**INTRODUZIONE**

**Classi fondamentali**-> **EuiccManager**: interfaccia all’eUICC e alla eSIM. È il punto di ingresso principale per le app per interagire con LPA (Local Profile Assistant)1. Infatti, le relative API richiedono un’app LPA correttamente implementata per funzionare.  
-> **EuiccService**: classe utile per connettersi ai servizi LPA.  
-> **EuiccCardManager**: classe che fornisce funzioni ES10x basate su GSMA RSP v2.02. Il chiamante delle relative API deve essere un LPA.

**Architettura RSP semplificata**

Immagine che contiene diagramma

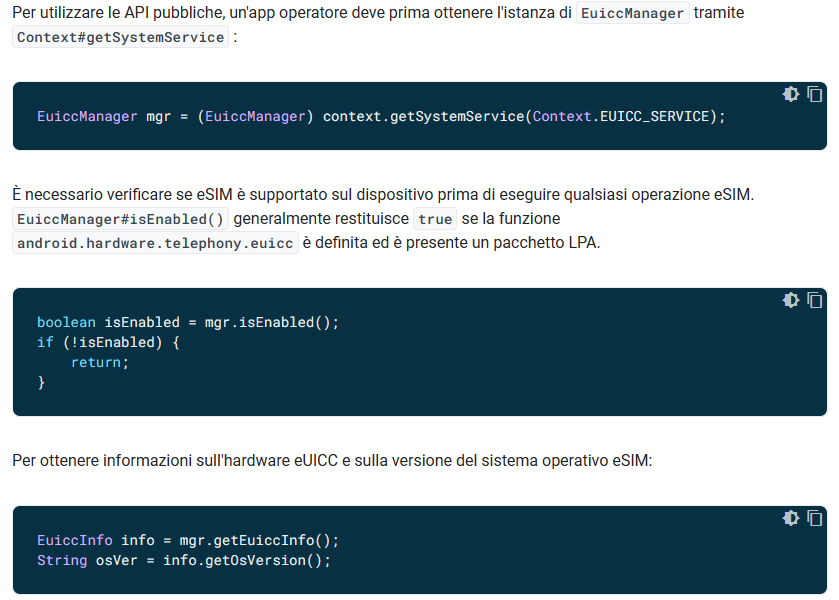
Descrizione generata automaticamente

*1 Un’app LPA è responsabile della gestione dei profili degli abbonati all’interno della rete mobile, inclusa la creazione, l’aggiornamento e la cancellazione dei profili degli abbonati. Può essere anche usata per il controllo dell’accesso alla rete e per la sicurezza dei dati degli abbonati.*

*2 Il protocollo RSP (Remote SIM Provisioning) permette di aggiornare le informazioni contenute sulla SIM card di un dispositivo mobile tramite una connessione di rete sicura, senza la necessità di rimuovere fisicamente la SIM card dal dispositivo. Le funzioni ES10x basate su GSMA RSP, quindi, sono funzioni utilizzate per la gestione delle credenziali di sicurezza degli utenti (e.g. l’autenticazione, l’autorizzazione, la gestione delle chiavi, la crittografia).*

**CREAZIONE DI UN’APP PER L’OPERATORE**

**EuiccManager**



Scaricamento di un abbonamento creato da codice di attivazione o codice QR:  
Avviene mediante l’API downloadSubscription(), che fa uso del callback PendingIntent (il quale deve essere BroadcastReceiver). In particolare:  
-> **downloadSubscription()**, oltre a consentire di gestire i download di abbonamenti (subscription) all’interno di un’applicazione Android, fornisce funzionalità per gestire la fatturazione degli abbonamenti stessi (e.g. verifica dello stato dell’abbonamento, richiesta di rinnovo dell’abbonamento).  
-> **PendingIntent** è un meccanismo di Android che consente a un’applicazione di richiedere l’esecuzione di un’azione in un momento successivo (e.g. quando l’utente fa clic su un’attività o una notifica).  
-> **BroadcastReceiver** è un componente del sistema Android che consente alle applicazioni di ricevere e rispondere a messaggi e notifiche provenienti dal sistema o da altre applicazioni. In altre parole, funge da ascoltatore per determinati eventi che si verificano all’interno del sistema Android (e.g. avvio del dispositivo o ricezione di un messaggio di testo). In particolare, quando si verifica un evento per cui un BroadcastReceiver è stato registrato, il sistema Android invia un messaggio di intent all’applicazione. Quest’ultima può quindi elaborare il messaggio di intent e prendere le opportune azioni in base al tipo di evento che si è verificato.  
-> Un **messaggio di intent** è un oggetto utilizzato per comunicare informazioni tra diverse componenti dell’applicazione o tra diverse applicazioni all’interno del sistema Android. In pratica, rappresenta un’azione da compiere o un evento che è avvenuto / deve avvenire nel sistema. Inoltre, include informazioni come il tipo di azione da eseguire, i dati da trasferire, il destinatario dell’intent, ma anche le autorizzazioni richieste dall’azione e l’URI (Uniform Resource Identifier) dell’intent.

Immagine che contiene testo

Descrizione generata automaticamente

- Inizialmente viene definito il BroadcastReceiver, che ha il metodo **onReceive()** che viene attivato quando l’applicazione viene colpita da un messaggio broadcast (i.e. un evento). In particolare, tale metodo verifica il tipo di messaggio che è stato ricevuto e, se corrisponde al tipo di messaggio che può essere gestito dal BroadcastReceiver, si salva le relative informazioni (resultCode, detailedCode, resultIntent).  
- Il BroadcastReceiver viene poi registrato con l’API **registerReceiver()**; il secondo parametro è un oggetto **IntentFilter** che specifica quali messaggi di broadcast si desidera ricevere (nel nostro caso “download\_subscription”).  
- **DownloadableSubscription** è una classe che rappresenta un abbonamento che può essere scaricato da un server mediante http e attivato su un dispositivo Android. Inoltre, consente di accedere alle informazioni relative all’abbonamento scaricabile (e.g. nome, descrizione, prezzo, durata, periodo di prova) e di gestire lo stato dell’abbonamento stesso (i.e. attivarlo, disattivarlo). Il metodo **forActivationCode()** di DownloadableSubscription consente di ottenere appunto un oggetto di tipo DownloadableSubscription a partire da un codice di attivazione.  
- Il metodo **getBroadcast()** della classe PendingIntent restituisce appunto un oggetto PendingIntent che rappresenta (e richiede l’esecuzione di) una determinata azione A (vedere il parametro **action** passato al costruttore di Intent). Quando A verrà effettivamente eseguita, verrà inviato un messaggio a tutti i ricevitori (come il BroadcastReceiver) che hanno dichiarato di essere interessati ad A.  
- Infine, tramite il metodo downloadSubscription(), viene gestito il download di un abbonamento. È qui che viene coinvolto il callback (i.e. il PendingIntent precedentemente dichiarato).

NB: chiaramente un abbonamento non è altro che un’offerta tariffaria che possiamo pagare per la nostra eSIM e che comprende una certa quantità di dati mobili, messaggi e minuti di telefonate.

Viceversa, se anziché scaricarlo vogliamo passare a un abbonamento a partire da un dato **ID di abbonamento**, la struttura del codice rimane pressoché la stessa:

Immagine che contiene testo

Descrizione generata automaticamente

**Errori risolvibili**  
A volte il sistema non è in grado di completare l’operazione eSIM ma l’errore può essere risolto dall’utente. Ad esempio, downloadSubscription potrebbe non riuscire se i metadati del profilo indicano che è necessario un codice di conferma dell’operatore.  
In casi come questo, il callback del chiamante viene invocato con **EuiccManager.EMBEDDED\_SUBSCRIPTION\_RESULT\_RESOLVABLE\_ERROR** (che è il valore restituito dall’operazione andata in errore) e viene chiamato il metodo **EuiccManager.startResolutionActivity()**. Nell’esempio esposto precedentemente, startResolutionActivity() attiva una schermata LPA User Interface (LUI) che consente all’utente di inserire il codice di conferma richiesto; dopo l’inserimento del codice si riprende l’operazione di download.  
Android 9 definisce questi errori risolvibili nella classe EuiccService, all’interno della quale sono definite stringe come queste:

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

**Privilegi di vettore**  
Badiamo che profili di abbonamento appartenenti a operatori diversi possono coesistere nell’eUICC di un dispositivo: in tal caso, l’operatore A non deve essere in grado di scaricare, abilitare o disabilitare un profilo proprio dell’operatore B.  
Per garantire che un profilo sia accessibile solo al suo proprietario, Android utilizza un meccanismo per concedere privilegi speciali (solo) all’app del proprietario del profilo (i.e. all’app dell’operatore). La piattaforma Android carica i certificati archiviati nel file delle regole di accesso (ARF) del profilo e concede l’autorizzazione (solo) alle app firmate da questi certificati per effettuare chiamate alle API di EuiccManager. Il processo di alto livello è descritto di seguito:  
1) L’operatore firma l’APK dell’app dell’operatore (= carrier app); lo strumento apksigner allega il certificato a chiave pubblica all’APK.  
2) L’operatore prepara un profilo, i cui metadati includono un ARF che contiene:  
 - Una firma (SHA-1/SHA-256) del certificato a chiave pubblica dell’app dell’operatore (obbligatorio).  
 - Il nome del package dell’app dell’operatore (fortemente consigliato).  
3) L’app tenta di eseguire un’operazione eUICC su un determinato profilo (e.g. scaricamento, abilitazione, disabilitazione del profilo).  
4) La piattaforma Android verifica che la firma dell’APK e la firma del certificato contenuto nell’ARF incluso nel profilo corrispondano. Eventualmente, verifica anche che il nome del package dell’app del chiamante e il nome del package dell’app dell’operatore corrispondano.  
5) Se i controlli di cui sopra vanno a buon fine, l’operazione eUICC nominata al punto (3) viene effettivamente concessa.

**CREAZIONE DI UN’APP PER L’LPA**

**Requisiti hardware/modem**  
-> L’LPA (Local Profile Assistant) e il sistema operativo eSIM sul chip eUICC devono supportare almeno GSMA RSP v2.0 o v2.2.  
-> È necessario pianificare l’utilizzo di server SM-DP+ e SM-DS.  
-> Il modem del dispositivo deve implementare i seguenti metodi:  
 - IRadio HAL v1.13: **setSimPower**, usato per attivare/disattivare l’alimentazione della scheda SIM.  
 - IRadio HAL v1.24: **getIccCardStatus**, usato per ottenere lo stato della scheda SIM.  
 - IRadioConfig HAL v1.05: **getSimSlotsStatus**, usato per ottenere lo stato dei slot per le schede SIM presenti sul dispositivo.  
-> Il modem deve riconoscere l’eSIM con il profilo predefinito abilitato come una SIM valida e mantenere la SIM accesa.  
-> Per un elenco completo dei requisiti del modem, vedere <https://source.android.com/docs/core/connect/esim-modem-requirements?hl=it>.

**EuiccService**Un LPA è costituito da due componenti separati (che possono essere implementati entrambi nello stesso APK): il **backend LPA** e **l’interfaccia utente LPA** (**LUI LPA**).

Backend LPA:  
Per implementare il backend LPA, è necessario estendere la classe EuiccService e dichiarare questo servizio (.EuiccServiceImpl) nel file **AndroidManifest.xml**, che è un file XML obbligatorio che descrive il package e le caratteristiche dell’applicazione e indica come essa deve essere eseguita.

**Immagine che contiene testo

Descrizione generata automaticamente**

Il servizio deve richiedere l’autorizzazione di sistema **android.permission.BIND\_EUICC\_SERVICE** per garantire che solo il sistema possa collegarsi a esso. Inoltre, deve includere un intent-filter con l’azione **android.service.euicc.EuiccService**, la cui priorità (android:priority) dovrebbe assumere un valore non nullo se sono presenti più implementazioni sul dispositivo.

*3 IRadio HAL v1.1 è un componente di livello Hardware Abstraction Layer di Android che fornisce un’interfaccia per le funzionalità radio basate su circuiti integrati per la gestione delle chiamate vocali.*

*4 IRadio HAL v1.2 è un componente di livello Hardware Abstraction Layer di Android che estende la funzionalità della versione 1.1 per supportare anche i servizi di messaggistica.*

*5 IRadioConfig HAL v1.0 è un componente di livello Hardware Abstraction Layer di Android che fornisce un’interfaccia per le impostazioni radio del dispositivo (e.g. configurazione della banda di frequenza, livello di potenza di trasmissione, sensibilità del ricevitore).*

Interfaccia utente LPA:  
Per implementare la LUI, è necessario fornire nel file AndroidManifest.xml un’attività per le seguenti azioni:  
-> **android.service.euicc.action.MANAGE\_EMBEDDED\_SUBSCRIPTIONS**-> **android.service.euicc.action.PROVISION\_EMBEDDED\_SUBSCRIPTION**

Immagine che contiene testo

Descrizione generata automaticamente

Come per il servizio esaminato per la parte backend, anche l’attività deve richiedere l’autorizzazione di sistema **android.permission.BIND\_EUICC\_SERVICE**; inoltre, dovrebbe avere un filtro intent con le azioni elencate sopra, la categoria **android.service.euicc.category.EUICC\_UI** e una priorità non nulla.

**EuiccCardManager**  
È l’interfaccia per comunicare col chip eSIM. Fornisce le funzioni ES106 e gestisce i comandi di richiesta/risposta APDU7 di basso livello e l’analisi ASN.1. Può essere chiamata solo da app con privilegi di sistema.

Immagine che contiene diagramma

Descrizione generata automaticamente

Le API di EuiccCardManager, per essere utilizzate, richiedono che il chiamante sia un LPA. Ciò significa che il chiamante deve estendere EuiccService ed essere dichiarato nel file AndroidManifest.xml, proprio come descritto precedentemente.

*6 ES10 (o ECMAScript 2019) è la decima versione dello standard ECMAScript per la programmazione in JavaScript.*

*7 I comandi APDU (Application Protocol Data Unit) costituiscono un protocollo di comunicazione standard per l’invio di comandi da un lettore di smart card a una smart card (e.g. carta SIM) e per la ricezione di risposte.*

Immagine che contiene testo

Descrizione generata automaticamente

-> In particolare, il metodo requestAllProfiles() recupera tutte le informazioni sui profili eSIM presenti sulla scheda eSIM.

Quando si utilizzano le API di EuiccCardManager, il chiamante (LPA) deve fornire un oggetto **Executor** attraverso cui viene richiamato il callback. L’Executor, di fatto, non è altro che un oggetto che ha la responsabilità di eseguire i task e, spesso e volentieri, viene utilizzato al posto di creare esplicitamente nuovi thread.

Immagine che contiene testo, Sito Web

Descrizione generata automaticamente

ICCID = num. identificativo della scheda SIM.

Immagine che contiene Sito Web

Descrizione generata automaticamente

**ATTIVAZIONE DI UN PROFILO eSIM CON UN’APP DELL’OPERATORE**

**Introduzione**  
Utilizzando un’app dell’operatore, è possibile attivare SIM e scaricare i profili chiamando direttamente downloadSubscription() o fornendo un codice di attivazione all’LPA. Quando un’app dell’operatore scarica un profilo tramite downoladSubscription(), può gestire il profilo stesso tramite un tag di metadati **BF76**8 che contiene informazioni sulla configurazione del profilo (e.g. l’identificatore del profilo, il tipo di autenticazione supportata, la configurazione della sicurezza e la configurazione delle funzionalità di roaming) e, in particolare, codifica le regole dei privilegi dell’operatore per il profilo:

Immagine che contiene testo

Descrizione generata automaticamente

Se un profilo non ha un tag BF76 o se il suo tag BF76 non corrisponde alla firma dell’app dell’operatore chiamante, il download viene rifiutato.

**Attivazione dell’eSIM utilizzando un codice di attivazione**  
Quando si vuole utilizzare un codice di attivazione per attivare un profilo eSIM, l’LPA lo recupera dall’app dell’operatore e scarica il profilo. In tal caso, l’LPA può controllare l’intero flusso dell’interfaccia utente, il che vuol dire che non viene visualizzata alcuna interfaccia utente dell’app dell’operatore. Questo approccio ignora il controllo del tag BF76 e gli operatori di rete non devono implementare l’intero flusso dell’interfaccia utente di attivazione eSIM.

**Definizione del servizio di provisioning eUICC del vettore**L’app LPA e l’operatore comunicano tramite due interfacce **AIDL**9: **IcarrierEuiccProvisioningService** e **IgetActivationCodeCallback**. In particolare:  
- L’app dell’operatore deve implementare un’interfaccia IcarrierEuiccProvisioningService ed esporla nella relativa dichiarazione AndroidManifest.xml (vedere pagina seguente).  
- L’LPA deve eseguire il binding a IcarrierEuiccProvisioningService e implementare IgetActivationCodeCallback.

Per ulteriori informazioni su come implementare un’interfaccia AIDL, consulta [Android Interface Definition Language (AIDL)  |  Android Developers](https://developer.android.com/guide/components/aidl#Defining).

*8 Il tag BF76 deve essere supportato dal sistema operativo eUICC e da SM-DP+.*

*9 Un’interfaccia AIDL (Android Interface Definition Language) è un’interfaccia che definisce un set di metodi che possono essere utilizzati per comunicare tra processi su piattaforma Android.*

Immagine che contiene testo

Descrizione generata automaticamente  
*Prototipo della dichiarazione AndroidManifest.xml in cui deve essere esposta l’interfaccia* *IcarrierEuiccProvisioningService.*

Immagine che contiene testo

Descrizione generata automaticamente

before

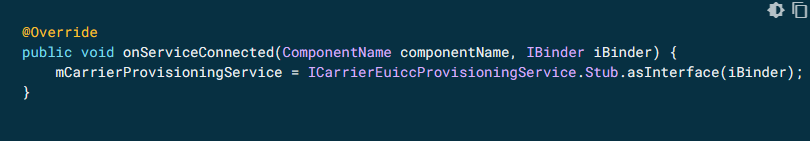
pass in

pass in

Immagine che contiene testo

Descrizione generata automaticamente

Esempio di implementazione LPA:  
Per eseguire il binding all’implementazione IcarrierEuiccProvisioningService dell’app dell’operatore, l’LPA deve copiare sia IcarrierEuiccProvisioningService.aidl che IgetActivationCodeCallback.aidl nel progetto e deve implementare **ServiceConnection**, che è una classe che serve per stabilire una connessione tra il client (nel nostro caso l’LPA) e il servizio (nel nostro caso l’app dell’operatore) e che consente al client di inviare comandi al servizio e ricevere dati da esso. Questa classe contiene due metodi principali: **onServiceConnected()** e **onServiceDisconnected()**: il primo viene chiamato quando la connessione al servizio è stata stabilita con successo, mentre il secondo viene chiamato quando la connessione al servizio viene interrotta in modo inaspettato.



Dopo l’associazione all’implementazione IcarrierEuiccProvisioningService dell’app dell’operatore, l’LPA chiama getActivationCode() o getActivationCodeForEid() per ottenere il codice di attivazione dell’app dell’operatore. La differenza tra getactivationCode() e getActivationCodeForEid() è che quest’ultimo consente a un operatore di preassociare un profilo all’EID del dispositivo prima che inizi il download.

Immagine che contiene testo

Descrizione generata automaticamente

Esempio di implementazione per l’app dell’operatore:  
Affinché l’LPA si colleghi all’app dell’operatore, quest’ultima deve copiare sia IcarrierEuiccProvisioningService.aidl che IgetActivationCodeCallback.aidl nel progetto e dichiarare il servizio IcarrierEuiccProvisioningService nel file AndroidManifest.xml. Il servizio deve richiedere l’autorizzazione di sistema **android.permission.WRITE\_EMBEDDED\_SUBSCRIPTIONS** per garantire che solo LPA possa collegarsi a esso.  
Il servizio delle includere anche un intent-filter con l’azione **android.service.euicc.action.BIND\_CARRIER\_PROVISIONING\_SERVICE**.

Immagine che contiene testo

Descrizione generata automaticamente

Per implementare il servizio dell’app dell’operatore, bisogna creare un servizio, estendere la classe **Stub** e implementare i metodi getActivationCode() e getActivationCodeForEid(). L’LPA può quindi chiamare entrambi i metodi per recuperare il codice di attivazione del profilo. L’app dell’operatore dovrebbe rispondere chiamando IgetActivationCodeCallback.onSuccess() col codice di attivazione se il codice è stato recuperato correttamente dal server dell’operatore. In caso di esito negativo, l’app dell’operatore dovrebbe rispondere con IgetActivationCodeCallback.onFailure().

Di seguito è riportato uno sketch dell’implementazione della classe **CarrierEuiccProvisioningService.java**:

import android.service.euicc.ICarrierEuiccProvisioningService;  
import android.service.euicc.ICarrierEuiccProvisioningService.Stub;  
import android.service.euicc.IGetActivationCodeCallback;  
  
public class CarrierEuiccProvisioningService extends Service {  
    private final ICarrierEuiccProvisioningService.Stub binder =  
        new Stub() {  
          @Override  
          public void getActivationCode(IGetActivationCodeCallback callback) throws RemoteException {  
            String activationCode = // do whatever work necessary to get an activation code (HTTP requests to carrier server, fetch from storage, etc.)  
            callback.onSuccess(activationCode);  
          }  
  
          @Override  
          public void getActivationCodeForEid(String eid, IGetActivationCodeCallback callback) throws RemoteException {  
            String activationCode = // do whatever work necessary (HTTP requests, fetch from storage, etc.)  
            callback.onSuccess(activationCode);  
          }  
      }  
}

**Avvio dell’interfaccia utente dell’app dell’operatore nel flusso di attivazione LPA**  
LPA può avviare l’interfaccia utente di un’app dell’operatore. Ciò è utile in quanto un’app dell’operatore potrebbe richiedere informazioni aggiuntive all’utente prima di fornire un codice di attivazione all’LPA. Di seguito è descritto il processo per l’avvio dell’interfaccia utente di un’app dell’operatore nell’LPA:  
1) L’LPA avvia il flusso di attivazione dell’app dell’operatore inviando il messaggio di intent **android.service.euicc.action.START\_CARRIER\_ACTIVATION** all’app dell’operatore; il ricevitore dell’app dell’operatore deve essere protetto nella dichiarazione AndroidManifest.xml con **android:permission=”android.permission.WRITE\_EMBEDDED\_SUBSCRIPTIONS”** per evitare di ricevere messaggi di intent da app non LPA.

Immagine che contiene testo

Descrizione generata automaticamente

- context.getPackageManager().resolveActivity(): recupera informazioni su una determinata activity del sistema (che verrà anche restituita in output); tale attività sarà associata all’azione relativa all’Intent (carrierAppIntent).  
 - setClassName(): imposta il nome della classe dell’attività di destinazione (packageName).  
 - startActivityForResult(): avvia un Intent (carrierAppIntent) per avviare un’altra attività e attende il risultato di quest’ultima.

2) L’app dell’operatore esegue il proprio lavoro (e.g. accesso dell’utente) utilizzando la propria interfaccia utente.  
3) L’app dell’operatore risponde all’LPA chiamando **setResult(int, Intent)** e **finish()**.  
 - Se l’app dell’operatore risponde con **RESULT\_OK**, LPA continua il flusso di attivazione.  
 - Se l’app dell’operatore si arresta in modo anomalo o risponde con **RESULT\_CANCELED**, l’LPA annulla il flusso di attivazione dell’eSIM.  
 - Se l’app dell’operatore risponde con qualcosa di diverso da RESULT\_OK o RESULT\_CANCELED, l’LPA lo considera un errore.

**Avvio del flusso di attivazione LPA in un’app dell’operatore**  
Le app dell’operatore possono utilizzare le API eUICC per avviare una LUI per l’attivazione di eSIM. Di seguito è descritto il processo per l’avvio del flusso di attivazione LPA in un’app dell’operatore:  
1) L’LPA deve dichiarare un’attività che includa un intent-filter con l’azione **android.service.euicc.action.START\_EUICC\_ACTIVATION**. La priorità dell’intent-filter dovrebbe essere impostata su un valore non nullo se sul dispositivo sono presenti più implementazioni. Per esempio:

Immagine che contiene testo

Descrizione generata automaticamente

2) L’app dell’operatore esegue il proprio lavoro (e.g. accesso dell’utente) utilizzando la propria interfaccia utente.  
3) L’app dell’operatore risponde all’LPA chiamando **setResult(int, Intent)** e **finish()**.  
4) D’altra parte, l’LPA si collega all’implementazione IcarrierEuiccProvisioningService dell’app dell’operatore per recuperare il codice di attivazione e scaricare il profilo corrispondente.  
5) Quando il flusso di attivazione LPA è completo, LPA risponde all’app dell’operatore con un risultato, che l’app dell’operatore gestisce in **onActivityResult(int, int, Intent)**.  
 - Se l’LPA riesce a scaricare il nuovo profilo eSIM, risponde con **RESULT\_OK**.  
 - Se l’utente annulla l’attivazione del profilo eSIM nell’LPA, quest’ultimo risponde con **RESULT\_CANCELED**.  
 - Se l’LPA risponde con qualcosa di diverso da RESULT\_OK o RESULT\_CANCELED, l’app dell’operatore lo considera un errore.

**RECAP**

**Introduzione**  
-> Classi fondamentali: EuiccManager, EuiccService, EuiccCardManager.

**Creazione di un’app per l’operatore**  
-> Scaricamento di un abbonamento: new BroadcastReceiver() 🡪 registerReceiver() 🡪 DownloadableSubscription.forActivationCode() 🡪 new Intent() 🡪 PendingIntent.getBroadcast() 🡪 downloadSubscription().

**Creazione di un’app per l’LPA**  
-> Definizione dell’LPA: AndroidManifest.xml (tag service per il backend LPA; tag activity per l’interfaccia utente LPA).  
-> Ottenimento dei profili su eUICC: new ResultCallback<EuiccProfileInfo[]>() 🡪 requestAllProfiles().  
-> Caricamento di un package di profili vincolati su eUICC: loadBoundProfilePackage().  
-> Passaggio a un profilo diverso con un determinato ICCID: switchToProfile().  
-> Ottenimento dell’indirizzo SM-DP+: requestDefaultSmdpAddress().  
-> Recupero di un elenco di notifiche degli eventi: listNotifications().

**Attivazione di un profilo eSIM con un’app dell’operatore**  
-> Cosa deve implementare l’app dell’operatore: interfaccia (AIDL) IcarrierEuiccProvisioningService.aidl 🡪 metodi getActivationCode() & getActivationCodeForEid().  
-> Cosa deve implementare l’LPA: interfaccia (AIDL) IgetActivationCodeCallback.aidl 🡪 metodi onSuccess() & onFailure().  
-> Connessione tra LPA e app dell’operatore stabilita con successo: onServiceConnected().  
-> Avvio dell’interfaccia utente dell’app dell’operatore nel flusso di attivazione LPA: new Intent(“android…START\_CARRIER\_ACTIVATION”).setPackage() 🡪 getPackageManager().resolveActivity() 🡪 setClassName() 🡪 startActivityForResult() 🡪 l’app dell’operatore esegue il proprio lavoro 🡪 setResult() 🡪 finish().  
-> Avvio del flusso di attivazione LPA in un’app dell’operatore (step successivi a quelli già elencati): l’LPA si collega a IcarrierEuiccProvisioningService 🡪 onActivityResult().