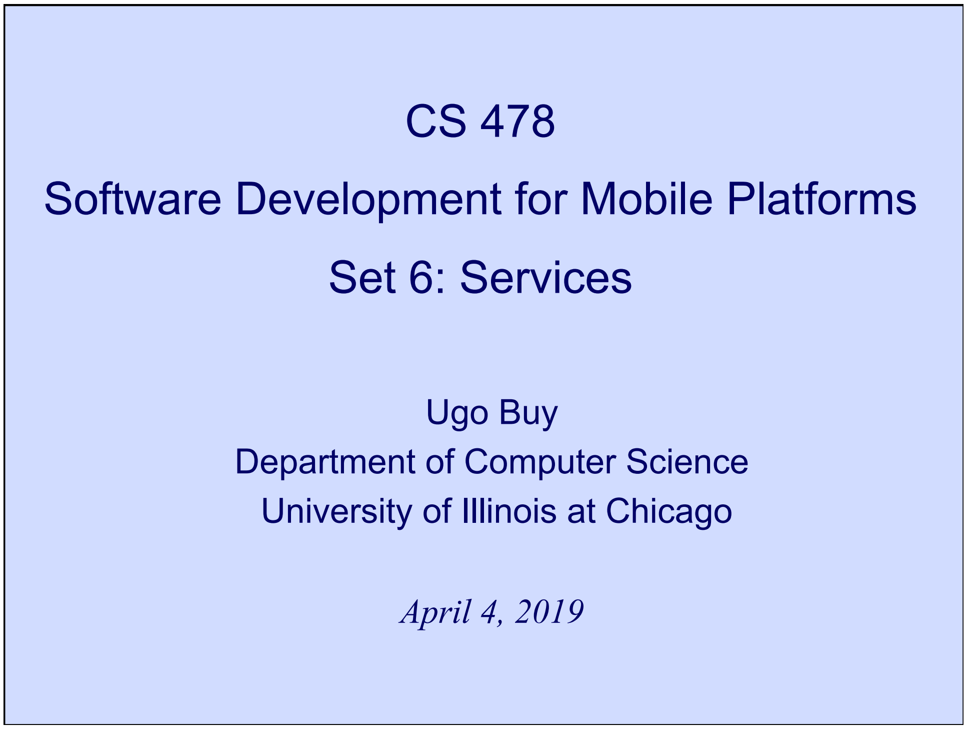


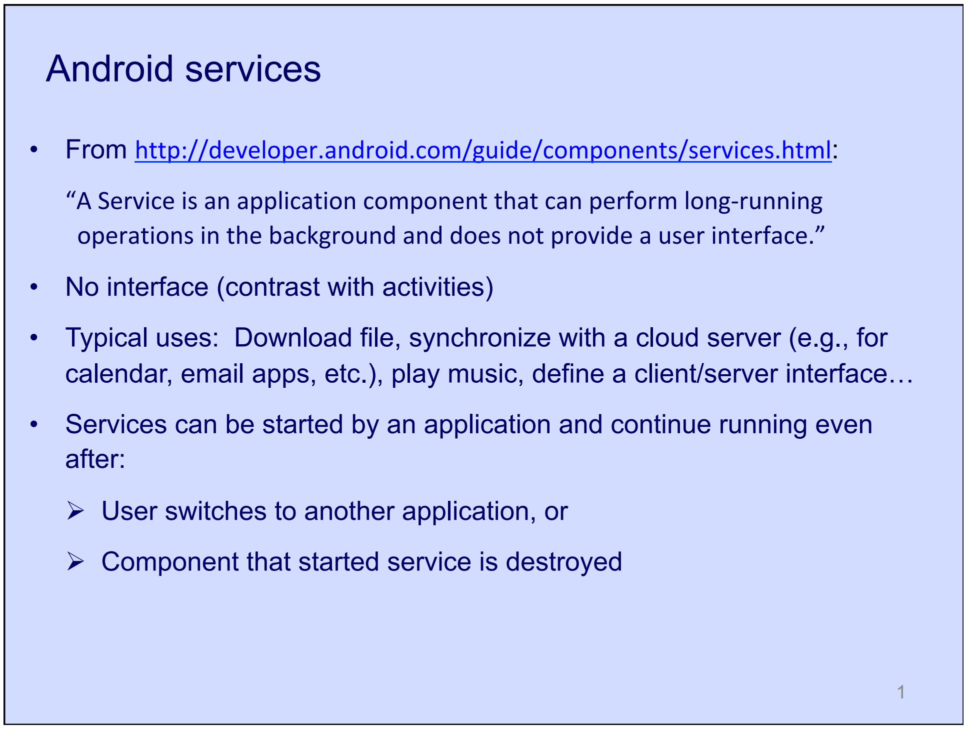
CS 478

Software Development for Mobile Platforms

Set 6: Services

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

April 4, 2019



Android services

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

  "A Service is an application component that can perform long-running operations in the background and does not provide a user interface."
- No interface (contrast with activities)
- Typical uses: Download file, synchronize with a cloud server (e.g., for calendar, email apps, etc.), play music, define a client/server interface…
- Services can be started by an application and continue running even after:
  - User switches to another application, or
  - Component that started service is destroyed

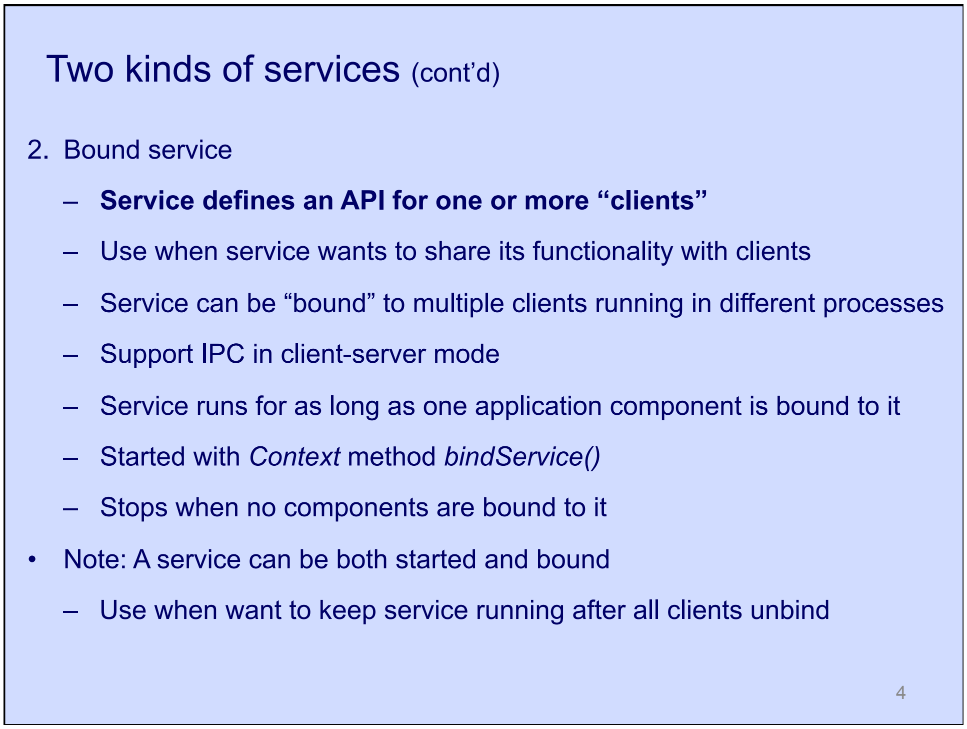
Basic facts about services

Two kinds of services

1. Started service:
   - Further distinction for started services:
     - A. Foreground service
     - B. Background service
   - Can run indefinitely, even after component that started service is destroyed
   - Does not return a result
   - Starts with *Context* method *startService()*, passing an intent
   - Can stop by calling *stopSelf()*, or be stopped by another component with call to *stopService()*

Started services: Background vs. foreground status

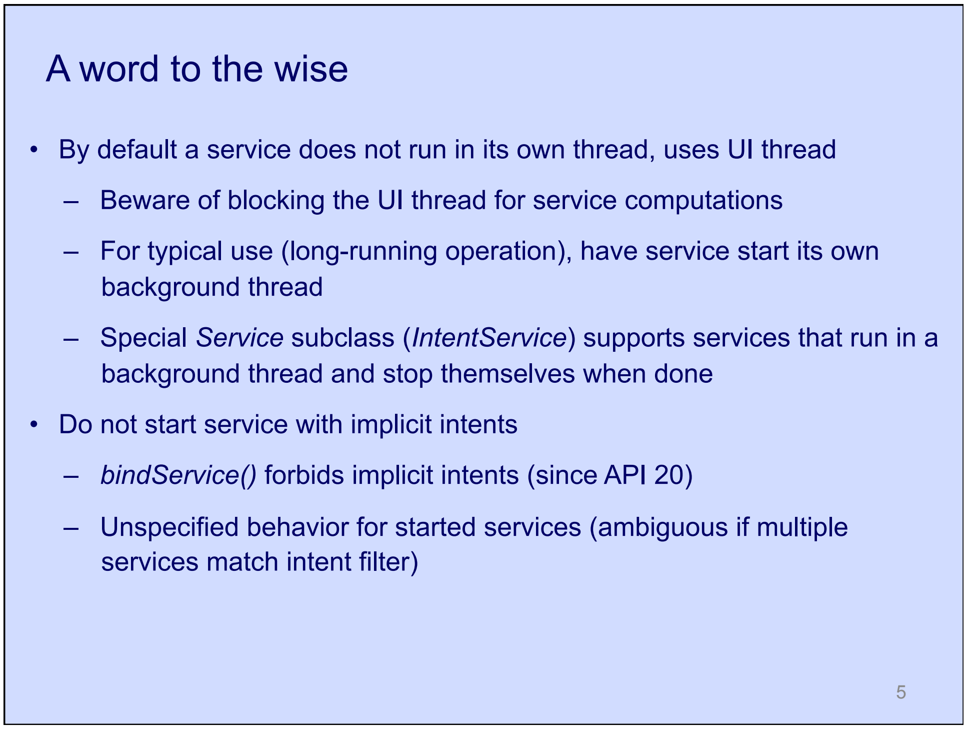
- Background service: No effect on user experience
  - Backing up device on cloud
  - Synchronize with cloud service (e.g., for calendar, email, contact, etc.)
  - Listen for goals scored by favorite soccer team
- Foreground service affects UX
  - Music playing service
- By default service runs in background
  - Elevate to foreground status by calling *startForeground()* in *onCreate* or *onStartCommand()* callbacks
- OS now routinely kills background services



Two kinds of services (cont'd)

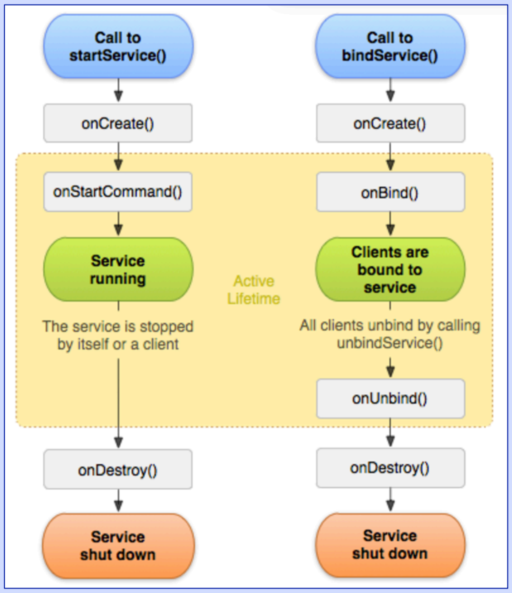
2. Bound service
   - **Service defines an API for one or more "clients"**
   - Use when service wants to share its functionality with clients
   - Service can be "bound" to multiple clients running in different processes
   - Support IPC in client-server mode
   - Service runs for as long as one application component is bound to it
   - Started with *Context* method *bindService()*
   - Stops when no components are bound to it

- Note: A service can be both started and bound
  - Use when want to keep service running after all clients unbind



A word to the wise

- By default a service does not run in its own thread, uses UI thread
  - Beware of blocking the UI thread for service computations
  - For typical use (long-running operation), have service start its own background thread
  - Special *Service* subclass (*IntentService*) supports services that run in a background thread and stop themselves when done
- Do not start service with implicit intents
  - *bindService()* forbids implicit intents (since API 20)
  - Unspecified behavior for started services (ambiguous if multiple services match intent filter)

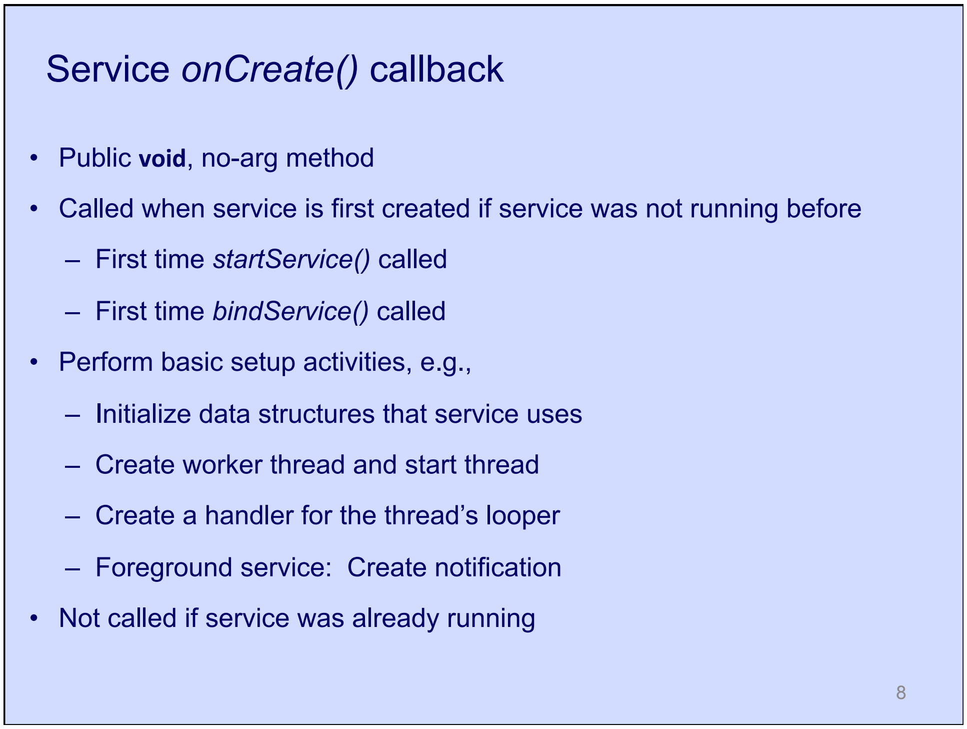


Lifetime of started and bound services

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

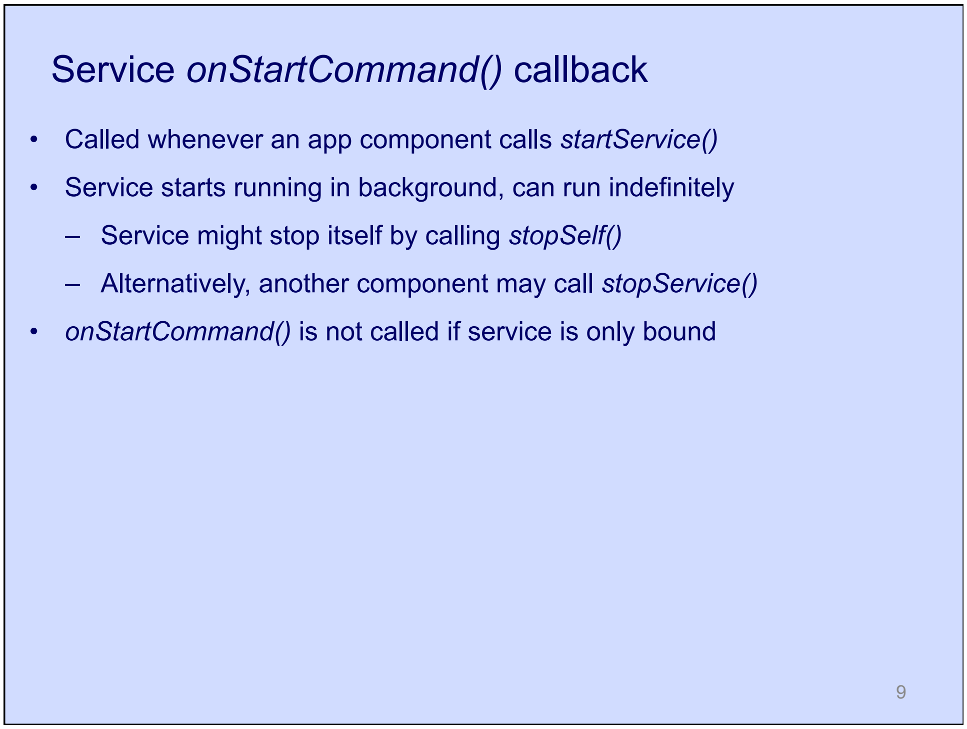
Creating services

- Extend framework class *Service* or *IntentService* (for services using a background thread)
  - If you extend *Service*, you should probably create a background thread anyways
  - Otherwise, service will run in app's UI thread, possibly blocking it
- Override key service lifecycle methods
  - *onCreate(),* called component calls *startService()* the first time
  - *onStartCommand(),* called whenever component calls *startService()*
  - *onBind(),* called when component calls *bindService()* method
  - *onDestroy()*, called when service stopped
  - *onHandleIntent()* – for *IntentService* subclasses only



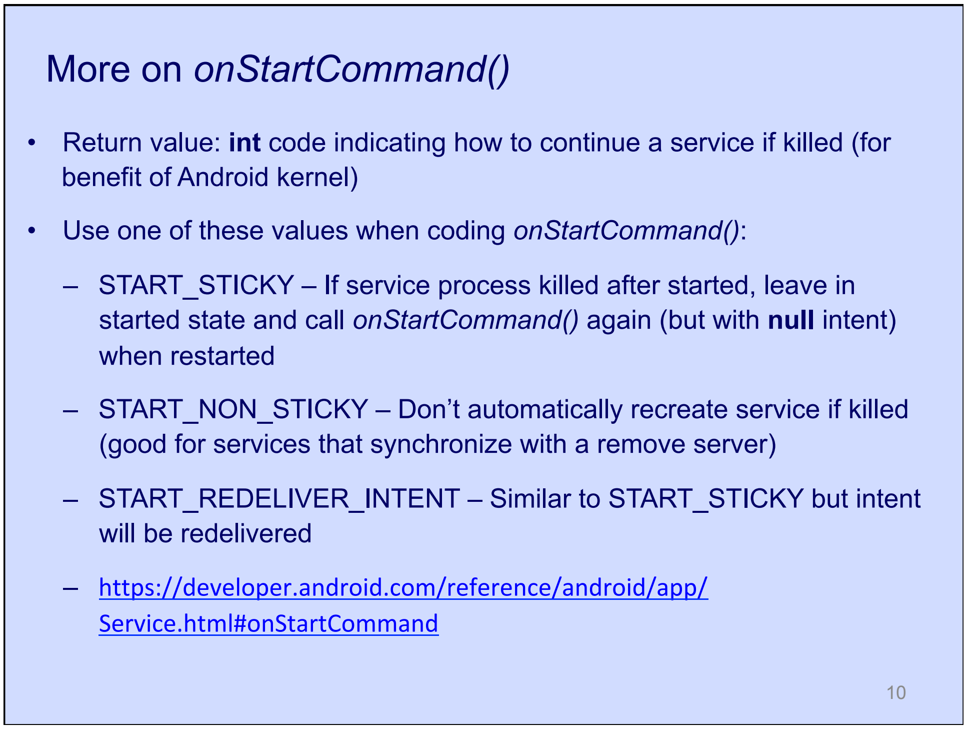
Service *onCreate()* callback

- Public **void**, no-arg method
- Called when service is first created if service was not running before
  - First time *startService()* called
  - First time *bindService()* called
- Perform basic setup activities, e.g.,
  - Initialize data structures that service uses
  - Create worker thread and start thread
  - Create a handler for the thread’s looper
  - Foreground service: Create notification
- Not called if service was already running



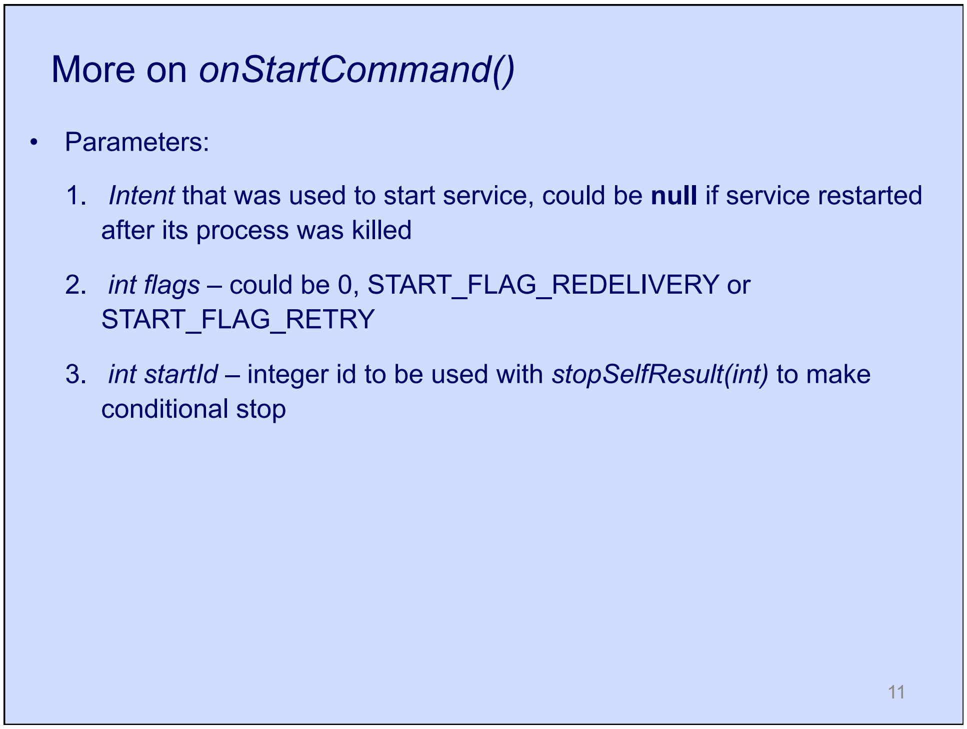
Service *onStartCommand()* callback

- Called whenever an app component calls *startService()*
- Service starts running in background, can run indefinitely
  - Service might stop itself by calling *stopSelf()*
  - Alternatively, another component may call *stopService()*
- *onStartCommand()* is not called if service is only bound



More on *onStartCommand()*

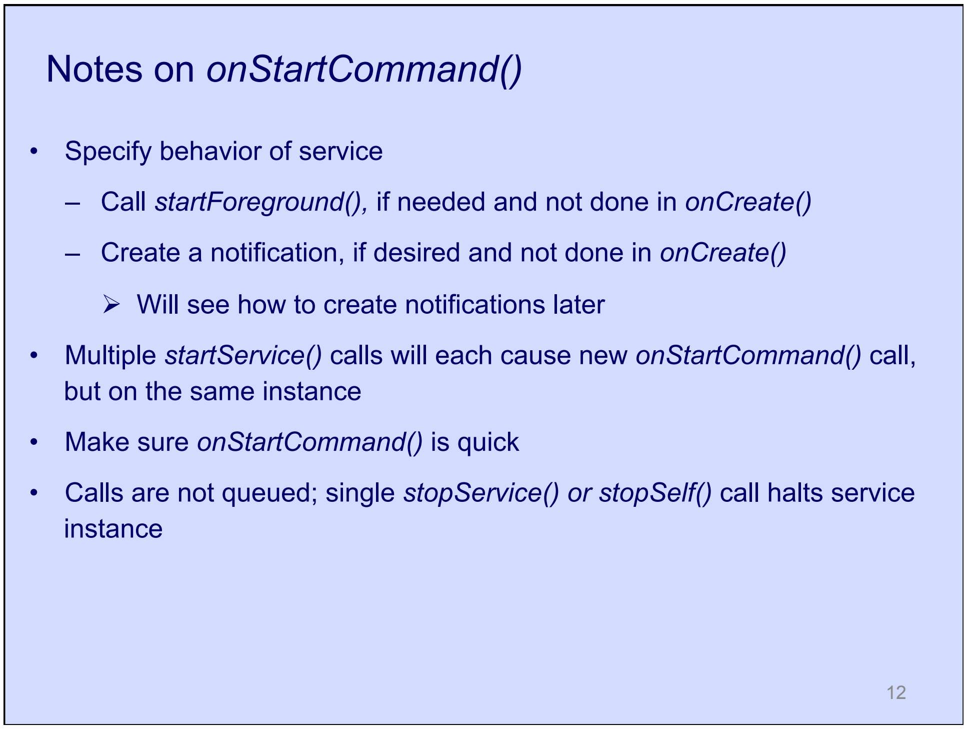
- Return value: **int** code indicating how to continue a service if killed (for benefit of Android kernel)
- Use one of these values when coding *onStartCommand()*:
  - START_STICKY – If service process killed after started, leave in started state and call *onStartCommand()* again (but with **null** intent) when restarted
  - START_NON_STICKY – Don't automatically recreate service if killed (good for services that synchronize with a remove server)
  - START_REDELIVER_INTENT – Similar to START_STICKY but intent will be redelivered
  - https://developer.android.com/reference/android/app/Service.html#onStartCommand



More on *onStartCommand()*

- Parameters:

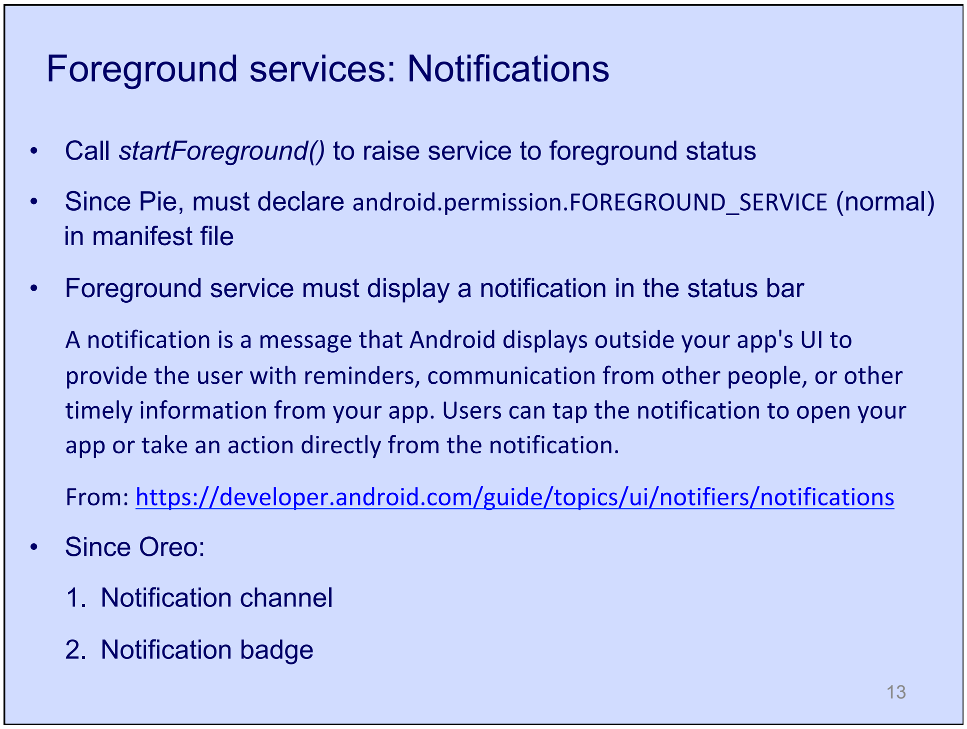
1. *Intent* that was used to start service, could be **null** if service restarted after its process was killed
2. *int flags* – could be 0, START_FLAG_REDELIVERY or START_FLAG_RETRY
3. *int startId* – integer id to be used with *stopSelfResult(int)* to make conditional stop



Notes on *onStartCommand()*

- Specify behavior of service
  - Call *startForeground(),* if needed and not done in *onCreate()*
  - Create a notification, if desired and not done in *onCreate()*
    - ➢ Will see how to create notifications later
- Multiple *startService()* calls will each cause new *onStartCommand()* call, but on the same instance
- Make sure *onStartCommand()* is quick
- Calls are not queued; single *stopService() or stopSelf()* call halts service instance

12



Foreground services: Notifications

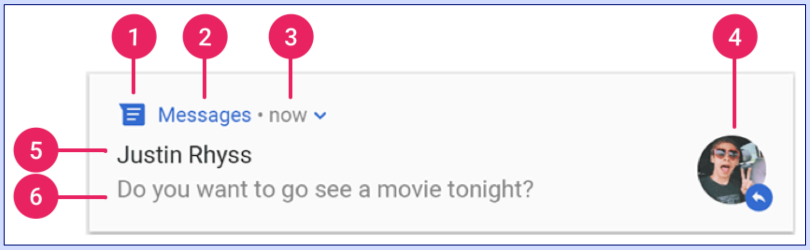
- Call *startForeground()* to raise service to foreground status
- Since Pie, must declare android.permission.FOREGROUND_SERVICE (normal) in manifest file
- Foreground service must display a notification in the status bar

  A notification is a message that Android displays outside your app's UI to provide the user with reminders, communication from other people, or other timely information from your app. Users can tap the notification to open your app or take an action directly from the notification.

  From: https://developer.android.com/guide/topics/ui/notifiers/notifications
- Since Oreo:
  1. Notification channel
  2. Notification badge

13

Notification structure in Android



Messages • now

Justin Rhyss

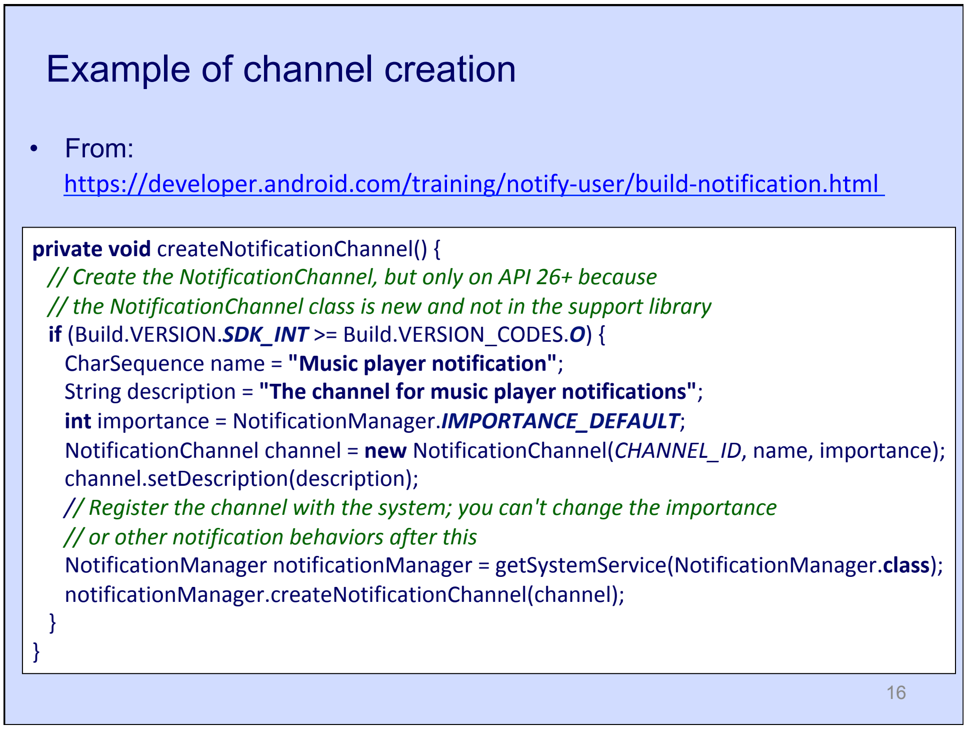
Do you want to go see a movie tonight?

1. Small icon (use *setSmallIcon()*)
2. Program name (OS supplied)
3. Time generated (OS supplied but modifiable with *setWhen()*)
4. Big icon (use *setLargeIcon()*)
5. Title (use *setContentTitle()*)
6. Text (use *setContentText()*)

From: https://developer.android.com/guide/topics/ui/notifiers/notifications

Notification channel

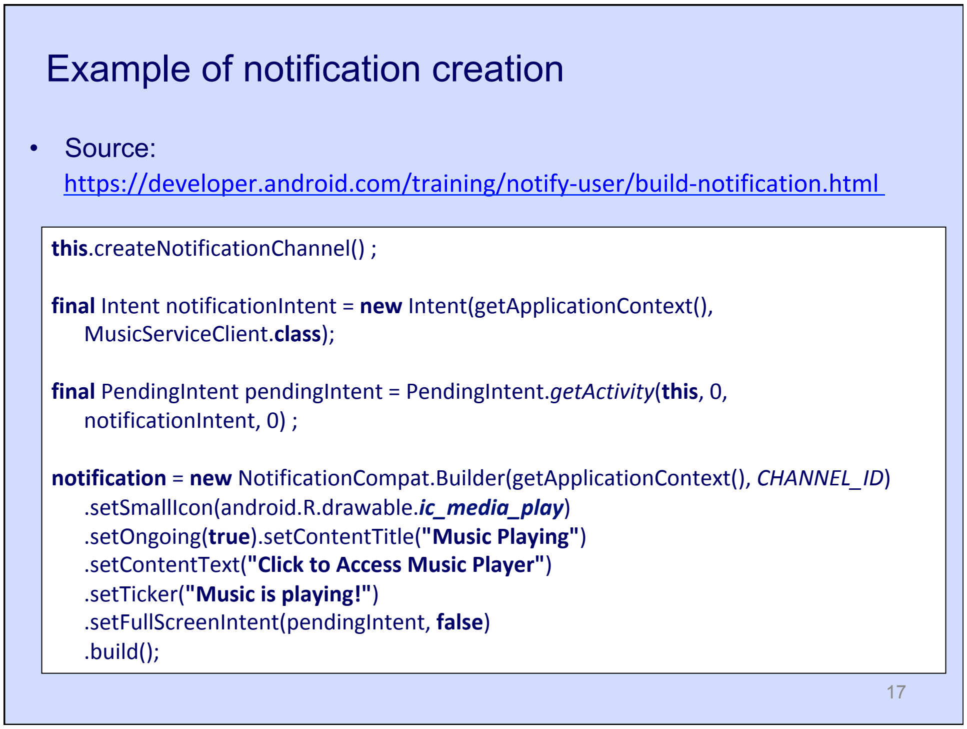
- Channel: Define visual and audible signals associated with notifications in a given channel
  - Vibrate? Make sound? Heads-up display?
- To build a notification:
  - Declare implementation dependency in build.gradle file: implementation "com.android.support:support-compat:28.0.0"
  - Create a channel (e.g., in service's *onCreate()* method)
  - Use **new** NotificationCompat.Builder(<callingAppContext>)
  - Set various parameters in *NotificationCompat.Builder* instance
  - Call *build()* on *NotificationCompat.Builder* instance
- https://developer.android.com/training/notify-user/build-notification



Example of channel creation

- From: https://developer.android.com/training/notify-user/build-notification.html

```
private void createNotificationChannel() {
  // Create the NotificationChannel, but only on API 26+ because
  // the NotificationChannel class is new and not in the support library
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    CharSequence name = "Music player notification";
    String description = "The channel for music player notifications";
    int importance = NotificationManager.IMPORTANCE_DEFAULT;
    NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name, importance);
    channel.setDescription(description);
    // Register the channel with the system; you can't change the importance
    // or other notification behaviors after this
    NotificationManager notificationManager = getSystemService(NotificationManager.class);
    notificationManager.createNotificationChannel(channel);
  }
}
```



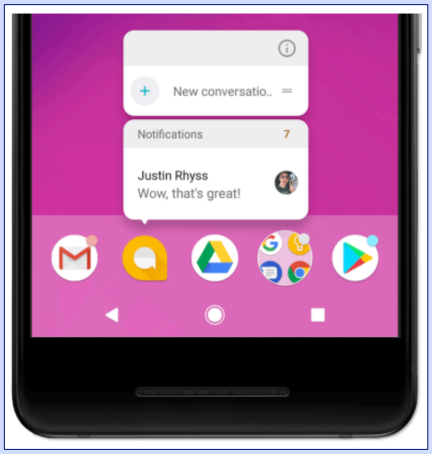
Example of notification creation

- Source: https://developer.android.com/training/notify-user/build-notification.html

```
this.createNotificationChannel() ;

final Intent notificationIntent = new Intent(getApplicationContext(),
    MusicServiceClient.class);

final PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
    notificationIntent, 0) ;

notification = new NotificationCompat.Builder(getApplicationContext(), CHANNEL_ID)
    .setSmallIcon(android.R.drawable.ic_media_play)
    .setOngoing(true).setContentTitle("Music Playing")
    .setContentText("Click to Access Music Player")
    .setTicker("Music is playing!")
    .setFullScreenIntent(pendingIntent, false)
    .build();
```

Notification badges

- Badges show on app icons in launcher window
- Long click on icon shows notification text
- API: *setShowBadge(true)* on notification channel



New conversatio..

Notifications 7

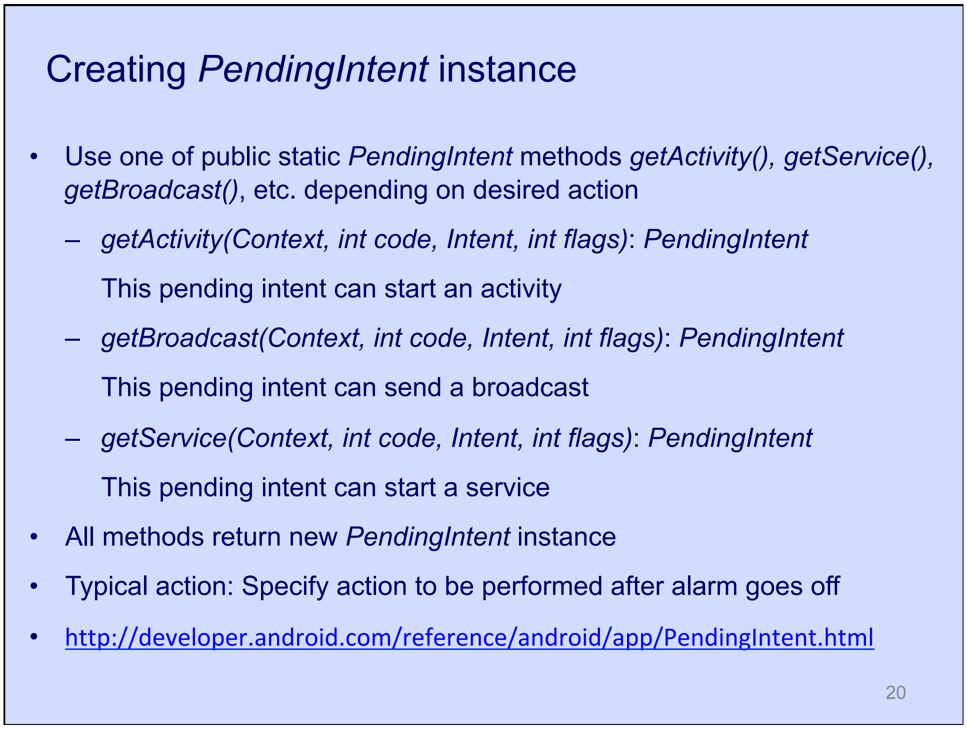
Justin Rhyss

Wow, that's great!

From: https://developer.android.com/training/notify-user/badges

Started services and *PendingIntent*

- Pending intents specify delayed actions
- *PendingIntent* = Intent + Action
- Action is one of
  - Starting an activity
  - Sending a broadcast
  - Starting a service
- Action is triggered some time after *PendingIntent* created
- Instances of *PendingIntent* class allow started service to communicate results back to caller
- http://developer.android.com/reference/android/app/PendingIntent.html



Creating *PendingIntent* instance

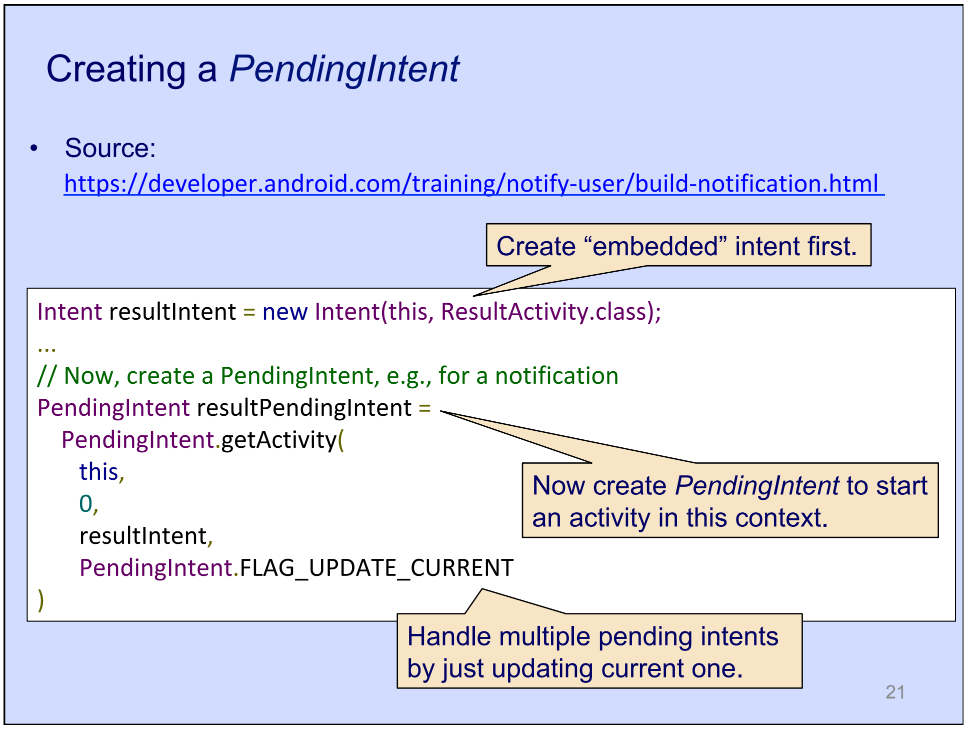
- Use one of public static *PendingIntent* methods *getActivity(), getService(), getBroadcast()*, etc. depending on desired action
  - *getActivity(Context, int code, Intent, int flags)*: *PendingIntent*

    This pending intent can start an activity
  - *getBroadcast(Context, int code, Intent, int flags)*: *PendingIntent*

    This pending intent can send a broadcast
  - *getService(Context, int code, Intent, int flags)*: *PendingIntent*

    This pending intent can start a service
- All methods return new *PendingIntent* instance
- Typical action: Specify action to be performed after alarm goes off
- http://developer.android.com/reference/android/app/PendingIntent.html

20



Creating a *PendingIntent*

- Source: https://developer.android.com/training/notify-user/build-notification.html

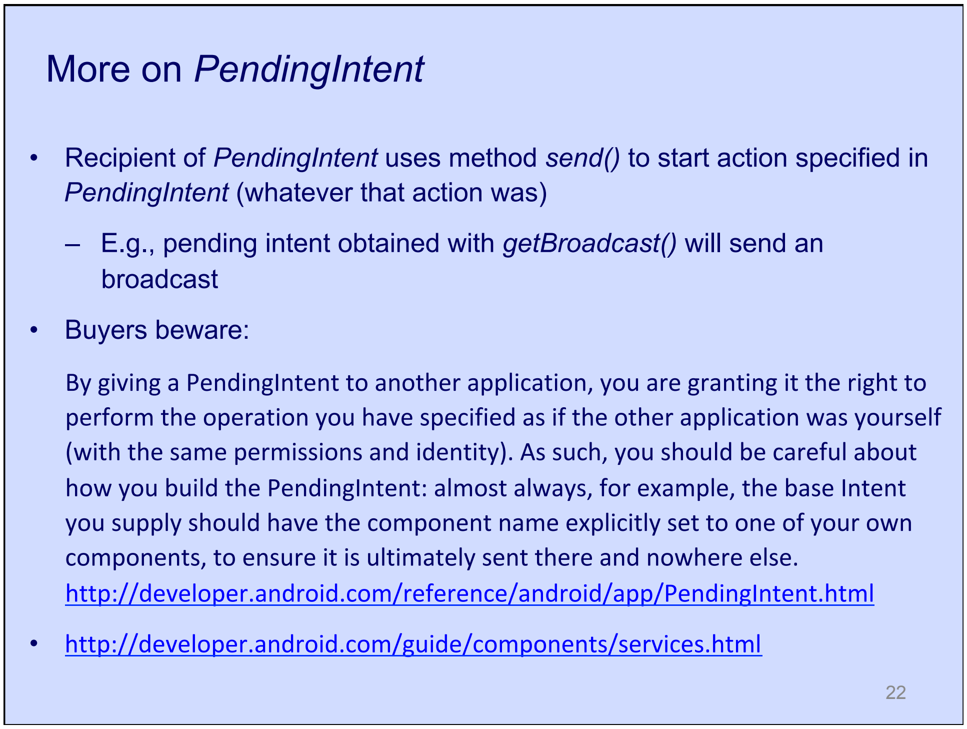
Create "embedded" intent first.

```
Intent resultIntent = new Intent(this, ResultActivity.class);
...
// Now, create a PendingIntent, e.g., for a notification
PendingIntent resultPendingIntent =
   PendingIntent.getActivity(
     this,
     0,
     resultIntent,
     PendingIntent.FLAG_UPDATE_CURRENT
)
```

Now create *PendingIntent* to start an activity in this context.

Handle multiple pending intents by just updating current one.

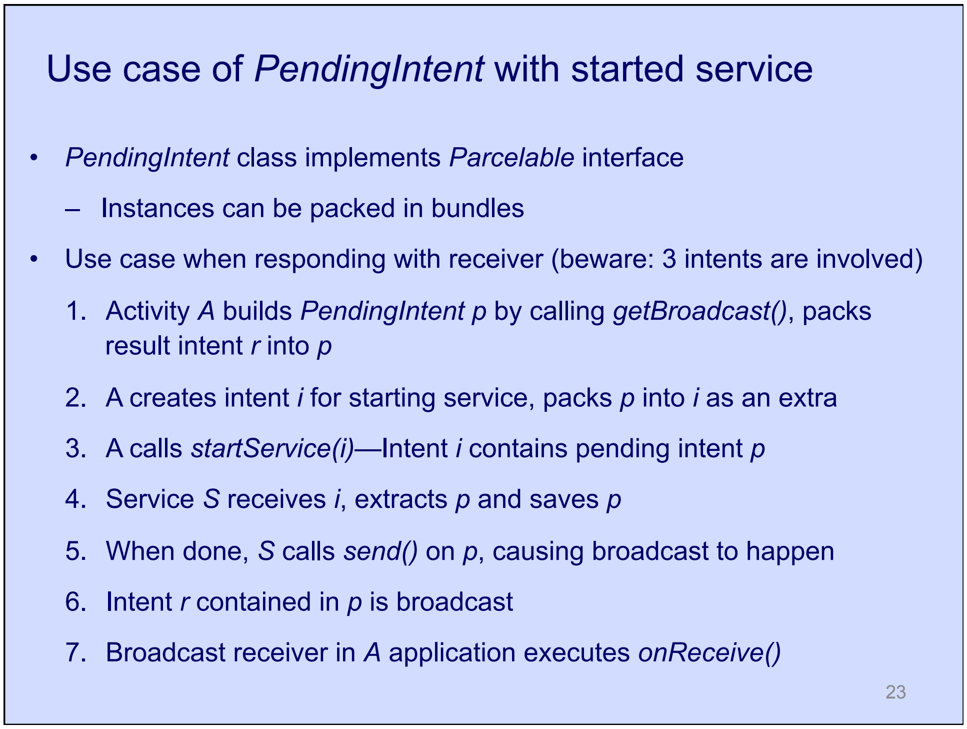
21



More on *PendingIntent*

- Recipient of *PendingIntent* uses method *send()* to start action specified in *PendingIntent* (whatever that action was)
  - E.g., pending intent obtained with *getBroadcast()* will send an broadcast
- Buyers beware:

  By giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity). As such, you should be careful about how you build the PendingIntent: almost always, for example, the base Intent you supply should have the component name explicitly set to one of your own components, to ensure it is ultimately sent there and nowhere else. http://developer.android.com/reference/android/app/PendingIntent.html
- http://developer.android.com/guide/components/services.html



Use case of *PendingIntent* with started service

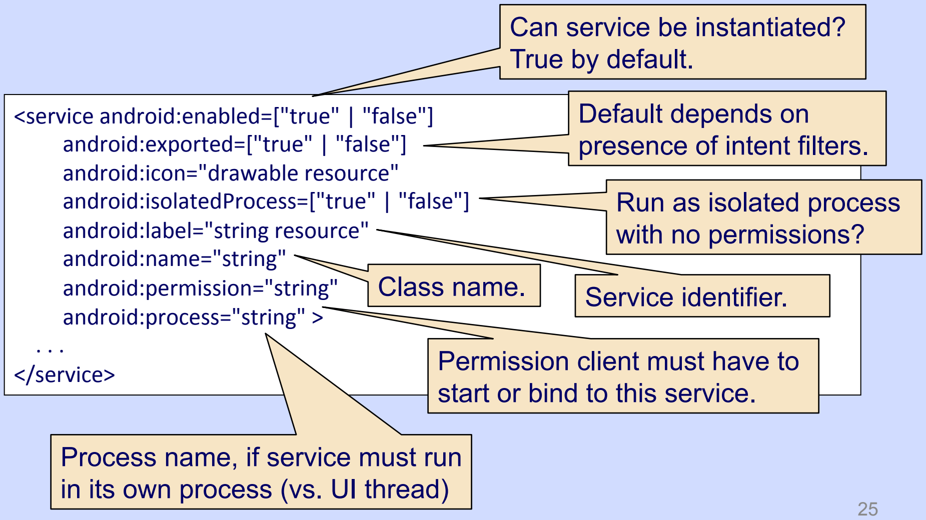
- *PendingIntent* class implements *Parcelable* interface
  - Instances can be packed in bundles
- Use case when responding with receiver (beware: 3 intents are involved)
  1. Activity *A* builds *PendingIntent p* by calling *getBroadcast()*, packs result intent *r* into *p*
  2. A creates intent *i* for starting service, packs *p* into *i* as an extra
  3. A calls *startService(i)*—Intent *i* contains pending intent *p*
  4. Service *S* receives *i*, extracts *p* and saves *p*
  5. When done, *S* calls *send()* on *p*, causing broadcast to happen
  6. Intent *r* contained in *p* is broadcast
  7. Broadcast receiver in *A* application executes *onReceive()*

Service *onDestroy()* callback

- Called by system when service no longer needed, must be destroyed
  - Do not call directly
- Specify clean up actions (e.g., shut down threads, unregister receivers, etc.)
- No params, **void** return value
- Only callback for service termination; there is no *onStop()* equivalent for services

The <service> tag

- Declaring a service in *AndroidManifest.xml* file



```
<service android:enabled=["true" | "false"]
      android:exported=["true" | "false"]
      android:icon="drawable resource"
      android:isolatedProcess=["true" | "false"]
      android:label="string resource"
      android:name="string"
      android:permission="string"
      android:process="string" >
   . . .
</service>
```

Can service be instantiated? True by default.

Default depends on presence of intent filters.

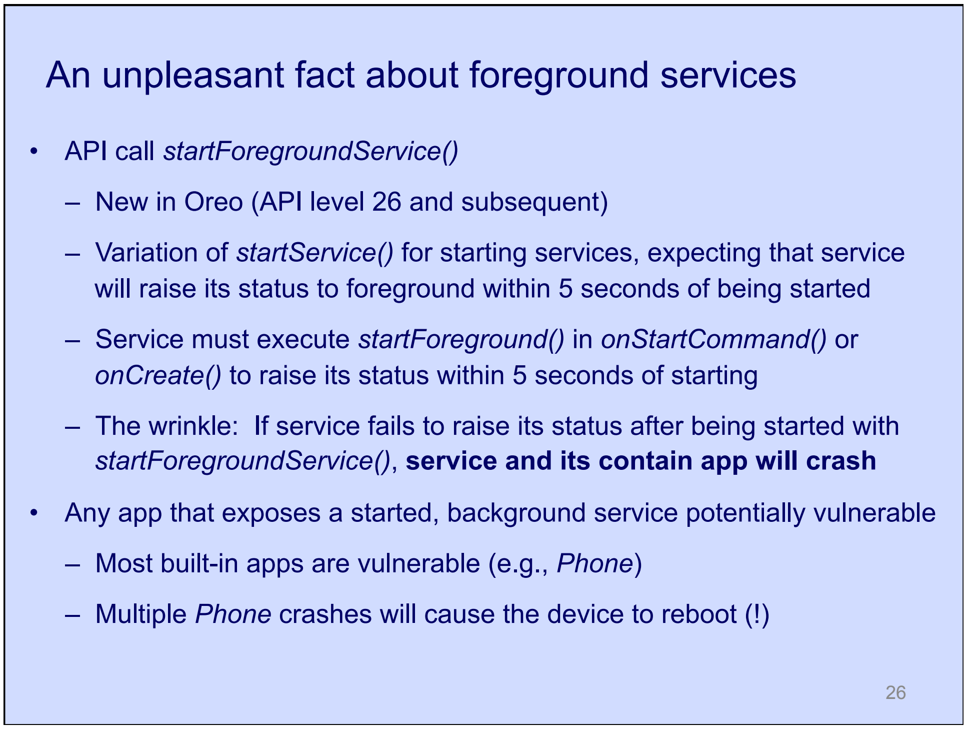
Run as isolated process with no permissions?

Class name.

Service identifier.

Permission client must have to start or bind to this service.

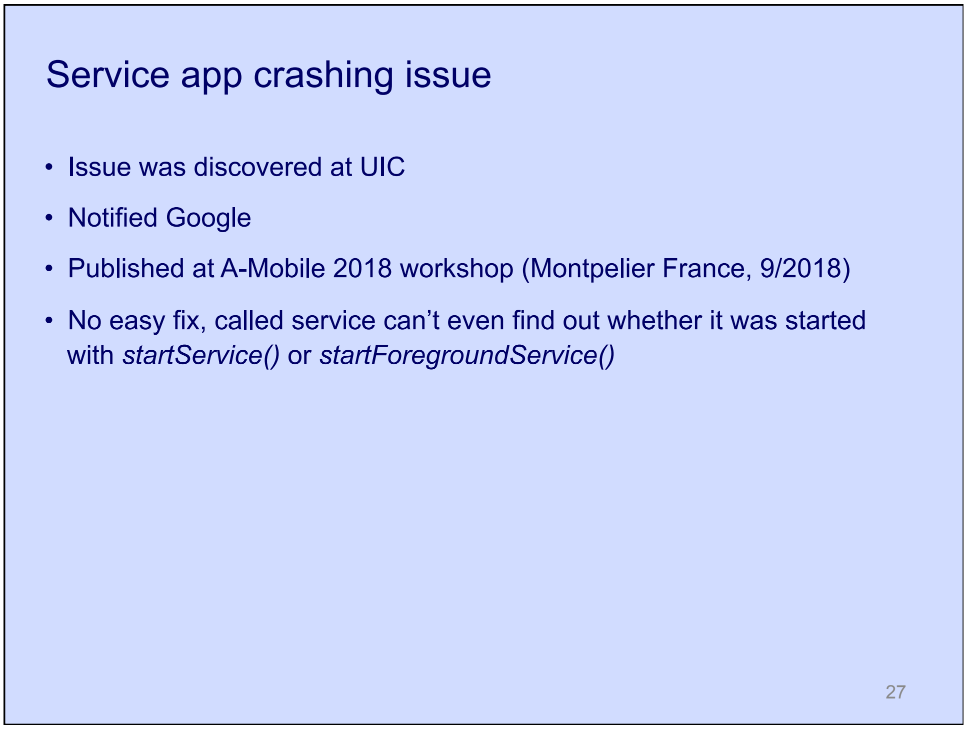
Process name, if service must run in its own process (vs. UI thread)



An unpleasant fact about foreground services

- API call *startForegroundService()*
  - New in Oreo (API level 26 and subsequent)
  - Variation of *startService()* for starting services, expecting that service will raise its status to foreground within 5 seconds of being started
  - Service must execute *startForeground()* in *onStartCommand()* or *onCreate()* to raise its status within 5 seconds of starting
  - The wrinkle: If service fails to raise its status after being started with *startForegroundService()*, **service and its contain app will crash**
- Any app that exposes a started, background service potentially vulnerable
  - Most built-in apps are vulnerable (e.g., *Phone*)
  - Multiple *Phone* crashes will cause the device to reboot (!)

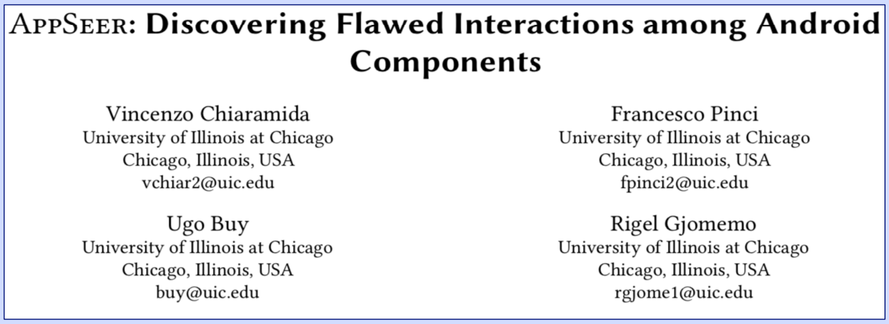
26



Service app crashing issue

- Issue was discovered at UIC
- Notified Google
- Published at A-Mobile 2018 workshop (Montpelier France, 9/2018)
- No easy fix, called service can't even find out whether it was started with *startService()* or *startForegroundService()*

27



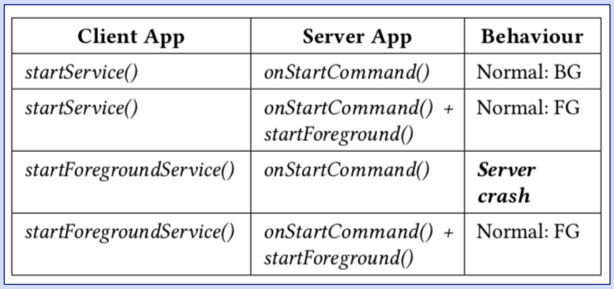
AppSeer: Discovering Flawed Interactions among Android Components

Vincenzo Chiaramida
University of Illinois at Chicago
Chicago, Illinois, USA
vchiar2@uic.edu

Francesco Pinci
University of Illinois at Chicago
Chicago, Illinois, USA
fpinci2@uic.edu

Ugo Buy
University of Illinois at Chicago
Chicago, Illinois, USA
buy@uic.edu

Rigel Gjomemo
University of Illinois at Chicago
Chicago, Illinois, USA
rgjome1@uic.edu



Client App	Server App	Behaviour
startService()	*onStartCommand()*	Normal: BG
startService()	*onStartCommand()* + *startForeground()*	Normal: FG
startForegroundService()	*onStartCommand()*	***Server crash***
startForegroundService()	*onStartCommand()* + *startForeground()*	Normal: FG

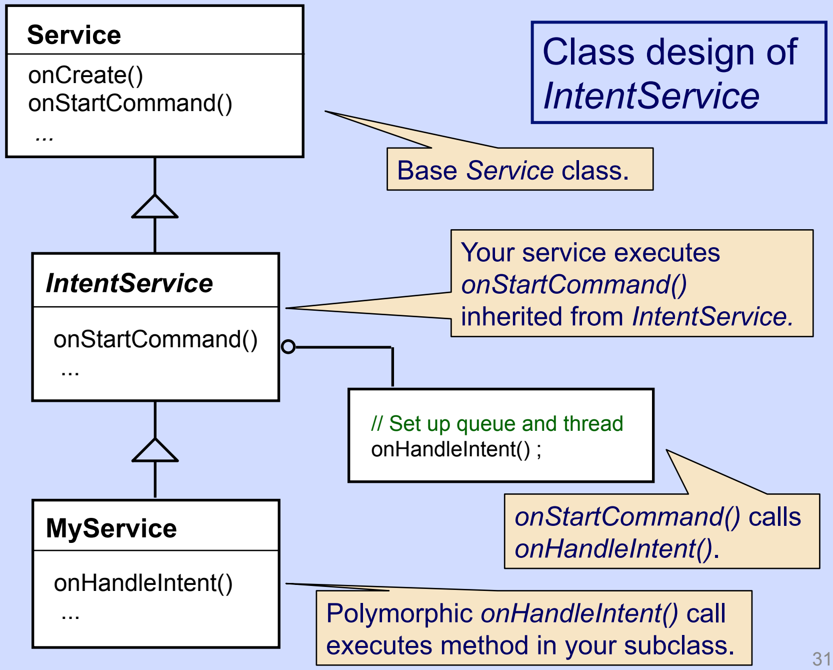
Services vs. threads

- Normally, service runs in UI thread
  - Dangerous if the service does long computations
- Abstract class *IntentService* uses background thread to run service
  - *IntentService* = A *Service* with a worker thread and a job queue
  - Clients call *startService()* or *startForegroundService()*, passing intents
  - Intents are queued in *HandlerService*’s job queue
  - First job in job passed to new method *onHandleIntent()*
  - Jobs are processed sequentially
  - Service lives for as long as there are queued jobs

Java code of *IntentService*

- Abstract class *IntentService* is a subclass of *Service*
- *IntentService.onStartCommand()* creates worker thread and job queue,
- *IntentService.onStartCommand()* also calls new method *onHandleIntent()*
- Define your service as a subclass of *IntentService*
  - **class** MyService **extends** IntentService ...
  - Define method *onHandleIntent(Intent)*, which will run in worker thread

Class design of *IntentService*



Service
onCreate()
onStartCommand()
...

IntentService
onStartCommand()
...

MyService
onHandleIntent()
...

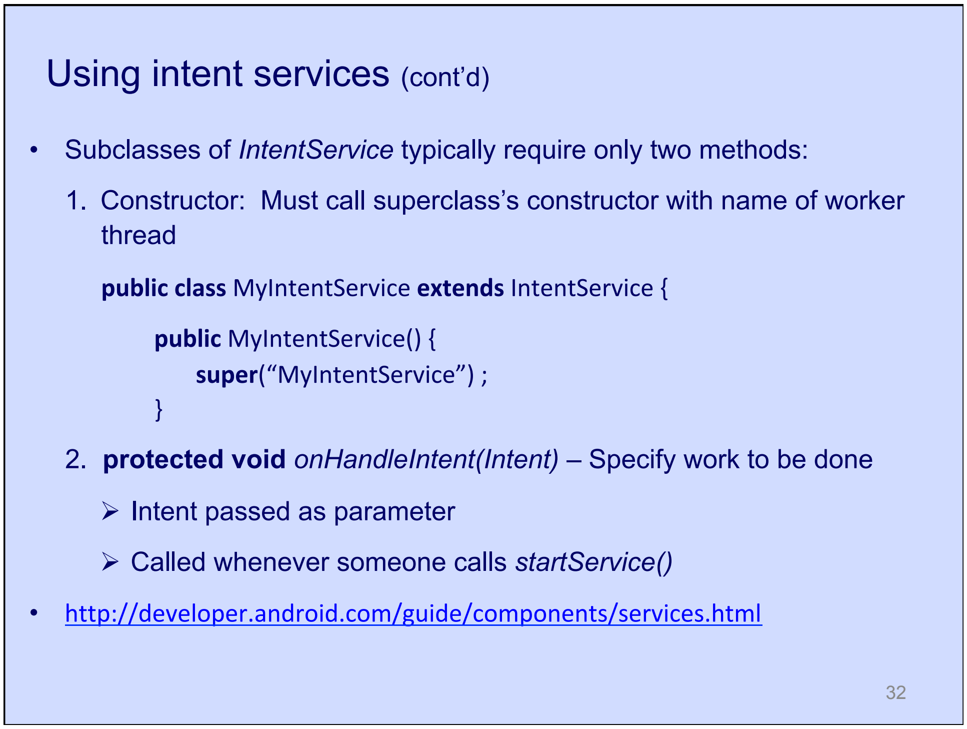
Base *Service* class.

Your service executes *onStartCommand()* inherited from *IntentService.*

```
// Set up queue and thread
onHandleIntent() ;
```

onStartCommand() calls *onHandleIntent()*.

Polymorphic *onHandleIntent()* call executes method in your subclass.



Using intent services (cont'd)

- Subclasses of *IntentService* typically require only two methods:

1. Constructor: Must call superclass's constructor with name of worker thread

```
public class MyIntentService extends IntentService {

        public MyIntentService() {
              super("MyIntentService") ;
        }
```

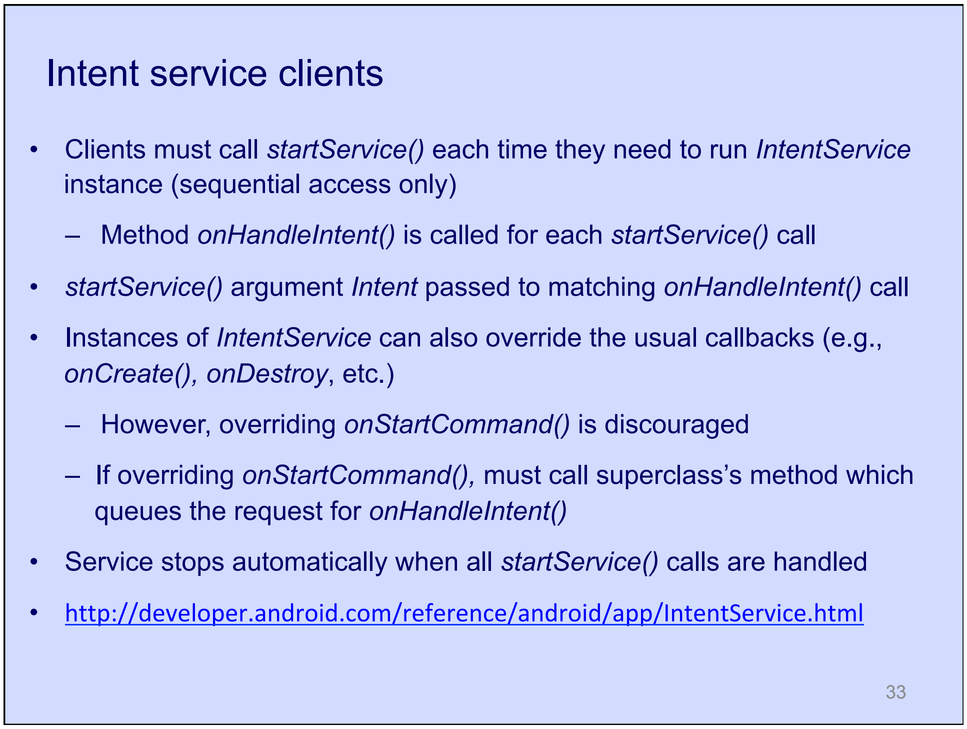
2. **protected void** *onHandleIntent(Intent)* – Specify work to be done

   ➢ Intent passed as parameter

   ➢ Called whenever someone calls *startService()*

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

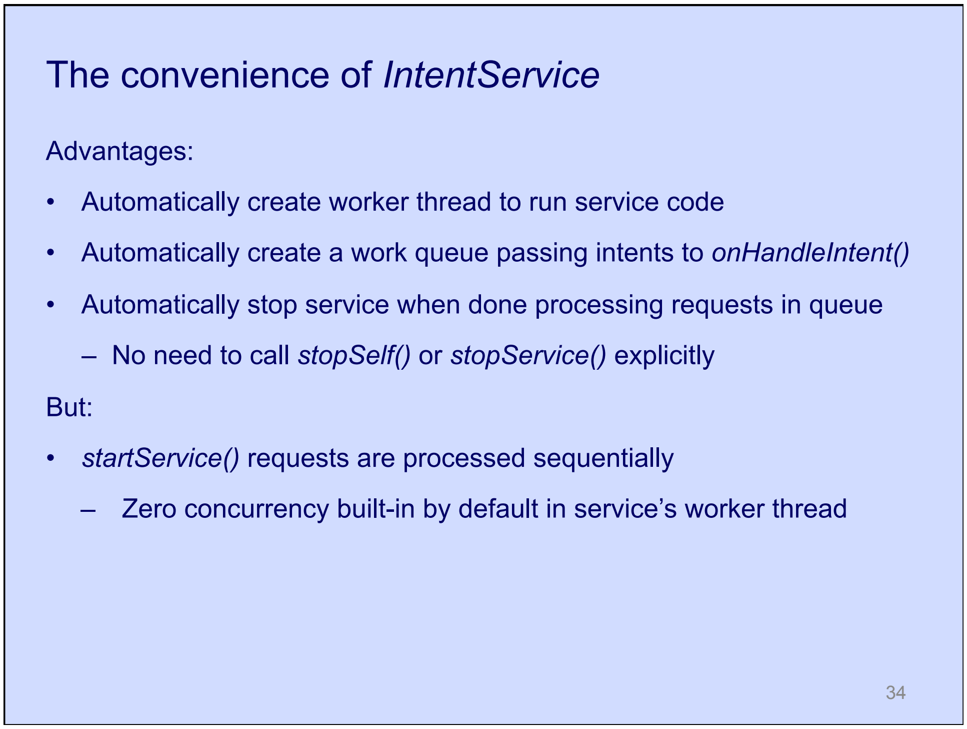
32



Intent service clients

- Clients must call *startService()* each time they need to run *IntentService* instance (sequential access only)
  – Method *onHandleIntent()* is called for each *startService()* call
- *startService()* argument *Intent* passed to matching *onHandleIntent()* call
- Instances of *IntentService* can also override the usual callbacks (e.g., *onCreate(), onDestroy*, etc.)
  – However, overriding *onStartCommand()* is discouraged
  – If overriding *onStartCommand(),* must call superclass's method which queues the request for *onHandleIntent()*
- Service stops automatically when all *startService()* calls are handled
- http://developer.android.com/reference/android/app/IntentService.html

33



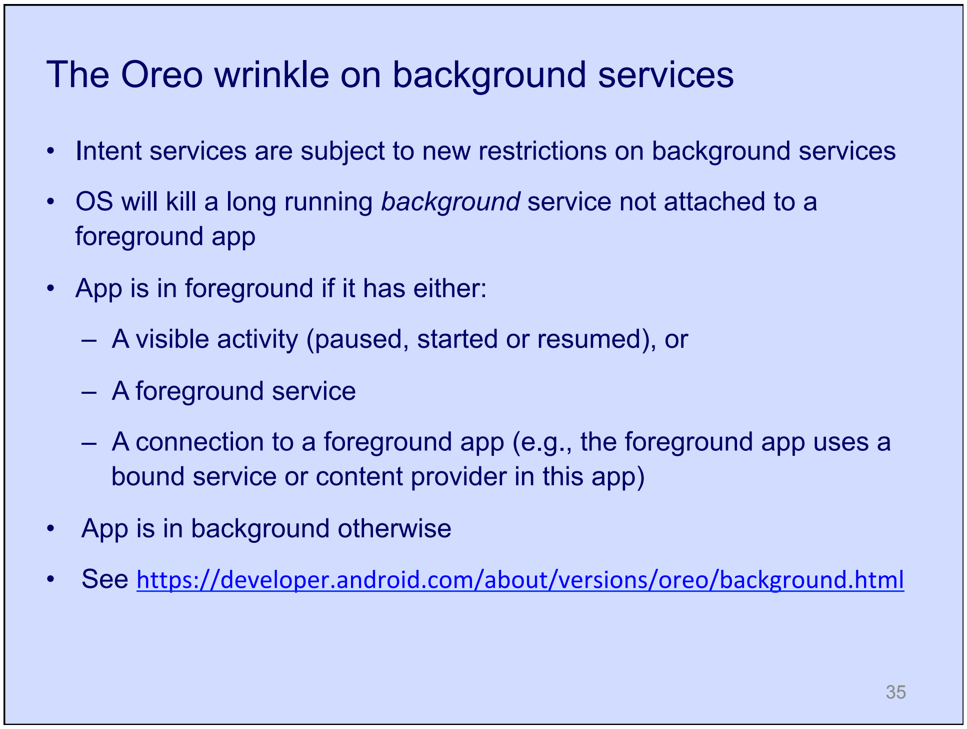
The convenience of *IntentService*

Advantages:

- Automatically create worker thread to run service code
- Automatically create a work queue passing intents to *onHandleIntent()*
- Automatically stop service when done processing requests in queue
  - No need to call *stopSelf()* or *stopService()* explicitly

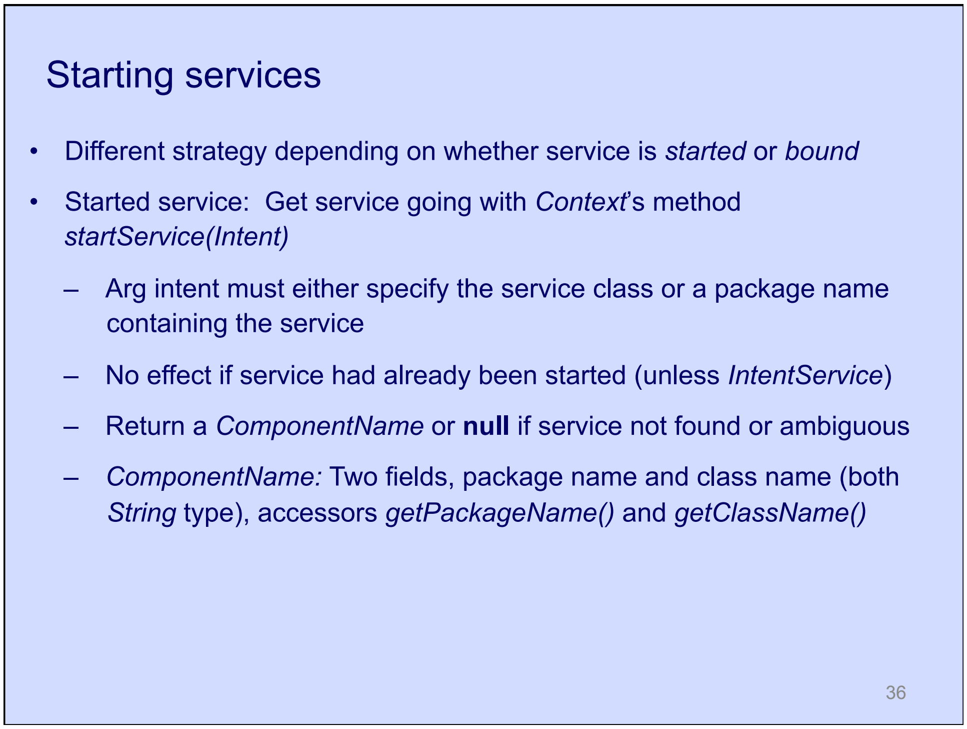
But:

- *startService()* requests are processed sequentially
  - Zero concurrency built-in by default in service's worker thread



The Oreo wrinkle on background services

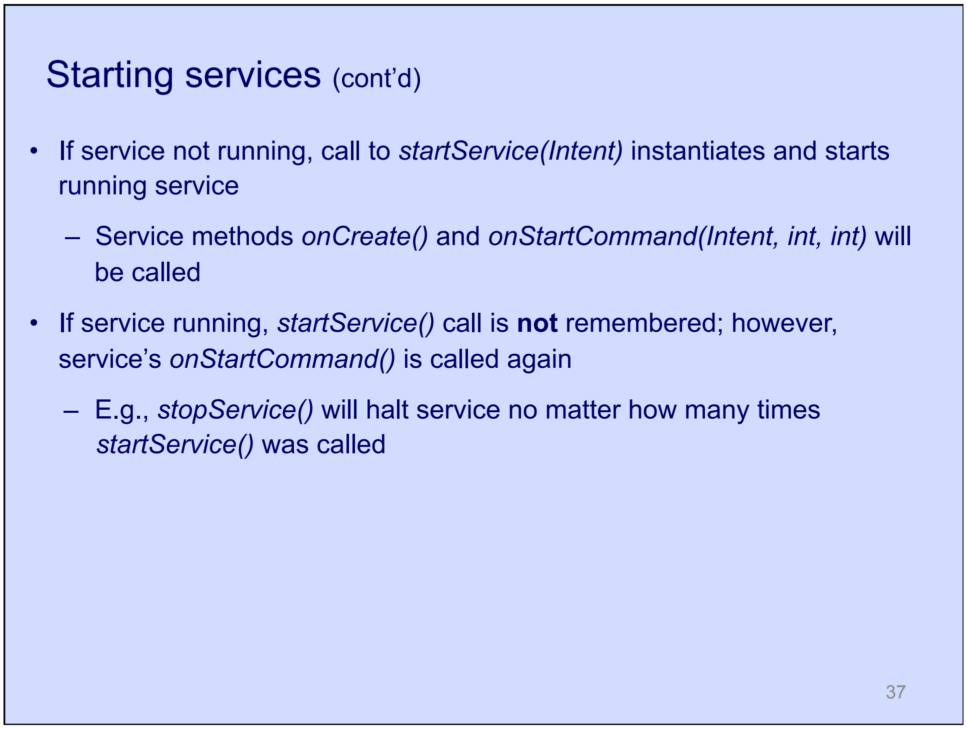
- Intent services are subject to new restrictions on background services
- OS will kill a long running *background* service not attached to a foreground app
- App is in foreground if it has either:
  - A visible activity (paused, started or resumed), or
  - A foreground service
  - A connection to a foreground app (e.g., the foreground app uses a bound service or content provider in this app)
- App is in background otherwise
- See https://developer.android.com/about/versions/oreo/background.html



Starting services

- Different strategy depending on whether service is *started* or *bound*
- Started service: Get service going with *Context*'s method *startService(Intent)*
  - Arg intent must either specify the service class or a package name containing the service
  - No effect if service had already been started (unless *IntentService*)
  - Return a *ComponentName* or **null** if service not found or ambiguous
  - *ComponentName:* Two fields, package name and class name (both *String* type), accessors *getPackageName()* and *getClassName()*

36



Starting services (cont'd)

- If service not running, call to *startService(Intent)* instantiates and starts running service
  - Service methods *onCreate()* and *onStartCommand(Intent, int, int)* will be called
- If service running, *startService()* call is **not** remembered; however, service's *onStartCommand()* is called again
  - E.g., *stopService()* will halt service no matter how many times *startService()* was called

37

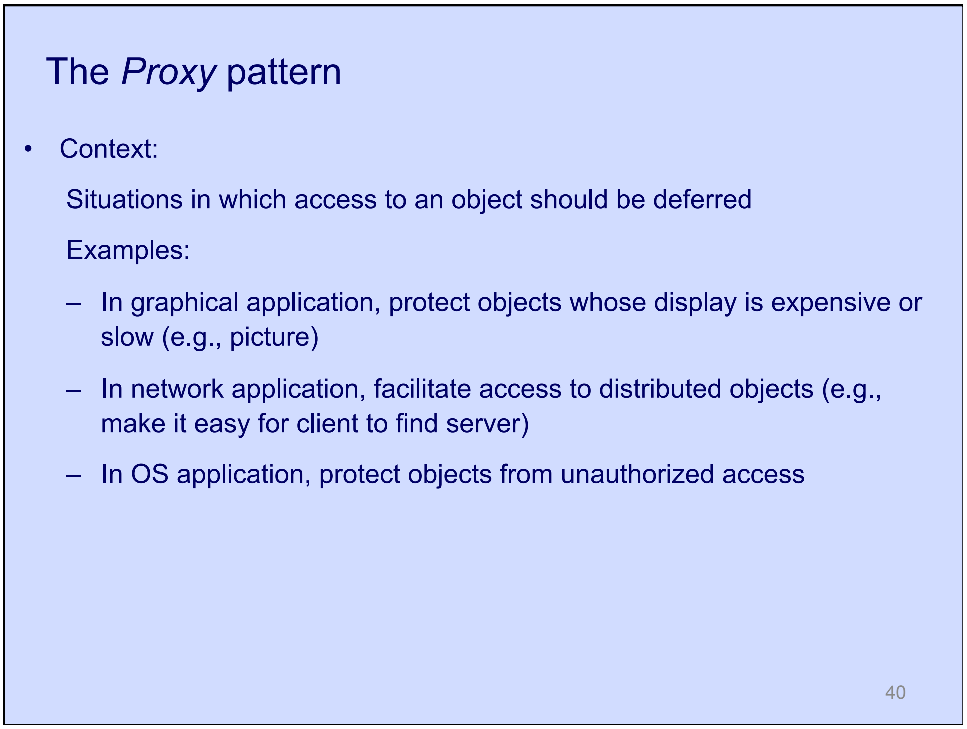
Bound services

- Goal: Create a persistent connection between client and service
- Client side: Request to connect to a service by calling *bindService()*
  - Client could be activity or service
- Client eventually receives a proxy object that implements *IBinder* interface
- Henceforth, client uses service API by making local calls on proxy object
- **Proxy makes remote calls to service's API look like local client calls**
- Android transfers calls to remote service and return values back to client
- If service declares permissions, client must first acquire permissions
- http://developer.android.com/guide/components/bound-services.html

Bound services: The client side

- Based on *Proxy* pattern by the Go4
- Outline – We'll see:

1. *Proxy* pattern
2. Protocol for binding to service, using service and unbinding from service



The *Proxy* pattern

- Context:

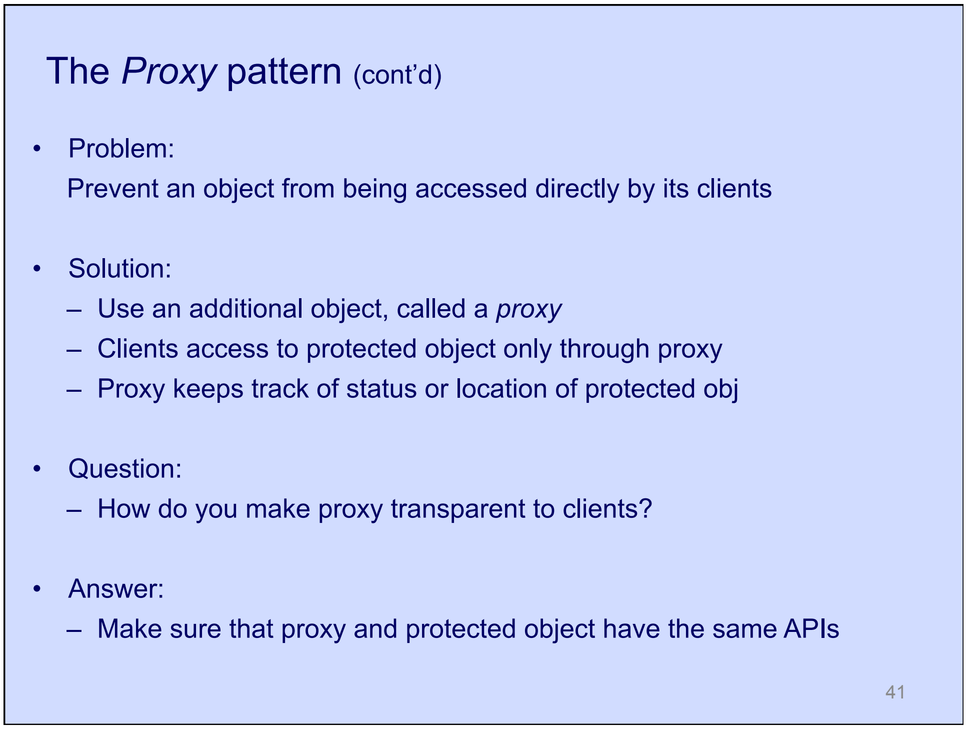
  Situations in which access to an object should be deferred

  Examples:

  – In graphical application, protect objects whose display is expensive or slow (e.g., picture)

  – In network application, facilitate access to distributed objects (e.g., make it easy for client to find server)

  – In OS application, protect objects from unauthorized access



The *Proxy* pattern (cont’d)

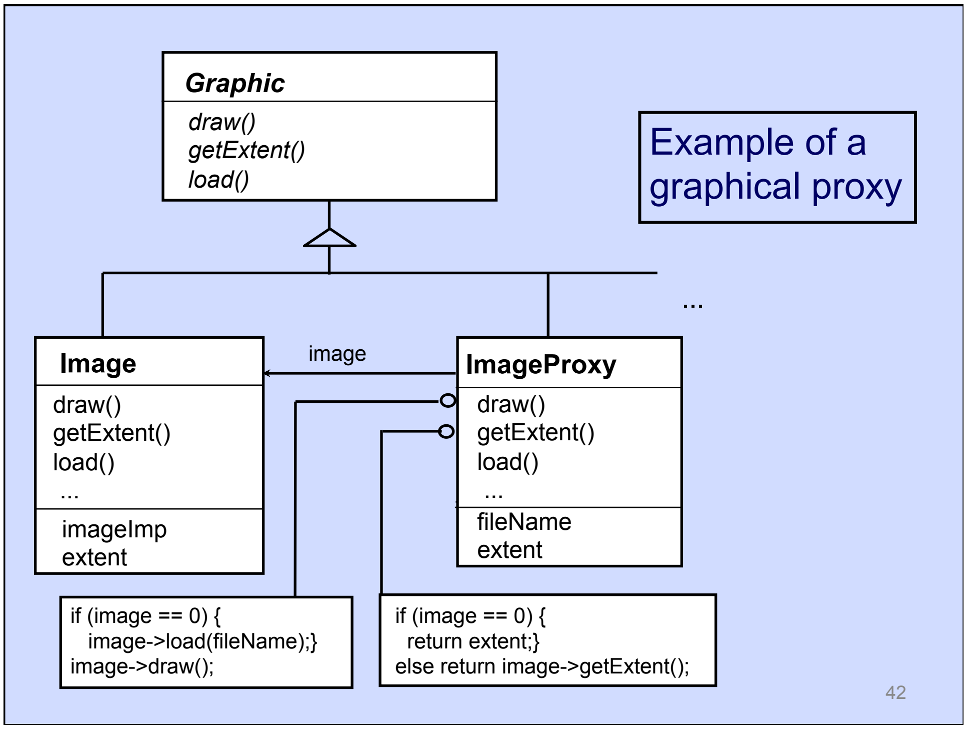
- Problem:

  Prevent an object from being accessed directly by its clients

- Solution:
  – Use an additional object, called a *proxy*
  – Clients access to protected object only through proxy
  – Proxy keeps track of status or location of protected obj

- Question:
  – How do you make proxy transparent to clients?

- Answer:
  – Make sure that proxy and protected object have the same APIs



Example of a graphical proxy

Graphic

draw()
getExtent()
load()

Image

draw()
getExtent()
load()
...

imageImp
extent

image

ImageProxy

draw()
getExtent()
load()
...

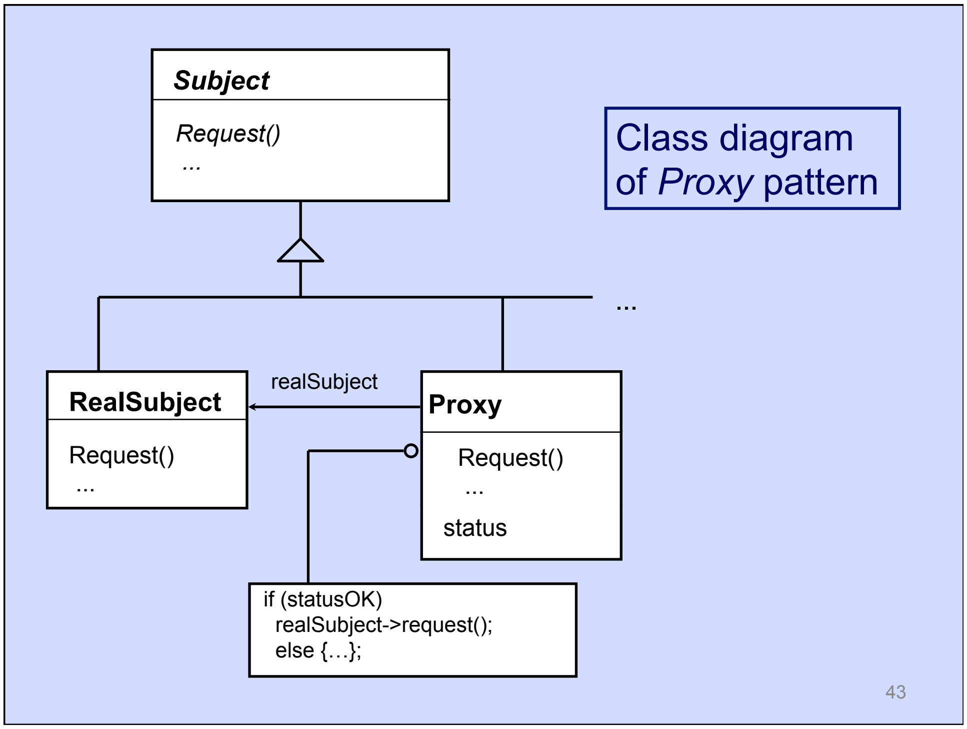
fileName
extent

...

```
if (image == 0) {
  image->load(fileName);}
image->draw();
```

```
if (image == 0) {
  return extent;}
else return image->getExtent();
```

42



Class diagram of *Proxy* pattern

Subject

Request()
...

RealSubject

Request()
...

realSubject

Proxy

Request()
...

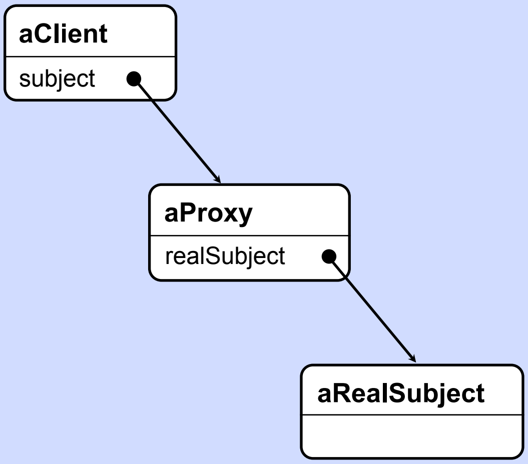
status

...

```
if (statusOK)
  realSubject->request();
  else {…};
```

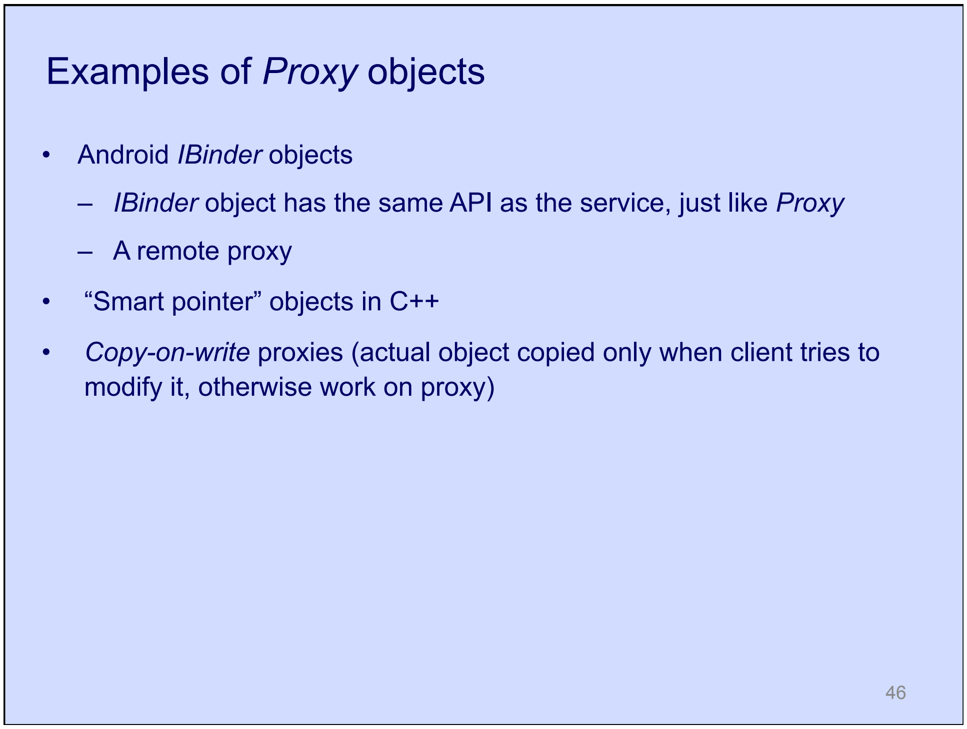
43

Object diagram of *Proxy* pattern



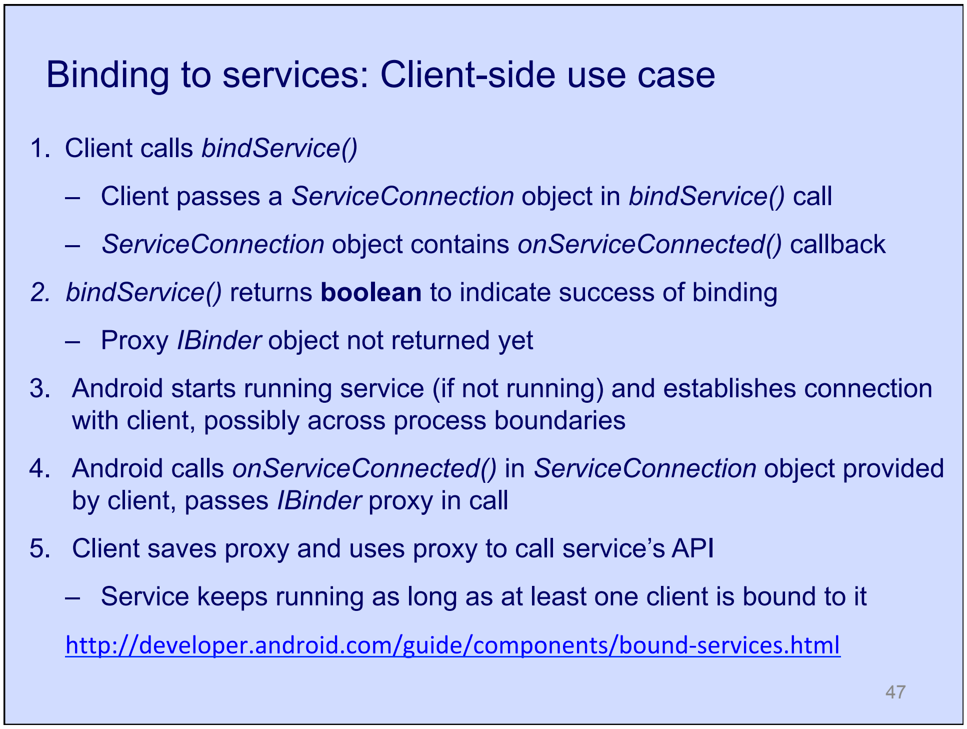
Responsibilities of *Proxy* object

- Maintain reference to object accessed
- Provide appropriate interface to clients (by delegating to protected object when necessary)
- Control access to the protected object
- Kinds of proxies (and added resp.):
  - — remote (hide location of real object)
  - — virtual (provide cache information)
  - — protection (check access rights)



Examples of *Proxy* objects

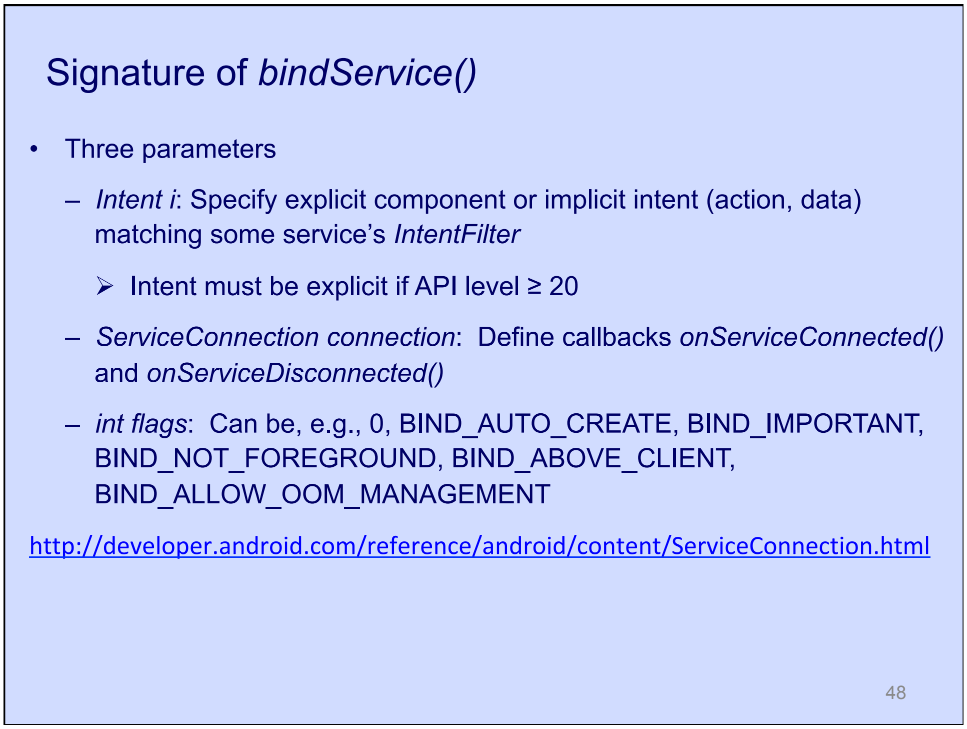
- Android *IBinder* objects
  - *IBinder* object has the same API as the service, just like *Proxy*
  - A remote proxy
- "Smart pointer" objects in C++
- *Copy-on-write* proxies (actual object copied only when client tries to modify it, otherwise work on proxy)



Binding to services: Client-side use case

1. Client calls *bindService()*
   - Client passes a *ServiceConnection* object in *bindService()* call
   - *ServiceConnection* object contains *onServiceConnected()* callback
2. *bindService()* returns **boolean** to indicate success of binding
   - Proxy *IBinder* object not returned yet
3. Android starts running service (if not running) and establishes connection with client, possibly across process boundaries
4. Android calls *onServiceConnected()* in *ServiceConnection* object provided by client, passes *IBinder* proxy in call
5. Client saves proxy and uses proxy to call service's API
   - Service keeps running as long as at least one client is bound to it

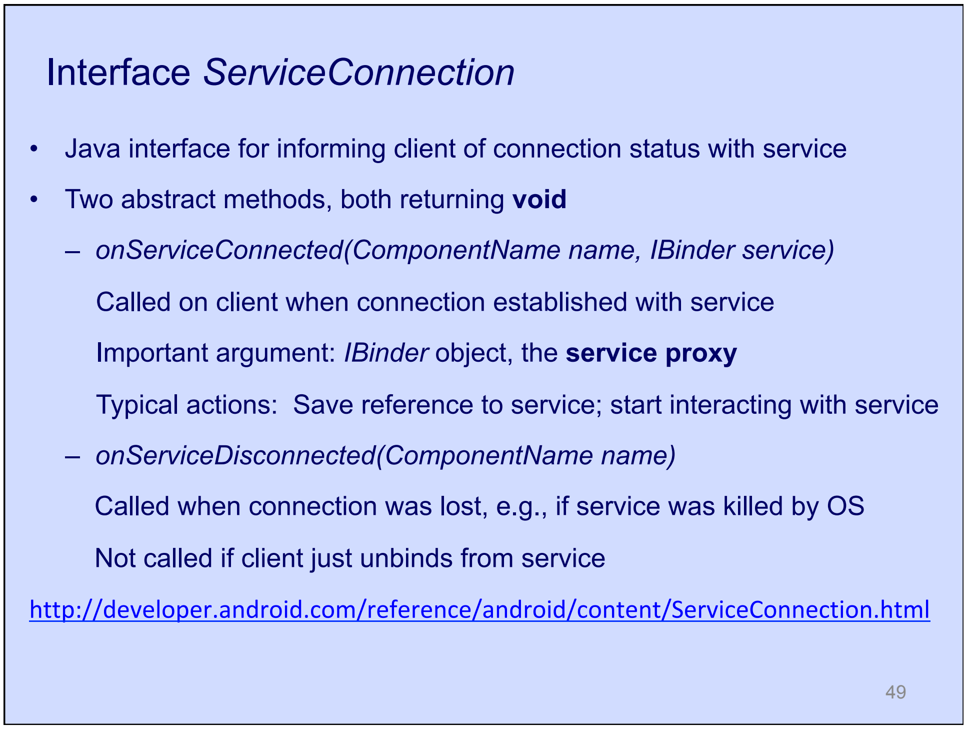
http://developer.android.com/guide/components/bound-services.html



Signature of *bindService()*

- Three parameters
  - *Intent i*: Specify explicit component or implicit intent (action, data) matching some service's *IntentFilter*
    - ➢ Intent must be explicit if API level ≥ 20
  - *ServiceConnection connection*: Define callbacks *onServiceConnected()* and *onServiceDisconnected()*
  - *int flags*: Can be, e.g., 0, BIND_AUTO_CREATE, BIND_IMPORTANT, BIND_NOT_FOREGROUND, BIND_ABOVE_CLIENT, BIND_ALLOW_OOM_MANAGEMENT

http://developer.android.com/reference/android/content/ServiceConnection.html



Interface *ServiceConnection*

- Java interface for informing client of connection status with service
- Two abstract methods, both returning **void**
  - *onServiceConnected(ComponentName name, IBinder service)*

    Called on client when connection established with service

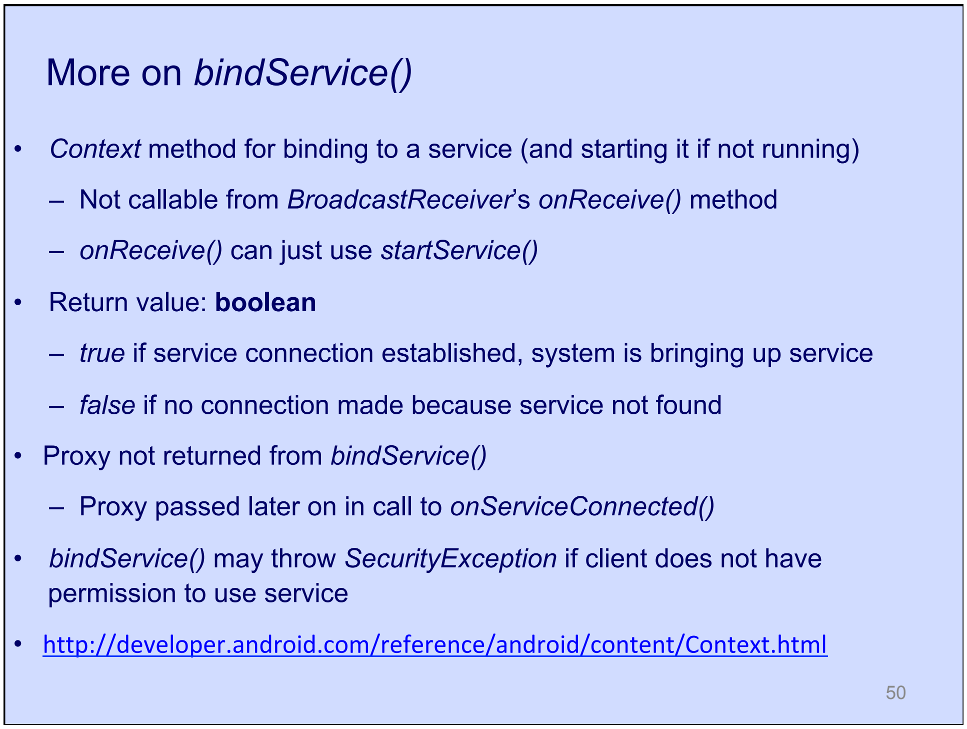
    Important argument: *IBinder* object, the **service proxy**

    Typical actions: Save reference to service; start interacting with service
  - *onServiceDisconnected(ComponentName name)*

    Called when connection was lost, e.g., if service was killed by OS

    Not called if client just unbinds from service

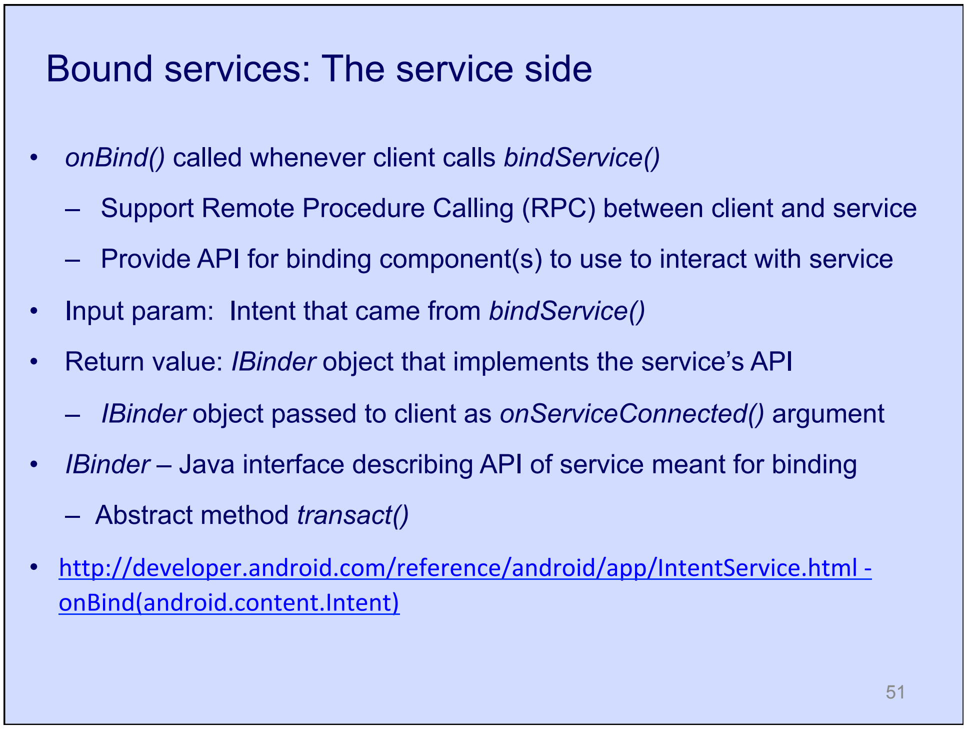
http://developer.android.com/reference/android/content/ServiceConnection.html



More on *bindService()*

- *Context* method for binding to a service (and starting it if not running)
  - Not callable from *BroadcastReceiver*'s *onReceive()* method
  - *onReceive()* can just use *startService()*
- Return value: **boolean**
  - *true* if service connection established, system is bringing up service
  - *false* if no connection made because service not found
- Proxy not returned from *bindService()*
  - Proxy passed later on in call to *onServiceConnected()*
- *bindService()* may throw *SecurityException* if client does not have permission to use service
- http://developer.android.com/reference/android/content/Context.html

50



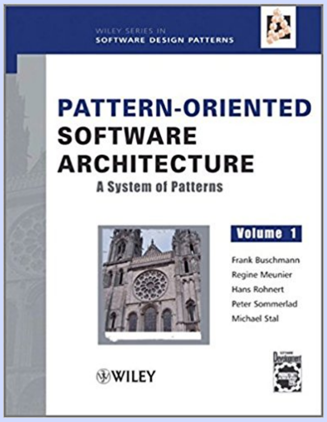
Bound services: The service side

- *onBind()* called whenever client calls *bindService()*
  - Support Remote Procedure Calling (RPC) between client and service
  - Provide API for binding component(s) to use to interact with service
- Input param: Intent that came from *bindService()*
- Return value: *IBinder* object that implements the service's API
  - *IBinder* object passed to client as *onServiceConnected()* argument
- *IBinder* – Java interface describing API of service meant for binding
  - Abstract method *transact()*
- http://developer.android.com/reference/android/app/IntentService.html - onBind(android.content.Intent)

51

Broker design pattern (Go5)

- Interactions between between client and bound service follow *Broker* pattern
- *Broker* is part of Gang-of-five (Go5) system
  - Buschmann, Meunier, Rohnert, Sommerlad, and Stal
- Go5 system: Pattern for software architecture (e.g., *Layers*, *MVC*, *Pipes and Filters, Blackboard*)



WILEY SERIES IN SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A System of Patterns

Volume 1

Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

WILEY

Broker pattern (Go5)

Name

Broker

Applicability

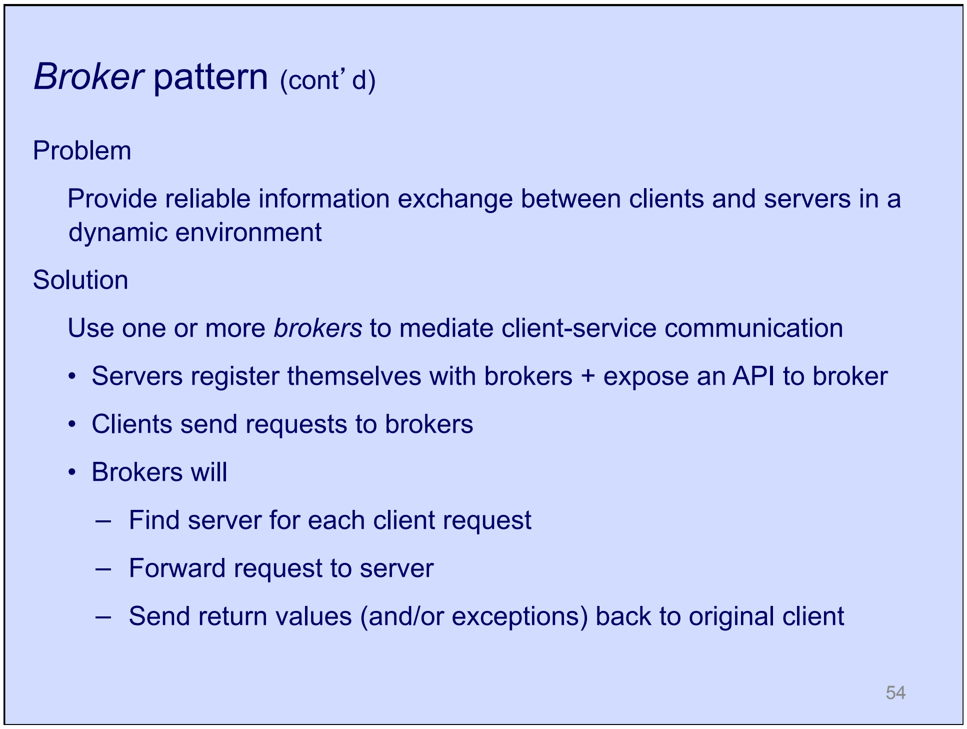
Group of *server* objects supporting *clients* in distributed environment

Example:

- The WWW: Clients (browsers) access server pages via unique URLs

Requirements

- Clients and servers are added and removed dynamically
- Insulate clients from protocol, implementation and location of servers
- Clients should access remote services transparently, regardless of service location, implementation language, and protocol



Broker pattern (cont’ d)

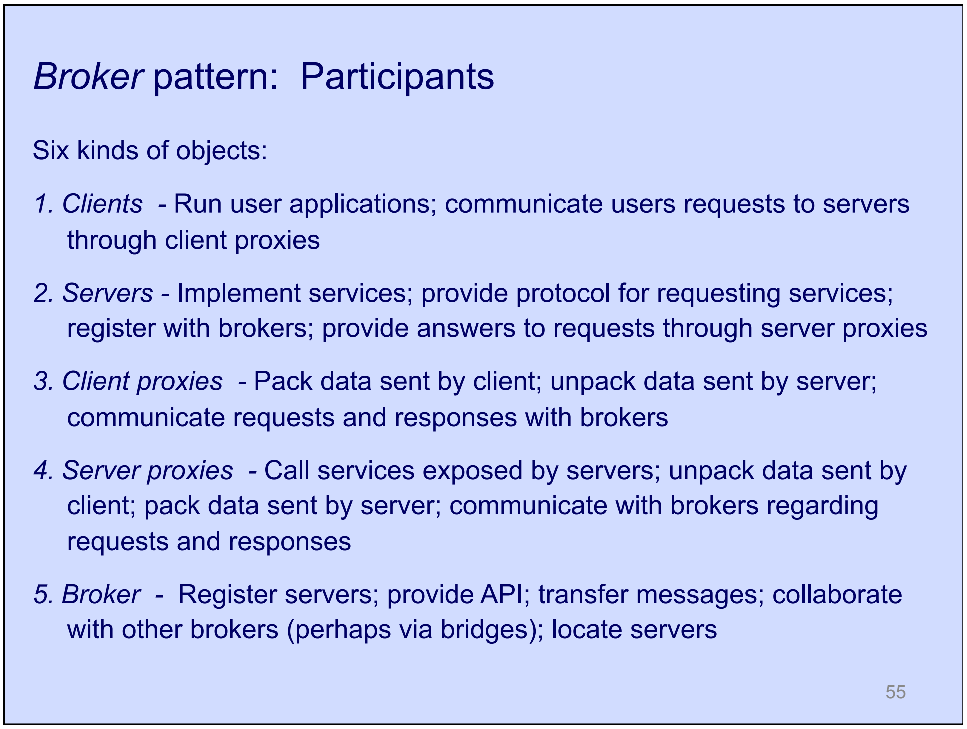
Problem

Provide reliable information exchange between clients and servers in a dynamic environment

Solution

Use one or more *brokers* to mediate client-service communication

- Servers register themselves with brokers + expose an API to broker
- Clients send requests to brokers
- Brokers will
  - Find server for each client request
  - Forward request to server
  - Send return values (and/or exceptions) back to original client



Broker pattern: Participants

Six kinds of objects:

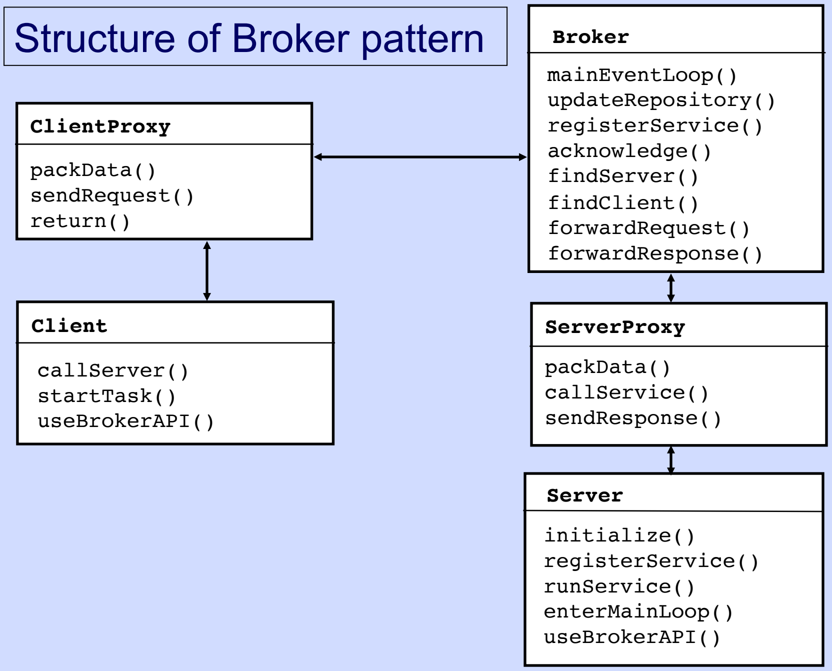
1. *Clients* - Run user applications; communicate users requests to servers through client proxies
2. *Servers* - Implement services; provide protocol for requesting services; register with brokers; provide answers to requests through server proxies
3. *Client proxies* - Pack data sent by client; unpack data sent by server; communicate requests and responses with brokers
4. *Server proxies* - Call services exposed by servers; unpack data sent by client; pack data sent by server; communicate with brokers regarding requests and responses
5. *Broker* - Register servers; provide API; transfer messages; collaborate with other brokers (perhaps via bridges); locate servers

Broker pattern: Participants (cont' d)

Additional kind of objects:

6. Bridges - Mediate between heterogeneous brokers

Structure of Broker pattern



ClientProxy

packData()
sendRequest()
return()

Client

callServer()
startTask()
useBrokerAPI()

Broker

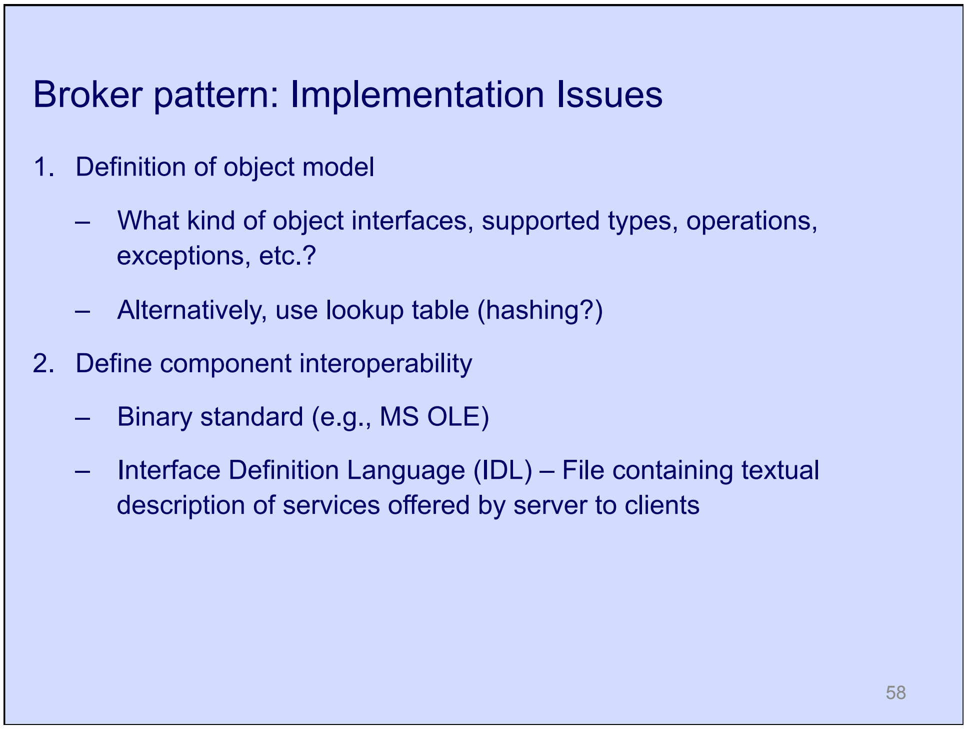
mainEventLoop()
updateRepository()
registerService()
acknowledge()
findServer()
findClient()
forwardRequest()
forwardResponse()

ServerProxy

packData()
callService()
sendResponse()

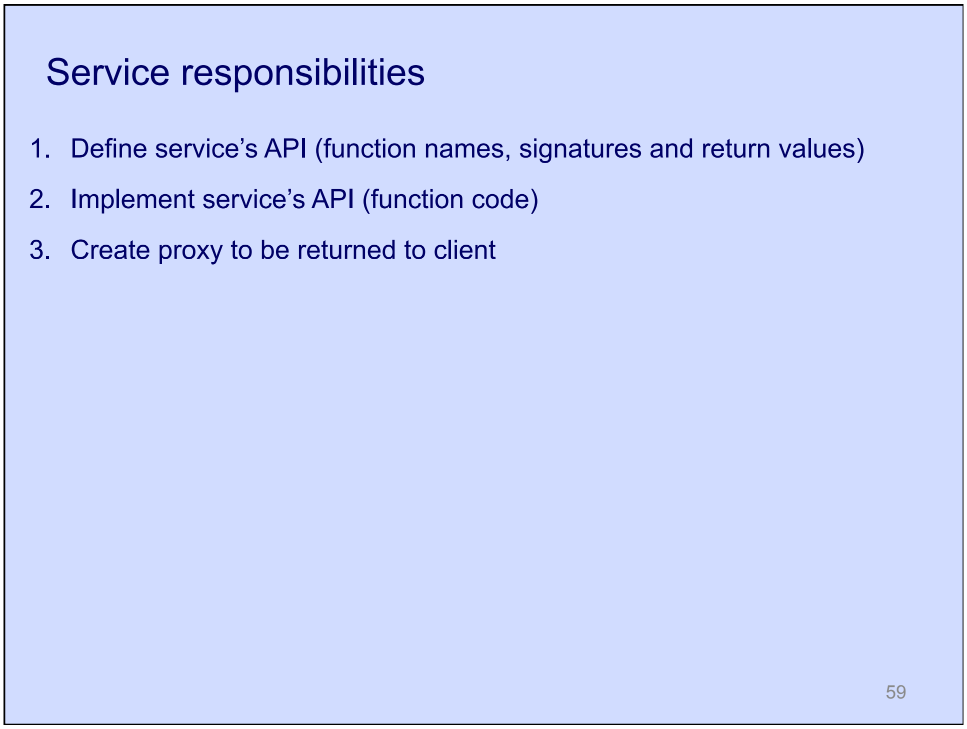
Server

initialize()
registerService()
runService()
enterMainLoop()
useBrokerAPI()



Broker pattern: Implementation Issues

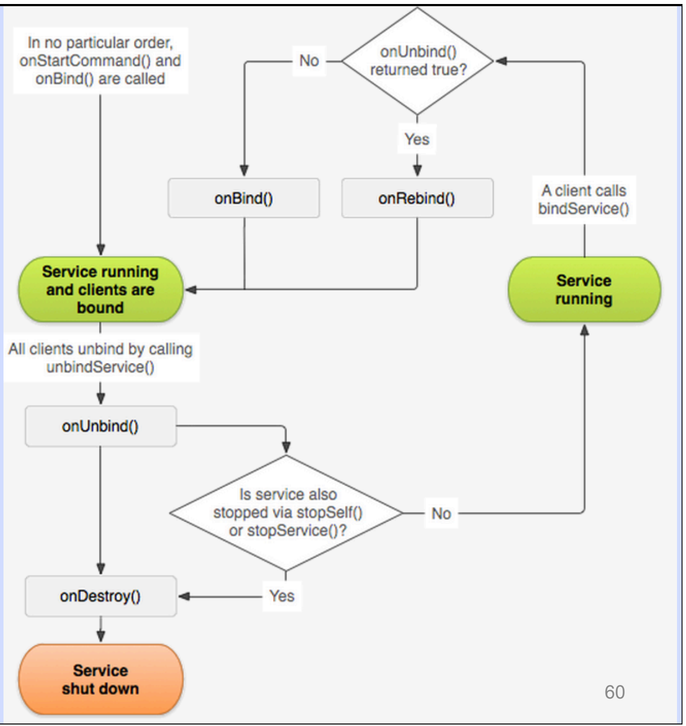
1. Definition of object model
   - What kind of object interfaces, supported types, operations, exceptions, etc.?
   - Alternatively, use lookup table (hashing?)
2. Define component interoperability
   - Binary standard (e.g., MS OLE)
   - Interface Definition Language (IDL) – File containing textual description of services offered by server to clients



Service responsibilities

1. Define service’s API (function names, signatures and return values)
2. Implement service’s API (function code)
3. Create proxy to be returned to client

The lifetime of a bound service



In no particular order, onStartCommand() and onBind() are called

No

onUnbind() returned true?

Yes

onBind()

onRebind()

A client calls bindService()

Service running and clients are bound

Service running

All clients unbind by calling unbindService()

onUnbind()

Is service also stopped via stopSelf() or stopService()?

No

onDestroy()

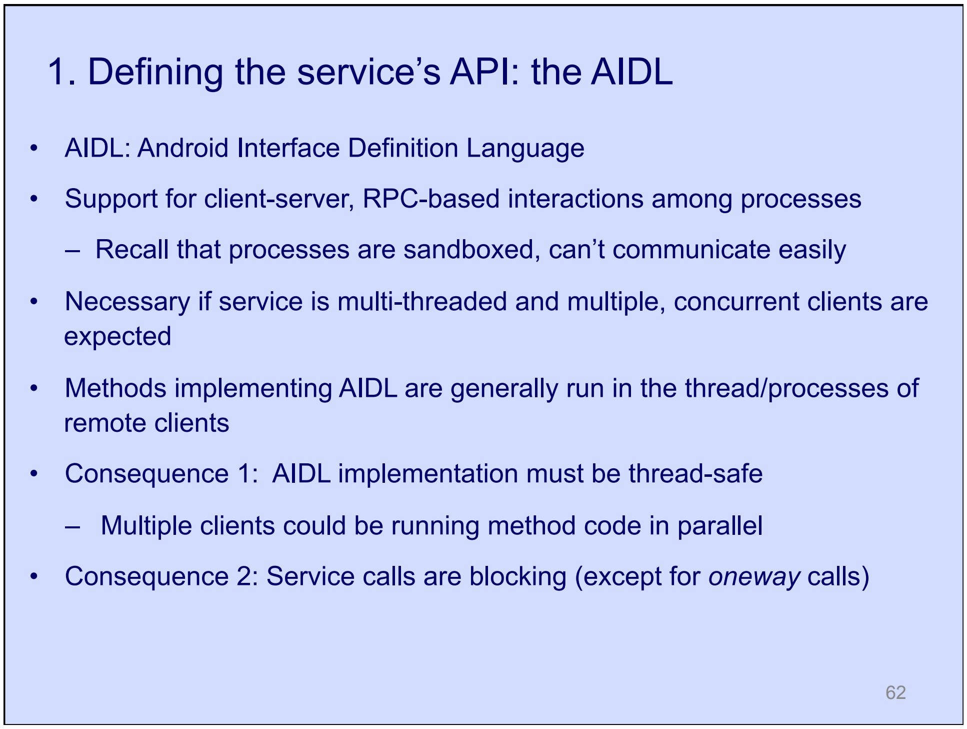
Yes

Service shut down

1. Defining service’s API: Interface *IBinder*

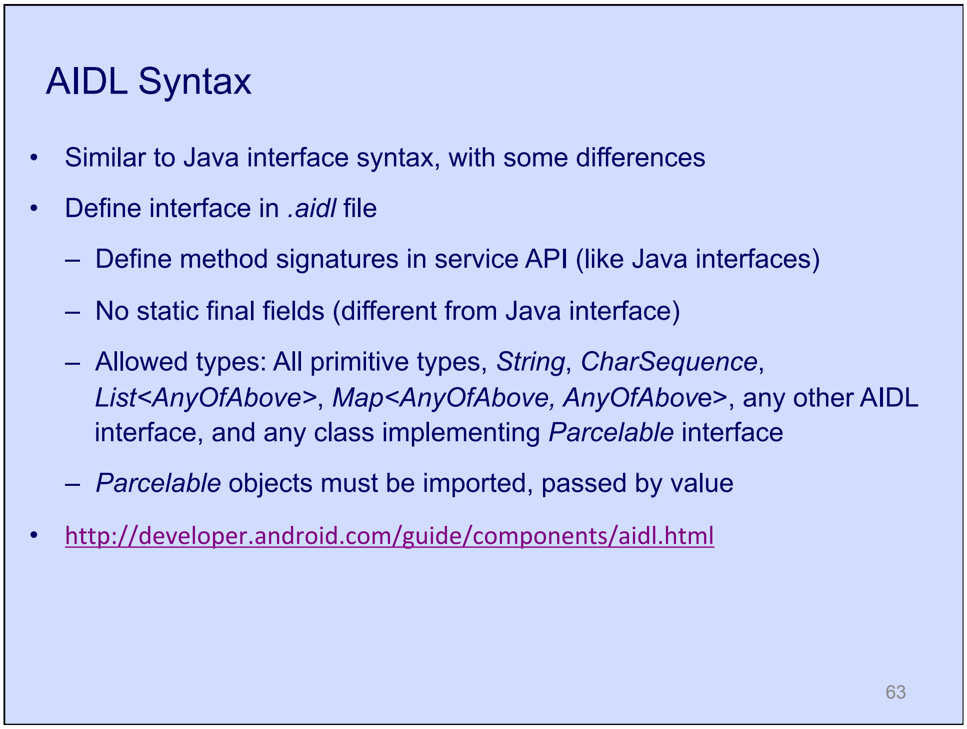
- Three ways to create an *IBinder* implementation
  1. (Local services only): Implement *Binder* subclass directly
     - ➢ Abstract class *Binder* implements Java interface *IBinder*
  2. Define *Handler* in bound service; have *Messenger* object deliver messages from client(s) to bound service across process boundaries
     - ➢ Good for non-local services (clients are different apps); however…
     - ➢ ... service will process requests sequentially
  3. Use Android Interface Definition Language (AIDL)
     - ➢ Good for non-local services; requests processed concurrently
- Here focus on (3), most general

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



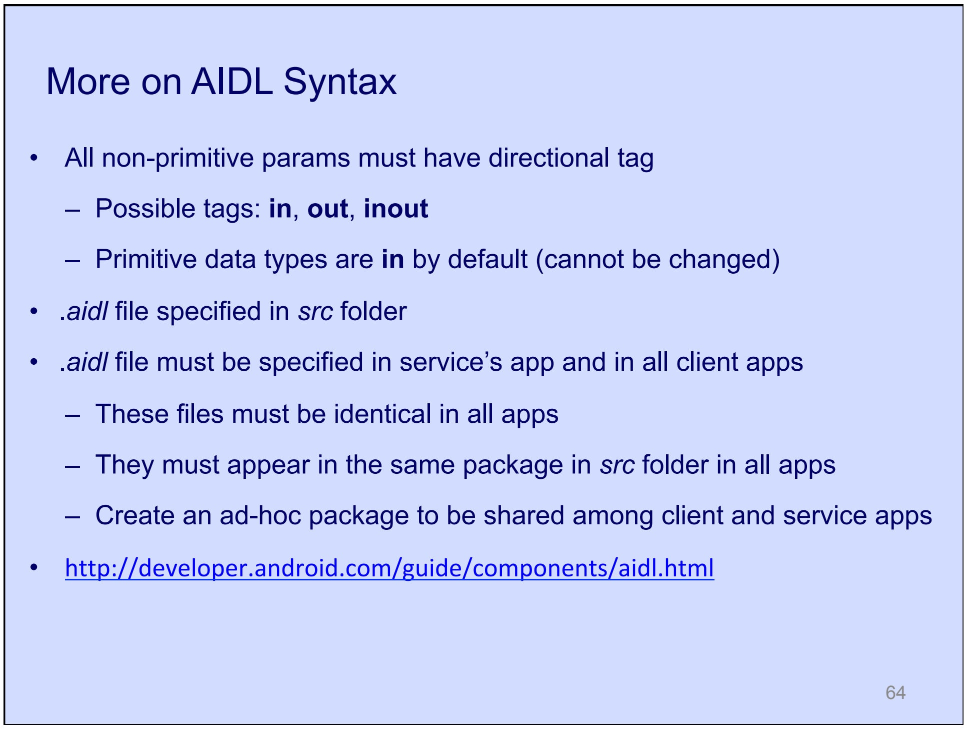
1. Defining the service’s API: the AIDL

- AIDL: Android Interface Definition Language
- Support for client-server, RPC-based interactions among processes
  - Recall that processes are sandboxed, can’t communicate easily
- Necessary if service is multi-threaded and multiple, concurrent clients are expected
- Methods implementing AIDL are generally run in the thread/processes of remote clients
- Consequence 1: AIDL implementation must be thread-safe
  - Multiple clients could be running method code in parallel
- Consequence 2: Service calls are blocking (except for *oneway* calls)



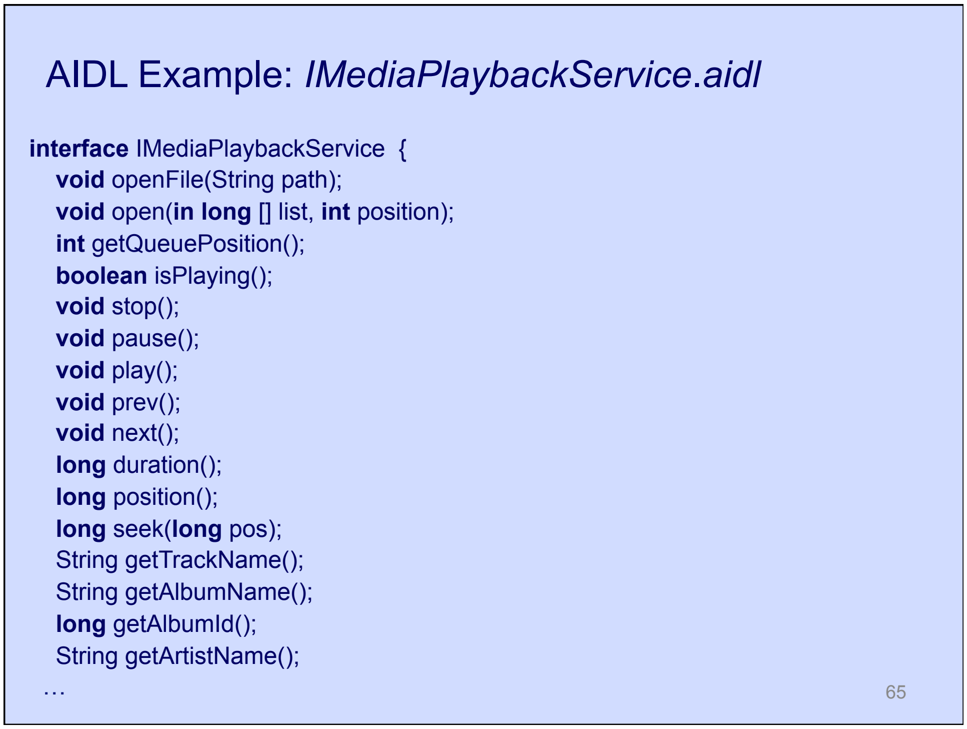
AIDL Syntax

- Similar to Java interface syntax, with some differences
- Define interface in *.aidl* file
  - Define method signatures in service API (like Java interfaces)
  - No static final fields (different from Java interface)
  - Allowed types: All primitive types, *String*, *CharSequence*, *List<AnyOfAbove>*, *Map<AnyOfAbove, AnyOfAbove>*, any other AIDL interface, and any class implementing *Parcelable* interface
  - *Parcelable* objects must be imported, passed by value
- http://developer.android.com/guide/components/aidl.html



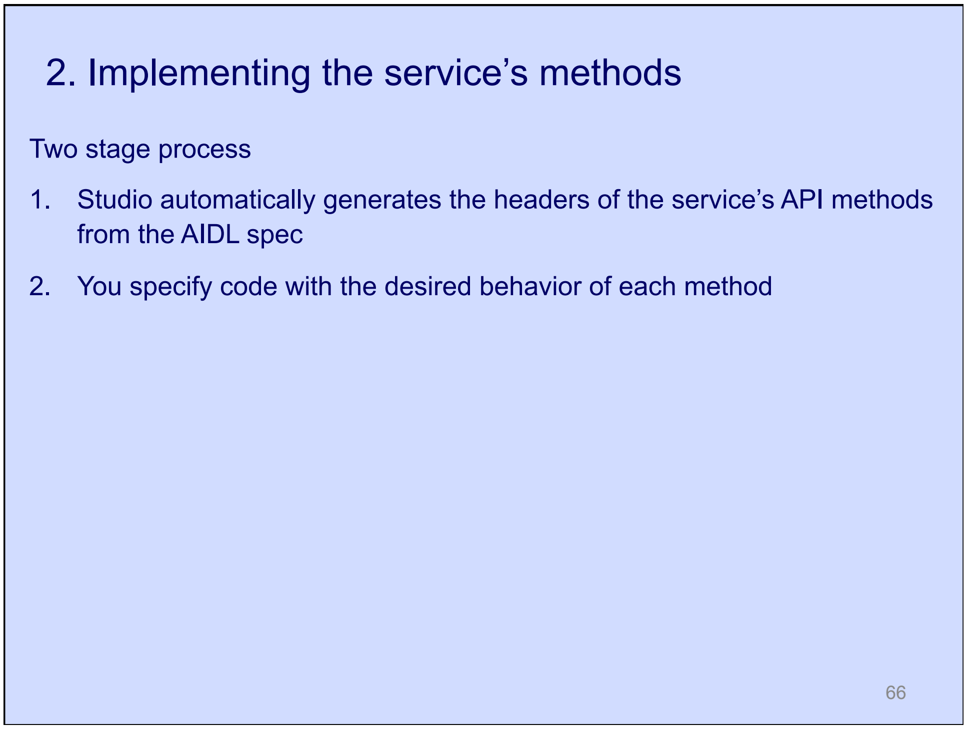
More on AIDL Syntax

- All non-primitive params must have directional tag
  - Possible tags: **in**, **out**, **inout**
  - Primitive data types are **in** by default (cannot be changed)
- *.aidl* file specified in *src* folder
- *.aidl* file must be specified in service's app and in all client apps
  - These files must be identical in all apps
  - They must appear in the same package in *src* folder in all apps
  - Create an ad-hoc package to be shared among client and service apps
- http://developer.android.com/guide/components/aidl.html



AIDL Example: *IMediaPlaybackService.aidl*

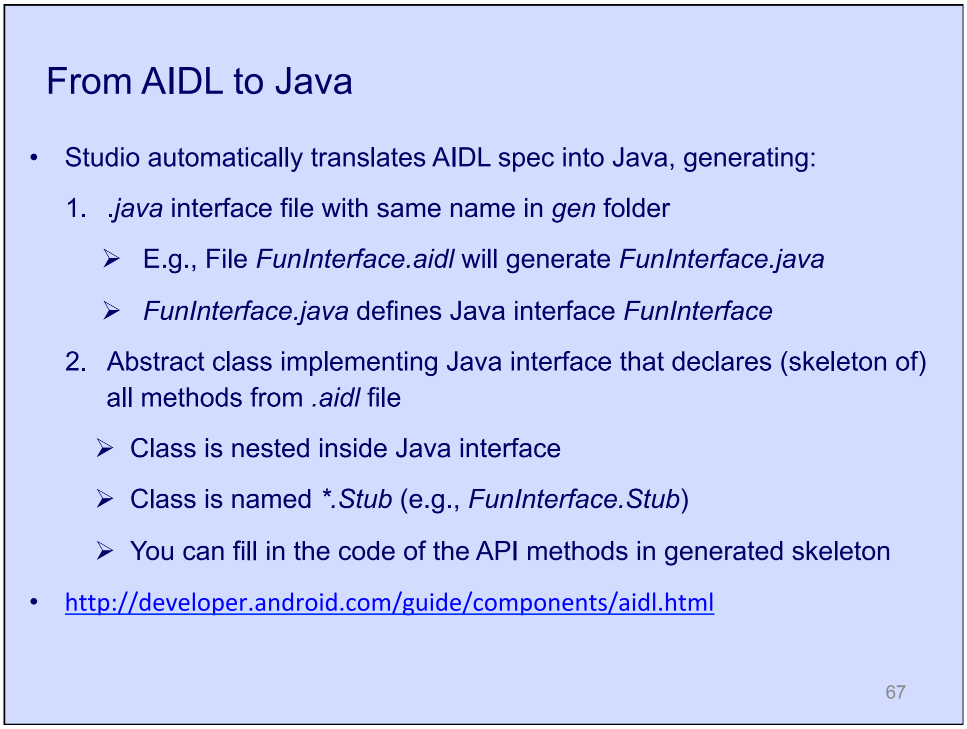
```
interface IMediaPlaybackService  {
   void openFile(String path);
   void open(in long [] list, int position);
   int getQueuePosition();
   boolean isPlaying();
   void stop();
   void pause();
   void play();
   void prev();
   void next();
   long duration();
   long position();
   long seek(long pos);
   String getTrackName();
   String getAlbumName();
   long getAlbumId();
   String getArtistName();
 …
```



2. Implementing the service's methods

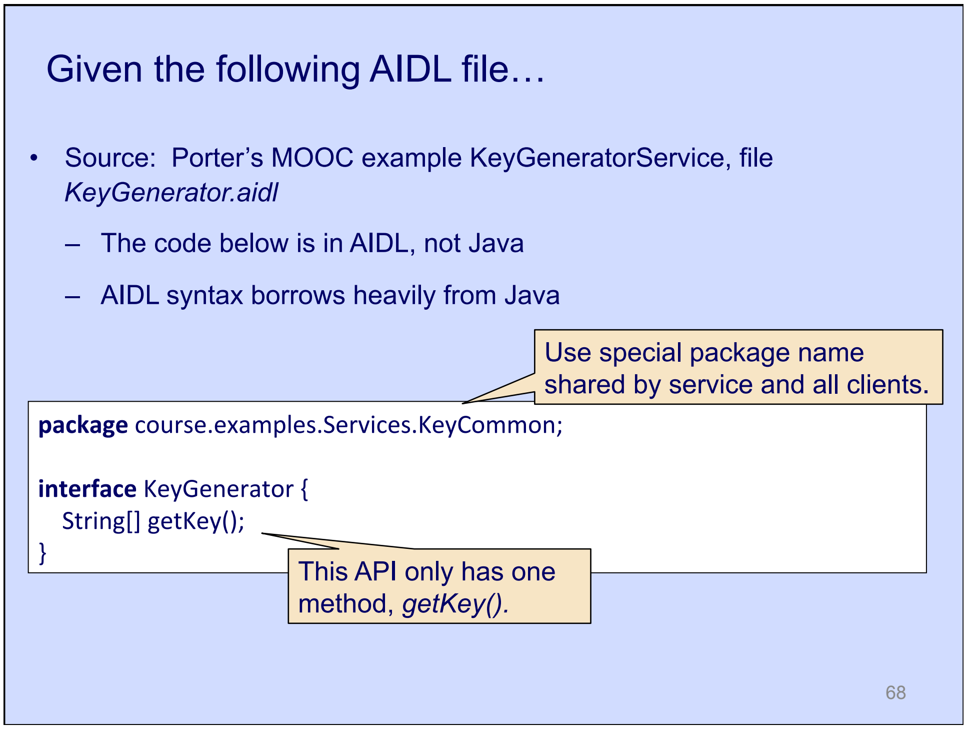
Two stage process

1. Studio automatically generates the headers of the service's API methods from the AIDL spec
2. You specify code with the desired behavior of each method



From AIDL to Java

- Studio automatically translates AIDL spec into Java, generating:
  1. *.java* interface file with same name in *gen* folder
     - ➢ E.g., File *FunInterface.aidl* will generate *FunInterface.java*
     - ➢ *FunInterface.java* defines Java interface *FunInterface*
  2. Abstract class implementing Java interface that declares (skeleton of) all methods from *.aidl* file
     - ➢ Class is nested inside Java interface
     - ➢ Class is named **.Stub* (e.g., *FunInterface.Stub*)
     - ➢ You can fill in the code of the API methods in generated skeleton
- http://developer.android.com/guide/components/aidl.html



Given the following AIDL file…

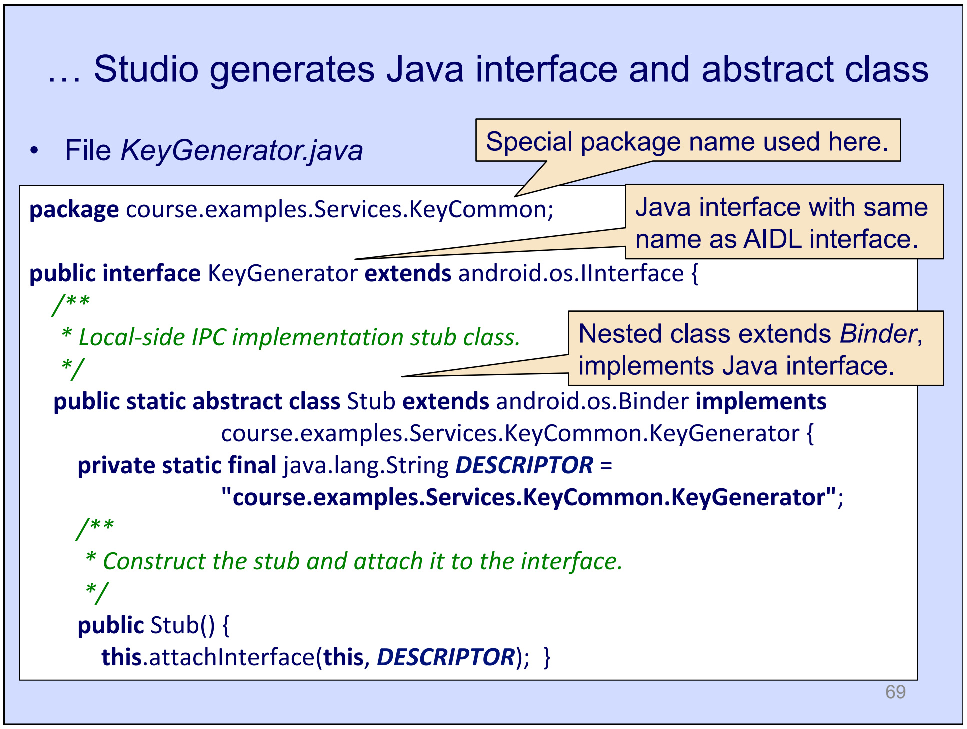
- Source: Porter’s MOOC example KeyGeneratorService, file *KeyGenerator.aidl*
  - The code below is in AIDL, not Java
  - AIDL syntax borrows heavily from Java

Use special package name shared by service and all clients.

```
package course.examples.Services.KeyCommon;

interface KeyGenerator {
   String[] getKey();
}
```

This API only has one method, *getKey()*.

68



… Studio generates Java interface and abstract class

- File *KeyGenerator.java*

Special package name used here.

Java interface with same name as AIDL interface.

Nested class extends *Binder*, implements Java interface.

```
package course.examples.Services.KeyCommon;

public interface KeyGenerator extends android.os.IInterface {
  /**
   * Local-side IPC implementation stub class.
   */
  public static abstract class Stub extends android.os.Binder implements
                      course.examples.Services.KeyCommon.KeyGenerator {
    private static final java.lang.String DESCRIPTOR =
                      "course.examples.Services.KeyCommon.KeyGenerator";
    /**
     * Construct the stub and attach it to the interface.
     */
    public Stub() {
      this.attachInterface(this, DESCRIPTOR);  }
```

69

3. Writing *onBind()* method

- Service creates instance of *Stub* object,
- Method *onBind()* returns object

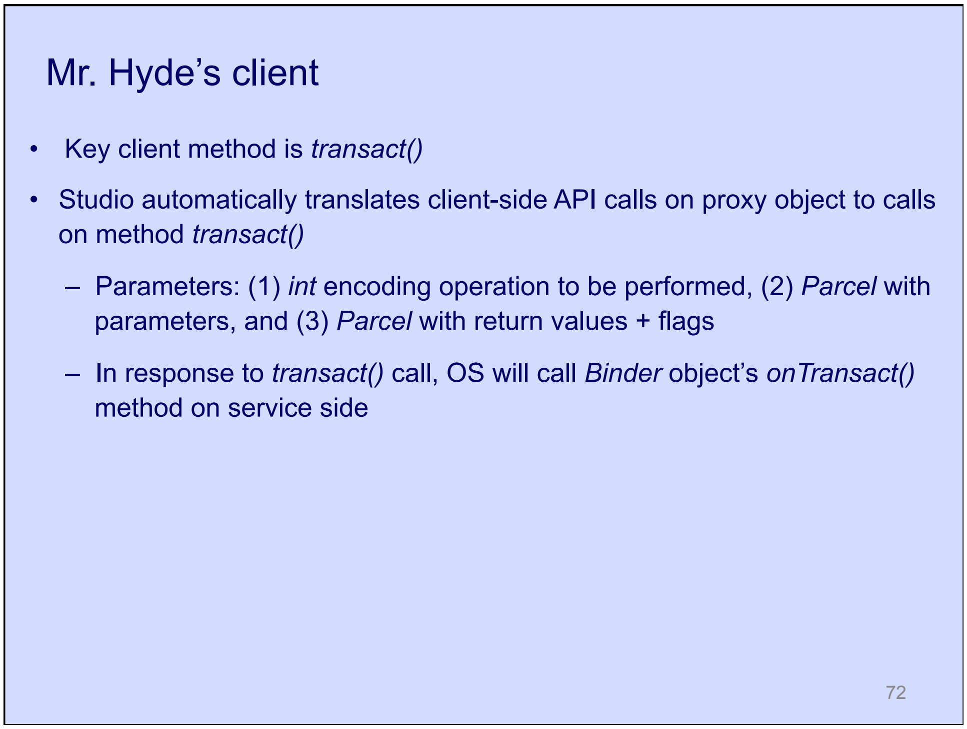
```
// Implement the Stub for this Object
private final KeyGenerator.Stub mBinder = new KeyGenerator.Stub() {
  // Implement the remote method
  public String[] getKey() {
      String[] s;
    // Generate unique id
    ...
    s = new String[]{ id.toString()};
    return s;
  }
};
// Return the Stub defined above
@Override
public IBinder onBind(Intent intent) {  return mBinder; }
```

Implement API in *Stub* class, which implements interface *IBinder.*

onBind() return *IBinder* object, that will be passed to *onServiceConnected()*

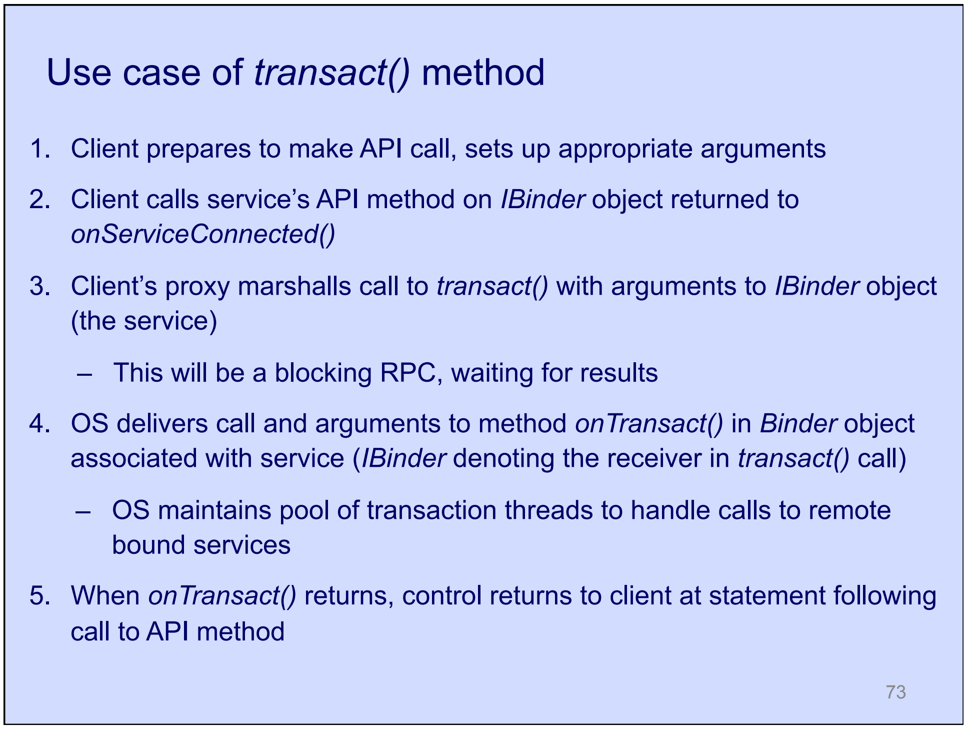
Dr. Jekyll and Mr. Hyde of bound services

- At programmer's level the service and clients interact through the methods declared in the service's AIDL spec – Very nice! (Dr. Jekyll)
- But client calls + their data are automatically marshalled into parcels passed to *transact()* method in client, whence Android passes these parcels to *onTransact()* method in service – A bit messy (Mr. Hyde)
- The whole process is transparent to the programmer



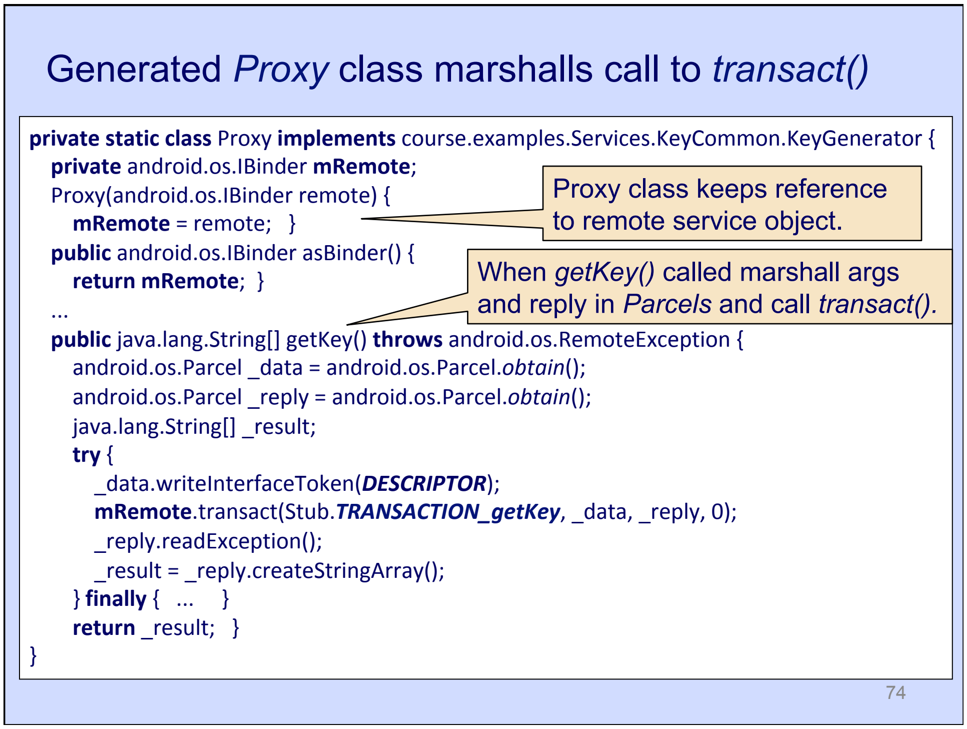
Mr. Hyde’s client

- Key client method is *transact()*
- Studio automatically translates client-side API calls on proxy object to calls on method *transact()*
  - Parameters: (1) *int* encoding operation to be performed, (2) *Parcel* with parameters, and (3) *Parcel* with return values + flags
  - In response to *transact()* call, OS will call *Binder* object’s *onTransact()* method on service side



Use case of *transact()* method

1. Client prepares to make API call, sets up appropriate arguments
2. Client calls service’s API method on *IBinder* object returned to *onServiceConnected()*
3. Client’s proxy marshalls call to *transact()* with arguments to *IBinder* object (the service)
   - This will be a blocking RPC, waiting for results
4. OS delivers call and arguments to method *onTransact()* in *Binder* object associated with service (*IBinder* denoting the receiver in *transact()* call)
   - OS maintains pool of transaction threads to handle calls to remote bound services
5. When *onTransact()* returns, control returns to client at statement following call to API method

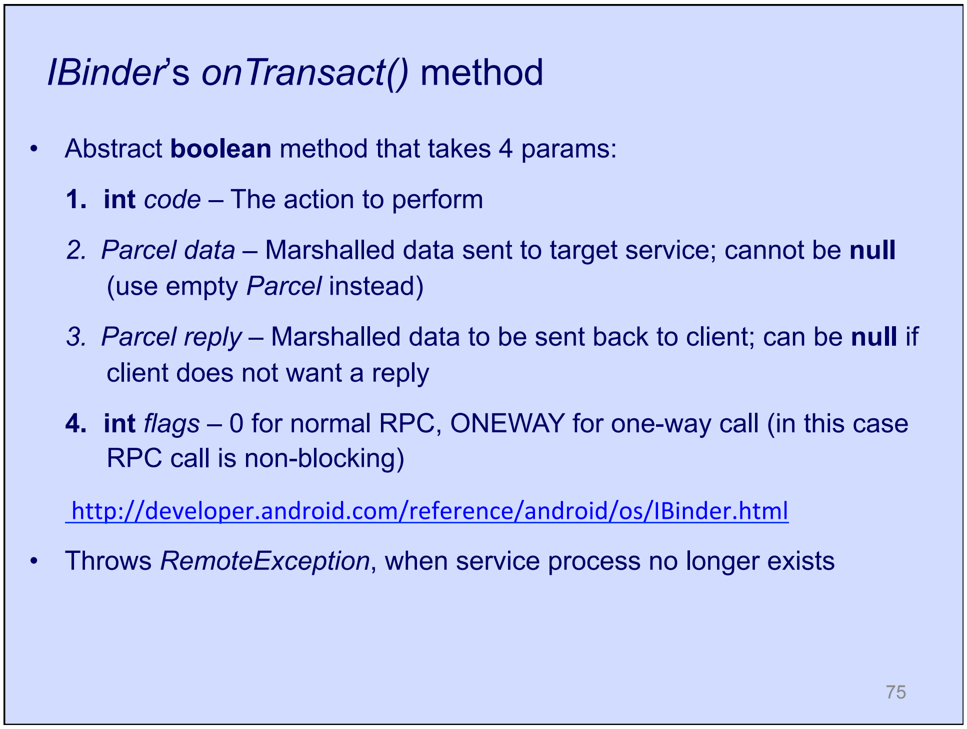


Generated *Proxy* class marshalls call to *transact()*

```
private static class Proxy implements course.examples.Services.KeyCommon.KeyGenerator {
   private android.os.IBinder mRemote;
   Proxy(android.os.IBinder remote) {
      mRemote = remote;   }
   public android.os.IBinder asBinder() {
      return mRemote;  }
   …
   public java.lang.String[] getKey() throws android.os.RemoteException {
      android.os.Parcel _data = android.os.Parcel.obtain();
      android.os.Parcel _reply = android.os.Parcel.obtain();
      java.lang.String[] _result;
      try {
         _data.writeInterfaceToken(DESCRIPTOR);
         mRemote.transact(Stub.TRANSACTION_getKey, _data, _reply, 0);
         _reply.readException();
         _result = _reply.createStringArray();
      } finally {   …    }
      return _result;   }
}
```

Proxy class keeps reference to remote service object.

When *getKey()* called marshall args and reply in *Parcels* and call *transact().*



IBinder's *onTransact()* method

- Abstract **boolean** method that takes 4 params:

  1. **int** *code* – The action to perform

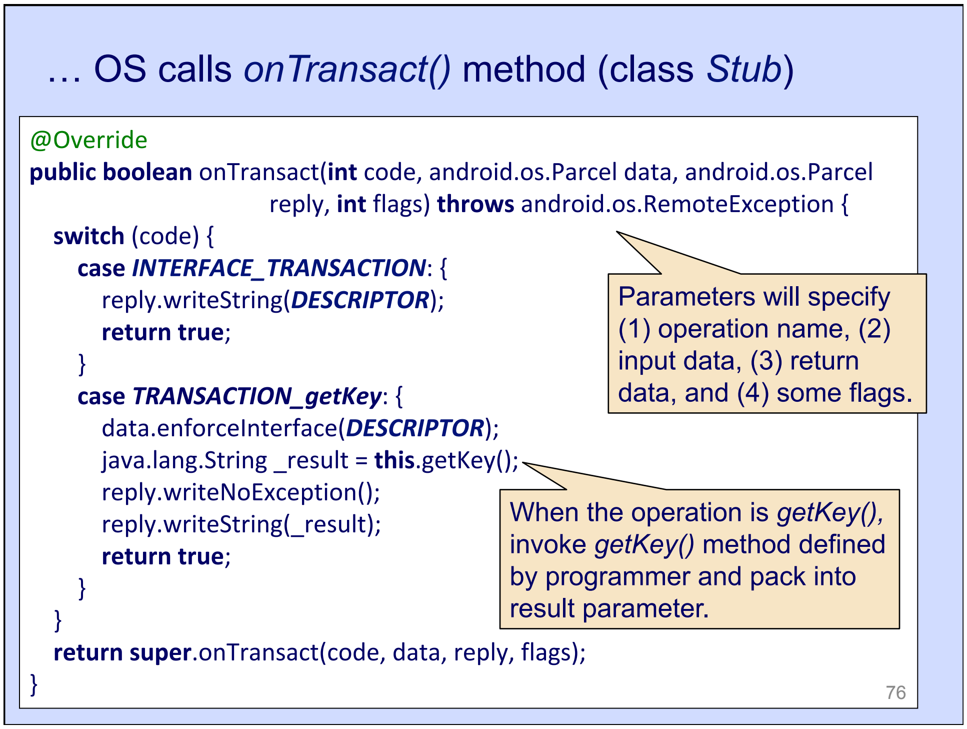
  2. *Parcel data* – Marshalled data sent to target service; cannot be **null** (use empty *Parcel* instead)

  3. *Parcel reply* – Marshalled data to be sent back to client; can be **null** if client does not want a reply

  4. **int** *flags* – 0 for normal RPC, ONEWAY for one-way call (in this case RPC call is non-blocking)

  http://developer.android.com/reference/android/os/IBinder.html

- Throws *RemoteException*, when service process no longer exists

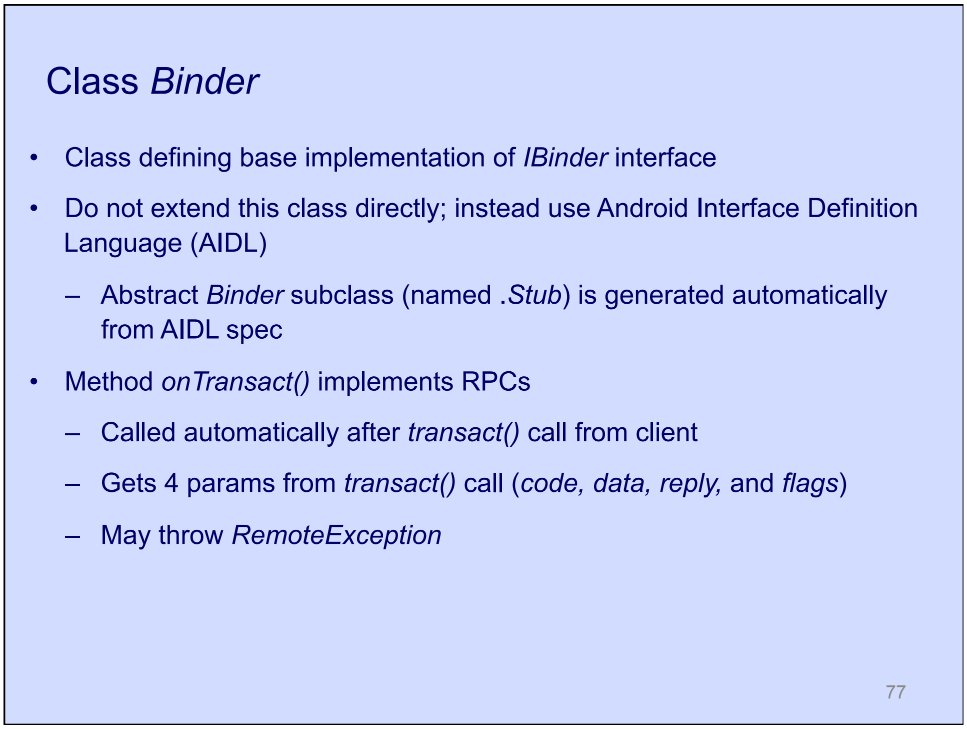


… OS calls *onTransact()* method (class *Stub*)

```
@Override
public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel
                    reply, int flags) throws android.os.RemoteException {
  switch (code) {
    case INTERFACE_TRANSACTION: {
      reply.writeString(DESCRIPTOR);
      return true;
    }
    case TRANSACTION_getKey: {
      data.enforceInterface(DESCRIPTOR);
      java.lang.String _result = this.getKey();
      reply.writeNoException();
      reply.writeString(_result);
      return true;
    }
  }
  return super.onTransact(code, data, reply, flags);
}
```

Parameters will specify (1) operation name, (2) input data, (3) return data, and (4) some flags.

When the operation is *getKey(),* invoke *getKey()* method defined by programmer and pack into result parameter.



Class *Binder*

- Class defining base implementation of *IBinder* interface
- Do not extend this class directly; instead use Android Interface Definition Language (AIDL)
  - Abstract *Binder* subclass (named *.Stub*) is generated automatically from AIDL spec
- Method *onTransact()* implements RPCs
  - Called automatically after *transact()* call from client
  - Gets 4 params from *transact()* call (*code, data, reply,* and *flags*)
  - May throw *RemoteException*

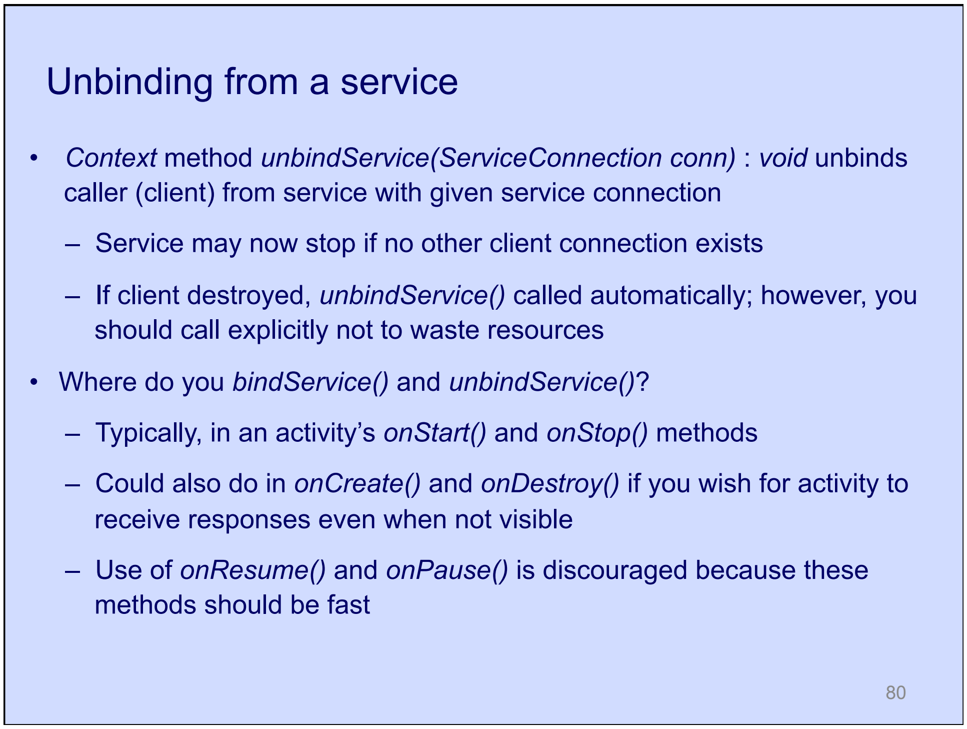
Summary: The use case of service binding

1. Client creates *ServiceConnection* object, defining 2 methods
2. Client calls *bindService()*, passing an intent, the *ServiceConnection* object, and flags; call returns immediately
3. If not running, OS instantiates service and calls *onCreate()* on new instance
4. OS calls *onBind()*, passing intent; connection established
5. OS calls back *onServiceConnected()* in *ServiceConnection* object created by client; *onServiceConnected()* runs in client
   - Client receives reference to service as an *IBinder* object passed as argument to *onServiceConnected()* call

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

Stopping a service

- *Context* method *stopService(Intent)* : *boolean* stops service matching intent if service is running
  - *true* if service was found and stopped
  - *false* service not found, or service was found but was not running
- May throw *SecurityException* if calling element does not have permission to stop the service
- Will not actually stop the service if service is bound to a client and BIND_AUTO_CREATE was set upon service binding



Unbinding from a service

- *Context* method *unbindService(ServiceConnection conn)* : *void* unbinds caller (client) from service with given service connection
  - Service may now stop if no other client connection exists
  - If client destroyed, *unbindService()* called automatically; however, you should call explicitly not to waste resources
- Where do you *bindService()* and *unbindService()*?
  - Typically, in an activity's *onStart()* and *onStop()* methods
  - Could also do in *onCreate()* and *onDestroy()* if you wish for activity to receive responses even when not visible
  - Use of *onResume()* and *onPause()* is discouraged because these methods should be fast