



Programmation Android

IV. Cycle de vie d'une activité

Plan

- 1 **Vue globale du cycle de vie d'une activité**
- 2 **Séquence de démarrage d'une activité**
- 3 **onPause et onResume**
- 4 **Arrêt et redémarrage**
- 5 **Destruction**
- 6 **Restauration**
- 7 ***Lifecycle-Aware Components (Jetpack)***

Plan

- 1 **Vue globale du cycle de vie d'une activité**
- 2 Séquence de démarrage d'une activité
- 3 onPause et onResume
- 4 Arrêt et redémarrage
- 5 Destruction
- 6 Restauration
- 7 *Lifecycle-Aware Components (Jetpack)*

Cycle de vie d'une activité

Pas de méthode `main()`

- Vous aurez remarqué que nos programmes ne possèdent pas de méthode `main()`
- Sous Android, c'est l'OS qui gère l'intégralité du cycle de vie des activités : initialisation, démarrage, prise/perde de focus, destruction...

The Activity Lifecycle



État d'une activité

Au fil des interactions utilisateurs et des événements de l'OS...

Une activité peut être dans un des 4 états suivants

- **resumed** : en avant plan
- **paused** : perte de focus mais encore visible (encore entièrement en mémoire)
- **stopped** : complètement obscurcie par une autre activité (peut être tuée pour récupérer de la mémoire)
- **off** : non démarrée. Le système peut terminer une activité en pause ou stoppée, elle repasse alors dans l'état off et devra être entièrement redémarrée

Principe de gestion du cycle de vie

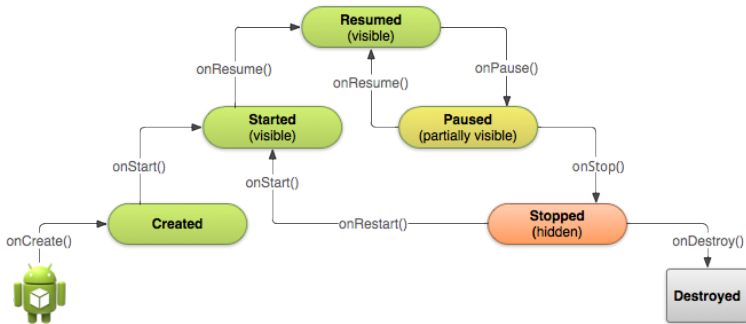
Passage d'un état à un autre \Rightarrow **appel automatique (*callback*) d'une méthode** spécifique de l'activité, par exemple `onCreate`

Coder les bons traitements dans les bonnes méthodes est essentiel pour la robustesse et la performance de l'application

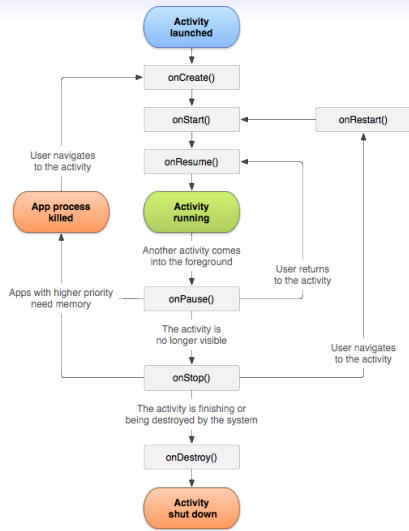
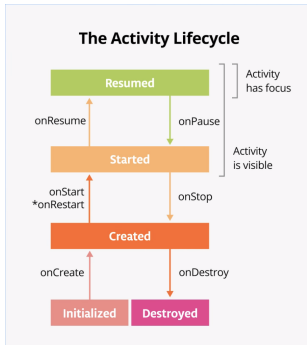
Bien coder les *Lifecycle callback methods* permet d'éviter

- un plantage si l'utilisateur reçoit un coup de téléphone ou navigue vers une autre appli
- la consommation inutile de ressources système si l'utilisateur n'utilise pas activement l'appli
- la perte des données utilisateur de l'appli si ce dernier quitte puis revient sur celle-ci plus tard
- un plantage ou la perte de la progression de l'utilisateur lors de certains événements, comme le passage de portrait à paysage

Cycle de vie d'une application Android



Cycle de vie : interactions utilisateur



Gestion du cycle de vie

Gestion cycle de vie ⇒ redéfinition des méthodes clés

- `onCreate` : initialisation
- `onStart` : démarrage
- `onResume` : prise de focus
- `onStop` : perte de focus
- `onDestroy` : destruction

Il n'est pas obligatoire de tout redéfinir, `onCreate` peut suffire

Lors de la redéfinition : toujours commencer par appeler le code défini dans la classe mère, avec `super.[lifecycleMethod]()`

Doc android.app.Activity

[▶ Activity](#)

The entire lifecycle of an activity is defined by the following Activity methods. All of these are hooks that you can override to do appropriate work when the activity changes state. All activities will implement `onCreate(Bundle)` to do their initial setup; many will also implement `onPause()` to commit changes to data and otherwise prepare to stop interacting with the user. **You should always call up to your superclass when implementing these methods.**

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);

    protected void onStart();

    protected void onRestart();

    protected void onResume();

    protected void onPause();

    protected void onStop();

    protected void onDestroy();
}
```

Doc android.app.Activity

▶ Activity

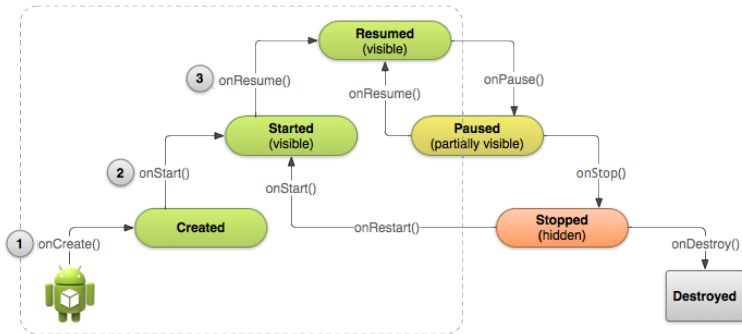
Method	Description	Killable?	Next
<code>onCreate()</code>	Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a <code>Bundle</code> containing the activity's previously frozen state, if there was one. Always followed by <code>onStart()</code> .	No	<code>onStart()</code>
<code>onRestart()</code>	Called after your activity has been stopped, prior to it being started again. Always followed by <code>onStart()</code>	No	<code>onStart()</code>
<code>onStart()</code>	Called when the activity is becoming visible to the user. Followed by <code>onResume()</code> if the activity comes to the foreground, or <code>onStop()</code> if it becomes hidden.	No	<code>onResume()</code> or <code>onStop()</code>
<code>onResume()</code>	Called when the activity will start interacting with the user. At this point your activity is at the top of the activity stack, with user input going to it. Always followed by <code>onPause()</code> .	No	<code>onPause()</code>
<code>onPause()</code>	Called when the system is about to start resuming a previous activity. This is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, etc. Implementations of this method must be very quick because the next activity will not be resumed until this method returns. Followed by either <code>onResume()</code> if the activity returns back to the front, or <code>onStop()</code> if it becomes invisible to the user.	Pre-HONEYCOMB	<code>onResume()</code> or <code>onStop()</code>
<code>onStop()</code>	Called when the activity is no longer visible to the user, because another activity has been resumed and is covering this one. This may happen either because a new activity is being started, an existing one is being brought in front of this one, or this one is being destroyed. Followed by either <code>onRestart()</code> if this activity is coming back to interact with the user, or <code>onDestroy()</code> if this activity is going away.	Yes	<code>onRestart()</code> or <code>onDestroy()</code>
<code>onDestroy()</code>	The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called <code>finish()</code> on it, or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the <code>isFinishing()</code> method.	Yes	<i>nothing</i>

Plan

- 1 Vue globale du cycle de vie d'une activité
- 2 Séquence de démarrage d'une activité**
- 3 onPause et onResume
- 4 Arrêt et redémarrage
- 5 Destruction
- 6 Restauration
- 7 *Lifecycle-Aware Components (Jetpack)*

Démarrage :

`onCreate` → `onStart` → `onResume`



Deux scénarios de démarrage

1. Par un launcher : fichier `AndroidManifest.xml`

```
<activity android:name=".MainActivity" android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```


2. Par un appel externe utilisant `startActivity`


- Un intent a été créé ⇒ l'OS démarra l'activité

Lorsque l'activité est démarrée, le système instancie l'activité et appelle sa méthode `onCreate`

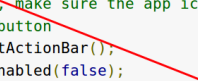
Exemple de redéfinition de onCreate

```
TextView mTextView; // Member variable for text view in the layout

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); 

    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity); 

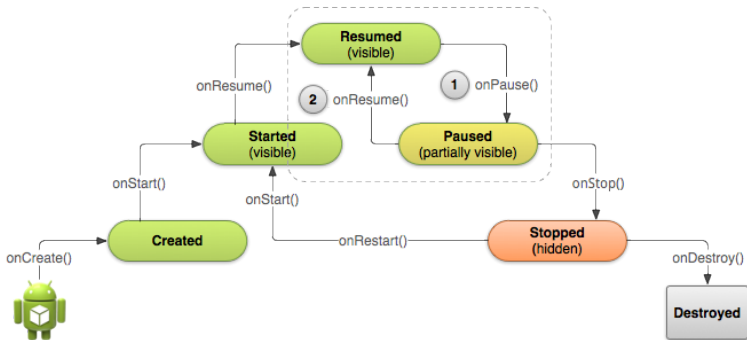
    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);

    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);  API level 5
    }
}
```

Plan

- 1 Vue globale du cycle de vie d'une activité
- 2 Séquence de démarrage d'une activité
- 3 onPause et onResume**
- 4 Arrêt et redémarrage
- 5 Destruction
- 6 Restauration
- 7 *Lifecycle-Aware Components (Jetpack)*

(1) Perte de focus → Paused



Intervient lorsqu'une autre activité non bloquante visuellement passe au premier plan (sélection, dialogues, etc.)
Premier signe que l'utilisateur va quitter l'application

Que faire dans onPause


Relâcher les ressources et stopper l'utilisation du CPU :

- Stopper les animations et tout ce qui peut consommer du CPU
- Enregistrer les changements non sauvés (uniquement ce qui a un intérêt pour l'application, pas de données utilisateur)
- rendre au système la main sur les ressources utilisées : réseau, senseurs (e.g. GPS, caméra).
- et d'une manière générale, éviter les traitements coûteux en CPU

Exemple

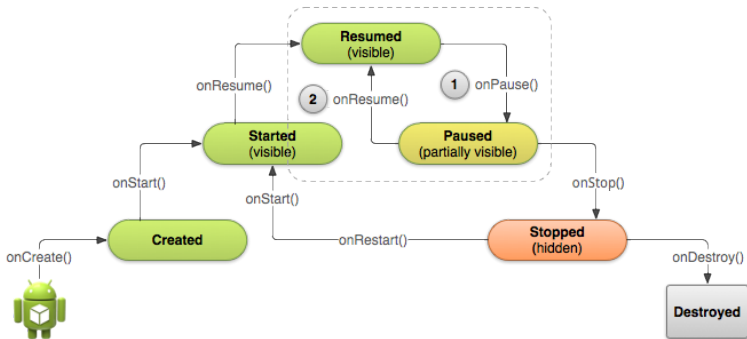
```
@Override
public void onPause() {
    super.onPause(); // Always call the superclass method first

    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release()
        mCamera = null;
    }
}
```



Note : l'instance de l'activité est gardée en mémoire : pas besoin de réinitialiser les composants utilisés

(2) Reprise de focus → onResume



Intervient à chaque prise de focus, y compris la première fois

Exemple

```
@Override
public void onResume() {
    super.onResume(); // Always call the superclass method first

    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

Méthode appelée à chaque prise de focus, y compris la première fois

Plan

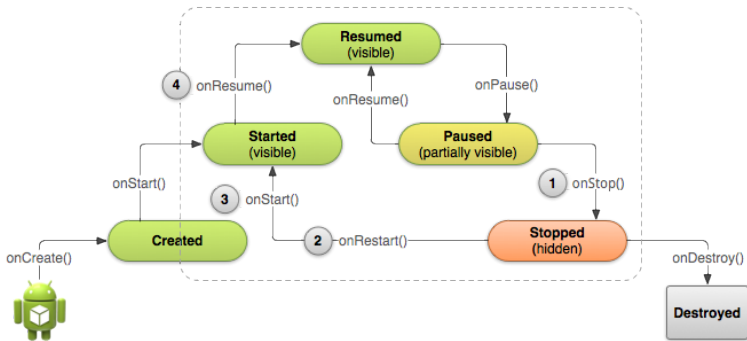
- 1 Vue globale du cycle de vie d'une activité
- 2 Séquence de démarrage d'une activité
- 3 onPause et onResume
- 4 Arrêt et redémarrage**
- 5 Destruction
- 6 Restauration
- 7 *Lifecycle-Aware Components (Jetpack)*

Arrêt et redémarrage

Scénarios

- L'utilisateur bascule sur une autre application puis revient via le menu des applications
- Une action de l'activité démarre une autre activité : l'activité est stoppée dès que la deuxième est créée.
- Appel téléphonique

Arrêt et redémarrage



L'activité reste en mémoire : pour des applications simples, `onPause` peut être suffisant

Exemple

```
@Override
protected void onStop() {
    super.onStop(); // Always call the superclass method first

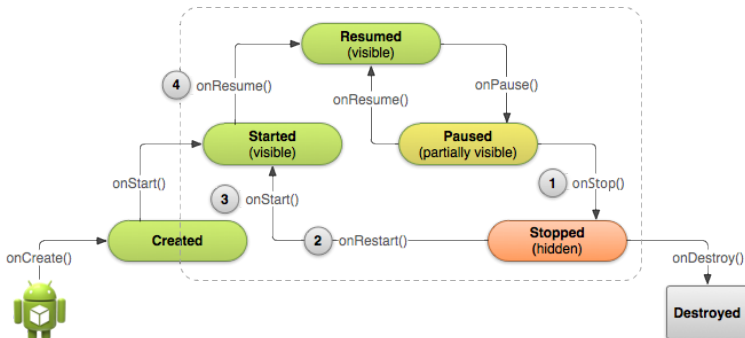
    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    getContentResolver().update(
        mUri, // The URI for the note to update.
        values, // The map of column names and new values to apply to them.
        null, // No SELECT criteria are used.
        null // No WHERE columns are used.
    );
}
```

MAJ BD

Note : la sauvegarde des états des Vues **possédant un android:id** est automatique (Bundle) : e.g. valeurs des champs texte

onRestart → onStart



La plupart des applications ne redéfinisse pas `onRestart`

Exemple

```
@Override
protected void onStart() {
    super.onStart(); // Always call the superclass method first

    // The activity is either being restarted or started for the first time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager =
        (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);

    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS, and use an intent
        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS action
        // to take the user to the Settings screen to enable GPS when they click "OK"
    }
}

@Override
protected void onRestart() {
    super.onRestart(); // Always call the superclass method first

    // Activity being restarted from stopped state
}
```

Plan

- 1 Vue globale du cycle de vie d'une activité
- 2 Séquence de démarrage d'une activité
- 3 onPause et onResume
- 4 Arrêt et redémarrage
- 5 Destruction**
- 6 Restauration
- 7 *Lifecycle-Aware Components (Jetpack)*

Destruction complète

→ instance de l'activité perdue

arrêts normaux

- L'utilisateur a supprimé l'activité ou cliqué son bouton **Close**
- L'activité a appelée sa méthode `finish()`

arrêts forcés

- le système détruit l'activité pour récupérer des ressources
- rotation de l'écran : destruction puis reconstruction, i.e. pour l'utilisation d'un layout spécifique

onDestroy

`onDestroy()` is called before the activity is destroyed. The system invokes this callback either because:

1. the activity is finishing (due to the user completely dismissing the activity or due to `finish()` being called on the activity), or
2. the system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)

```
@Override
public void onDestroy() {
    super.onDestroy(); // Always call the superclass

    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}
```

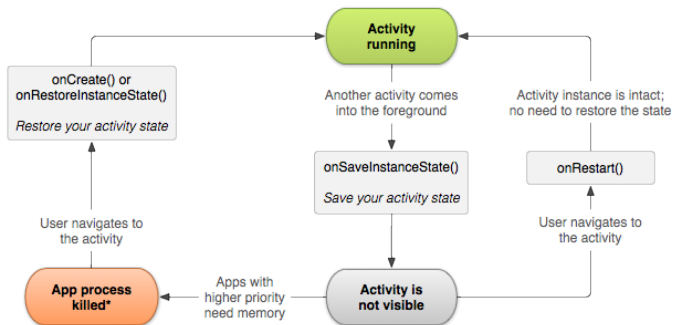
Note: do not count on this method being called as a place for saving data! For example, if an activity is editing data in a content provider, those edits should be committed in either `onPause()` or `onSaveInstanceState(Bundle)`, not here. This method is usually implemented to free resources like threads that are associated with an activity, so that a destroyed activity does not leave such things around while the rest of its application is still running. There are situations where the system will simply kill the activity's hosting process without calling this method (or any others) in it, so it should not be used to do things that are intended to remain around after the process goes away.

Dernière chance de relâcher des ressources

Plan

- 1 Vue globale du cycle de vie d'une activité
- 2 Séquence de démarrage d'une activité
- 3 onPause et onResume
- 4 Arrêt et redémarrage
- 5 Destruction
- 6 Restauration**
- 7 *Lifecycle-Aware Components (Jetpack)*

Sauvegarde / Restauration

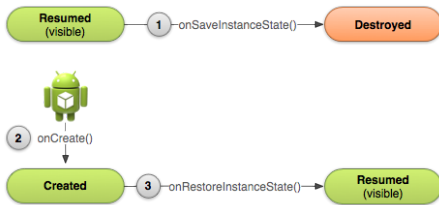


*Activity instance is destroyed, but the state from `onSaveInstanceState()` is saved

Restauration dans l'état où l'on a quitté : 2 cas →

- Activité détruite puis recréée, restauration nécessaire
- l'activité est stoppée, puis réactivée : état intact

Sauvegarde de données en cas d'arrêt forcé



(1) `onSaveInstanceState (Bundle)` : appelée après `onStop` (avant si `v. < Android 9 Pie`). Le `Bundle` est prérempli pour les Views ayant un ID. \Rightarrow `onSaveInstanceState` est utilisée pour faire des sauvegardes supplémentaires, utilisées plus tard par `onRestoreInstanceState (Bundle)` (3) lorsque l'activité est recréée (2) suite à une destruction.

Exemple

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

Restauration de l'état d'une activité (2) et/ou (3)



Restauration de l'état d'une activité

Dans la méthode `onCreate` :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) { ←
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

Restauration de l'état d'une activité

Mieux ! Dans la méthode `onRestoreInstanceState` :

```
public void onRestoreInstanceState(Bundle savedInstanceState) {  
    // Always call the superclass so it can restore the view hierarchy  
    super.onRestoreInstanceState(savedInstanceState);  
  
    // Restore state members from saved instance  
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);  
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);  
}
```

Pas besoin de tester que le `Bundle` n'est pas `null` car cette méthode n'est appelée que si c'est le cas.

Plan

- 1 Vue globale du cycle de vie d'une activité
- 2 Séquence de démarrage d'une activité
- 3 onPause et onResume
- 4 Arrêt et redémarrage
- 5 Destruction
- 6 Restauration
- 7 **Lifecycle-Aware Components (Jetpack)**

En pratique, il y a des limites

Tous les composants ayant besoin de réagir au cycle de vie sont gérés dans les méthodes de l'activité



- Mauvaise organisation du code
- Facilite l'introduction d'erreurs

Lifecycle-Aware Components (Jetpack)

Le package `androidx.lifecycle` de la librairie Jetpack fournit des classes, interfaces et annotations permettant de coder des composants qui réagissent automatiquement au changement d'état du cycle de vie d'une activité.

Peut nécessiter l'ajout d'une dépendance dans le fichier build de gradle du module app [voir ici pour l'ajouter](#)

Sans la librairie (*location use case*)

```
class MyLocationListener {
    public MyLocationListener(Context context, Callback callback) {
        // ...
    }

    void start() {
        // connect to system location service
    }

    void stop() {
        // disconnect from system location service
    }
}

class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    @Override
    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, (location) -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        myLocationListener.start();
        // manage other components that need to respond
        // to the activity lifecycle
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
        // manage other components that need to respond
        // to the activity lifecycle
    }
}
```

onStop peut finir avant que onStart soit achevée

```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, location -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        Util.checkUserStatus(result -> {
            // what if this callback is invoked AFTER activity is stopped?
            if (result) {
                myLocationListener.start();
            }
        });
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
    }
}
```

androidx.lifecycle.Lifecycle

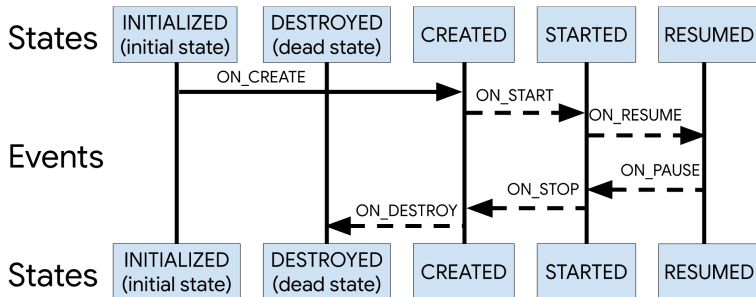
Solution : gérer et propager les événements du cycle de vie grâce à une instance de *LifeCycle* créée pour l'activité

LifeCycle modélise l'état du cycle de vie d'un composant (activité) et permet aux autres objets de l'observer.

Contient deux énumérations principales

- `LifeCycle.Event` : événements du cycle de vie.
- `LifeCycle.State` : état actuel du composant suivi.

androidx.lifecycle.Lifecycle



Exemples (*deprecated*)

```
public class CameraComponent implements LifecycleObserver {  
  
    ...  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)  
    public void initializeCamera() {  
        if (camera == null) {  
            getCamera();  
        }  
    }  
  
    ...  
}
```

```
public class JavaCameraComponent implements LifecycleObserver {  
  
    ...  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)  
    public void releaseCamera() {  
        if (camera != null) {  
            camera.release();  
            camera = null;  
        }  
    }  
  
    ...  
}
```

Il faut faire le lien dans onCreate

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    CameraComponent c = new CameraComponent();
    getLifecycle().addObserver(c);
}
```

DefaultLifecycleObserver est maintenant conseillé

```
public class MyObserver implements DefaultLifecycleObserver {  
    @Override  
    public void onResume(LifecycleOwner owner) {  
        connect()  
    }  
  
    @Override  
    public void onPause(LifecycleOwner owner) {  
        disconnect()  
    }  
}  
  
myLifecycleOwner.getLifecycle().addObserver(new MyObserver());
```

L'interface LifecycleOwner

Elle est implémentée par `AppCompatActivity` et `Fragment`

⇒ définit la méthode `getLifecycle()` qui renvoie l'instance de `LifeCycle` associée à l'activité : à donner aux objets intéressés

```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, getLifecycle(), location -> {
            // update UI
        });
        Util.checkUserStatus(result -> {
            if (result) {
                myLocationListener.enable();
            }
        });
    }
}
```


MyLocationListener

```
class MyLocationListener implements DefaultLifecycleObserver {
    private boolean enabled = false;
    public MyLocationListener(Context context, Lifecycle lifecycle, Callback callback)
        ...
    }

    @Override
    public void onStart(LifecycleOwner owner) {
        if (enabled) {
            // connect
        }
    }

    public void enable() {
        enabled = true;
        if (lifecycle.getCurrentState().isAtLeast(STARTED)) {
            // connect if not connected
        }
    }

    @Override
    public void onStop(LifecycleOwner owner) {
        // disconnect if connected
    }
}
```

LifecycleRegistry

On peut implémenter son propre LifecycleOwner :

```
public class MyActivity extends Activity implements LifecycleOwner {
    private LifecycleRegistry lifecycleRegistry;

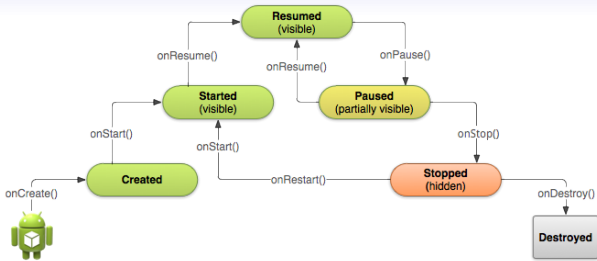
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        lifecycleRegistry = new LifecycleRegistry(this);
        lifecycleRegistry.markState(Lifecycle.State.CREATED);
    }

    @Override
    public void onStart() {
        super.onStart();
        lifecycleRegistry.markState(Lifecycle.State.STARTED);
    }

    @NonNull
    @Override
    public Lifecycle getLifecycle() {
        return lifecycleRegistry;
    }
}
```

Résumé global



Ce cours reprend largement les tutoriels en ligne proposés par Google:

▶ [Android developers](#)