Lezione 23

Programmazione Android

- Utilizzo di codice nativo
  - Motivazioni
  - C e Java
    - Java Native Interface (JNI)
- Accesso ai servizi Google
  - Google APIs for Android

Codice nativo

Motivazioni

- Android incoraggia l'uso di codice Java (compilato in DEX) per favorire la portabilità e la sicurezza
  - Grazie all'esecuzione all'interno di una VM
- Tuttavia, in certi casi la programmazione **nativa** è indispensabile
  - Prestazioni più alte
  - Librerie pre-esistenti scritte in C / C++
  - Accesso a livello di bit all'hardware

Svantaggi

- Maggiore complessità di sviluppo
- Assenza di portabilità
  - È necessario fornire versioni compilate del codice nativo per ciascuno dei processori su cui la vostra app girerà
    - In pratica, di solito ARM, MIPS e a volte Intel
    - È possibile limitare sul market la distribuzione ai soli dispositivi dotati di CPU “giusta”
- Interfaccia complicata con il mondo di oggetti Java
  - Necessità di fare marshalling di tutto

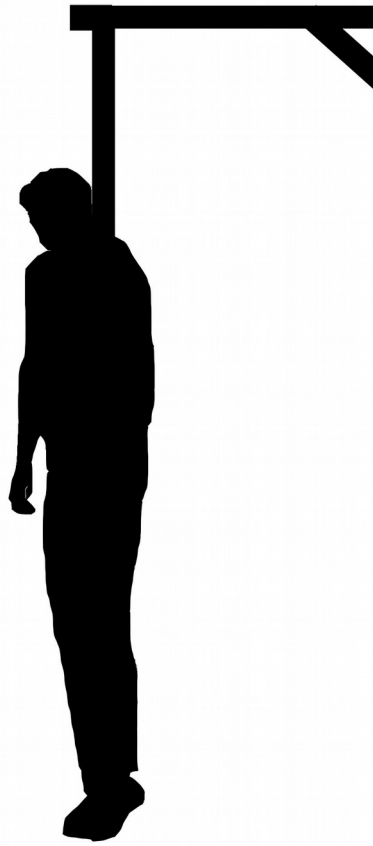
Svantaggi

Android NDK

The NDK is a toolset that allows you to implement parts of your app using native-code languages such as C and C++. For certain types of apps, this can be helpful so you can reuse existing code libraries written in these languages, but most apps do not need the Android NDK.

Before downloading the NDK, you should understand that **the NDK will not benefit most apps**. As a developer, you need to balance its benefits against its drawbacks. Notably, using native code on Android generally does not result in a noticable performance improvement, but it always increases your app complexity. In general, you should only use the NDK if it is essential to your app —never because you simply prefer to program in C/C++.

Typical good candidates for the NDK are CPU-intensive workloads such as game engines, signal processing, physics simulation, and so on. When examining whether or not you should develop in native code, think about your requirements and see if the Android framework APIs provide the functionality that you need.



Approccio generale

- Android **supporta** la scrittura di applicazioni ***interamente*** in C
- Tuttavia, la cosa è considerata normalmente poco praticabile
  - Usata solo in alcuni framework per i giochi o in applicazioni di controllo automatico
  - Più spesso, si realizzano **librerie** scritte in C che offrono funzioni con interfaccia **semplice** che si occupano della sola parte CPU-intensive o “delicata”
    - Number crunching, accesso all'hardware

Approccio generale

- Per produrre codice nativo per Android, è necessario installare una *toolchain* aggiuntiva rispetto al solito
- Cosiddetto **NDK** – Native Development Kit
- Disponibili versioni per Mac, Windows, Linux
- Ciascuna in varianti a 32 e a 64 bit
- Supporto variegato in base al tipo di CPU
- Integrato in Eclipse e in Android Studio
  - Limitato testing, a volte sorprese

Compatibilità ABI

- Ciascuna delle architetture hardware supportate richiede la sua ABI
- Ogni implementazione ha qualche sottile incompatibilità…

ABI	Supported Instruction Set(s)	Notes
armeabi	ARMV5TE and later Thumb-1	No hard float.
armeabi-v7a	armeabi Thumb-2 VFPv3-D16 Other, optional	Incompatible with ARMv5, v6 devices.
arm64-v8a	AArch-64	
x86	x86 (IA-32) MMX SSE/2/3 SSSE3	No support for MOVBE or SSE4.
X86_64	x86-64 MMX SSE/2/3 SSSE3 SSE4.1, 4.2 POPCNT	
mips	MIPS32r1 and later	Uses hard-float, and assumes a CPU:FPU clock ratio of 2:1 for maximum compatibility. Provides neither micromips nor MIPS16.
mips64	MIPS64r6	

C e Java

Java Native Interface

- Java prevede già a livello di definizione del linguaggio la possibilità di interfacciarsi con codice **nativo**
  - Tipicamente, ma non necessariamente, scritto in C
  - In sostanza: moduli oggetto (file.o) contenenti codice compilato, e con le convenzioni di chiamata del C
    - Parametri passati tramite lo stack
    - Push e pop a cura del chiamante
    - Convenzioni sulla modalità di restituzione dei risultati
- Codice nativo raccolto in una **libreria** (file.so)

Java Native Interface

- La **Java Native Interface** (JNI) è l'insieme di specifiche che descrivono come interfacciare codice Java e C
- È una parte standard del linguaggio Java
  - E infatti, è usata per implementare tutta l'interfaccia con il sistema operativo ospite
    - Accesso ai file, alle socket, alle primitive grafiche...
- Android usa JNI esattamente come si fa in Java "versione desktop"

Funzioni di JNI

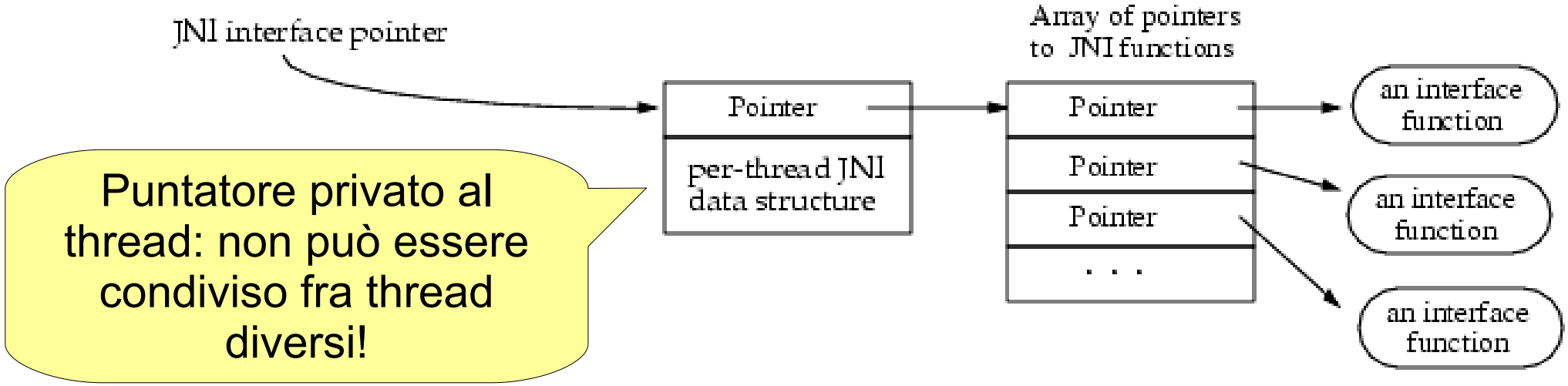
- JNI fornisce funzioni C per:
  - Creare oggetti Java, leggere e scrivere valori nei loro campi
    - Funzioni specializzate per i tipi comuni: numerici, array e stringhe
  - Chiamare metodi statici e di istanza
  - Generare e catturare eccezioni
  - Caricare classi Java a runtime e ispezionarle tramite *reflection*
  - Controllare a runtime i tipi “veri” degli oggetti

Funzioni di JNI

- JNI fornisce funzioni C per:
  - Creare oggetti Java, leggere e scrivere valori nei loro campi
    - Funzioni specializzate per i tipi comuni: numerici, array e stringhe
  - Chiamare metodi statici e di istanza
  - Generare e catturare eccezioni
  - Caricare classi Java a runtime e ispezionarle tramite *reflection*
  - Controllare a runtime i tipi “veri” degli oggetti

Architettura di JNI

- Le funzioni offerte da JNI (da chiamare in C) sono offerte tramite un puntatore a un array di puntatori a funzione
- Il puntatore da usare viene passato come argomento al vostro codice C



Packaging del codice C

- Le vostre funzioni C devono essere compilate in una libreria a caricamento dinamico
  - file.so, visto che Android gira su kernel Linux
- Devono essere **rientranti** e **thread-aware**
  - Ovvero, eseguibili contemporaneamente da più thread
  - Opzioni del GCC: -D_REENTRANT oppure -D_POSIX_C_SOURCE
- Il file .so deve essere incluso nel vostro progetto
  - Nella directory lib/

Caricamento della libreria

- È necessario che la vostra libreria C sia caricata in memoria prima di poterne chiamare le funzioni
- Java offre il metodo statico System.loadLibrary()
- Per essere sicuri che la libreria sia caricata al momento giusto, è pratica comune inserire la chiamata a loadLibrary() in un **inizializzatore statico di classe**
  - In pratica: quando la VM carica la vostra classe, esegue anche il loadLibrary()

Caricamento della libreria

- Esempio:

```
package it.unipi.di.sam.nativa;

class Nativa {
   static {
      System.loadLibrary("mylib");
   }

   …
}
```

Inizializzatore statico.
È naturalmente possibile caricare più librerie diverse, inserendo più chiamate a loadLibrary(). Il sistema tiene traccia delle librerie già caricate.

Caricamento della libreria

- Esempio:

```
package it.unipi.di.sam.nativa;

class Nativa {
   static {
      System.loadLibrary("mylib");
   }

   …
}
```

Il nome "base" della libreria viene rimappato secondo le convenzioni del SO. Nel nostro caso, sarà lib/lib**mylib**.so (con eventuale gestione della versione stile Linux)

Dichiarazione di metodi nativi

- In Java, il qualificatore **native** indica al compilatore che un metodo non ha un corpo proprio
- All'invocazione del metodo, verrà eseguita una funzione C associata, il cui codice si trova in una libreria caricata in precedenza

```
class Nativa {
  native int somma(int a, int b);
  …
}
```

Binding dei nomi

- La corrispondenza dei nomi fra metodo Java e funzione C è data da un processo di *mangling*
  - Simile a quello usato di solito dai compilatori C++
  - Esistono due forme alternative
    - Se un metodo è overloaded, il *mangling* incorporerà la sequenza di descrittori di tipo degli argomenti (nome lungo)
    - Altrimenti, viene effettuato solo il *mangling* del nome del metodo (nome breve)
    - La VM prova prima a cercare il nome “breve”, se non lo trova cerca quello “lungo”
  - Visto che il nome è a nostra discrezione... KISS

Binding dei nomi

- Regole di mangling:
  - Il nome C inizia con il prefisso Java_
  - Segue il nome fully-qualified della classe
    - A sua volta *mangled*
  - Segue un _
  - Segue il nome del metodo
    - A sua volta *mangled*
  - Per i nomi lunghi, segue un __ (doppio _) e poi i codici di tipo degli argomenti
    - A loro volta *mangled*

Binding dei nomi

- Problema!
  - Java consente di avere pressoché tutti i caratteri di UNICODE in un identificatore
  - C consente solo ASCII 7 bit
- Una parte del processo di *mangling* viene quindi applicata a questi identificatori
  - Nomi di package, di classi, di metodi

Binding dei nomi

- Ogni carattere UNICODE non-ASCII7 viene codificato con _0*xxxx*, dove *xxxx* rappresenta il codice UNICODE in esadecimale (lettere minuscole) del carattere incriminato
- Il carattere "_" viene codificato con _1
- Nelle codifiche di tipi, si usano anche i caratteri ";" e "["; questi sono codificate rispettivamente con _2 e _3
- Nei nomi di classi fully-qualified, lo "/" è codificato da "_"

Binding dei nomi

- Le segnature in Java sono codificate (internamente) con la sintassi **(**$T_1 T_2$...**)**R
  - T_n = *tipi degli argomenti*
  - R = *tipo del risultato*
- A noi interessa solo la parte con i tipi degli argomenti
  - Il nome C “lungo” terminerà con __$T_1 T_2$…

Binding dei nomi

- I tipi T sono indicati da un codice alfabetico
  - Z – boolean
  - B – byte
  - C – char
  - D – double
  - F – float
  - I – int
  - J – long
  - S - short
  - V – void
  - L – object
    - seguito dal nome fully qualified di una classe
      - elementi separati da / e terminato da ;
  - [– array
    - Seguito dalla codifica del tipo degli elementi

Binding dei nomi - esempio -

```
package it.unipi.di.sam.nativa;

class Nativa {
   native int somma(int a, int b);
}
```

Java

C

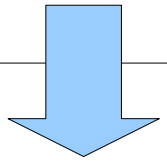
```
Java_it_unipi_di_sam_nativa_Nativa_somma__II
```

Binding dei nomi - esempio -

```
package it.unipi.di.sam.nativa;

class Nativa {
   native void stampa(String[] s, Vector v);
}
```

Java



C

```
Java_it_unipi_di_sam_nativa_Nativa_stampa__3
Ljava_lang_String_2Ljava_util_Vector_2
```

Argomenti

- Alla funzione C (quella con il nome *mangled*) vengono passati due argomenti in più rispetto a quelli del metodo Java:
  - Un puntatore alla struttura **JNIEnv**
  - Un puntatore a **this**
- Tutti gli altri argomenti sono *marshalled*
  - Convertiti da valori Java a valori C
  - Spesso, con limitazioni...

Marshalling dei tipi

- L'header **jni.h** definisce nomi C per i tipi Java

Tipo Java	Tipo C	Note
boolean	jboolean	Unsigned, 8 bit
byte	jbyte	Signed, 8 bit
char	jchar	Unsigned, 16 bit
short	jshort	Signed, 16 bit
int	jint	Signed, 32 bit
long	jlong	Signed, 64 bit
float	jfloat	32 bit
double	jdouble	64 bit
void	jvoid	Non esistono valori

Marshalling dei tipi

- L'header **jni.h** definisce nomi C per i tipi Java

Tipo Java	Tipo C	Note
boolean	jboolean	Unsigned, 8 bit
byte	jbyte	
char	jchar	
short	jshort	
int	jint	
long	jlong	
float	jfloat	
double	jdouble	64 bit
void	jvoid	Non esistono valori

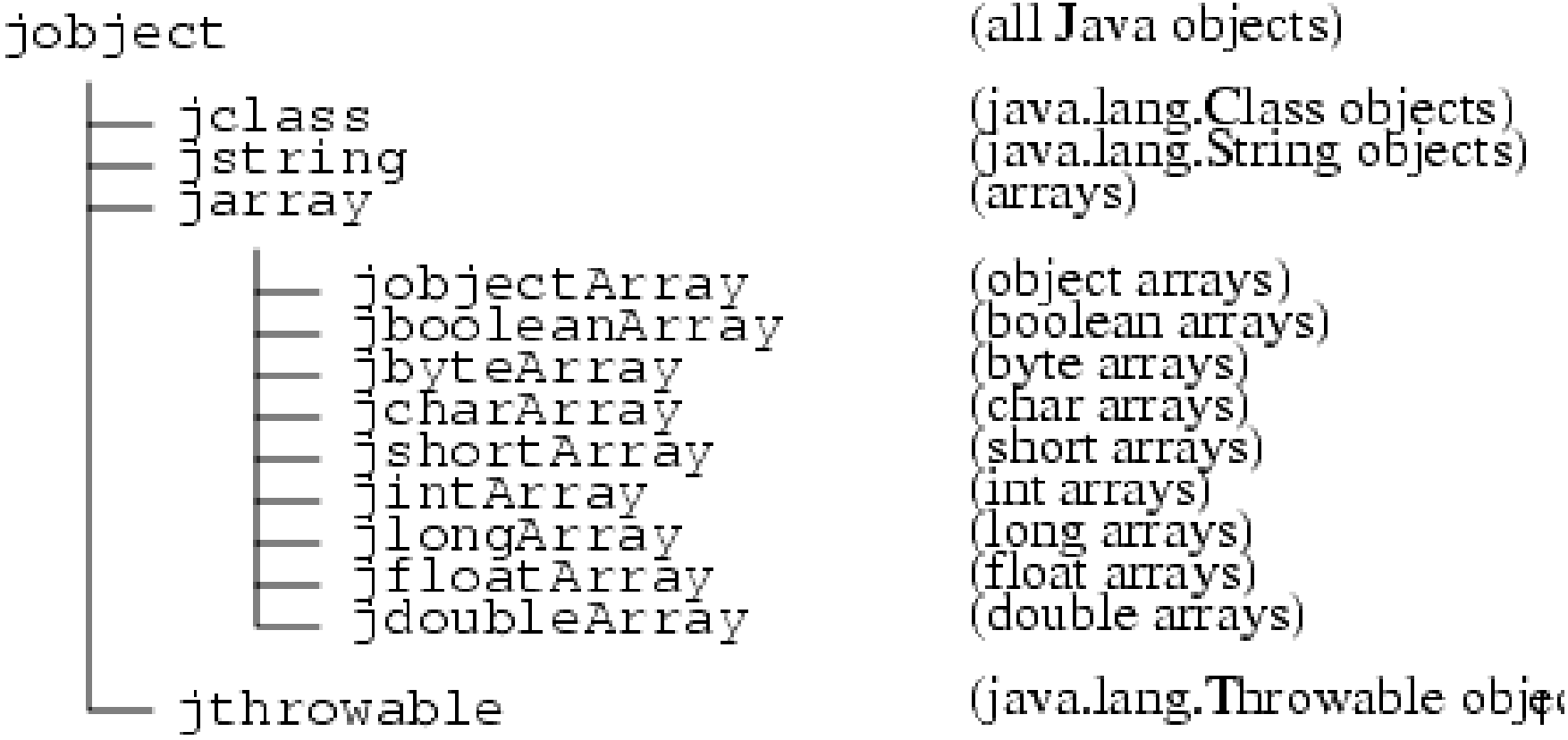
Jni.h contiene altre definizioni utili, per esempio:

```
#define JNI_FALSE 0
#define JNI_TRUE 1
typedef jint jsize;
```

Date un'occhiata!

Marshalling dei tipi

- Per i tipi riferimento si usa una “gerarchia”



Ma nella pratica, spesso si usa jobject e via!

Esempio

- In definitiva, le segnature vengono trasformate così:

```
package it.unipi.di.sam.nativa;

class Nativa {
   native int somma(int a, int b);
}
```

```
#include <jni.h>
jint Java_it_unipi_di_sam_nativa_Nativa_somma__II(
   JNIEnv *env, jobject this, jint a, jint b) { … }
```

Accedere al mondo Java da C

- Se la vostra funzione si limita a leggere i valori degli argomenti (di tipi **base**), fare qualche conto, e restituire un risultato, siete a cavallo
  - Il 90% degli esempi che si trova in giro è così!
  - char, int, double ecc. vengono **copiati** fra Java e C
- Ma in casi reali, dovrete fare accesso agli oggetti Java da C
  - Gli altri oggetti sono passati **per riferimento**
- Tutti questi accessi devono essere **mediati** da apposite funzioni offerte da JNIEnv

Accedere al mondo Java da C

- Per accedere ai **campi** e ai **metodi** di un oggetto Java (di cui si ha il riferimento) occorre usare degli identificatori numerici (in realtà, sono puntatori C)
  - jfieldID – id di un campo
  - jmethodID – id di un metodo
- I valori vengono ottenuti chiamando funzioni di JNIEnv:

```
jfieldID GetFieldID(JNIEnv *env, jclass clazz,
                    const char *name, const char *sig);
```

Accedere al mondo Java da C

- Una volta ottenuto il jfieldID di un campo, si legge il valore con un'altra funzione di JNIEnv

```
NativeType Get<type>Field(JNIEnv *env,
                          jobject obj, jfieldID fieldID);
```

- È una famiglia di funzioni
  - jint GetIntField(), jdouble GetDoubleField(), ecc.
- Esistono funzioni di JNIEnv per una varietà di compiti
  - Leggere e scrivere campi, allocare e liberare oggetti, istanziare classi, chiamare metodi, ecc.

Esempio

Parte Java

```
class FieldAccess {
  static int si;
  String s;

  private native void accessFields();

  public static void main(String args[]) {
    FieldAccess c = new FieldAccess();
    FieldAccess.si = 100;
    c.s = "abc";
    c.accessFields();
    System.out.println("J  FieldAccess.si = " + FieldAccess.si);
    System.out.println("J  c.s = \"" + c.s + "\"");
  }

  static {
    System.loadLibrary("MyImpOfFieldAccess");
  }
}
```

Esempio

Parte C

```
#include <stdio.h>
#include <jni.h>
#include "FieldAccess.h"

JNIEXPORT void JNICALL Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
  jclass cls = (*env)->GetObjectClass(env, obj);
  jfieldID fid;
  jstring jstr;
  const char *str;
  jint si;

  fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
  if (fid == 0) return;
  si = (*env)->GetStaticIntField(env, cls, fid);
  printf("C  FieldAccess.si = %d\n", si);
  (*env)->SetStaticIntField(env, cls, fid, 200);

  fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");
  if (fid == 0) return;
  jstr = (*env)->GetObjectField(env, obj, fid);
  str = (*env)->GetStringUTFChars(env, jstr, 0);
  printf("C  c.s = \"%s\"\n", str);
  (*env)->ReleaseStringUTFChars(env, jstr, str);

  jstr = (*env)->NewStringUTF(env, "123");
  (*env)->SetObjectField(env, obj, fid, jstr);
}
```

Esempio

Generato dal sistema di build del vostro IDE, oppure (a mano) dal comando **javah -jni <classe>**

Parte C

```
#include <stdio.h>
#include <jni.h>
#include "FieldAccess.h"

JNIEXPORT void JNICALL Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
  jclass cls = (*env)->GetObjectClass(env, obj);
  jfieldID fid;
  jstring jstr;
  const char *str;
  jint si;

  fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
  if (fid == 0) return;
  si = (*env)->GetStaticIntField(env, cls, fid);
  printf("C  FieldAccess.si = %d\n", si);
  (*env)->SetStaticIntField(env, cls, fid, 200);

  fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");
  if (fid == 0) return;
  jstr = (*env)->GetObjectField(env, obj, fid);
  str = (*env)->GetStringUTFChars(env, jstr, 0);
  printf("C  c.s = \"%s\"\n", str);
  (*env)->ReleaseStringUTFChars(env, jstr, str);

  jstr = (*env)->NewStringUTF(env, "123");
  (*env)->SetObjectField(env, obj, fid, jstr);
}
```

Alcune altre funzioni utili

- FindClass
- GetSuperClass
- NewGlobalRef / DeleteGlobalRef
- NewLocalRef
- IsSameObject
- AllocObject
- NewObject
- GetObjectClass
- IsInstanceOf
- Call<ε/NonVirtual/Static><type>Method<ε/a/v>
- <Get/Set><ε/Static><type>Field
- NewString<ε/UTF> / GetStringLength<ε/UTF> / GetStringChars<ε/UTF> / ReleaseStringChars<ε/UTF>
- New<type>Array / Get<type>ArrayElements / Release<type>ArrayElements
- GetJavaVM

La struttura JavaVM

- La funzione di JNIEnv GetJavaVM restituisce un puntatore a una struttura JavaVM che “rappresenta” la macchina virtuale Java
  - JavaVM offre a sua volta una tabella di puntatori a funzione che consentono di manipolare la VM
- Tuttavia, Android non usa la JavaVM!
- Non sempre la specifica JNI e il comportamento di Android sono consistenti
  - In particolare: Android usa una sola VM per tutto, con il metodo dei zygote Dalvik, o precompila con ART

Local e Global reference

- La maggior parte dei valori usati nel codice C saranno riferimenti **locali**
  - Validi per tutto il tempo di esecuzione della funzione C
- Se però allocate nuovi oggetti, e volete restituirli al chiamante, non potranno essere riferimenti locali
  - Verrebbero distrutti all'uscita
- Per questo motivo si possono dichiarare riferimenti **globali**
  - Che però vanno deallocati a mano: niente garbage collection

Consigli della nonna

- Consiglio pragmatico:
  - Cercate di strutturare l'interfaccia fra Java e C in modo che tutta la parte “complicata” sia fatta in Java
  - La parte in C dovrebbe solo fare number crunching su tipi base, oppure accesso all'hardware
    - Ma quest'ultimo solo se programmate Android custom!
  - Non appena avete oggetti o eccezioni che attraversano l'interfaccia Java/C, correte il rischio di fare grossa confusione
- Soprattutto: il codice C è **unsafe!**
  Potete andare in crash!

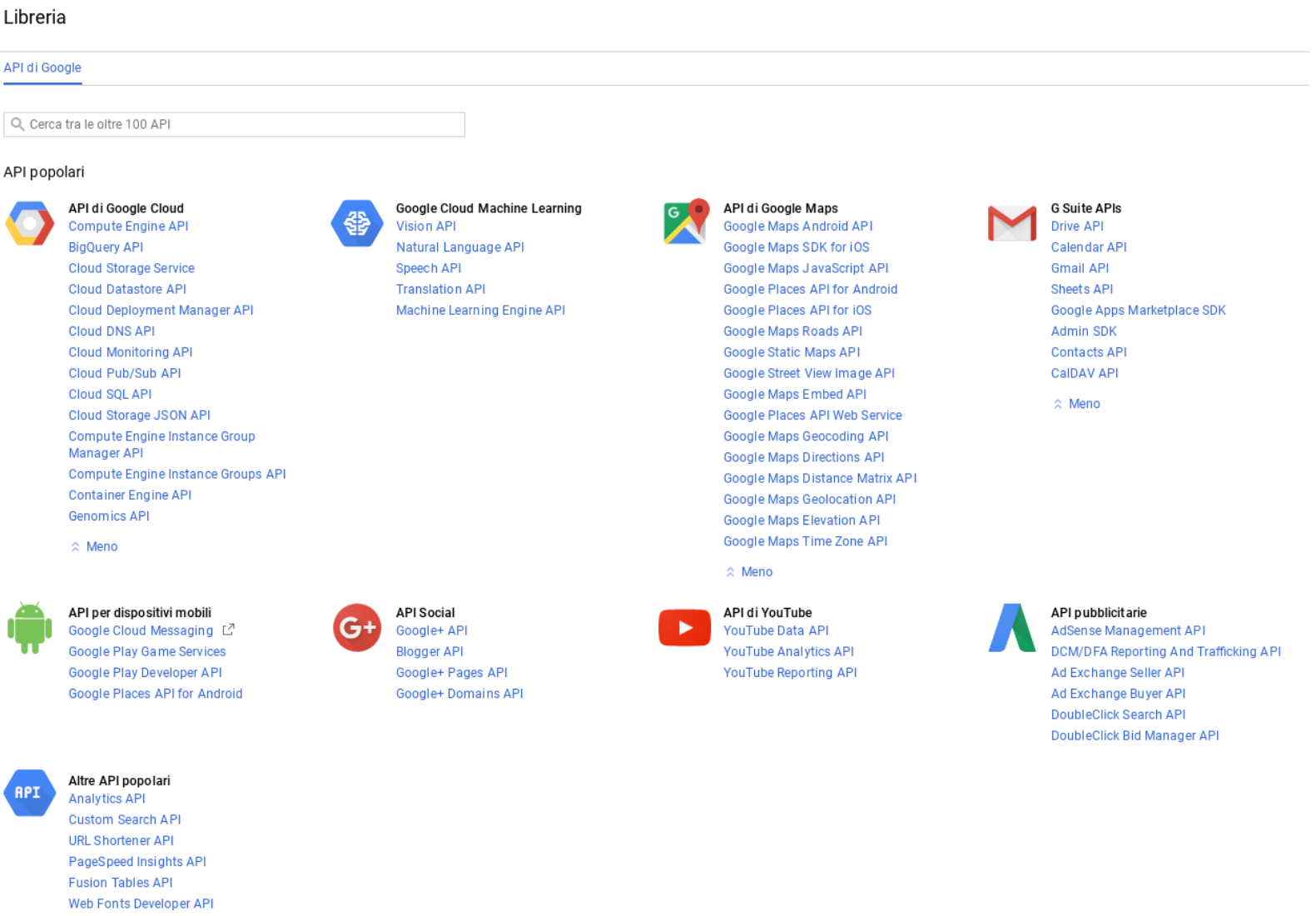
Temi non trattati

- JNI consente di interfacciarsi con C++, oltre che con C
- Eccezioni (throw e catch) in C
- Creazione di thread (POSIX) dal C
- Weak references in C
- Monitor, regioni critiche, concorrenza
- Manipolazioni esplicite dello stack

Google APIs

Google APIs

- Oltre 100 APIs
- Ciascuna con dozzine o centinaia di metodi
- Così tanti servizi da richiedere un motore di ricerca interno con una sezione delle API più popolari!
- Marketplace di APIs a pagamento



API Explorer

https://developers.google.com/apis-explorer

Google

Search for services, methods, and recent requests... Loading...

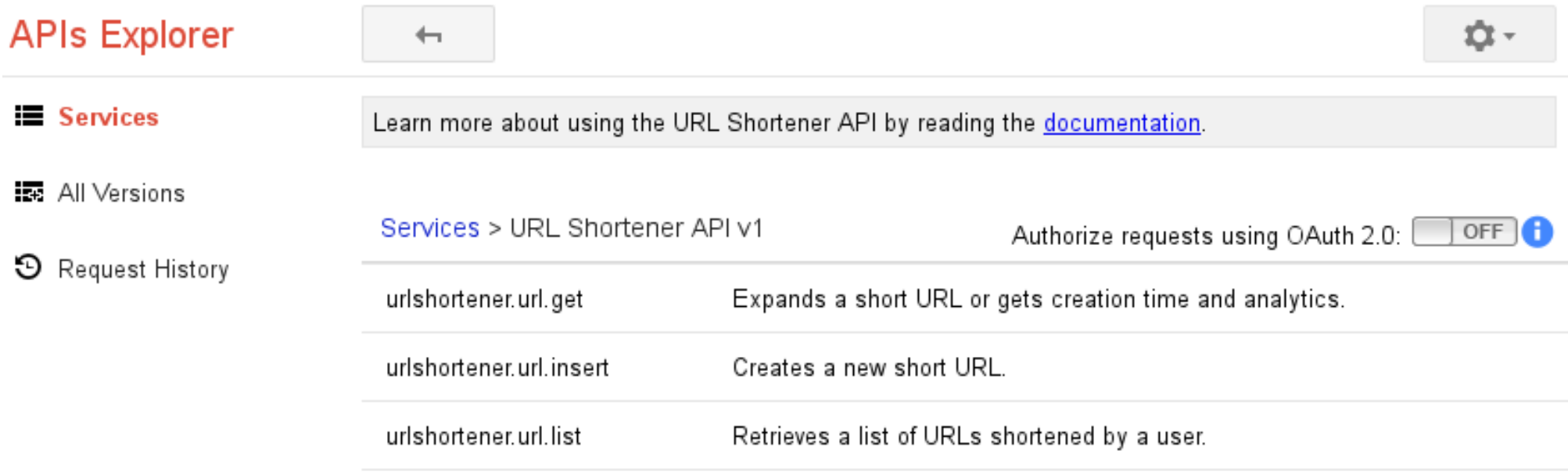
APIs Explorer

- Services
- All Versions
- Request History

Accelerated Mobile Pages (AMP) URL API	v1	This API contains a single method, batchGet. Call this method to retrieve the AMP URL (and equivalent AMP Cache URL) for given public URL(s).
Ad Exchange Buyer API	v1.4	Accesses your bidding-account information, submits creatives for validation, finds available direct deals, and retrieves performance reports.
Ad Exchange Buyer API II	v2beta1	Accesses the latest features for managing Ad Exchange accounts and Real-Time Bidding configurations.
Ad Exchange Seller API	v2.0	Accesses the inventory of Ad Exchange seller users and generates reports.
Admin Reports API	reports_v1	Fetches reports for the administrators of G Suite customers about the usage, collaboration, security, and risk for their users.
AdSense Host API	v4.1	Limited Availability Generates performance reports, generates ad codes, and provides publisher management capabilities for AdSense Hosts.
AdSense Management API	v1.4	Accesses AdSense publishers' inventory and generates performance reports.
APIs Discovery Service	v1	Provides information about other Google APIs, such as what APIs are available, the resource, and method details for each API.
BigQuery API	v2	A data platform for customers to create, manage, share and query data.
BigQuery Data Transfer Service API	v1	Transfers data from partner SaaS applications to Google BigQuery on a scheduled, managed basis.
Blogger API	v3	Limited Availability API for access to the data within Blogger.
Books API	v1	Searches for books and manages your Google Books library.
Calendar API	v3	Manipulates events and other calendar data.
Cloud Monitoring API	v2beta2	Accesses Google Cloud Monitoring data.
Cloud Source Repositories API	v1	Access source code repositories hosted by Google.
Cloud Spanner API	v1	Cloud Spanner is a managed, mission-critical, globally consistent and scalable relational database service.
Cloud SQL Administration API	v1beta4	Creates and configures Cloud SQL instances, which provide fully-managed MySQL databases.
Cloud Storage JSON API	v1	Stores and retrieves potentially large, immutable data objects.

API Explorer

https://developers.google.com/apis-explorer

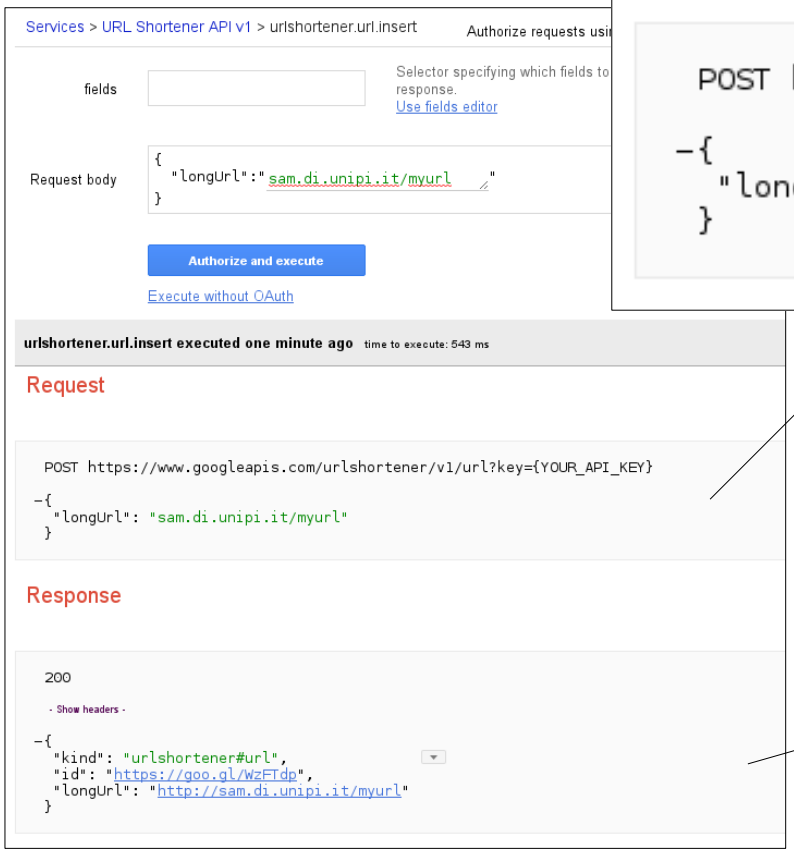


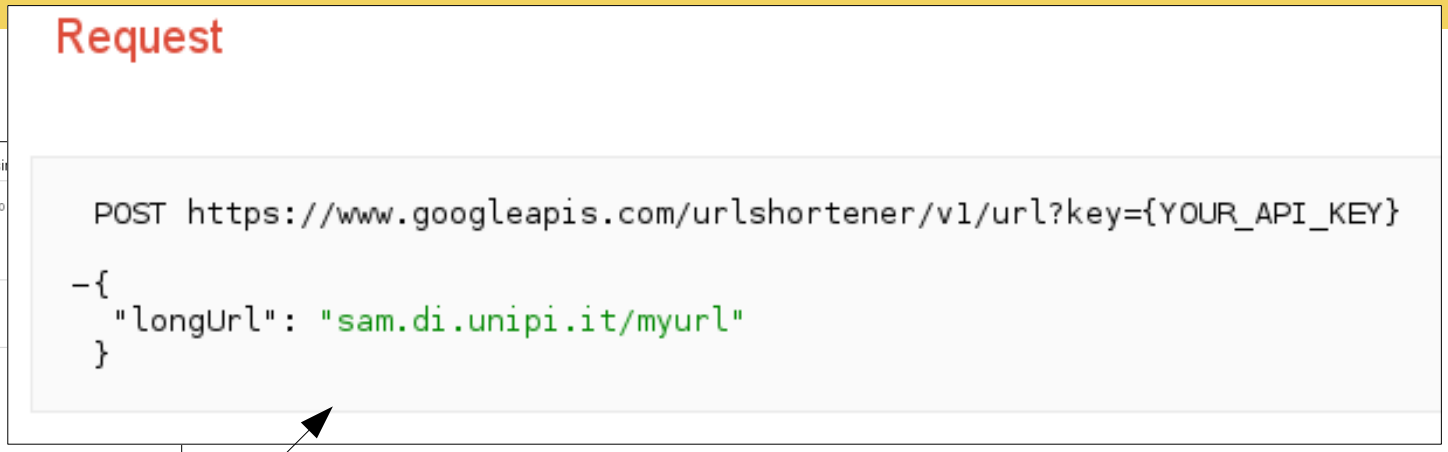
Ogni API offre un insieme di metodi che possono essere chiamati in stile REST.

Ogni chiamata incluse una **richiesta** (HttpRequest), tipicamente con un *payload* JSON che fornisce gli argomenti.

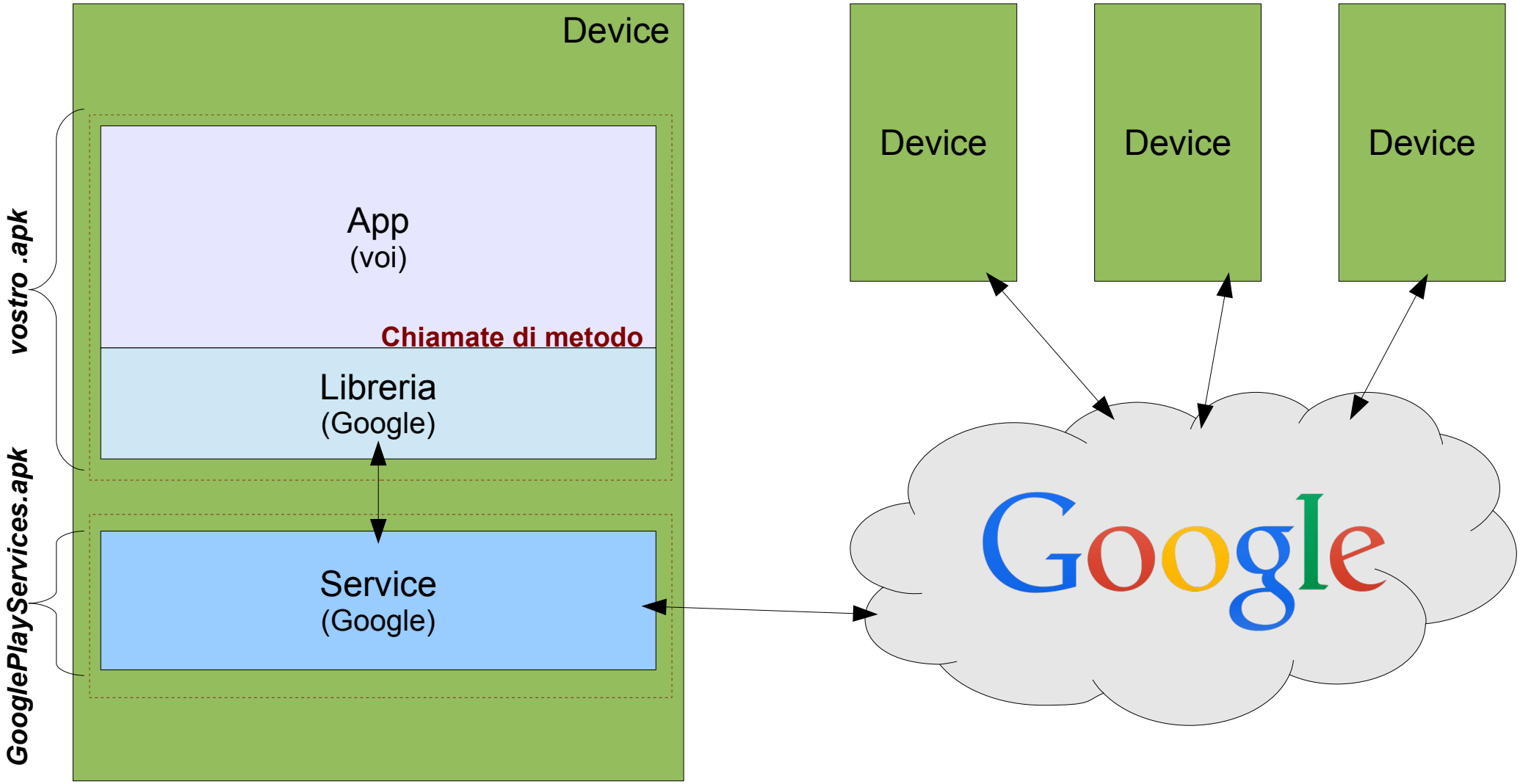
La **risposta** è una coppia <*codice*,*corpo*> in cui il *codice* è un error code HTTP (200=ok, 404=forbidden, ecc.), mentre il *corpo* è un oggetto JSON i cui campi rappresentanto il risultato della chiamata.

Esempio: Url shortener

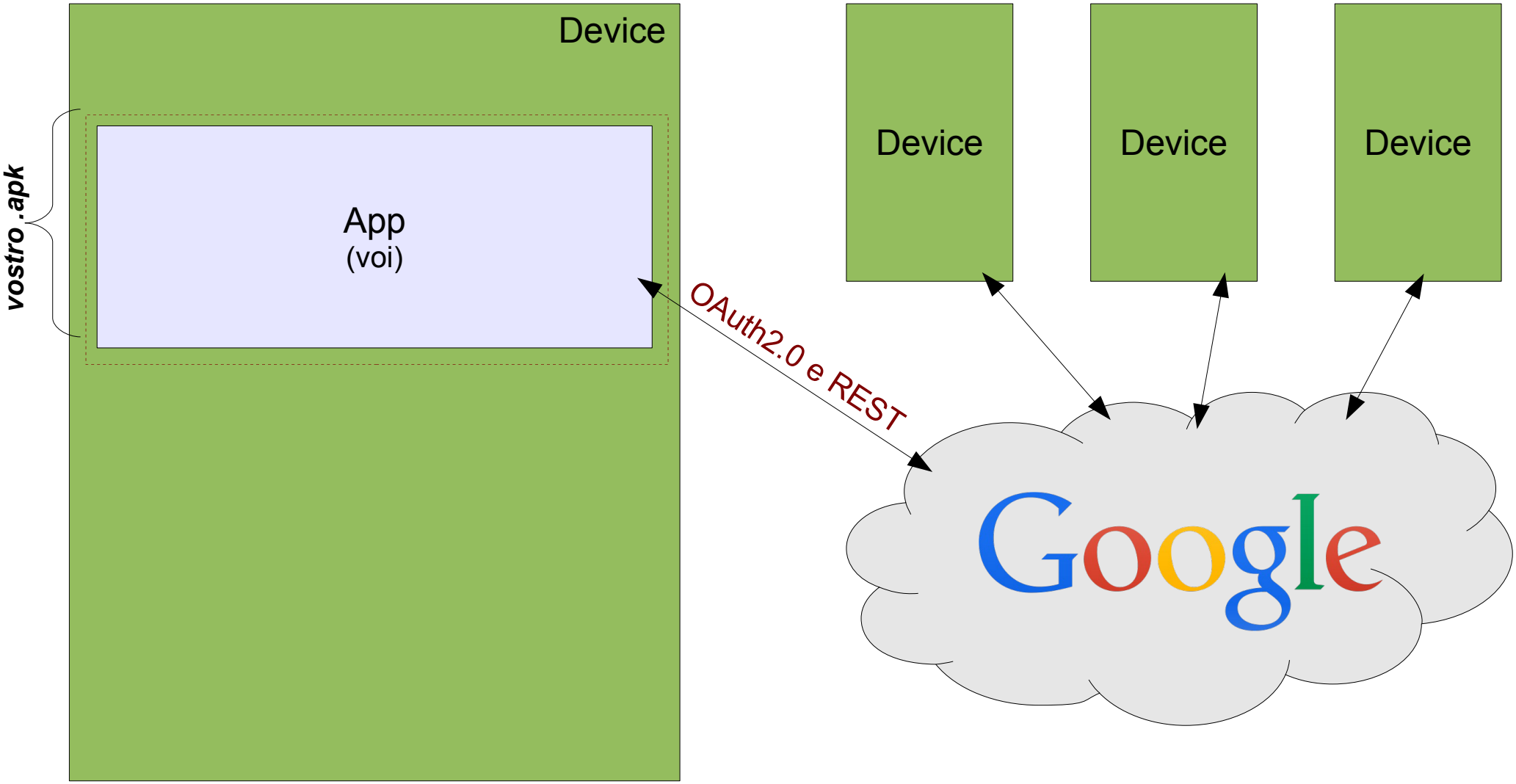




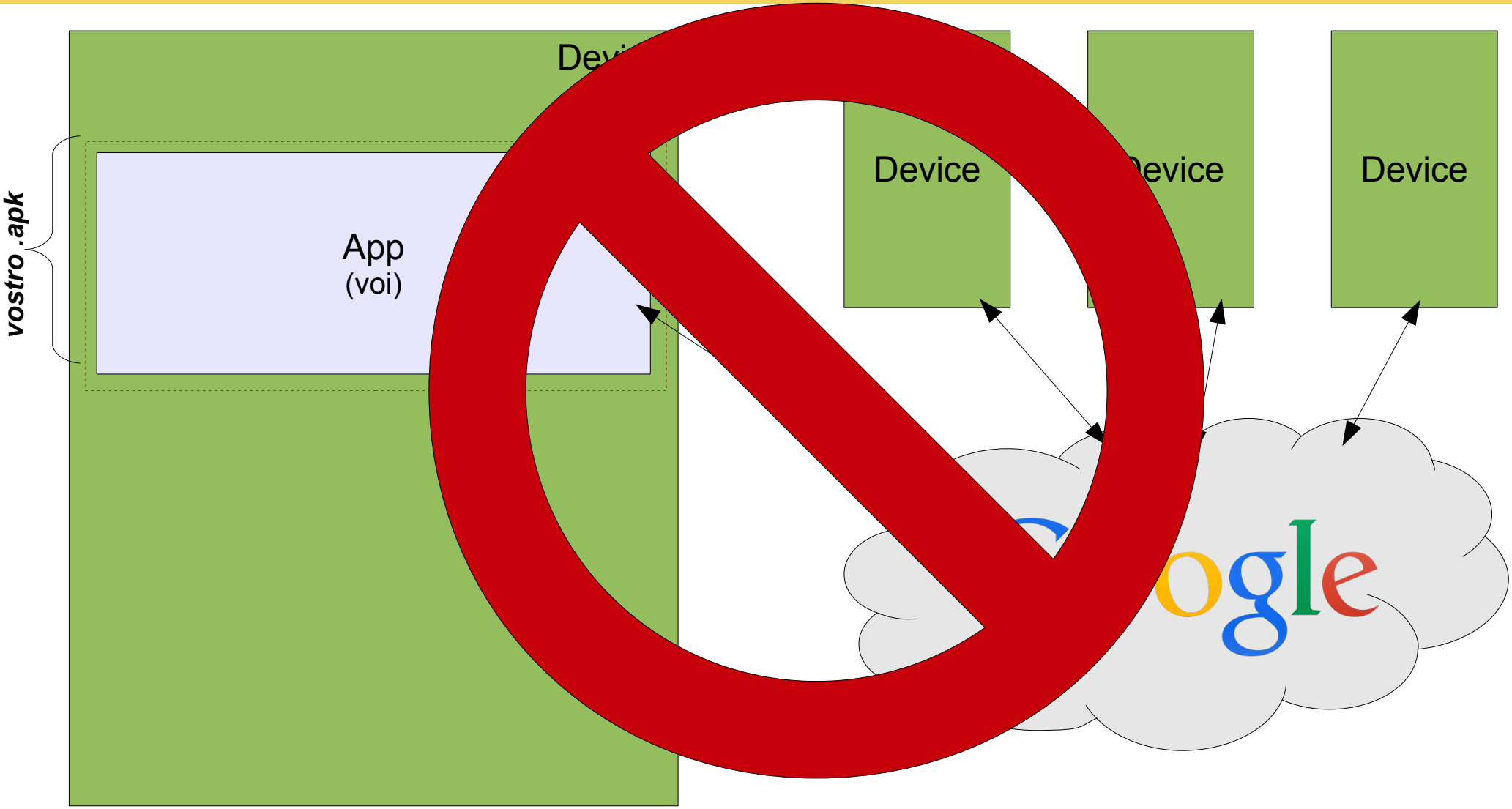
Architettura "Google APIs for Android"



Architettura alternativa



Architettura alternativa



Connessione a Google API

- Usiamo il pattern *Builder*, aggiungendo due interfacce e un gruppo di API:

```
GoogleApiClient gapi = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(new GoogleApiClient.ConnectionCallbacks() {
        public void onConnected(Bundle hint) {
            // Possiamo usare l'API
        }
        public void onConnectionSuspended(int res) {
            // Accesso all'API sospeso – ma tornerà!

        }
    })
    .addOnConnectionFailedListener(new GoogleApiClient.OnConnectionFailedListener() {
        public void onConnectionFailed(ConnectionResult res) {
                // Connession fallita. Addio mondo crudele!
        }
    })
    .addApi(Wearable.API) // Lista delle Google API a cui vogliamo accedere
    .build();
```

Spesso (ma non sempre) fatto nella onCreate() dell'activity

Connessione a GoogleAPI

- Si può anche, come al solito, fare implementare le interfacce all'Activity in questione, attivare più API insieme, ecc.
  - E magari scrivere di meno!
- Per esempio:

```
gapi = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .addApiIfAvailable(Wearable.API)
    .addApi(Drive.API)
    .addScope(Drive.SCOPE_FILE)
    .build();
```

Wear solo se disponibile.

Poi si può usare gapi.hasConnectedApi(Wearable.API) per scoprire a runtime se l'API è disponibile.

Drive sempre – altrimenti è un errore

Connessione a GoogleAPI

- L'ultimo passo è anche il più semplice!
- Per avviare la connessione alle API selezionate:

```
gapi.connect();
```

- Schemi tipici:
  - Uso solo durante l'attività
    - gapi.connect() nella onStart(), gapi.disconnect() nella onStop()
  - Uso solo in certi casi specifici
    - gapi.connect(); operazione; gapi.disconnect()
  - Uso continuo
    - gapi.connect() nella onCreate(), gapi.disconnect() nella onDestroy()

Connessione a GoogleAPI

- La gestione degli errori invece è una tragedia
  - Dopo una chiamata a connect() può darsi che…
    - Tutto bene: onConnected(hint)
    - Qualcosa storto: onConnectionFailed(res)
- In caso di fallimenti, è possibile che il sistema offra una ***risoluzione*** implicita
  - Esempio: il client non era autenticato
  - Spesso le risoluzioni necessitano di azione dell'utente
  - In ogni caso… possiamo esprimere l'*intenzione* di rimediare!

Connessione a GoogleAPI

- Esempio

```
public void onConnectionFailed(ConnectionResult res) {
    if (risincorso) {
        // Stiamo già cercando di risolvere l'errore
        return;
    } else if (res.hasResolution()) {
        try {
            risincorso = true;
            res.startResolutionForResult(this, REQUEST_RESOLVE_ERROR);
        } catch (SendIntentException e) {
            // Il tentativo di risoluzione ha causato errore. Riproviamo.
            gapi.connect();
        }
    } else {
        // Niente da fare, dialog d'errore all'utente
        risincorso = true;
        showErrorDialog(res.getErrorCode());
    }
}
```

GooglePlayServicesUtil.getErrorDialog(errorCode, this.getActivity(), REQUEST_RESOLVE_ERROR);

Connessione a GoogleAPI

- Se abbiamo avviato un tentativo di risoluzione, verrà chiamato più avanti il nostro onActivityResult() con RESULT_OK
  - via startResolutionForResult()
  - via GooglePlayServicesUtil.getErrorDialog()
- A quel punto, vale la pena di riprovare la connect()
- In caso contrario… siamo alla disperazione, errore permanente!

GoogleAPIs

- Alcune delle API disponibili:
  - Search – è Google!
  - Ads – pubblicità e simili
  - Analytics – analisi di traffico
  - AppInvite – gestione beta tester
  - Cast – Chromecast e simili
  - Auth e Identity – identità degli utenti
  - Drive – storage condiviso
  - Fit – informazioni sull'attività fisica
  - Games – aspetti sociali dei giochi
  - Location, Maps, Panorama e Nearby – geocose
  - Plus – accesso a G+
  - Wallet – pagamenti elettronici
  - Wear – dispositivi indossabili

GoogleAPIs

- Trovate un elenco completo di API a https://developers.google.com/products/
- Alcune richiedono particolari relazioni commerciali, o che registriate la vostra applicazione sulla Developer Console, o non sono rilevanti per Android

Operazioni su GoogleAPI

- Le operazioni su GoogleAPI sono per loro natura **asincrone** e **fallibili**
  - Per forza: passano da rete, sotto c'è REST
- Tutte le chiamate vengono fatte tramite metodi statici di classi di libreria corrispondenti alle API
  - Hanno come parametro `gapi`
  - In molti casi, restituiscono un PendingResult
  - Sul PendingResult si può agire in tre modi
    - Registrando una **callback** da chiamare quando il risultato è pronto
    - **Sospendendo** il thread in attesa che il risultato sia pronto
    - **Cancellando** l'operazione (e testando se è stata cancellata)

Operazioni su GoogleAPI
Callback

- Come di consueto...
  - Si chiama **setResultCallback**(ResultCallback<R> rc) sul PendingResult
  - Quando l'operazione sarà conclusa, il risultato (di tipo R) verrà passato a rc
  - R estende Result
    - Al momento, sono definite 71 sottoclassi di Result nella libreria, con i diversi risultati specifici di diverse chiamate
    - Result fornisce di suo solo getStatus(), la quale restituisce un'oggetto Status
      - Status a sua volta ha un codice di successo (successo / interrotto / fallito / cancellato), una stringa che rappresenta un messaggio di stato, e un PendingIntent

Operazioni su GoogleAPI
Callback

- Esempio:

```
private void loadFile(String filename) {
    Query q = new Query.Builder()
        .addFilter(Filters.eq(SearchableField.TITLE, filename))
        .build();
    Drive.DriveApi.query(gapi, q)
        .setResultCallback(new ResultCallback() {
            public void onResult(DriveApi.MetadataBufferResult r) {
                MetadataBuffer mdb=r.getMetaDataBuffer();
                if (mdb!=null) {
                    for (Metadata md: mdb) {
                        int size=md.getFileSize();
                        String desc=md.getDescription();
                        ...
                    }
                    mdb.release();
                }
            }
    });
}
```

Di setResultCallback() esiste anche la versione con timeout: scaduto il tempo indicato, la chiamata si considera fallita.

Operazioni su GoogleAPI
Sospensione

- Sul PendingResult si chiama la **await**()
- Il thread chiamante viene bloccato finché il risultato non è disponibile
  - Quindi: **mai** chiamare sul thread UI!
- La await() restituisce il risultato
  - Di tipo R, sottoclasse di Result
  - Lo stesso oggetto che sarebbe passato a onResult()

Di await() esiste anche la versione con timeout: scaduto il tempo indicato, la chiamata si considera fallita.

Operazioni su GoogleAPI
Cancellazione

- Su un PendingResult non ancora completato si può chiamare la **cancel**()
  - Se era stata impostata una callback, questa viene chiamata con un parametro Result
  - Se un thread era bloccato su una await(), viene risvegliato e si ritorna un Result
- In entrambi i casi, il Result conterrà come status code l'indicazione dell'interruzione
- Il metodo **isCanceled**() di PendingResult indica se il PendingResult è stato cancellato o meno