Android: Threads and Services

http://developer.android.com/guide/components/processes-and-threads.html

http://developer.android.com/guide/components/services.html

Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2018-2019)

Outline

Outline

Reference of Available Intents [http://developer.android.com/guide/appendix/app-intents.html]

This table describe some of the default applications and settings that Google provides in their standard Android implementation.

Target application	Intent URI	Intent action	Result
Browser	http://*web_address* https://*web_address*	VIEW	Open a browser window to the URL specified
	"" (empty string) http://*web_address* https://*web_address*	WEB_SEARCH	Opens the file at the location on the device in the browser
Dialer	tel: *phone_number*	CALL	Calls the entered phone number. This requires your application to request the following permission in your manifest: <uses-permission id="android.permission.CALL_PHONE" />
	tel:*phone_number* voicemail:	DIAL	Dials (but does not actually initiate the call) the number given (or the stored voicemail on the phone)
Google Maps	geo:*latitude*,*longitude* geo:*latitude*,*longitude*?z=*zoom* geo:0,0?q=*my+street+address* geo:0,0?q=*business+near+city*	VIEW	Opens the Maps application to the given location or query
Google Streetview	google.streetview:cbll=*lat*,*lng*&cbp=1,*yaw*,,*pitch*,*zoom*&mz=*mapZoom*	VIEW	Opens the Street View application to the given location. The URI scheme is based on the syntax used for Street View panorama information in Google Maps URLs

MainActivity.java

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinIntent.git]

Create menu:

MainActivity.kt:

```
1    override fun onCreateOptionsMenu(menu: Menu): Boolean {
2        menuInflater.inflate(R.menu.intent_menu, menu)
3        return super.onCreateOptionsMenu(menu)
4    }
```

intent_menu.xml:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/
       android">
3      <item android:id="@+id/menuWebpage"
4            android:title="@string/menu_webpage"/>
5      <item android:id="@+id/menuGoogle"
6            android:title="@string/menu_google"/>
7      <item android:id="@+id/menuPlace"
8            android:title="@string/menu_place"/>
9      <item android:id="@+id/menuDial"
10           android:title="@string/menu_dial"/>
11 </menu>
```

Manage the events:

```
1    override fun onOptionsItemSelected(item: MenuItem?): Boolean {
2        return when (item?.itemId) {
3            R.id.menuWebpage ->  IntentUtils.invokeWebBrowser(this)
4            R.id.menuGoogle -> IntentUtils.invokeWebSearch(this)
5            R.id.menuPlace -> IntentUtils.showPlace(this)
6            R.id.menuDial -> IntentUtils.dial(this)
7            else -> super.onOptionsItemSelected(item)
8        }
9    }
```

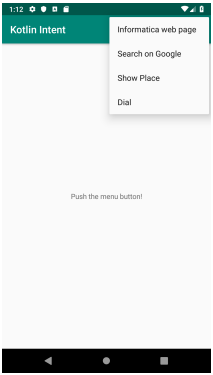
object IntentUtils

[http://developer.android.com/guide/components/intents-common.html]

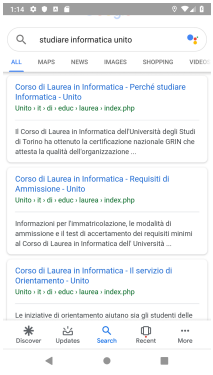
```
1     ...
2     fun invokeWebBrowser(activity: Activity): Boolean {
3         val intent = Intent(
4             Intent.ACTION_VIEW,
5             Uri.parse("http://magistrale.educ.di.unito.it")
6         )
7         activity.startActivity(intent)
8         return true
9     }
10
11    fun invokeWebSearch(activity: Activity): Boolean {
12        val intent = Intent(Intent.ACTION_WEB_SEARCH)
13        intent.putExtra(SearchManager.QUERY,
14                        "studiare informatica unito")
15        activity.startActivity(intent)
16        return true
17    }
```

```
1
2     fun dial(activity: Activity): Boolean {
3         val intent = Intent(
4             Intent.ACTION_DIAL,
5             Uri.parse("tel:12345678")
6         )
7         activity.startActivity(intent)
8         return true
9     }
10
11    fun showPlace(activity: Activity): Boolean {
12        val intent = Intent(
13            Intent.ACTION_VIEW,
14            Uri.parse("geo:45.090137, 7.659323?z=17")
15        )
16        activity.startActivity(intent)
17        return true
18    }
19    ...
```

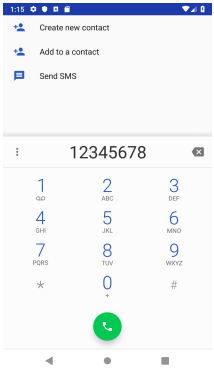
Execution:


1:12
Kotlin Intent
Informatica web page
Search on Google
Show Place
Dial
Push the menu button!


1:14
magistrale.educ.di.unito.it
di.unito.it
MENU
INTRANET
Corso di Laurea Magistrale in Informatica
"Il modo migliore per predire il futuro è inventarlo". - Alan Kay, parafrasando Cesare Pavese
Il Corso di Laurea Magistrale in Informatica (LM-18) dell'Università degli Studi di Torino (certificazione di qualità GRIN) rappresenta il percorso necessario per la completa specializzazione dell'informatico, facilitando il suo immediato inserimento nel mondo
ITALIAN
ENGLISH


1:14
studiare informatica unito
ALL
MAPS
NEWS
IMAGES
SHOPPING
Corso di Laurea in Informatica - Perché studiare Informatica - Unito
Unito › it › di › educ › laurea › index.php
Il Corso di Laurea in Informatica dell'Università degli Studi di Torino ha ottenuto la certificazione nazionale GRIN che attesta la qualità dell'organizzazione ...
Corso di Laurea in Informatica - Requisiti di Ammissione - Unito
Unito › it › di › educ › laurea › index.php
Informazioni per l'immatricolazione, le modalità di ammissione e il test di accertamento dei requisiti minimi al Corso di Laurea in Informatica dell' Università ...
Corso di Laurea in Informatica - Il servizio di Orientamento - Unito
Unito › it › di › educ › laurea › index.php
Le iniziative di orientamento aiutano sia gli studenti delle
Discover
Updates
Search
Recent
More


1:15
Try gas stations, ATMs
Health club
Centro Piero della Francesca
Università di Torino Dipartimento di...
Furniture store
Amedeo di Savoia
Fermata 186 - OSPEDALE AMEDEO DI SAVOIA
Ospedale Amedeo di Savoia
GO
Google


1:15
Create new contact
Add to a contact
Send SMS
12345678
1 QD
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
*
0 +
#

Outline

Processes and Threads

- When an application component starts
  - If the application does not have any other components running, then the Android system starts a new Linux process for the application with a single thread of execution.
    - By default, all components of the same application run in the same process and thread (called the "main" thread).
  - If an application component starts and there already exists a process for that application,[1] then the component is started within that process and uses the same thread of execution. However, you can arrange for
    - different components in your application to run in separate processes,[2] and
    - you can create additional threads for any process.

[1] Because another component from the application exists

[2] THESE SLIDES DO NOT ILLUSTRATE THIS POSSIBILITY.

Process lifecycle

The Android system tries to maintain an application process for as long as possible, but eventually needs to remove old processes to reclaim memory for new or more important processes. There are five levels in the importance hierarchy. The following list presents the different types of processes in order of importance (the first process is most important and is killed last):

1. **Foreground process.** A process that is required for what the user is currently doing.
2. **Visible process.** A process that doesn't have any foreground components, but still can affect what the user sees on screen.
3. **Service process.** A process that is running a service that has been started with the startService() method and does not fall into either of the two higher categories.
4. **Background process.** A process holding an activity that's not currently visible to the user (the activity's onStop() method has been called).
5. **Empty process.** A process that doesn't hold any active application components. The only reason to keep this kind of process alive is for caching purposes, to improve startup time the next time a component needs to run in it.

How apps work (by default)

- The system creates a thread for the application, called ‘main” or “UI thread”
  - It dispatches events to the user interface
- Everything happens in the UI thread
  - Long operations block the whole UI
  - No events can be dispatched
- Being blocked for more than 5 secs causes “application not responding” being presented to the user

Threads

- Android natively supports multi-threading
- An application can comprise concurrent threads
- Threads are managed like in Java by
  - Extending class Thread
  - Implementing interface Runnable
    - Method run() is executed when Thread.start() is launched

Rules to Android's single thread model

The Android UI toolkit is not thread-safe!

The following rules naturally arise.

1. Do not block the UI thread. I.e.:
   - If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).
2. Do not access the Android UI toolkit from outside the UI thread. I.e.:
   - All manipulations to the user interface should be done within the UI thread.
   - Do not manipulate your UI in a worker thread.

Example (Bad)

[Violation of rule 1]

```
    override fun onClick(v: View) {
        val bitmap = loadImageFromNetwork("http://example.com/image.png")
        myImageView.setImageBitmap(bitmap)
    }
```

[Violation of rule 2]

```
    override fun onClick(v: View) {
        Thread(Runnable {
            val bitmap = loadImageFromNetwork("http://example.com/image.png")
            mImageView.setImageBitmap(bitmap)
        }).start()
    }
```

[Violation of ?]

```
    override fun onClick(v: View) {
        var bitmap: Bitmap
        Thread(Runnable {
            bitmap = loadImageFromNetwork("http://example.com/image.png")
        }).start()
        myImageView.setImageBitmap(bitmap)
    }
```

How can we fix the problem?

- Message-passing like mechanism for thread communication
- Each thread is associated with a queue of messages
- Different ways to access the UI thread from other threads
  - Activity.runOnUiThread(Runnable)
  - View.post(Runnable)
  - View.postDelayed(Runnable, long)
- The Runnable is sent to the UI thread and run within it
  - It is invoked on a View from outside the UI thread

Example (Fixed)

You can fix the previous code by using the View.post(Runnable) method:

```
override fun onClick(v: View) {
    Thread(Runnable {
        val bitmap = loadImageFromNetwork("http://example.com/image.png")
        mImageView.post(Runnable {
            mImageView.setImageBitmap(bitmap)
        })
    }).start()
}
```

Now this implementation is thread-safe: the network operation is done from a separate thread while the ImageView is manipulated from the UI thread.

Example (Fixed)

You can fix the previous code by using the View.post(Runnable) method:

```
override fun onClick(v: View) {
    Thread(Runnable {
        val bitmap = loadImageFromNetwork("http://example.com/image.png")
        mImageView.post(Runnable {
            mImageView.setImageBitmap(bitmap)
        })
    }).start()
}
```

Now this implementation is thread-safe: the network operation is done from a separate thread while the ImageView is manipulated from the UI thread.

- As the complexity of the operation grows, this kind of code can get complicated and difficult to maintain.
  - To handle more complex interactions with a worker thread, you might consider using a Handler in your worker thread, to process messages delivered from the UI thread.
  - The best solution is to extend the AsyncTask class, which simplifies the execution of worker thread tasks that need to interact with the UI.

Asynchronous tasks

- Enable proper and easy use of the UI thread
  - Allow one to perform background operations and publish results on the UI thread without having to manipulate threads
- One must subclass AsyncTask and implement the doInBackground() method that runs in a pool of background threads
- An AsyncTask instance has to be created on the UI thread and can be executed only once
- To run the task call execute() from the UI thread
- We can cancel the task at any time from any thread (through method cancel())

Example (Fixed by using AsyncTask)

You can implement the previous example using AsyncTask this way:

```
1     override fun onClick(v: View) {
2         DownloadImageTask().execute("http://example.com/image.png")
3     }
4 private class DownloadImageTask : AsyncTask<String, Void, Bitmap>() {
5     /** The system calls this to perform work in a worker thread and delivers it the parameters given to AsyncTask.execute()  */
6     override fun doInBackground(vararg urls: String): Bitmap {
7         return loadImageFromNetwork(urls[0])
8     }
9     /** The system calls this to perform work in the UI thread and delivers the result from doInBackground()  */
10    override fun onPostExecute(result: Bitmap) {
11        mImageView.setImageBitmap(result)
12    }
13 }
```

Now the UI is safe and the code is simpler, because it separates the work into the part that should be done on a worker thread and the part that should be done on the UI thread.

Example (Fixed by using AsyncTask)

You can implement the previous example using AsyncTask this way:

```
1     override fun onClick(v: View) {
2         DownloadImageTask().execute("http://example.com/image.png")
3     }
4 private class DownloadImageTask : AsyncTask<String, Void, Bitmap>() {
5     /** The system calls this to perform work in a worker thread and delivers it the parameters given to AsyncTask.execute()  */
6     override fun doInBackground(vararg urls: String): Bitmap {
7         return loadImageFromNetwork(urls[0])
8     }
9     /** The system calls this to perform work in the UI thread and delivers the result from doInBackground()  */
10    override fun onPostExecute(result: Bitmap) {
11        mImageView.setImageBitmap(result)
12    }
13 }
```

Now the UI is safe and the code is simpler, because it separates the work into the part that should be done on a worker thread and the part that should be done on the UI thread.

- doInBackground() executes automatically on a worker thread
  - This step is used to perform long-running computations in background
  - The result of the computation must be returned by this step and passed back
- onPreExecute(), onPostExecute() and onProgressUpdate() are all invoked on the UI thread
  - The value returned by doInBackground() is sent to onPostExecute()
- We can call publishProgress() at anytime in doInBackground() to execute onProgressUpdate() on the UI thread

Example of Async Task

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinAsyncTask.git]

In MainActivity.kt we download a JSON with the list of information of a Flicker gallery:

```
1    ...
2        val strPhoto = getJSON(url, 10000)
3        val jsonAll = JSONObject(strPhoto)
4
5        val jsonPhotos = jsonAll.optJSONObject("photos")
6        val jsonPhoto = jsonPhotos.optJSONArray("photo")
7
8        lista.apply {
9            layoutManager = LinearLayoutManager(applicationContext)
10           adapter = MyListAdapter(jsonPhoto)
11       }
12   ...
```

and into the MyListAdapter we download all images we need.

Example of Async Task

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinAsyncTask.git]

In MyListAdapter we define an holder with some custom property:

```
1    ...
2    // Provide a reference to the views for each data item
3    class ViewHolder(val myView: View,
4                     var imageURL: String,
5                     var toDownload: Boolean = true,
6                     var bitmap: Bitmap? = null) : RecyclerView.ViewHolder(myView)
7    ...
```

Example of Async Task

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinAsyncTask.git]

Override the method to display the data at the specified position:

```
1   // Replace the contents of a view (invoked by the layout manager)
2   override fun onBindViewHolder(holder: MyListAdapter.ViewHolder, position: Int) {
3
4       // https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg
5       val imageData = jsonPhoto.optJSONObject(position)
6       val strURL = "https://farm" + imageData.optInt("farm", 1) +
7               ".staticflickr.com/" + imageData.optString("server", "") +
8               "/" + imageData.optString("id", "") + "_" + imageData.optString("secret", "") + ".jpg"
9
10      holder.myView.photoTitle.text = imageData.optString("title", "")
11      if (strURL == holder.imageURL && holder.bitmap != null) {
12          holder.myView.photo.setImageBitmap(holder.bitmap)
13      } else {
14          holder.apply {
15              imageURL = strURL
16              toDownload = true
17              myView.photo.setImageResource(R.drawable.empty)
18          }
19          DownloadAsyncTask().executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, holder)
20      }
21  }
```

Example of Async Task

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinAsyncTask.git]

Do the background work:

```
override fun doInBackground(vararg params: MyListAdapter.ViewHolder): MyListAdapter.ViewHolder {
    val viewHolder = params[0]
    try {
        if (viewHolder.toDownload) {
            viewHolder.toDownload = false
            val imageURL = URL(viewHolder.imageURL)
            viewHolder.bitmap = BitmapFactory.decodeStream(imageURL.openStream())
        }
    } catch (e: IOException) {
        viewHolder.bitmap = null
    }
    return viewHolder
}
```

Show the result:

```
override fun onPostExecute(result: MyListAdapter.ViewHolder) {
    if (result.bitmap == null) {
        result.myView.photo.setImageResource(R.drawable.empty)
    } else {
        result.myView.photo.setImageBitmap(result.bitmap)
    }
}
```

Caution: A problem you might encounter when using a worker thread is unexpected restarts in your activity due to a runtime configuration change [http://developer.android.com/guide/topics/resources/runtime-changes.html] (such as when the user changes the screen orientation), which may destroy your worker thread. To see how you can persist your task during one of these restarts and how to properly cancel the task when the activity is destroyed, see the source code for the Shelves [https://code.google.com/p/shelves/] sample application.

Outline

Broadcast receiver

- A component that allows us to register for system or application events
  - All registered receivers for an event are notified by the Android runtime once the event happens
- A broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work
  - A receiver can be registered via the manifest
    - Through tag <receiver>
  - We can also register a receiver via method Context.registerReceiver()
- The implementing class extends class BroadcastReceiver
  - Method onReceive() is called by the Android system
    - Once the code returns, the system considers the object to be finished and no longer active

System Events

- Several system events are defined as final static fields in class Intent
  [http://developer.android.com/reference/android/content/Intent.html]
- Other Android system classes also define events

Event	Description
Intent.ACTION_BOOT_COMPLETED	Boot completed. Requires the android.permission.RECEIVE_BOOT_COMPLETED permission.
Intent.ACTION_POWER_CONNECTED	Power got connected to the device.
Intent.ACTION_POWER_DISCONNECTED	Power got disconnected to the device.
Intent.ACTION_BATTERY_LOW	Triggered on low battery. Typically used to reduce activities in your app which consume power.
Intent.ACTION_BATTERY_OKAY	Battery status good again.

Dynamic invocations

- Intent objects are delivered to all interested parties
- Android finds the appropriate activity, service, or broadcast receiver to respond to the intent
  - Instantiates them if necessary
- Broadcast intents are delivered only to broadcast receivers, never to activities or services
  - An intent passed to startActivity() is delivered only to an activity, never to a service or broadcast receiver, etc.

sendBroadcast and sendOrderedBroadcast

- sendBroadcast
  - Broadcasts the given intent to all interested BroadcastReceivers
    - An optional required permission could be enforced
  - This call is asynchronous; it returns immediately
- sendOrderedBroadcast
  - Broadcasts the given intent to all interested BroadcastReceivers
    - Delivering them one at a time to allow preferred receivers to consume the broadcast before it is delivered to the others
  - This call is asynchronous; it returns immediately

It is an helper to register for and send broadcasts of Intents to local objects within your process—this is has a number of advantages over sending global broadcasts with sendBroadcast(Intent):

- Broadcasted data do not leave your app
  - No need to worry about leaking private data
- Other apps cannot communicate with these broadcasts
  - No need to worry about having security holes
- More efficient than sending a global broadcast through the system

Outline

Services

- Application components that perform long-running operations in background without a user interface
  - Must be declared in the manifest
  - Any app component can use a service in the same way as any component can use an activity
    - We can declare the service as private, in the manifest file, and block access from other applications
- Continue to run in background even if the user switches to another application
- A component can also bind to a service to interact with it and perform inter-process communication (IPC)

A service can essentially take two forms:

- **Started.** A service is "started" when an application component starts it by calling startService()
  - A started service performs a single operation and does not return a result to the caller
- **Bound.** A service is "bound" when an application component binds to it by calling bindService()
  - A bound service offers a client-server interface that allows components to interact with the service, even across processes with inter-process communication (IPC)
  - A bound service runs only as long as another application component is bound to it
  - Multiple components can bind to the service

Services can work both ways at the same time—it can be started (to run indefinitely) and also allow binding.

- It's simply a matter of whether you implement a couple callback methods:
  - onStartCommand() to allow components to start it, and
  - onBind() to allow binding.

Caution:

- A service runs in the main thread of its hosting process
  - The service does not create its own thread and does not run in a separate process
- If your service is going to do any CPU intensive work you should create a new thread within the service to do that work

Creating a Service

To create a service, you must create a subclass of Service (or one of its existing subclasses)—the most important callback methods you should override are:

- onStartCommand()
  - The system calls this method when another component requests that the service be started (through startService(Intent))
- stopSelf() or stopService(Intent)
  - For the self-termination of the service or for asking for the termination of a service from the outside
  - No need to implement these methods if we only want to provide binding
  - System-decided termination (i.e., memory shor tage)
- onCreate()
  - The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either onStar tCommand() or onBind())
    - If the service is already running, this method is not called
- onDestroy()
  - The system calls this method when the service is no longer used and is being destroyed

startService() vs bindService()

- If a service is started by invoking startService()
  - It keeps running until it stops itself or another component stops it
- If a service is created by invoking bindService() (and onStartCommand() is not called)
  - It only runs as long as a component is bound to it
  - When the service is unbound from all clients, it is destroyed

A started service is one that another component starts by calling startService(), resulting in a call to the service's onStartCommand() method. Traditionally, there are two classes you can extend to create a started service:

- **Service.** This is the base class for all services.
  - When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.
- **IntentService.** This is a subclass of Service that uses a worker thread to handle all start requests, one at a time.
  - This is the best option if you don't require that your service handle multiple requests simultaneously.

- Provides a straightforward solution for handling asynchronous requests (expressed through Intents)
  - All you need to do is implement onHandleIntent(), which receives the intent for each start request so you can do the background work.
- Clients send requests through startService(Intent)
  - The service is started as needed and handles Intents using a worker thread automatically
  - The service stops itself as soon it runs out of work
- All requests are handled by a single worker thread
  - Implementation of pattern "Work queue processor"

Foreground/Background services

- A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a status bar icon. Foreground services continue running even when the user isn't interacting with the app.
- A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

  Note: If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself is not in the foreground. In most cases like this, your app should use a scheduled job instead.

Scheduled services

- A JobScheduler (API level 21) launches the service
- The system schedules the jobs for execution at the appropriate times
- Google recommends that we use JobScheduler to execute background services

Outline

Service vs Thread

- A service can run in the background even when the user is not interacting with the application.
- A thread can perform work outside the main thread, but only while the user is interacting with the application.

Outline

Service Notifications

- Once running, a service can notify the user of events using
  - Toast notifications are messages that appear on the surface of the current window for a moment then disappear
  - Status bar notifications provide an icon in the status bar with a message, the user can select it to take an action
    - ⋆ This is the best technique when some background work has completed

Status Bar Notifications

- Used by background services to notify the occurrence of an event without interrupting the operations of the foreground activities
  - Display an icon on the Status Bar
  - Display a message in the Notification Window
  - Fire an event in case the user selects the notification

Toast Notifications

- A Toast Notification is a message that pops up on the surface of the window, and automatically fades out
  - Typically created by the foreground activity
  - Display a message text and then fades out
  - Does not accept events! (use Status Bar Notifications instead)

Outline

Android Interface Definition Language (AIDL)

[http://developer.android.com/guide/components/aidl.html]

AIDL supports Inter-Process Communication (IPC)

- AIDL is similar to other IDLs using
- Used to define the programming interface that both the client and service agree upon to communicate with each other
  - One process cannot normally access the memory of another process
  - Actors need to decompose their objects into primitives that the operating system can understand, and marshal the objects across the boundary
  - Android handles it for us with AIDL

Note. Using AIDL is necessary only if you allow clients from different applications to access your service for IPC and want to handle multithreading in your service.

- If you do not need to perform concurrent IPC across different applications, you should create your interface by implementing a Binder or,
- if you want to perform IPC, but do not need to handle multithreading, implement your interface using a Messenger.

Regardless, be sure that you understand Bound Services before implementing an AIDL.