utilisée dans le **ViewModel**, puis le **ViewModel** expose uniquement des objets **LiveData** immuables aux observateurs.
- Après avoir configuré la relation d'observateur, mettre à jour la valeur de l'objet **LiveData**, déclenche tous les observateurs.

```
package android.arch.lifecycle;

/**
 * {@link LiveData} which publicly exposes {@link #setValue(T)} and {@link #postValue(T)}
 *
 * @param <T> The type of data hold by this instance
 */
@WeakerAccess
public class MutableLiveData<T> extends LiveData<T> {
    @Override
    public void postValue(T value) { super.postValue(value); }

    @Override
    public void setValue(T value) { super.setValue(value); }
}
```

Mettre à jour les objets LiveData

- L'exemple montre une pression sur un bouton pour mettre à jour la valeur de **LiveData**.
- La mise à jour de l'objet **LiveData**, comme illustré par l'exemple ci-contre, déclenche tous les observateurs lorsque l'utilisateur appuie sur un bouton:
- L'appel de **setValue (T)** dans l'exemple permet aux observateurs d'appeler leurs méthodes **onChanged ()** avec la valeur John Doe.
- **setValue ()** ou **postValue ()** peuvent être appelés pour mettre à jour *mName* pour diverses raisons, notamment en réponse à une demande réseau ou à un chargement de base de données terminé; l'appel à **setValue ()** ou **postValue ()** déclenche des observateurs et met à jour l'interface utilisateur.
- Remarque: appeler la méthode **setValue (T)** pour mettre à jour l'objet **LiveData** à partir du thread principal.
- Si le code est exécuté dans un thread de travail, utiliser la méthode **postValue (T)** à la place pour mettre à jour l'objet LiveData.

```
button.setOnClickListener(new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        String anotherName = "John Doe";  
        model.getCurrentName().setValue(anotherName)  
    }  
});
```

Mettre à jour les objets LiveData

- Exemple

```
public class MainActivityViewModel extends ViewModel {  
  
    private String TAG = this.getClass().getSimpleName();  
    private String myRandomNumber;  
  
    public String getNumber() {  
        Log.i(TAG, msg: "Get number");  
        if (myRandomNumber == null) {  
            createNumber();  
        }  
        return myRandomNumber;  
    }  
  
    private void createNumber() {  
        Log.i(TAG, msg: "Create new number");  
        Random random = new Random();  
        myRandomNumber = "Number: " + (random.nextInt( bound: 10 - 1 ) + 1);  
    }  
}
```

```
public class MainActivityViewModel extends ViewModel {  
  
    private String TAG = this.getClass().getSimpleName();  
    private MutableLiveData<String> myRandomNumber;  
  
    public MutableLiveData<String> getNumber() {  
        Log.i(TAG, msg: "Get number");  
        if (myRandomNumber == null) {  
            myRandomNumber = new MutableLiveData<>();  
            createNumber();  
        }  
        return myRandomNumber;  
    }  
  
    private void createNumber() {  
        Log.i(TAG, msg: "Create new number");  
        Random random = new Random();  
        myRandomNumber.setValue("Number: " + (random.nextInt( bound: 10 - 1 ) + 1));  
    }  
}
```

Utiliser LiveData avec Room

- La bibliothèque de persistance **Room** prend en charge les requêtes observables, qui renvoient des objets **LiveData**.
- Les requêtes observables sont écrites dans le cadre d'un objet d'accès à la base de données (DAO).
- Room génère tout le code nécessaire pour mettre à jour l'objet **LiveData** lorsqu'une base de données est mise à jour.
 - Le code généré exécute la requête de manière asynchrone sur un thread d'arrière-plan si nécessaire.
- Ce modèle est utile pour synchroniser les données affichées dans une interface utilisateur avec les données stockées dans une base de données.

Gestion des dépendances

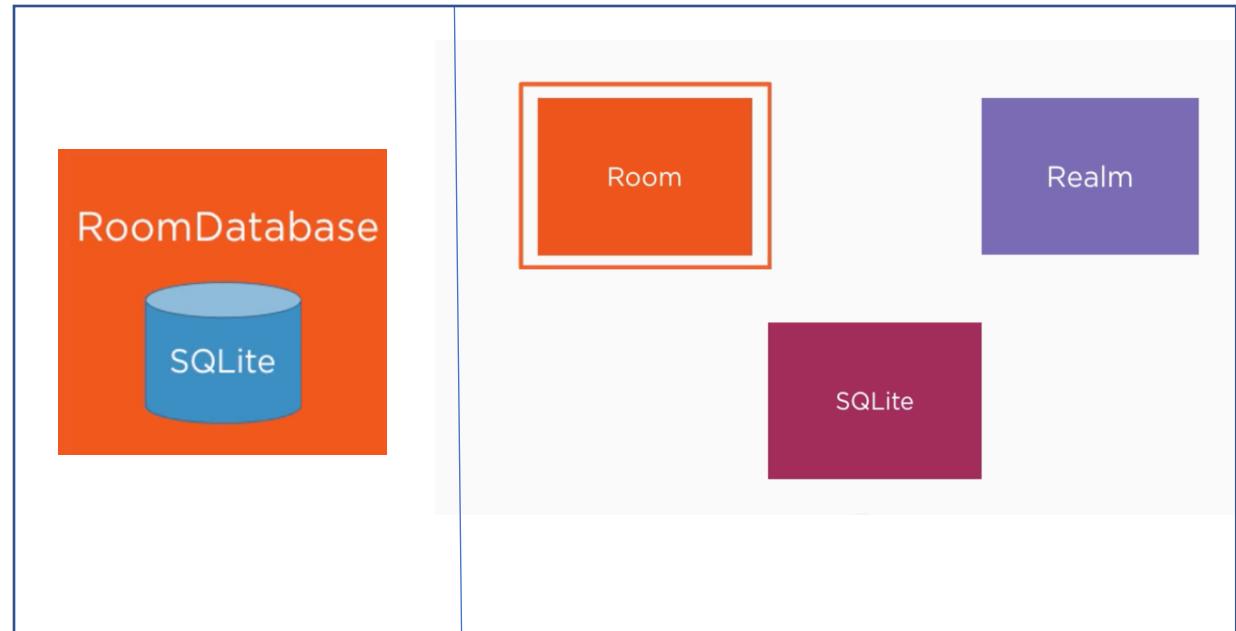
```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:27.1.1'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    implementation 'com.android.support:design:27.1.1'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'

    def lifecycle_version = "1.1.1"
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"
    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version"
}
```

ROOM

Qu'est ce que Room ?

- Les applications qui gèrent des quantités non négligeables de données structurées peuvent bénéficier de la persistance de ces données localement via **Room**.
 - Exemple : Mettre en cache des éléments de données pertinents de sorte que lorsque l'appareil ne peut pas accéder au réseau, l'utilisateur puisse toujours parcourir ce contenu lorsqu'il est hors ligne.
- Une bibliothèque qui gère les données persistantes comme une couche au dessus de SQLite.
- Fournit une couche d'abstraction par rapport à **SQLite** pour permettre une manipulation simplifiée des BDs.
- Constitue une bibliothèque ORM (Object Relation Mapping)

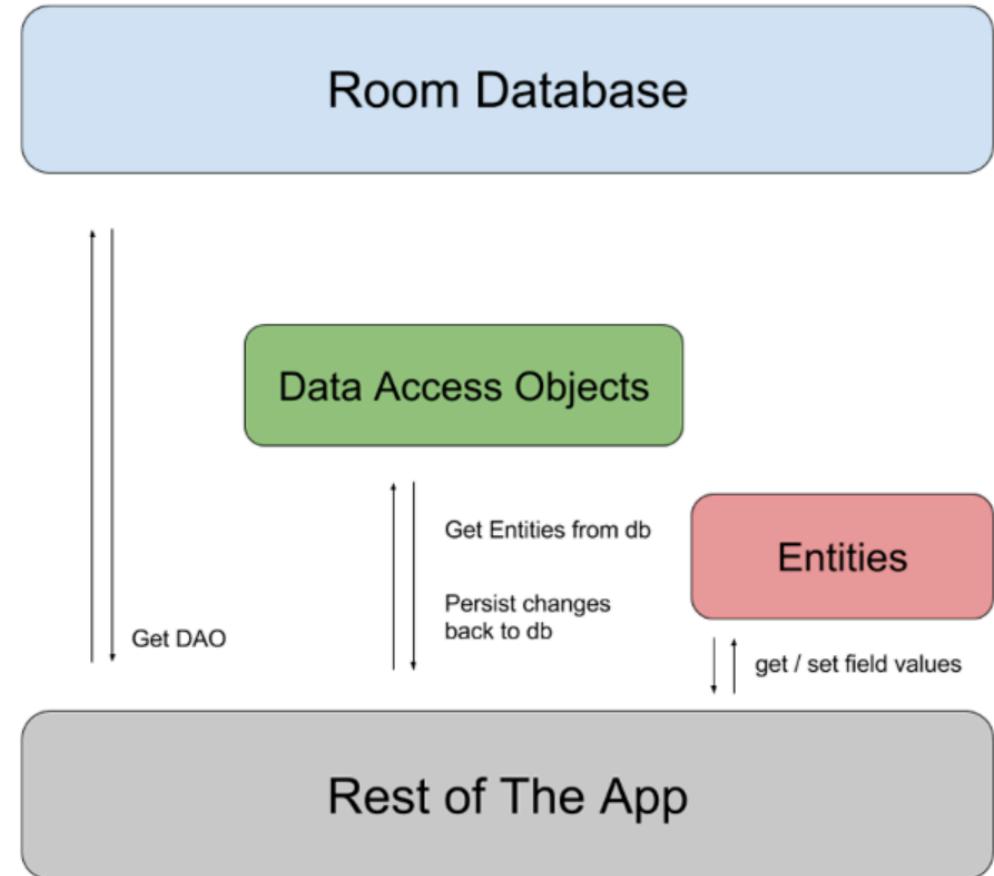


Pourquoi Room ?

| SQLite | Room |
|--------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| Manipulation de requêtes brutes. | Pas de requêtes brutes. |
| Pas de vérification pendant la compilation des requêtes brutes. | Vérification à la compilation des traitements SQLite. |
| Beaucoup de duplication (redondances) pour les conversions entre les requêtes SQL et les objets-données | Lien direct entre les objets de la base de données et les objets de l'application (Java) sans aucun code (redondant) |
| Les APIs SQLLine sont offrent des traitements bas-niveau: nécessite beaucoup de temps et d'effort de développement | Room facilite la tâche de manipulation des BDs surtout quand utilisée avec ViewModel et LiveData. |

Les composants de Room

- Les entités (**Entities**): représentent des tables dans la base de données de votre application
 - Définit le schéma des tables de la BD.
 - Annotées avec **@Entity**.
- Les objets d'accès aux données (**DAO**): fournissent des méthodes que l'application utilise pour interroger, mettre à jour, insérer et supprimer des données dans la base de données.
 - Contiennent les méthodes pour accéder à la BD.
 - Annotées avec **@Dao**.
- Base de données (**Database**): La classe de base de données qui contient la base de données et sert de point d'accès principal pour la connexion sous-jacente aux données persistantes de l'application.
 - La classe définissant la BD.
 - Annotée avec **@Database**.



Installer Room

- Pour utiliser **Room**, ajouter les dépendances suivantes au fichier **build.gradle** de l'application:

```
dependencies {  
    def room_version = "2.2.6"  
  
    implementation "androidx.room:room-runtime:$room_version"  
    annotationProcessor "androidx.room:room-compiler:$room_version"  
  
    // optional - RxJava support for Room  
    implementation "androidx.room:room-rxjava2:$room_version"  
  
    // optional - Guava support for Room, including Optional and ListenableFuture  
    implementation "androidx.room:room-guava:$room_version"  
  
    // optional - Test helpers  
    testImplementation "androidx.room:room-testing:$room_version"  
}
```

Data Entity (Entité de données)

- Le code suivant définit une entité de données utilisateur.
- Chaque instance de **User** représente une ligne dans une table **user** dans la base de données de l'application.

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Data access object (DAO)- Objet d'accès aux données

- Le code suivant définit un DAO appelé ***UserDao***.
- ***UserDao*** fournit les méthodes que le reste de l'application utilise pour interagir avec les données de la table utilisateur.

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

Database- Base de données

- Le code suivant définit une classe ***AppDatabase*** pour contenir la base de données.
- ***AppDatabase*** définit la configuration de la base de données et sert de point d'accès principal de l'application aux données persistantes.
- La classe de base de données doit remplir les conditions suivantes:
 - La classe doit être annotée avec une annotation ***@Database*** qui inclut un tableau d'entités qui répertorie toutes les entités de données associées à la base de données.
 - La classe doit être *une classe abstraite* qui étend ***RoomDatabase***.
 - Pour chaque classe DAO associée à la base de données, la classe de base de données doit définir une méthode abstraite qui n'a aucun argument et renvoie une instance de la classe DAO.

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

Usage

- Une fois les entités de données, les DAO et l'objet de base de données définis, utiliser le code suivant pour créer une instance de la base de données:

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
    AppDatabase.class, "database-name").build();
```

- Ensuite
 - Utiliser les méthodes abstraites de l'**AppDatabase** pour obtenir une instance de DAO.
 - Utiliser les méthodes de l'instance DAO pour interagir avec la base de données:

```
UserDao userDao = db.userDao();
List<User> users = userDao.getAll();
```