



D u k e S y s t e m s

Android (revised)

Jeff Chase

Duke University

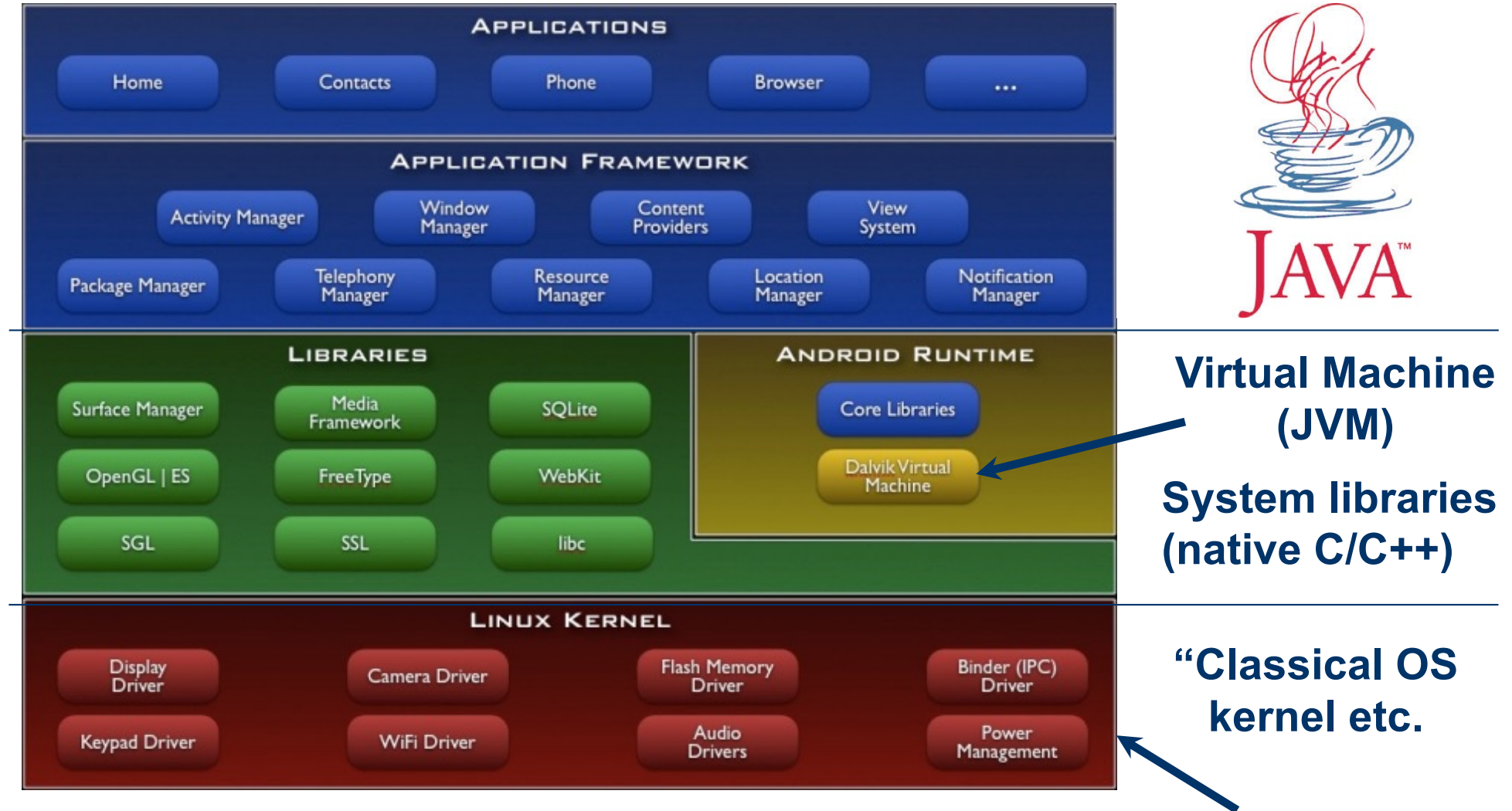
Android in a nutshell

- **Linux kernel inside:** Android uses the Unix OS abstractions with a few mods (e.g., they added “binder” IPC as a kernel module).
- **Apps run as multi-threaded Linux processes.** Typ. one process per-app per-device (per-user). The process runs whenever some part of the app is active --- then the system kills it for its memory.
- **Linux is hidden behind Java and Android APIs.** All apps run over Dalvik JVM [or now ART/AOT] with canned libraries (bionic).
- **Android subsystems run as services outside of the kernel.** Android environment combines various managers (running as processes) and a baked-in runtime library linked with all apps.
- **Event-driven app ecosystem.** Apps can integrate seamlessly with other apps, providing services, data, UI screens, etc. to one another.
- **Object/component model with object-oriented IPC and asynchronous event notification.** IDL/RPC with objects, permissions, reference counting, thread pools, **upcalls**.

Platforms are layered/nested



ANDROID



Reloaded with a few new kernel extensions (drivers).

Android: components

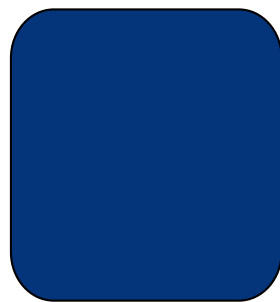
App Components

Android's application framework lets you create rich and innovative apps using a set of reusable components. This section explains how you can build the components that define the building blocks of your app and how to connect them together using intents.

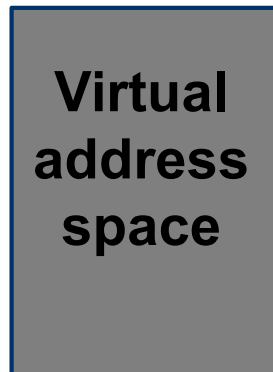
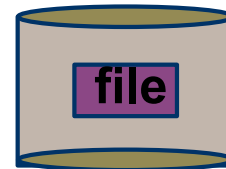
[INTENTS AND INTENT FILTERS](#) >

Some symbols

- Figures follow conventions established in previous lectures.

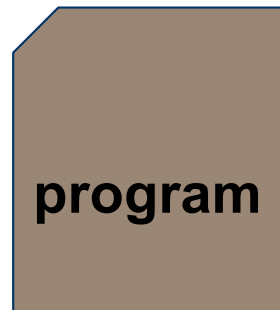


**Module
(component)**



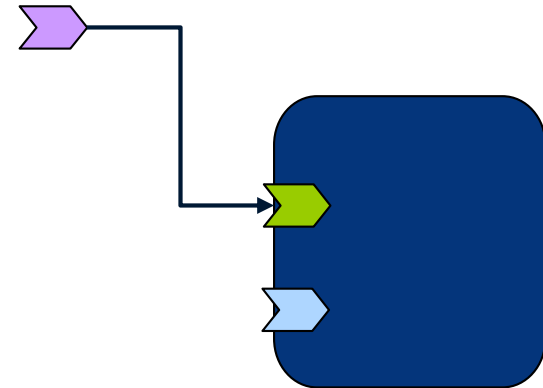
**Virtual
address
space**

**Light blue
indicates a
trusted
system
element**

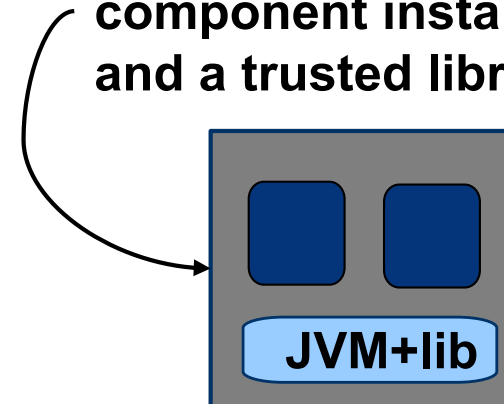


program

API / linkage / binding

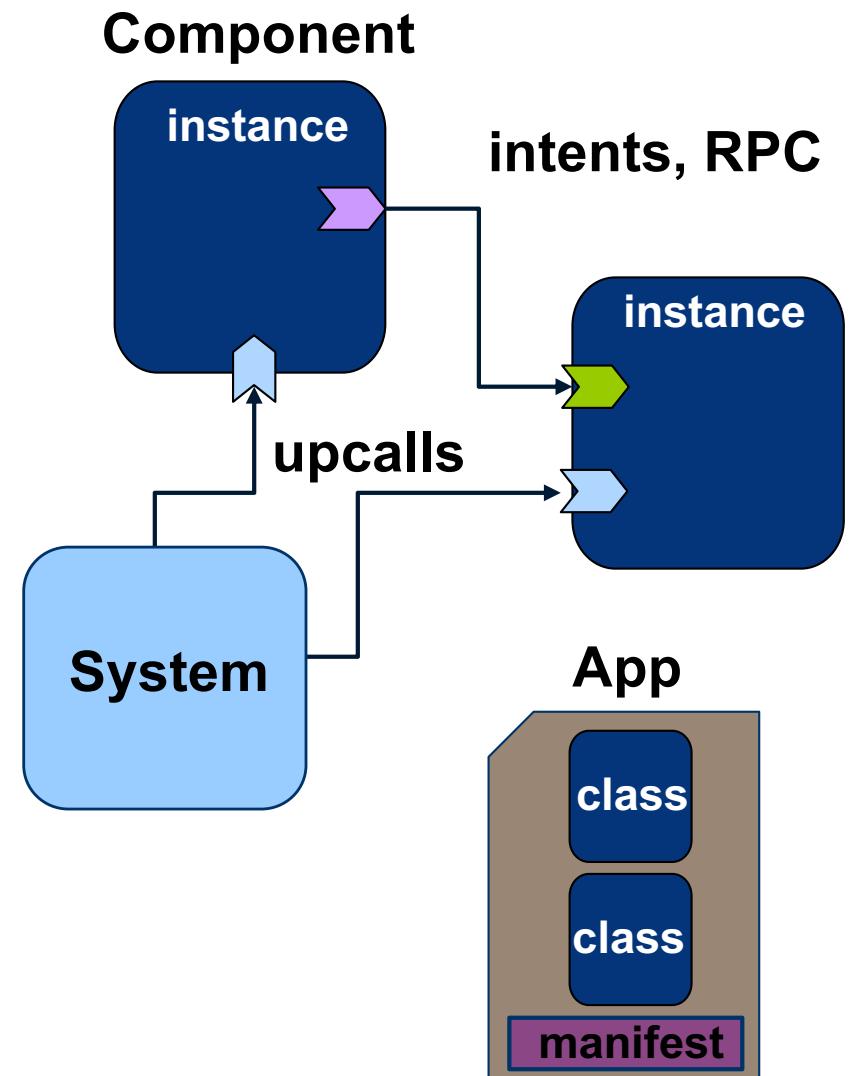


**A process/VAS with
component instances
and a trusted library.**



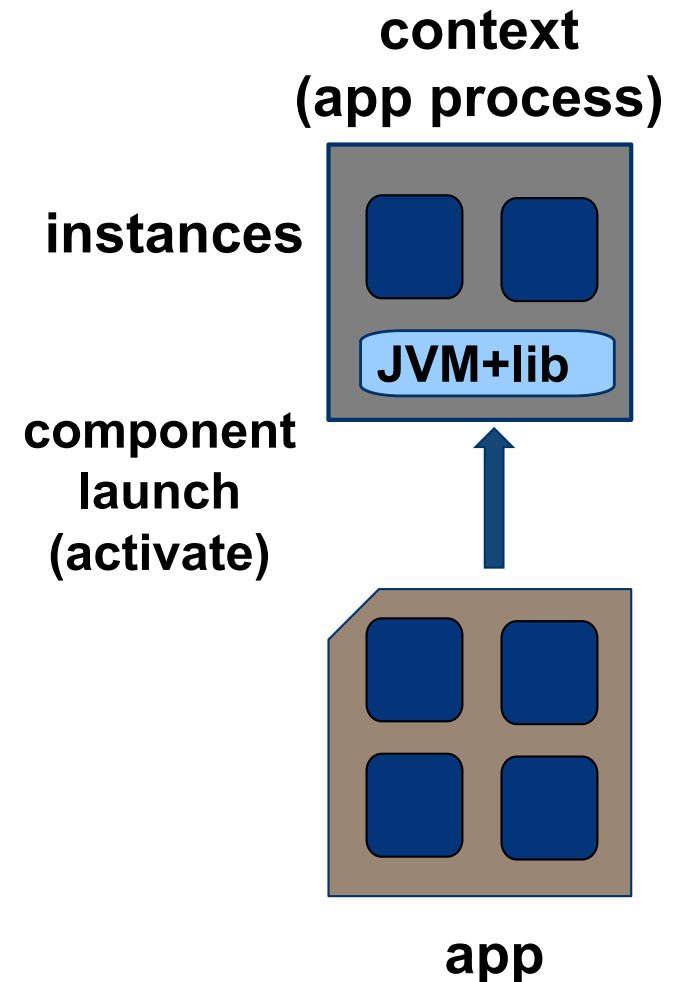
Android: components

- An app is a set of **components**.
 - App declares a metadata list (**manifest**) of its components, and what events (**intents**) they respond to.
- System instantiates and destroys components automatically, driven by various events in the system and UI.
- Components have **upcall** interfaces invoked from the system to notify them of lifecycle events
 - activate, deactivate, etc.
- Apps may interact with one another by **typed messages** among components.
 - **intents** and **binder RPC**



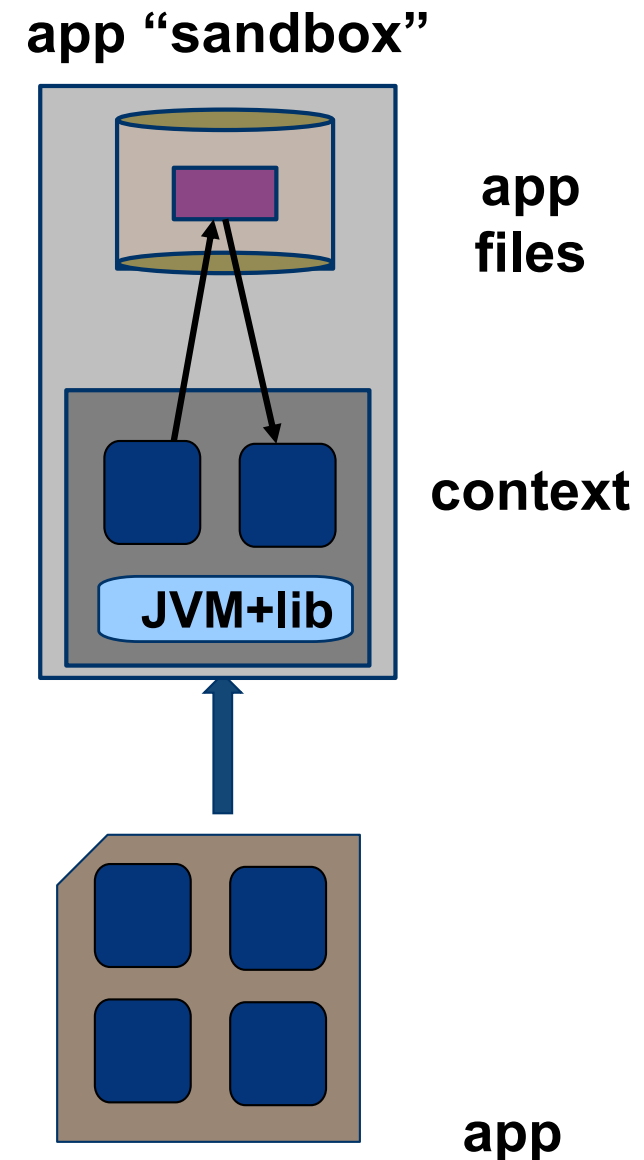
Components run in contexts

- **Components are Java code.**
 - A component is a **class** in an app.
 - Its name is (appname, classname).
 - Apps are named as Java packages.
- **Components / apps run in JVMs.**
 - Each component runs at most one **instance** in exactly one context.
 - Components may activate/deactivate independently of one another.
 - The context is a Linux **process** with a JVM and a trusted system library.
 - Android library defines **context** object with system API.



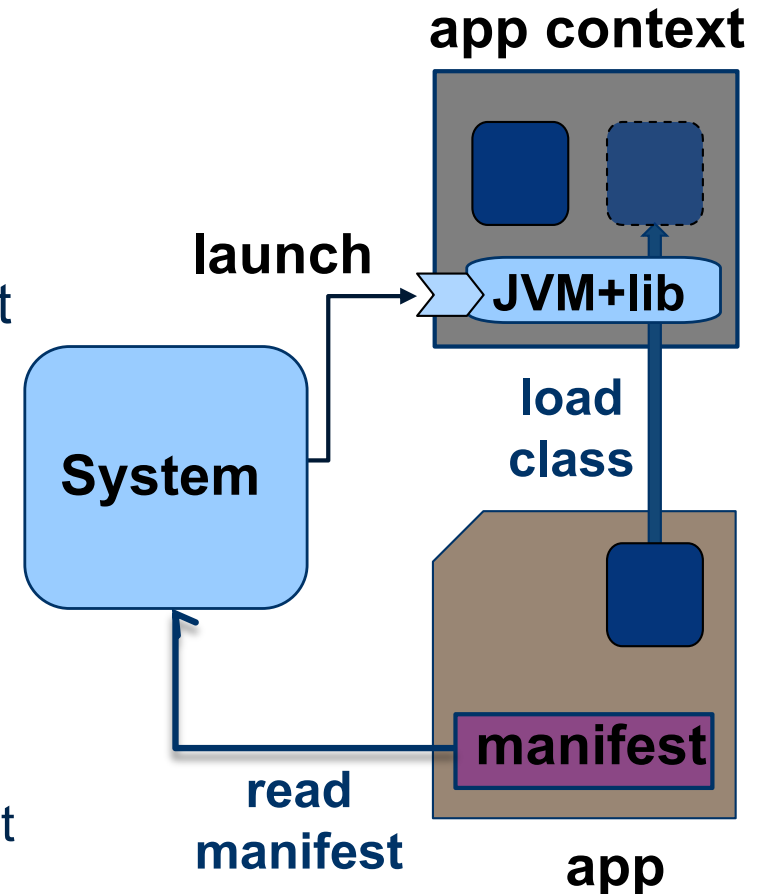
Apps are isolated

- Components in the same app (generally) share a context.
- Components in different apps are **always** in different contexts.
- Apps cannot reference each other's memory contexts (sandbox/lockbox):
 - “soft” JVM protection
 - “hard” process boundaries
- Apps interact only via IPC.
 - intents, events
 - service RPC and content put/get
- Apps run with distinct user IDs.
 - Principle of Least Privilege



Component launch

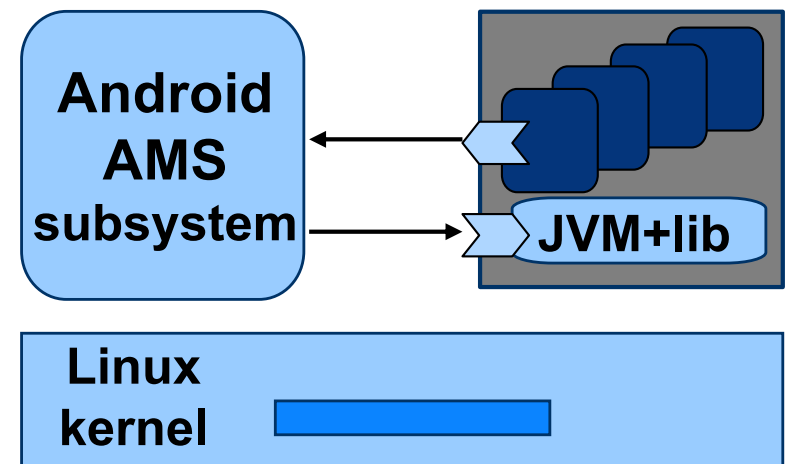
- To launch a component, the system:
 - Selects a JVM context to run it.
 - Tells JVM to instantiate the class.
- System communicates with the context via its system library.
- System obtains info about the component from app manifest.
 - Class → component type: the component class descends from a system base class.
 - List of event profiles (**intent filters**) that trigger component launch.



If there is no JVM context active for the component's app, then the system starts one.

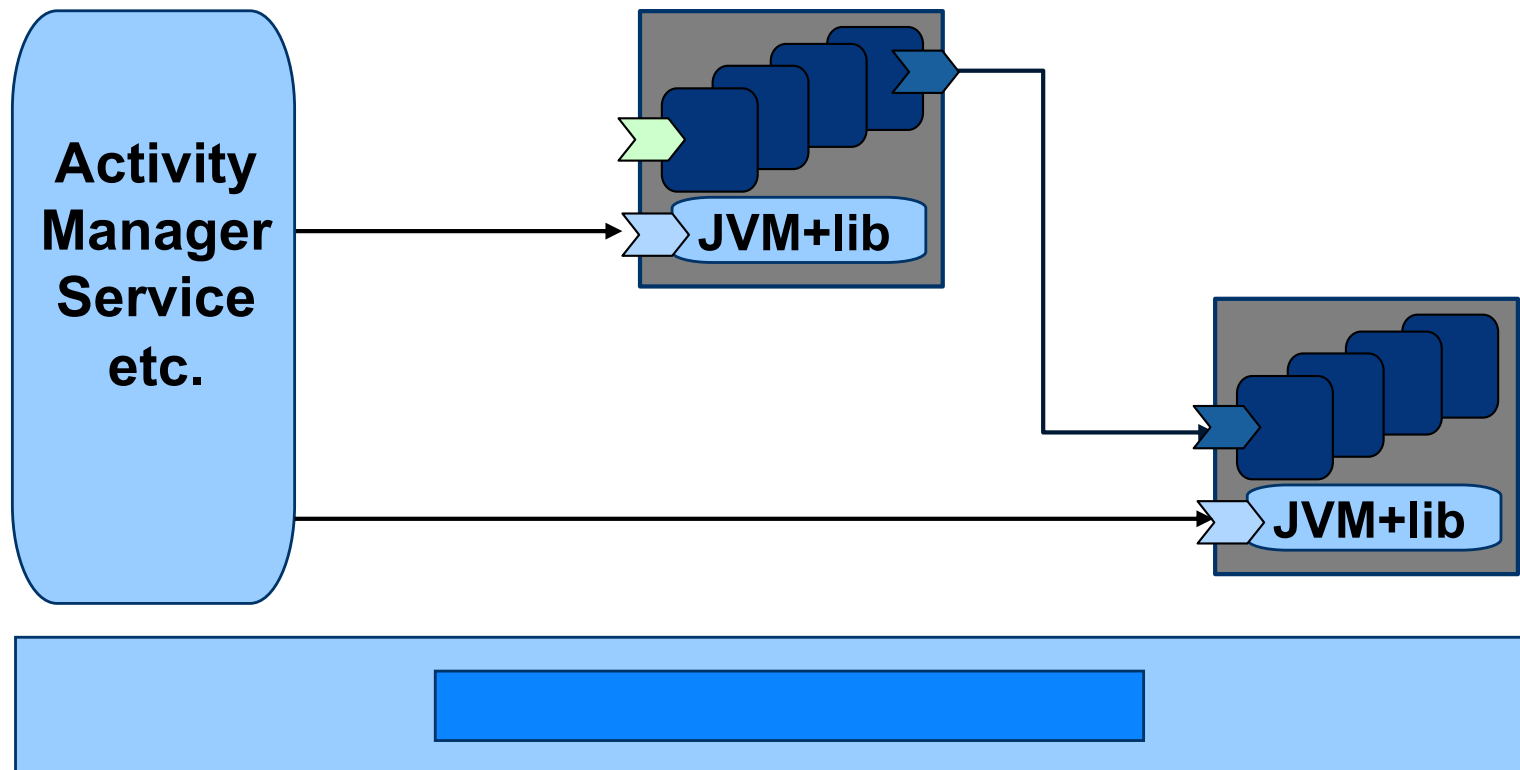
“Subsystems”

- A server process may provide trusted system functions to other processes, outside of the kernel.
 - E.g., this code is trusted, but like other processes it cannot manipulate the hardware state except by invoking the kernel.
- Example: Android **Activity Manager** subsystem provides many functions of Android, e.g., component launch and brokering of component interactions.
- With no special kernel support! It uses same syscalls as anyone else.
- AMS controls app contexts by forking them with a trusted lib, and issuing RPC commands to that lib.



“binder” message driver in kernel

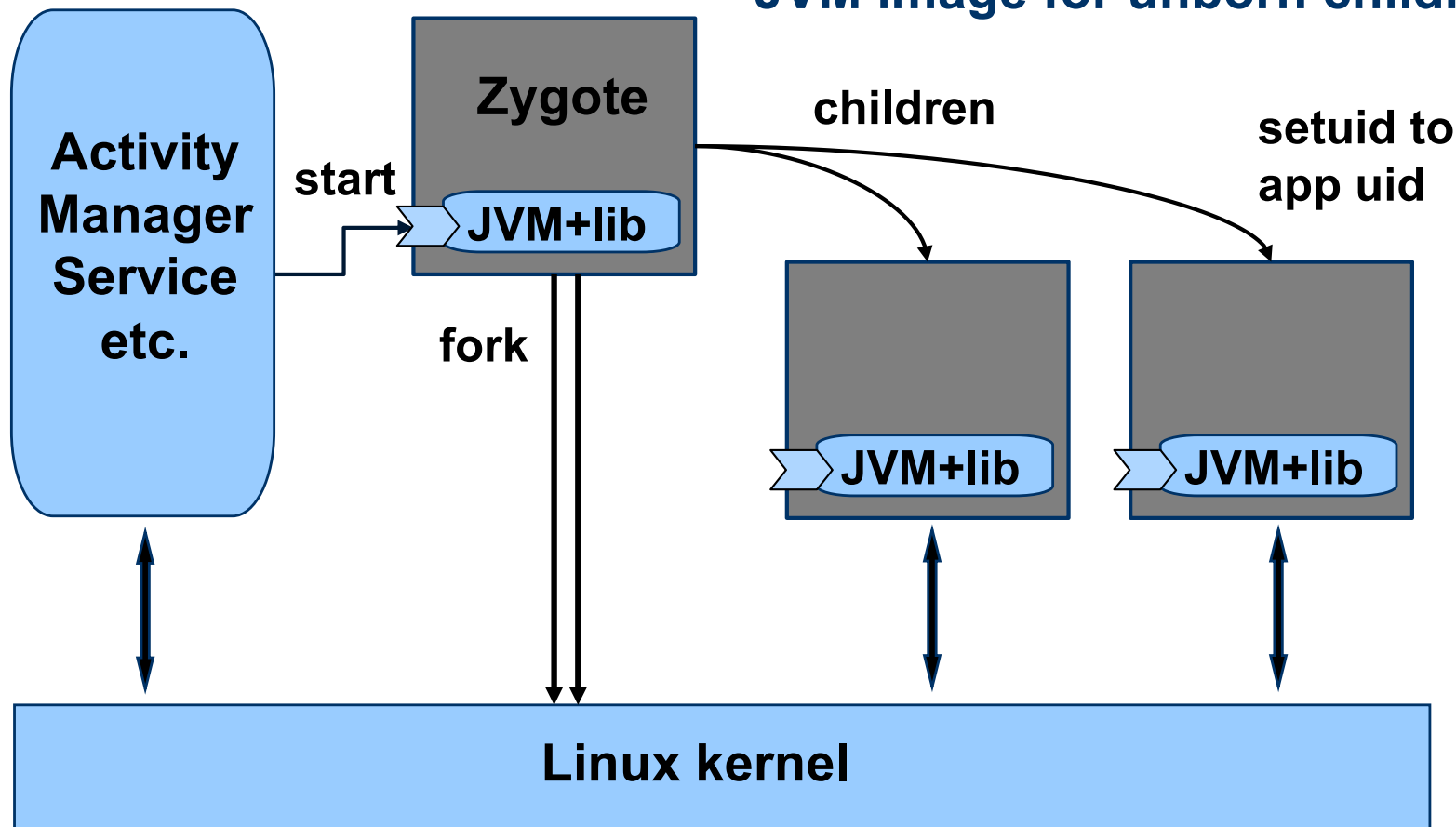
Android environment server



The Activity Manager keeps an RPC binding to every context. Apps call system APIs and receive events via binder RPC calls to/from Android Activity Manager etc.

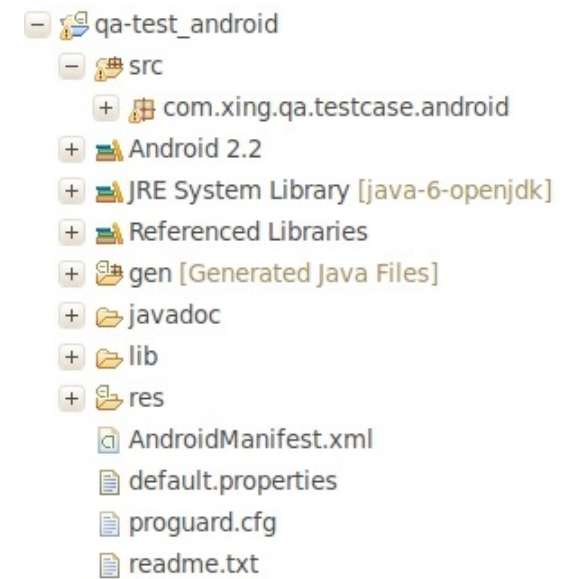
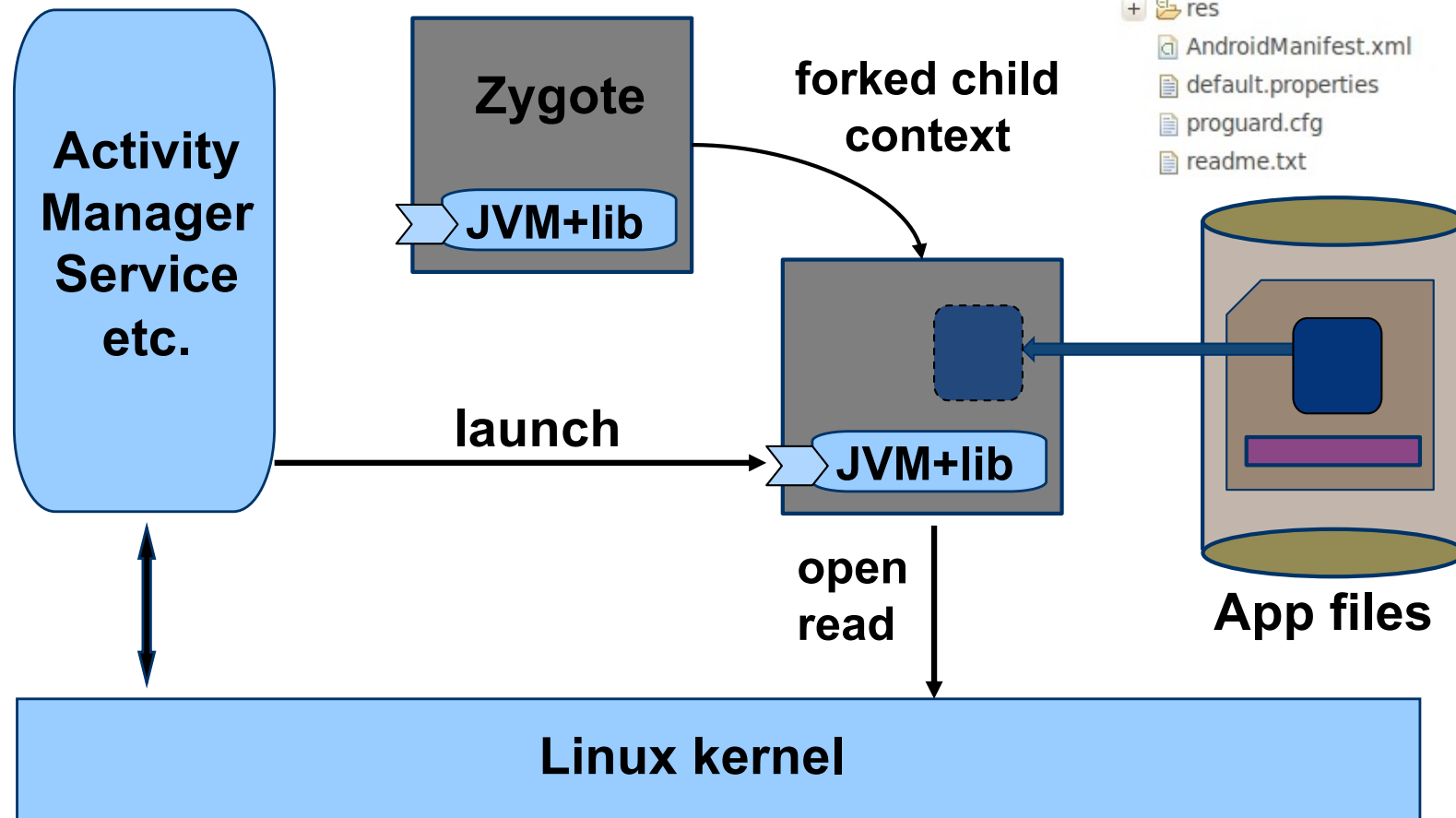
App launch

Zygote is a preinitialized “warm” JVM image for unborn children.



How do we launch the application's code? Exec?

App launch



No exec needed: all Android contexts run the same Linux program: the JVM. Fork is just right!



Platform abstractions

- Platforms provide “building blocks”...
- ...and APIs to use them.
 - Instantiate/create/allocate
 - Manipulate/configure
 - Attach/detach
 - Combine in uniform ways
 - Release/destroy

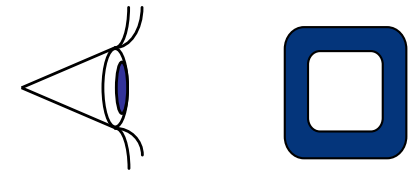


The choice of abstractions reflects a philosophy of how to build and organize software systems.

The four component types

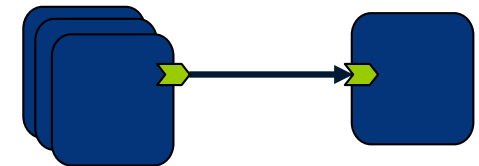
1. **Activity**. Display a screen.

- Push on a “back stack”, like a browser.
- May be launched to respond to intents sent by other apps.



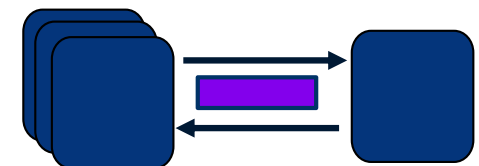
2. **Service**. Serve an API (e.g.).

- Export binder RPC interface to other apps.



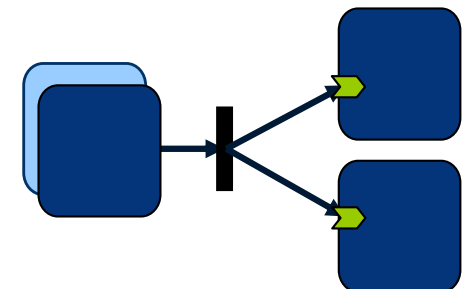
3. **Provider**. Get/put content objects.

- Serve a URI space with MIME types.
- E.g., backed by SQLite database tables.

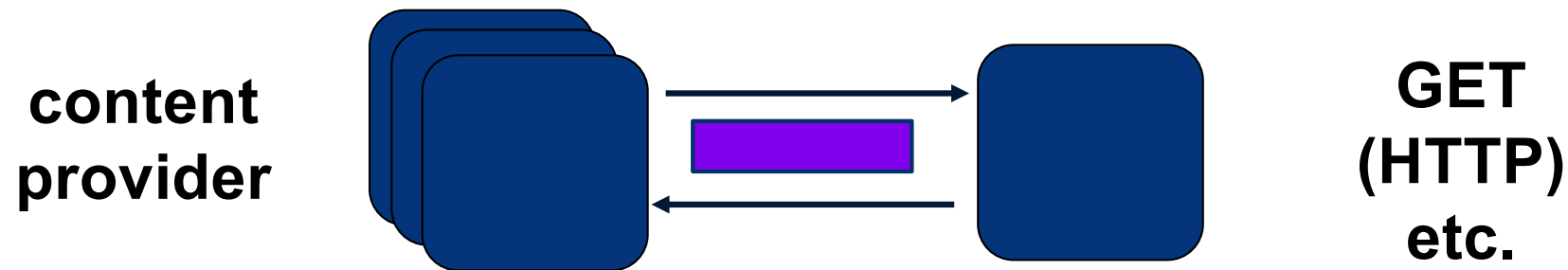
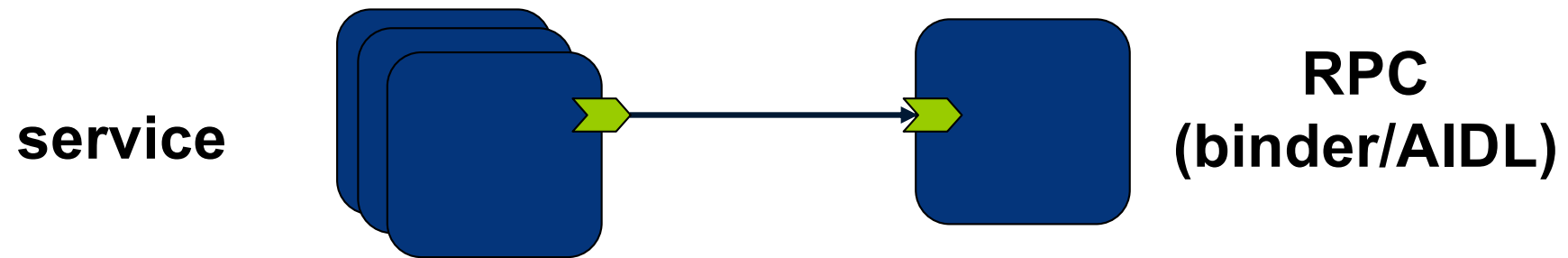


4. **Receiver**. Respond to event notifications.

- E.g., low battery, wifi up/down, storage full, USB insert, ...



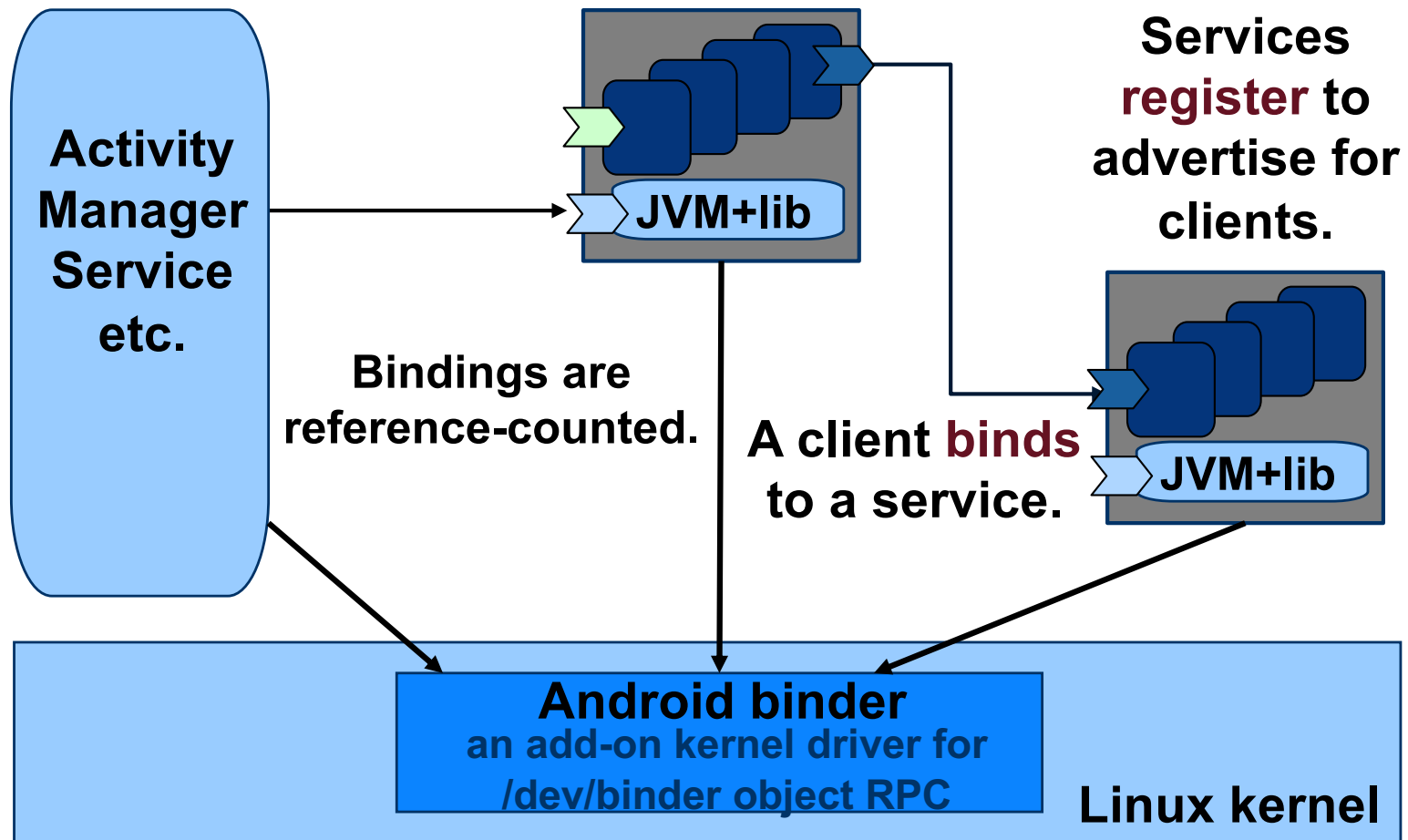
Apps offer services to other apps



Clients
initiate connection
and send requests.

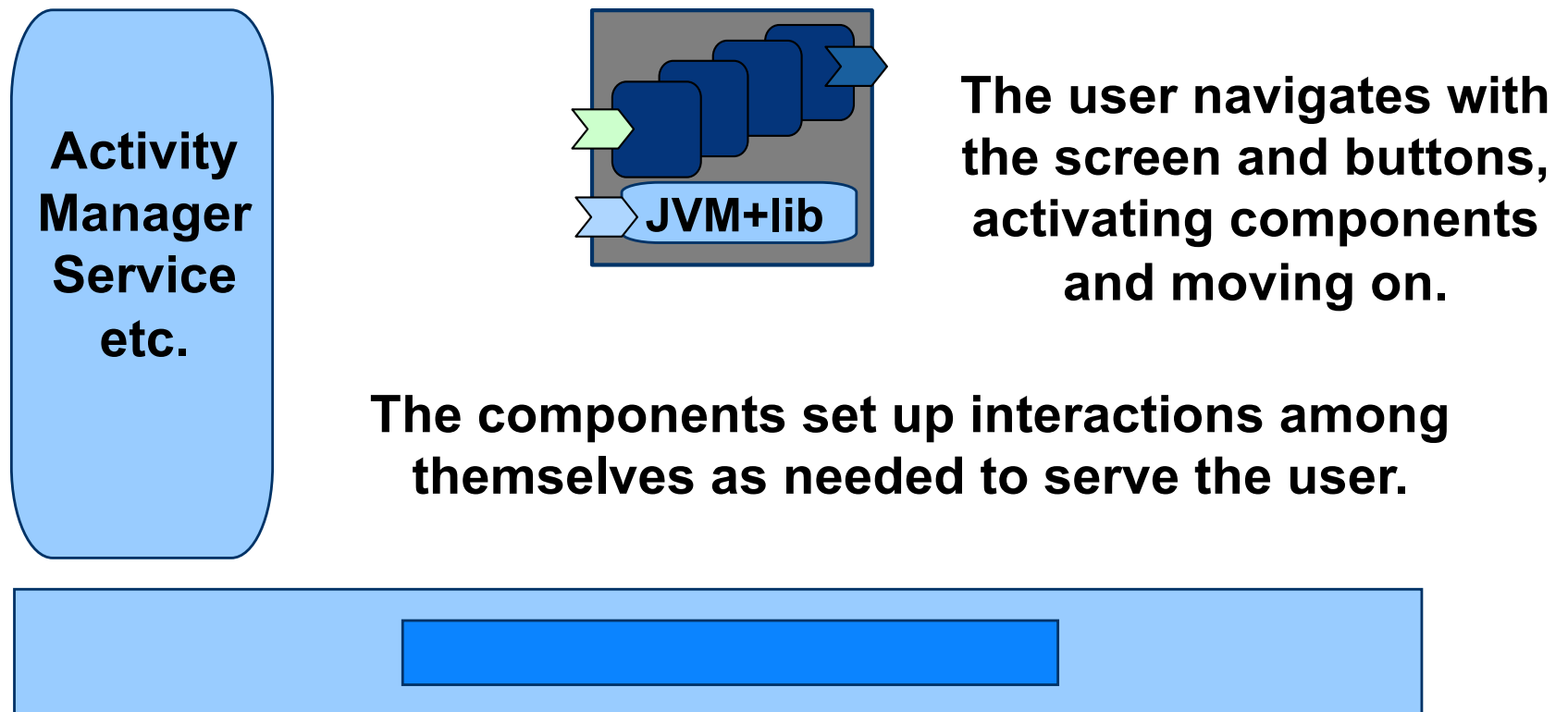
Server
listens for and
accepts clients,
handles requests,
sends replies

Binder: object-based RPC channels

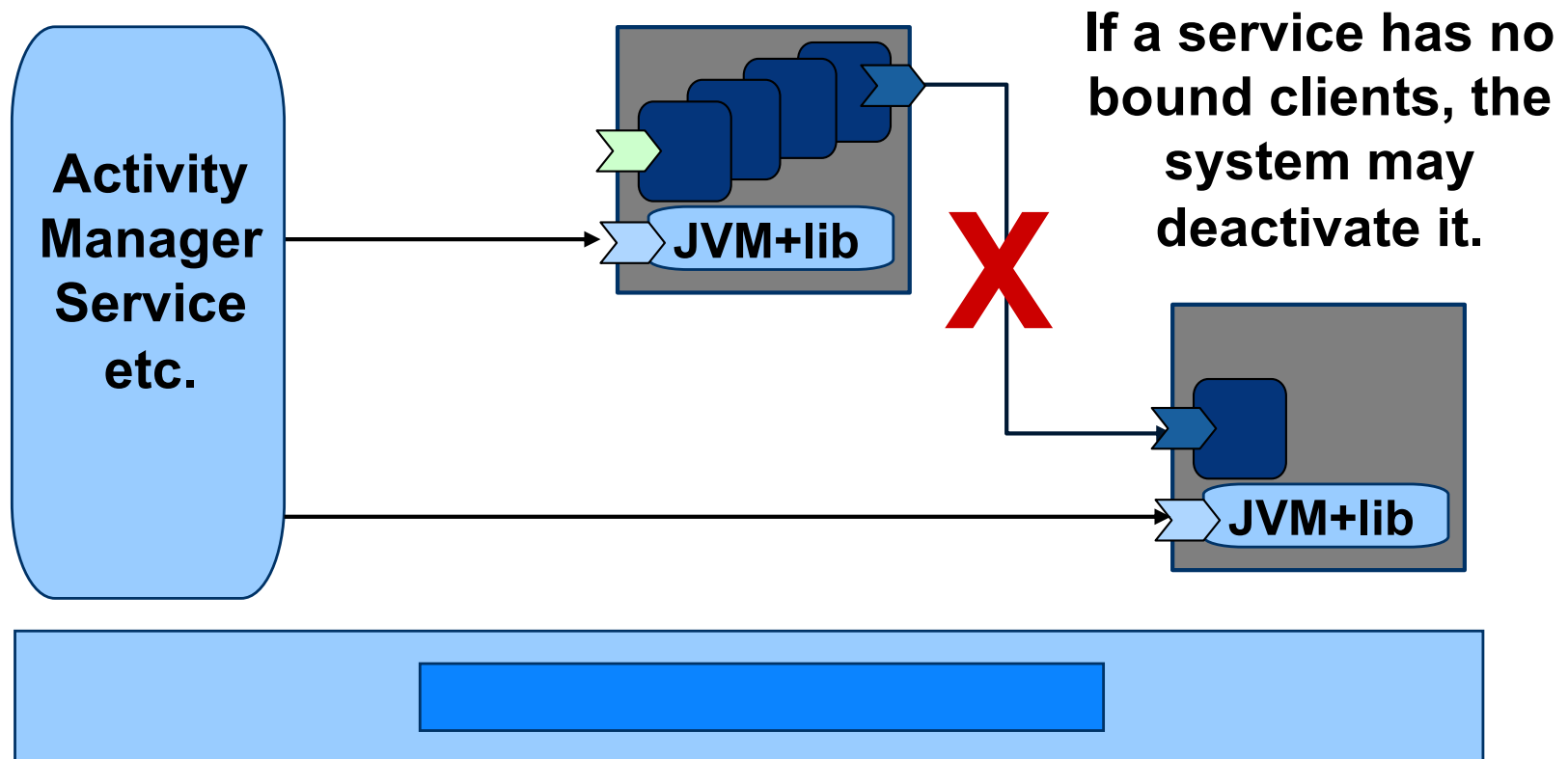


Android services and libraries communicate by sending messages through shared-memory channels set up by **binder**.

Deactivating components and apps

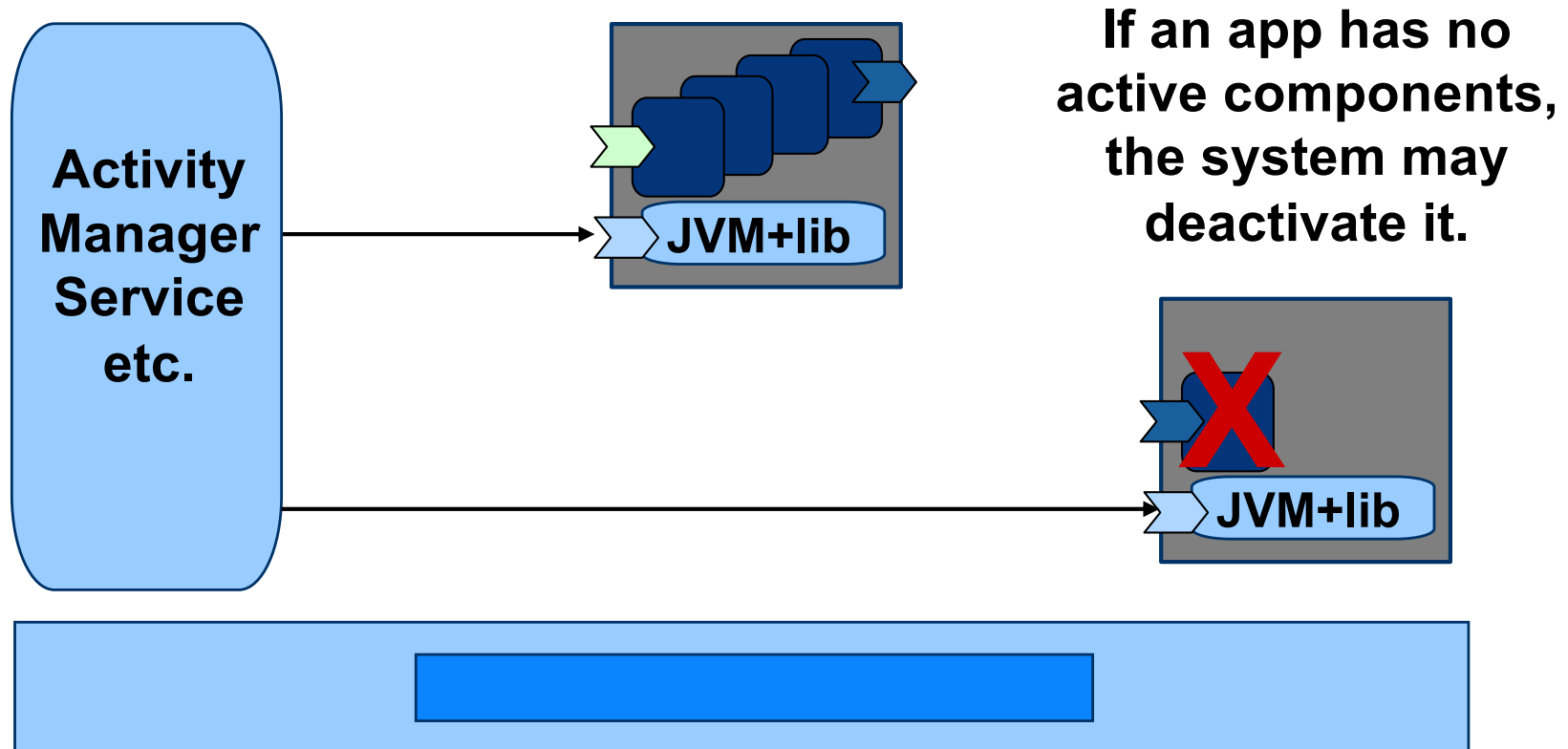


Deactivating components and apps



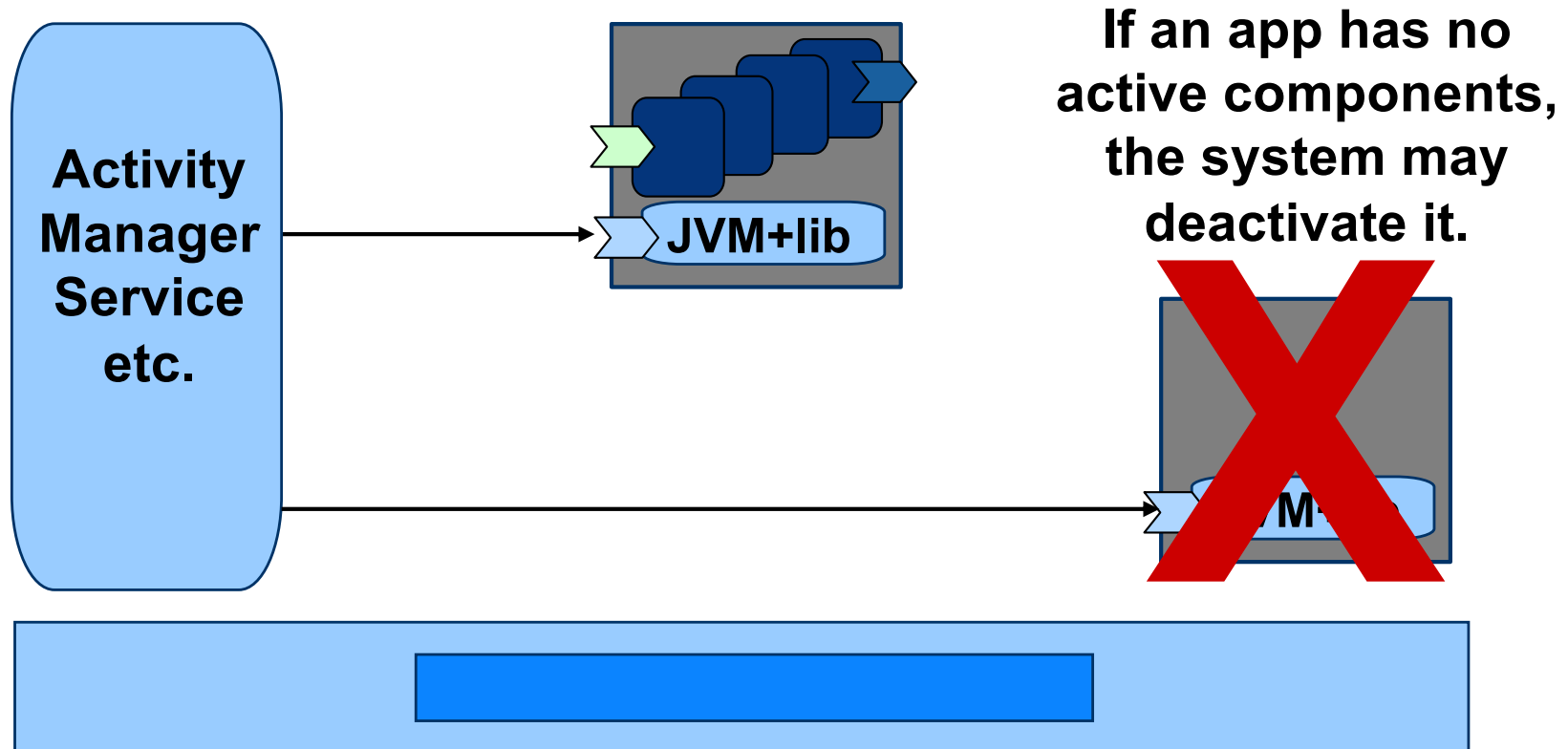
The Activity Manager decides when to deactivate components and tear down app contexts.

Deactivating components and apps



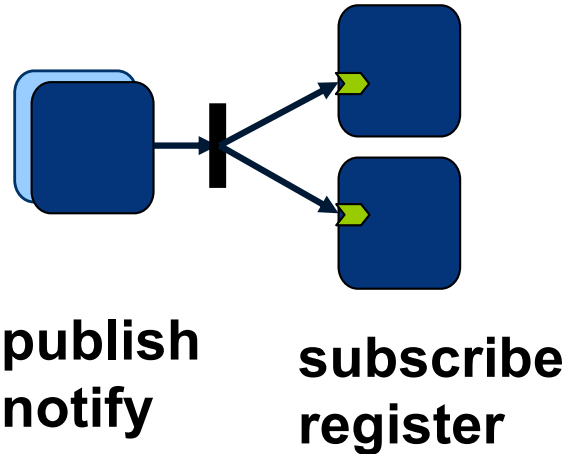
The Activity Manager decides when to deactivate components and tear down app contexts.

Deactivating components and apps



The Activity Manager decides when to deactivate components and tear down app contexts.

Concept: implicit invocation



Example: an Android **receiver** component can register to receive **broadcast intents** published by other components, or by the system.
E.g., ACTION_BATTERY_LOW

Event producers {**publish/raise/announce/signal**} events to notify {**consumers/receivers/subscribers**} of occurrences relating to a {**subject/topic/category**}.

The system invokes a registered **handler** method/procedure in each receiver. Delivery is **synchronous** or **asynchronous**.

Android: Intents

Intents are named events sent (“fired”) to components.

- Various system events can fire intents: tap on launch menu or app icon, change in device state, network connectivity, etc.
- An app component may fire an intent to request service from another.
- An intent may include data and may generate a result/response.
- **Every (external) component launch is triggered by an intent.**
- An Intent object includes an **action name** and optional data: URIs and/or key-value pairs (similar to environment variables).
- Some system APIs for apps to fire intents at one another:

[startActivity\(\)](#)
[startActivityForResult\(\)](#)
[onActivityResult\(\)](#)

[startService\(\)](#)
[bindService\(\)](#)
[sendBroadcast\(\)](#)
[sendOrderedBroadcast\(\)](#)

Intents

<https://developer.android.com/reference/android/content/Intent.html>

An intent is an **abstract description of an operation to be performed**. It can be used with `startActivity` to launch an Activity, `broadcastIntent` to send it to any interested BroadcastReceiver components, and `startService(Intent)` or `bindService(Intent, ServiceConnection, int)` to communicate with a background Service.

An Intent provides a facility for performing **late runtime binding between the code in different applications**. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed.

Intent filters and implicit intents

- An **explicit intent** names the target app+component directly. Else the intent is an **implicit intent**.
- Apps declare the intents that each component receives in **intent filters** in the app manifest.
- The system searches the filters for a suitable component(s) to receive an implicit intent: another example of implicit invocation.

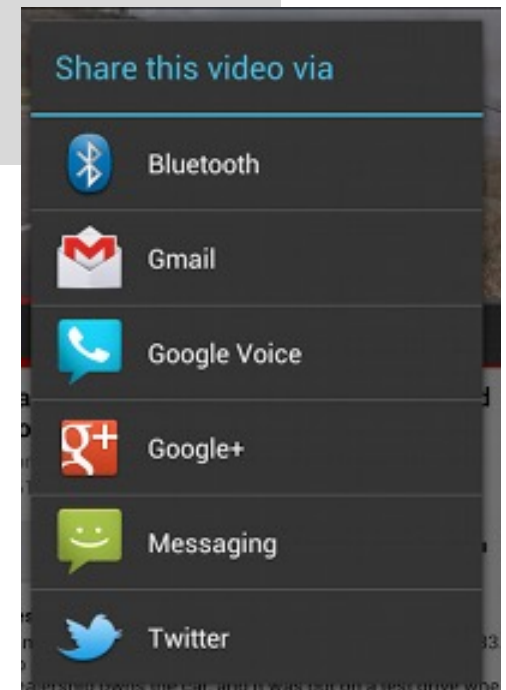
```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

Intent filters and implicit intents

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType(HTTP.PLAIN_TEXT_TYPE); // "text/plain" MIME type

// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

The sender of an implicit intent might also allow the user to **choose** the receiving app, by requesting the system to display a list of installed apps that handle that type of intent.



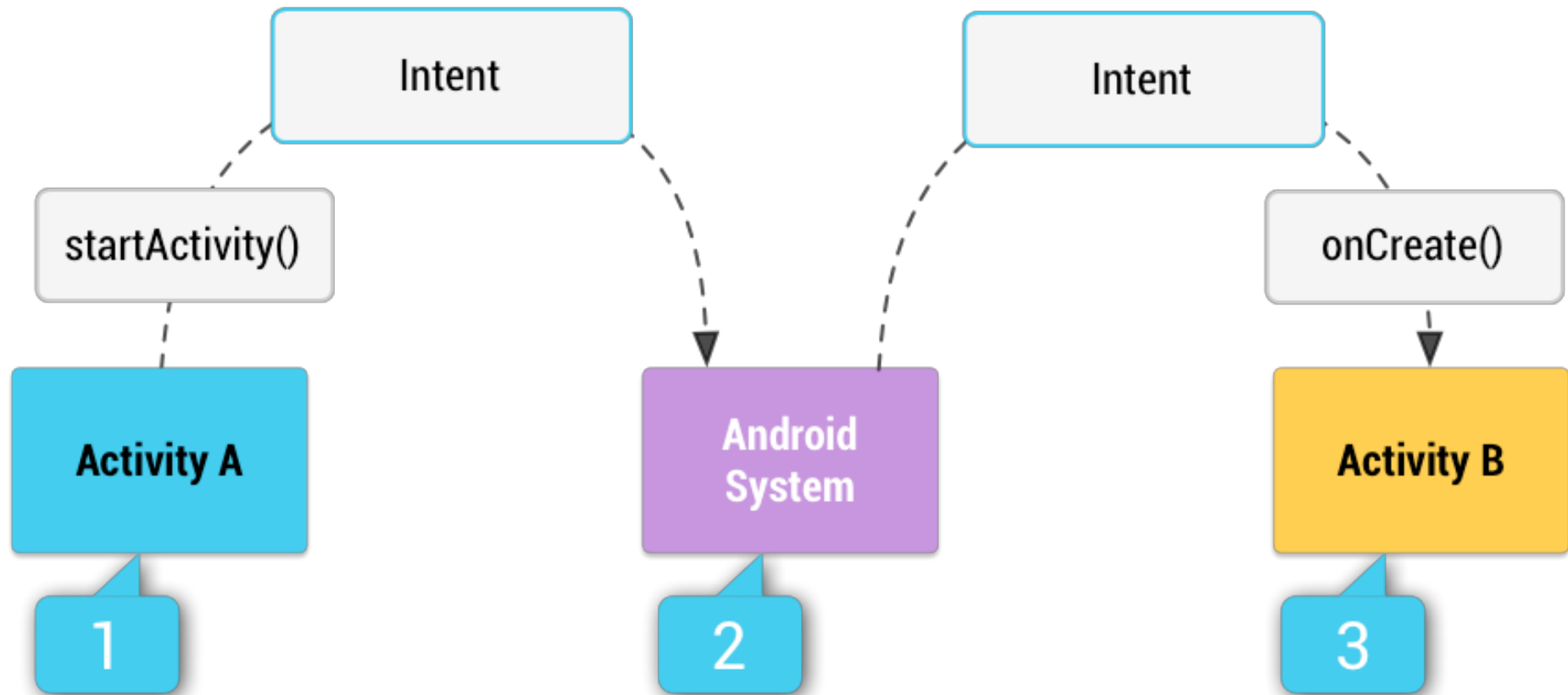


Figure 1. Illustration of how an implicit intent is delivered through the system to start another activity: **[1]** *Activity A* creates an [Intent](#) with an action description and passes it to [startActivity\(\)](#). **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its [onCreate\(\)](#) method and passing it the [Intent](#).

Intents: delivery

- Intent system is built above binder RPC, but intents indirect through the ActivityManagerService. (Once apps are bound, they can send binder RPCs directly to one another.)
- Incoming intents are (generally) handled by app's main (UI) thread.
- Components invoked via an intent are notified of the intent with an **upcall** to a standard method, which (generally) passes the intent.
- These standard component upcall methods are defined in the Android base class for the component type.
- Sender of an intent may wait for delivery to complete (and possibly return a result), or delivery may be **asynchronous** with sender.
- Some implicit intents are **broadcast intents** that may be delivered to multiple receivers, in parallel or in sequence.

Some standard intents

Constant	Target component	Action
<code>ACTION_CALL</code>	activity	Initiate a phone call.
<code>ACTION_EDIT</code>	activity	Display data for the user to edit.
<code>ACTION_MAIN</code>	activity	Start up as the initial activity of a task, with no data input and no returned output.
<code>ACTION_SYNC</code>	activity	Synchronize data on a server with data on the mobile device.
<code>ACTION_BATTERY_LOW</code>	broadcast receiver	A warning that the battery is low.
<code>ACTION_HEADSET_PLUG</code>	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
<code>ACTION_SCREEN_ON</code>	broadcast receiver	The screen has been turned on.
<code>ACTION_TIMEZONE_CHANGED</code>	broadcast receiver	The setting for the time zone has changed.

illustration only

Android service components

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

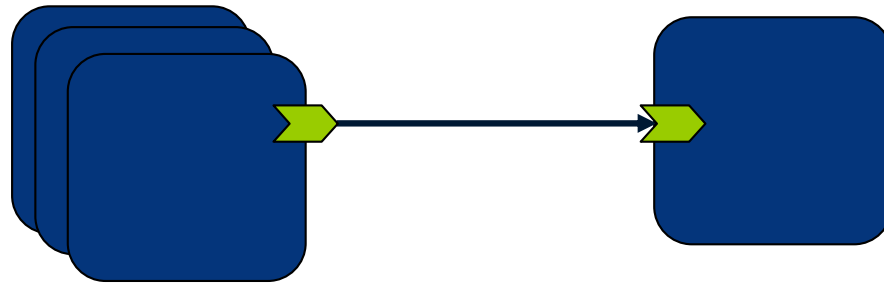
A Service is an application component that can perform long-running operations in the background and does not provide a user interface. For example, a service might ...play music, perform file I/O, or

Services may be “started” or “bound” or both:

Started. A service is "started" when an application component (such as an activity) starts it by calling [startService\(Intent,...\)](#). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

Bound. A service is "bound" when an application component binds to it by calling [bindService\(Intent,...\)](#). A bound service offers a client-server interface that allows components to interact with the service [to send requests and get results via binder RPC]. A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Binding to an Android service



public abstract boolean **bindService**
(Intent service, ServiceConnection conn, int flags)

Connect to an application service, creating it [instantiating the component] if needed. ...The given conn will receive the service [binder] object when it is created and be told [via upcalls] if it dies and restarts. ...

This function will throw `SecurityException` if you [the client app] do not have permission to bind to the given service.

Binding to an Android service



1. C uses **bindService** to fire an intent at S by its explicit name, passing C's ServiceConnection object **sc**.
2. Android forks a process for S if needed, and instantiates the Service component S within that process, if needed.
3. Android runtime in server (running on process main thread) receives the incoming intent.
4. If this is the first bound client of S, it upcalls the `onBind()` method of S, which returns a binder object **b** that can receive RPCs.
5. A reference to **b** is returned to C via a callback.
6. Android runtime upcalls C's **sc.onServiceConnected** on a binder thread, with a local "proxy" stub object referencing **b**.
7. Threads in C call methods on **b**, which may return other objects.

Concept: object-based RPC

- On the onBind() upcall, service returns an **object** reference. The object has an AIDL-defined interface.
- The client (via onServiceConnected) receives a stub/proxy object that “masquerades” as the remote object.
- Client invocations of the local proxy “become” RPC calls to the remote object.
- Other remotable (binder) objects may be passed as arguments to RPC calls, or returned as results.
- Allows applications to invoke one another’s objects in a flexible/transparent/safe way.
- [Example: SyncAdapter]

Structuring systems using upcalls

- Component instances have a **lifecycle**.
 - A sequence of events, e.g.: [birth, do-this, do-that, death]
- Lifecycle is (largely) managed by the system.
 - Components are acted upon by other components via intents.
 - And by the system: e.g., it decides when a component shall die.
 - These actions may cause lifecycle transitions.
 - Described by a finite state machine!
- Components are notified of lifecycle events and other operations by **upcalls**.
 - A component API method is invoked (on main thread) to notify the component and give it an opportunity to respond.
 - Implemented with MessageQueue/Handler under the hood.

Activity

- System **upcalls** component as its state changes due to user actions.
- An activity is **running** when it is in foreground, with a screen view facing the user.
- If another activity is started, the activity is **paused**.
- If a **paused** activity is not visible to the user, it is **stopped**.
- A **stopped** activity may be **destroyed**.
- An app process with no component instances may be killed.

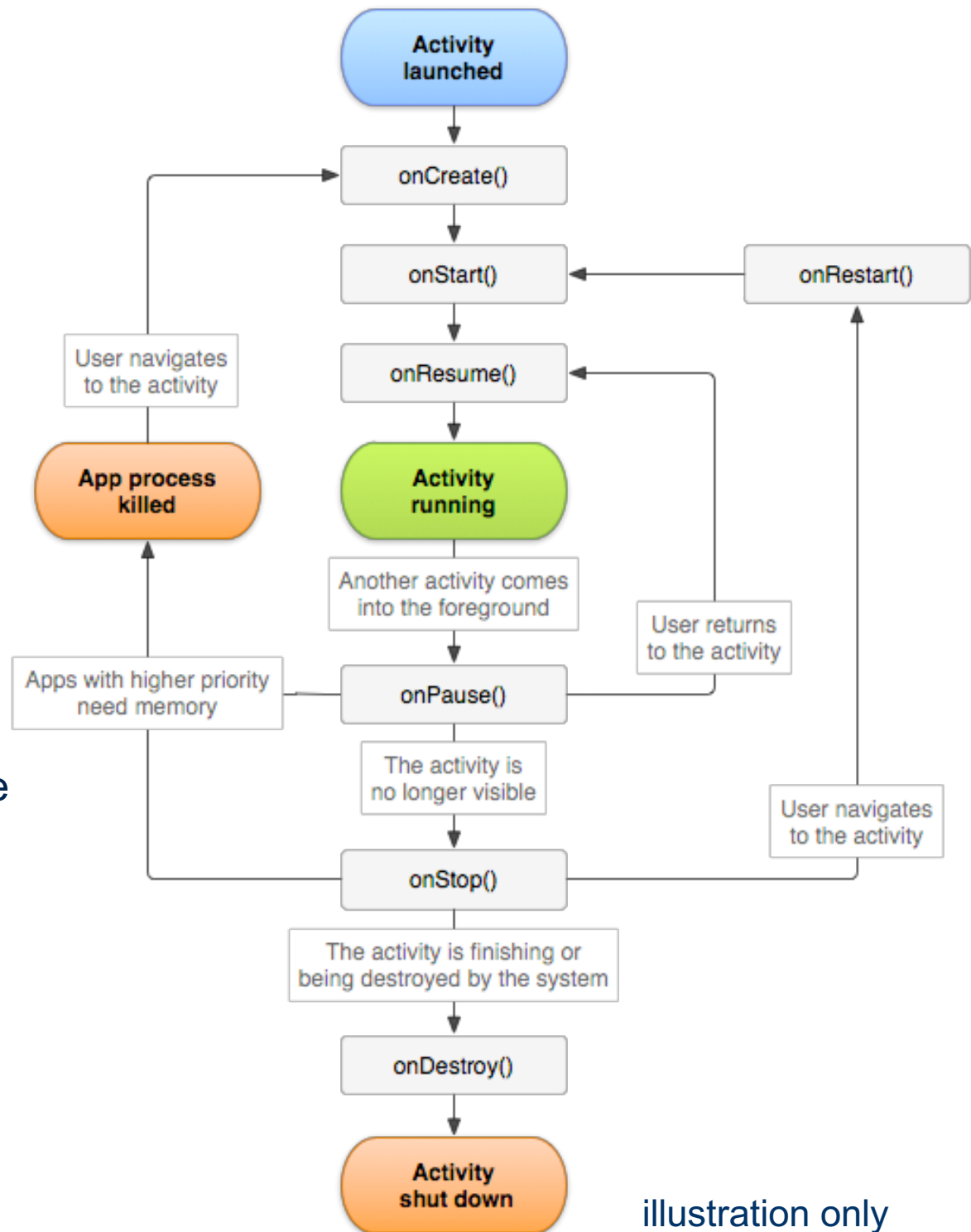


illustration only

Service

- Services may advertise one or more binder endpoints.
- Clients choose to bind/unbind (or unbind when stopped).
- A service with no bound clients may be shut down.

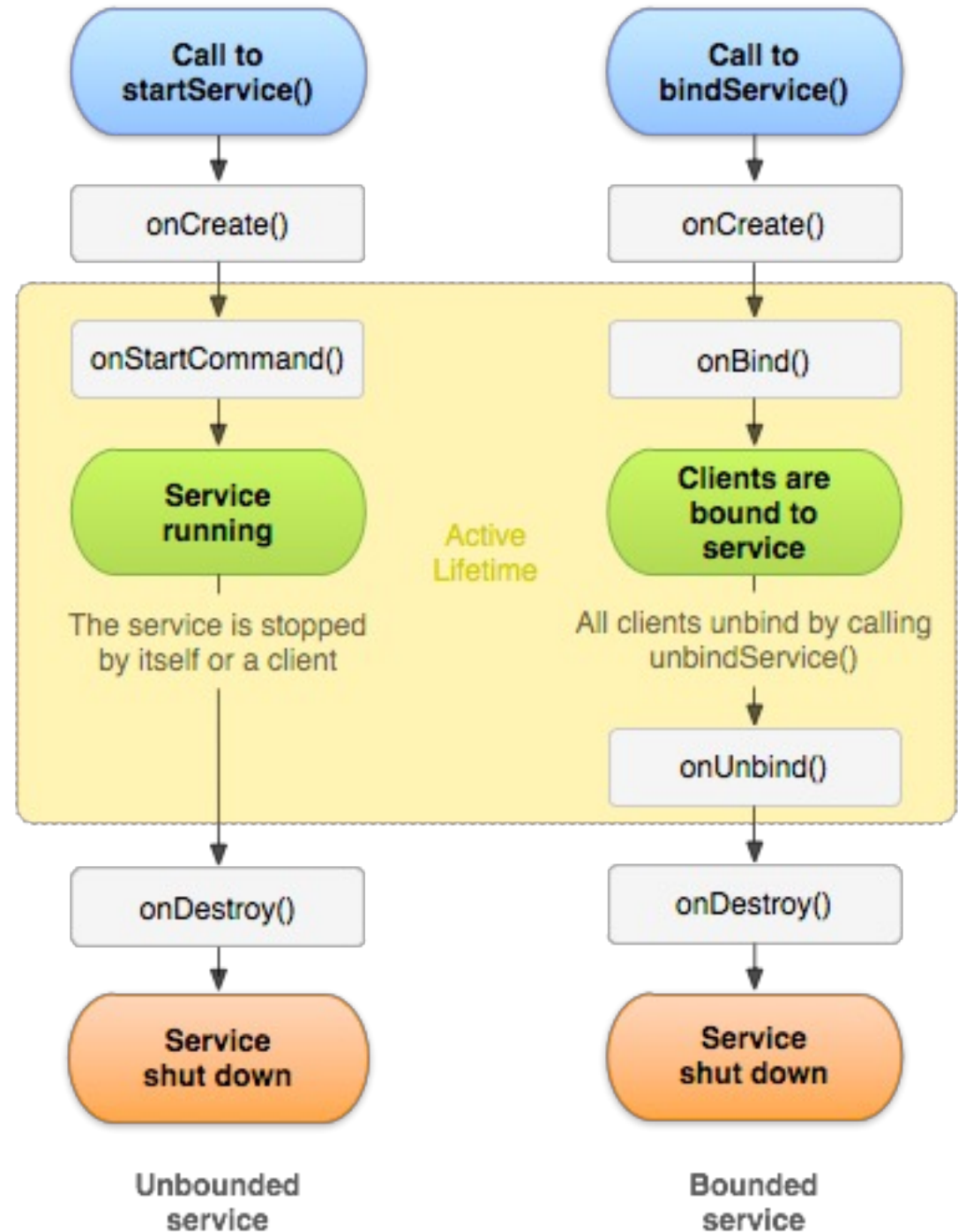
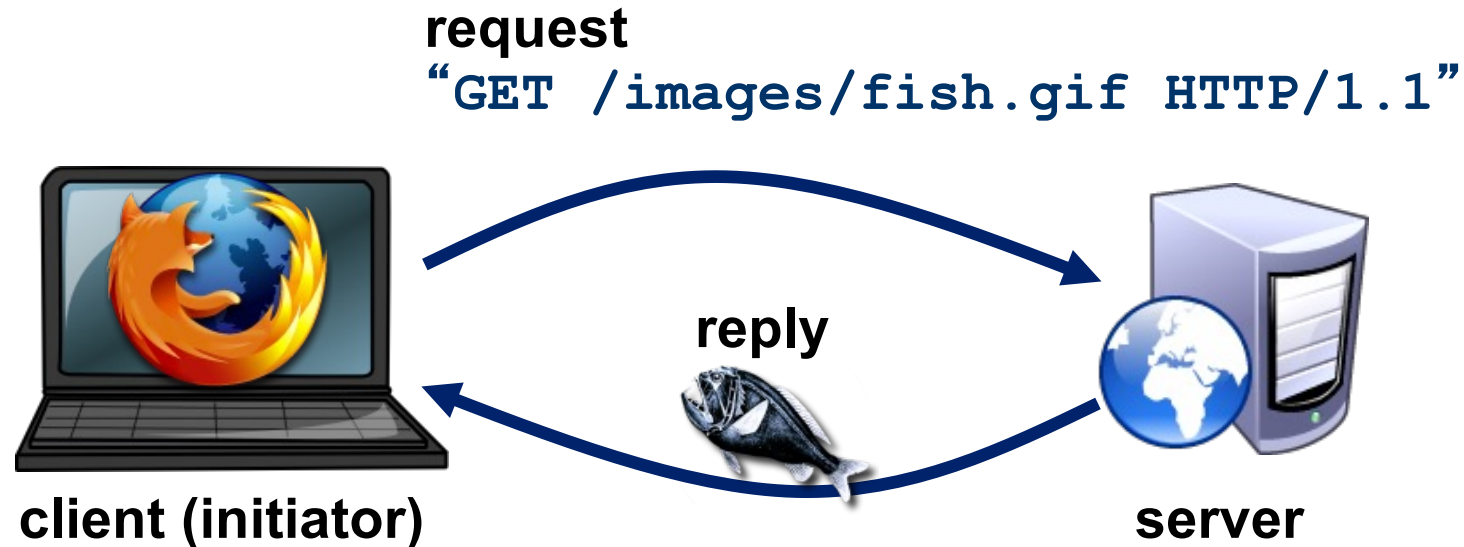


illustration only

Naming and protection for services

- Fundamental issue: **services require names**. Any cross-app interaction requires common names.
 - The server must advertise a name, and the client must know the server's name in order to contact it.
 - We also need some way to name the action requested.
 - In Android, we may also need to launch the component on-the-fly when an action is requested. So the system must be able to identify the component to launch.
- What about **protection**? Can any would-be client invoke an advertised service? Or can the server limit it?
- **These are fundamental questions**: Android is a nice example to illustrate some ways to handle these issues.

A simple, familiar example



```
sd = socket(...);  
connect(sd, name);  
write(sd, request...);  
read(sd, reply...);  
close(sd);
```

```
s = socket(...);  
bind(s, name);  
sd = accept(s);  
read(sd, request...);  
write(sd, reply...);  
close(sd);
```

Android protection model

- Each app runs with its own Linux userID.
 - Each app has a private space of files, processes, etc. that defines its “sandbox” and/or “lockbox”.
 - It does not matter that they run on behalf of the same user: each installed app receives its own userID.
- The system mediates app access to the sensors and UI.
 - GPS, camera, microphone, touch screen, etc.
 - Access requires permissions subject to user approval.
- Apps may interact (e.g., by invoking one another’s components) subject to permission checks.

Android: naming

- **Explicit intents** name the target component by its name.
- A component name is its Java **class name**.
- A fully qualified class name includes its **package name**.
- A fully qualified package name includes a DNS **domain name** of the originator, e.g., **com.example.ExampleActivity**
- Implicit intents name an **action** in a **category**. The sender need not know the app that receives it. Android defines various default actions, e.g.:

```
<intent-filter>  
  <action android:name="android.intent.action.SEND"/>  
  <category android:name="android.intent.category.DEFAULT"/>  
  <data android:mimeType="text/plain"/>  
</intent-filter>
```

Isolation and Sharing



Android Security Architecture

[<http://developer.android.com/guide/topics/security/permissions.html>]

“A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another application's files, performing network access, keeping the device awake, and so on.

Because each Android application operates in a **process sandbox**, applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities not provided by the basic sandbox. Applications statically declare the permissions they require, and the **Android system prompts the user for consent at the time the application is installed**. Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user experience to the detriment of security.”

Android permissions

- A **permission** is a named label, declared by the system or an app.
- Apps declare (in manifest) the permissions they want/require/use.
 - An app may request any declared permission by global name.
 - System grants requested permissions according to policy at app install time. Once installed the app's permissions don't change.
 - The system policy for granting a permission P depends on P's declared **protection level** (later).
- **Permissions protect interactions among apps:** each component declares the permissions its counterparties must possess.
 1. App A may declare an intent filter for its component C, and declare that any sender of the intent to C must have permission P.
 2. App A may fire an intent and declare that any component C that receives the intent must be part of an app that has permission P.

Permissions: examples

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"...>
  <uses-sdk android:minSdkVersion="14" />
  <uses-permission android:name="android.permission.READ_CALENDAR" />
  <uses-permission android:name="android.permission.WRITE_CALENDAR" />
  ...
</manifest>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.me.app.myapp" >
  <permission android:name="com.me.app.myapp.permission.DEADLY_ACTIVITY"
    android:label="@string/permlab_deadlyActivity"
    android:description="@string/permdesc_deadlyActivity"
    android:permissionGroup="android.permission-group.COST_MONEY"
    android:protectionLevel="dangerous" />
  ...
</manifest>
```

Granting permissions

- “At application install time, permissions requested by the application are granted to it by the package installer, based on checks against the signatures of the applications declaring those permissions and/or interaction with the user.”
- Each permission is either a system permission or it is from an app provider (whose key **signed** the app that declared it).
- The declaring app (or system) associates a **protection level** with the permission.
- The protection level drives system policy to grant permissions.
 - **normal**: granted on request
 - **dangerous**: requires user approval
 - **signature**: granted on request to apps signed by the same provider
 - **system**: granted only to apps installed on the system image

Content provider: permissions

Quote:

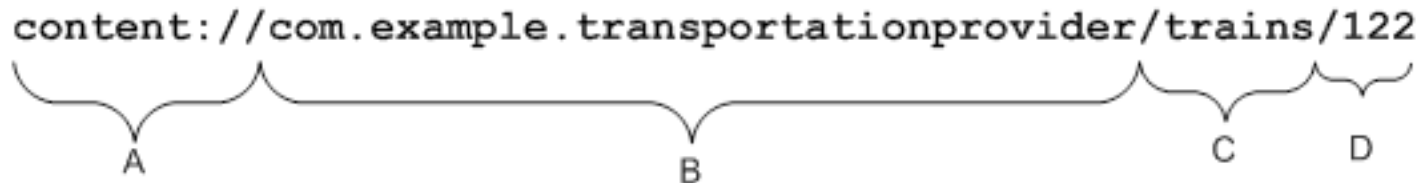
To retrieve data from a provider, your application needs "read access permission" for the provider. You can't request this permission at run-time; instead, you have to specify that you need this permission in your manifest, using the [<uses-permission>](#) element and the exact permission name defined by the provider. When you specify this element in your manifest, you are in effect "requesting" this permission for your application. When users install your application, they implicitly grant this request.

To find the exact name of the read access permission for the provider you're using, as well as the names for other access permissions used by the provider, look in the provider's documentation.

[<http://developer.android.com/guide/topics/providers/content-provider-basics.html>]

Content providers: naming and URIs

`content://com.example.transportationprovider/trains/122`

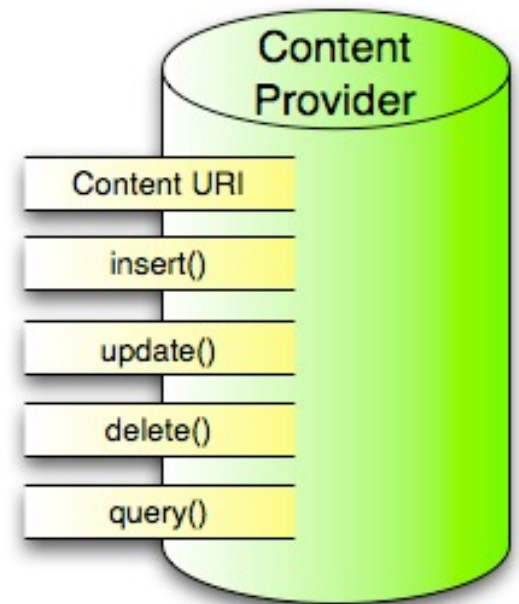


The diagram shows the URI `content://com.example.transportationprovider/trains/122` with four labels below it: A is under `content://`, B is under `com.example.transportationprovider`, C is under `trains`, and D is under `122`. Brackets connect each label to its corresponding part of the URI.

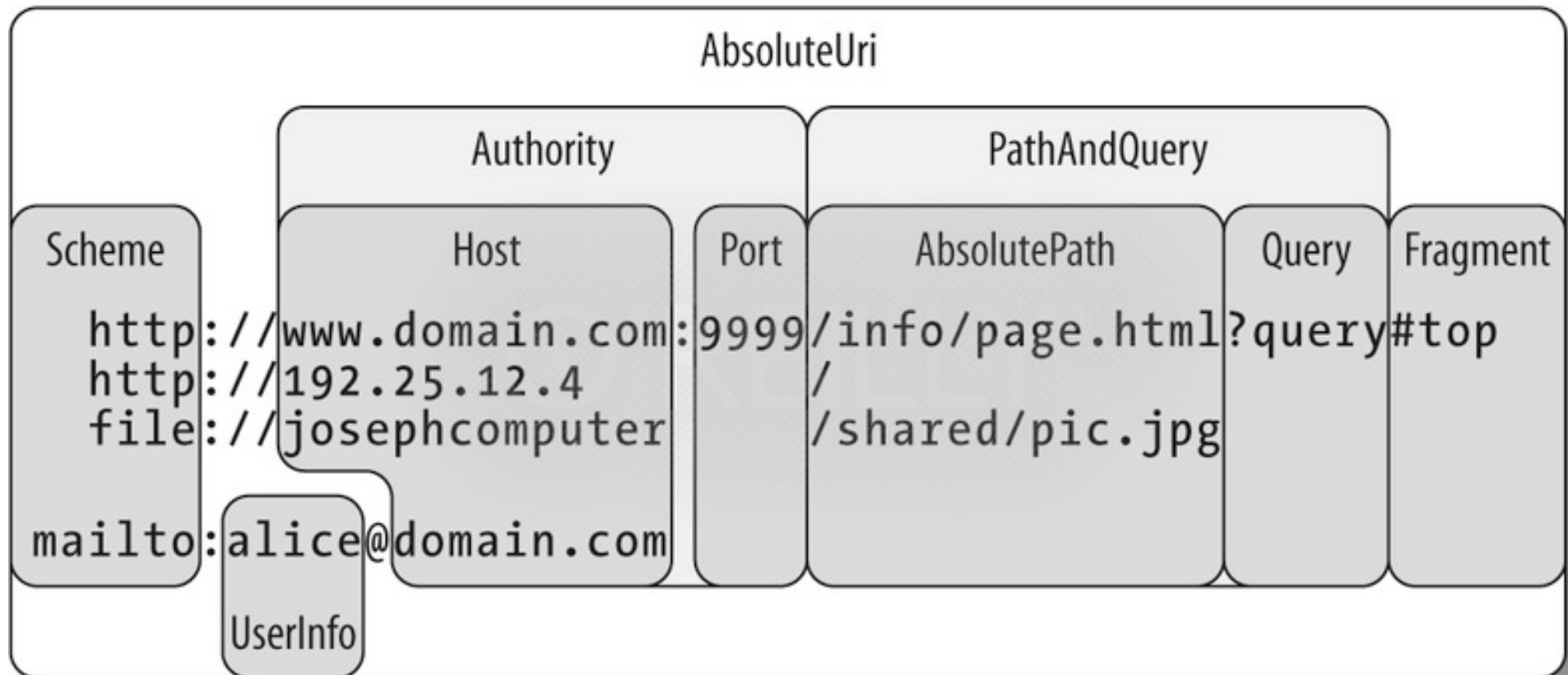
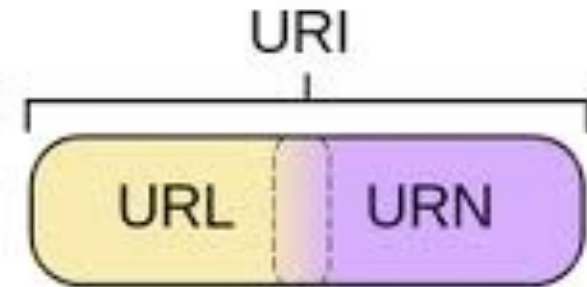
Define the provider's authority string, its content URIs, and column names....To avoid conflicts with other providers, you should use Internet domain ownership (in reverse) as the basis ...for Android package names...define your provider authority as an extension of the name of the package containing [it]...

Developers usually create content URIs from the authority by appending paths that point to individual tables...

By convention, providers offer access to a single row in a table by accepting a content URI with an ID value for the row at the end of the URI. ...



URIs and URLs

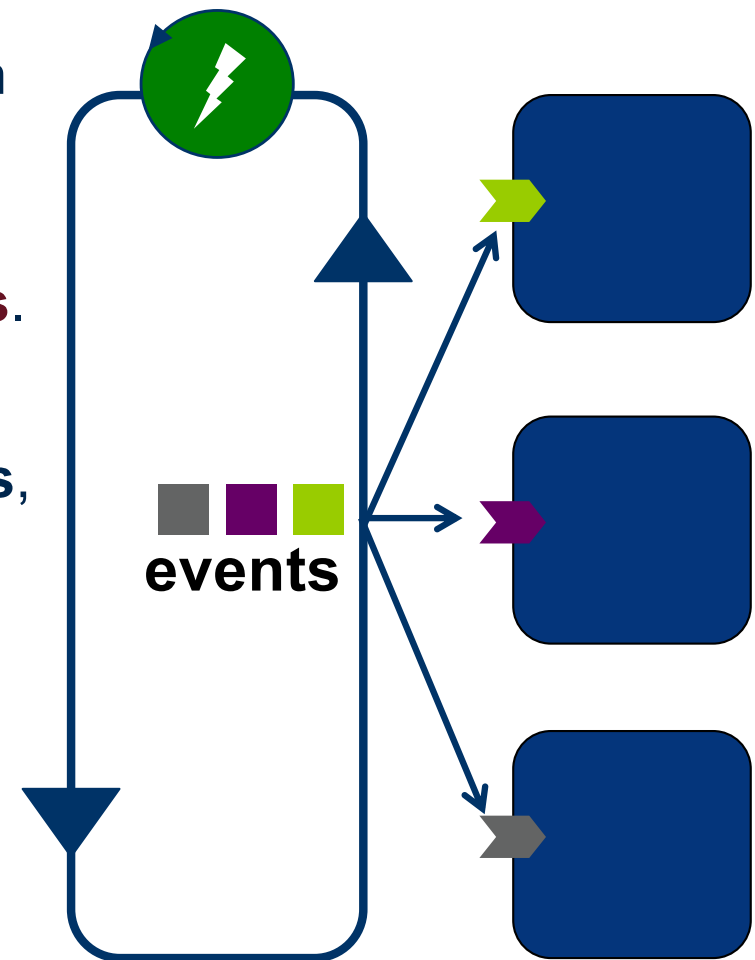


Android: what to know

- **Process model:** how app environment is built over Linux processes, using zygote, fork, and AMS subsystem.
- **RPC basics** as in binder: remote calls, stubs, proxy, threading model, kernel module, service binding by name, object-based RPC.
- **Components.** Names, cross-app invocation/integration, four kinds of components and their use. Secure cross-app launch via **intent** APIs.
- **Upcalls.** Components are event-driven: they are passive and receive upcalls on the app's main thread.
- **Lifecycle events.** Some upcalls come from the system to notify app of lifecycle events: activation/deactivation, bindings, etc.
- **Implicit invocation:** useful when an app doesn't know what other apps are installed or might be interested in an event. Examples: implicit intents, broadcast events.
- **Reference counting:** how Android tracks whether a component or app is active, enabling system to manage the app lifecycle.

Event-driven programming

- **Event-driven programming** is a design pattern for a thread's program.
- The thread receives and handles a sequence of typed **messages** or **events**.
 - Handle one event at a time, in order.
- In its **pure** form the thread **never blocks**, except to get the next event.
 - Blocks only if no events to handle (**idle**).
- We can think of the program as a set of **handler** routines for the event types.
 - The thread **upcalls** the handler to **dispatch** or “handle” each event.
- A handler should not block: if it does, the thread becomes unresponsive to events.



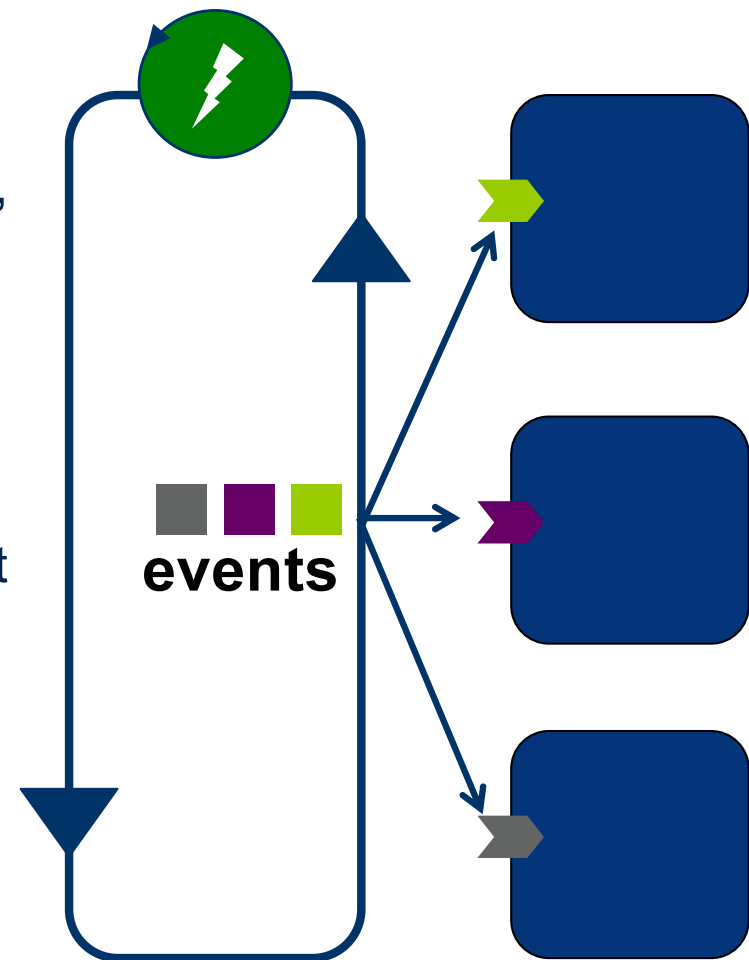
Dispatch events by invoking handlers (upcalls).

But what's an “event”?

- **A system can use an event-driven design pattern to handle any kind of asynchronous event.**
 - Arriving input (e.g., GUI clicks/swipes, requests to a server)
 - Notify that an operation started earlier is complete
 - E.g., I/O completion
 - Subscribe to events published/posted by other threads
 - Including status of children: stop/exit/wait, signals, etc.
- You can use an “event” to represent **any kind of message that drives any kind of action** in the receiving thread.
 - **In Android:** UI events, binder RPC, intents
- **But the system must be designed for it**, so that operations the thread requests **do not block**; the request returns immediately (“asynchronous”) and delivers a **completion event** later.

Android: threading model

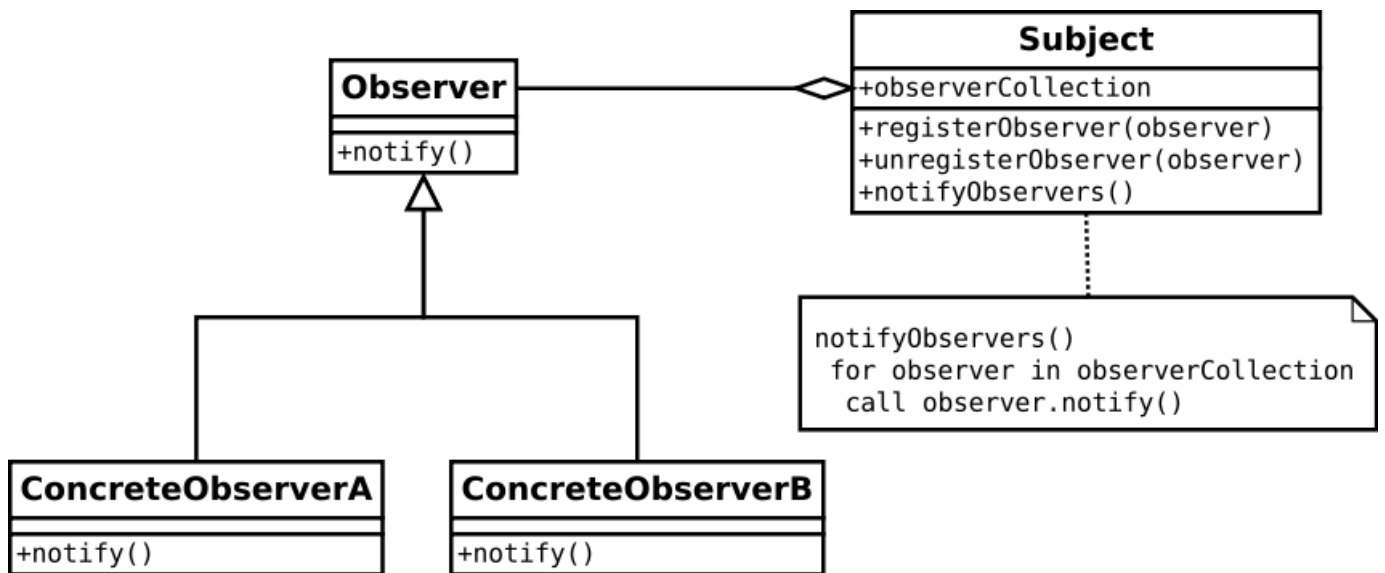
- An app is launched as a process when any of its components is first instantiated.
- The process **main thread** is event-driven, e.g., by User Interface (UI) events.
 - Also called the “**UI thread**”.
 - UI toolkit code is not **thread-safe**, so it should execute only on the UI thread.
 - UI thread should not block (except for next event), or app becomes unresponsive.
- UI thread also receives incoming events or calls (intents and **upcalls**), and launches and tears down components.
- An app may spawn other **background threads** (workers) for other uses.
- Binder RPC manages a **thread pool** per app/process.



Observer pattern

- The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- After notification, each observer carries out its task via a separate thread as to prevent blocking. It is mainly used to implement distributed event handling systems.

Observer pattern



Observer pattern

```
import java.util.Observable;
import java.util.Scanner;

class EventSource extends Observable implements Runnable {
    public void run() {
        while (true) {
            String response = new Scanner(System.in).next();
            setChanged();
            notifyObservers(response);
        }
    }
}
```

```
import java.util.Observable;
import java.util.Observer;

public class MyApp {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        eventSource.addObserver(new Observer() {
            public void update(Observable obj, Object arg) {
                System.out.println("Received response: " + arg);
            }
        });

        new Thread(eventSource).start();
    }
}

/*
interface Observer{
    public void update(Observable obj, Object arg);
}*/
```

Threads in Android

Three examples/models for use of threads in Android.

1. **Main thread (UI thread):** receives UI events and other upcall events on a single incoming message queue. Illustrates event-driven pattern: thread blocks only to wait for next event.
2. **ThreadPool:** an elastic pool of threads that handle incoming calls from clients: Android supports “binder” request/response calls from one application to another. When a request arrives, a thread from the pool receives it, handles it, responds to it, and returns to the pool to wait for the next request.
3. **AsyncTask:** the main thread can create an AsyncTask thread to perform some long-running activity without blocking the UI thread. The AsyncTask thread sends progress updates to the main thread on its message queue.

These patterns are common in many other systems as well.

Adapted from Android Developer Guide

Summary: By default, all components of the same application run in the same process and thread (called the "main" thread). The main thread controls the UI, so it is also called the UI thread. If the UI thread blocks then the application stops responding to the user. You can create additional background threads for operations that block, e.g., I/O, to avoid doing those operations on the UI thread. The background threads can interact with the UI by posting messages/tasks/events to the UI thread.

Details: When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution called the **main thread**.

All components that run in the same process are initialized by its main thread, and system upcalls to those components (onCreate, onBind, onStart,...) run on the main thread.

The main thread is also called the **UI thread**. It is in charge of dispatching events to user interface widgets and interacting with elements of the Android UI toolkit. For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, to set its pressed state and redraw itself.

If you have operations that might require blocking, e.g., to perform I/O like network communication or database access, you should run them in separate threads. A thread that is not the UI thread is called a **background thread** or "worker" thread.

Adapted from Android Developer Guide

Your app should never block the UI thread. When the UI thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to “hang” or “freeze”.

Even worse, if the app blocks the UI thread for more than a few seconds, Android presents the user with the infamous "**application not responding**" (ANR) dialog. The user might then decide to quit your application and uninstall it.

In a correct Android program the UI thread blocks only to wait for the next event, when it has nothing else to do (it is idle). If you have an operation to perform that might block for any other reason, then you should arrange for a background/worker thread to do it.

Additionally, the Android **UI toolkit is not thread-safe**: if multiple threads call a module that is not thread-safe, then the process might crash. A correct app manipulates the user interface only from a single thread, the UI thread. So: your app must not call UI widgets from a worker thread.

So how can a worker thread interact with the UI, e.g., to post status updates? Android offers several ways for a worker to post operations to run on the UI thread.

Note: this concept of a single event-driven main/UI thread appears in other systems too.

Android UI toolkit is not thread-safe

So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Thus, there are simply two rules to Android's single thread model:

- Do not block the UI thread
- Do not access the Android UI toolkit from outside the UI thread

Better performance through threading

<https://developer.android.com/topic/performance/threads>

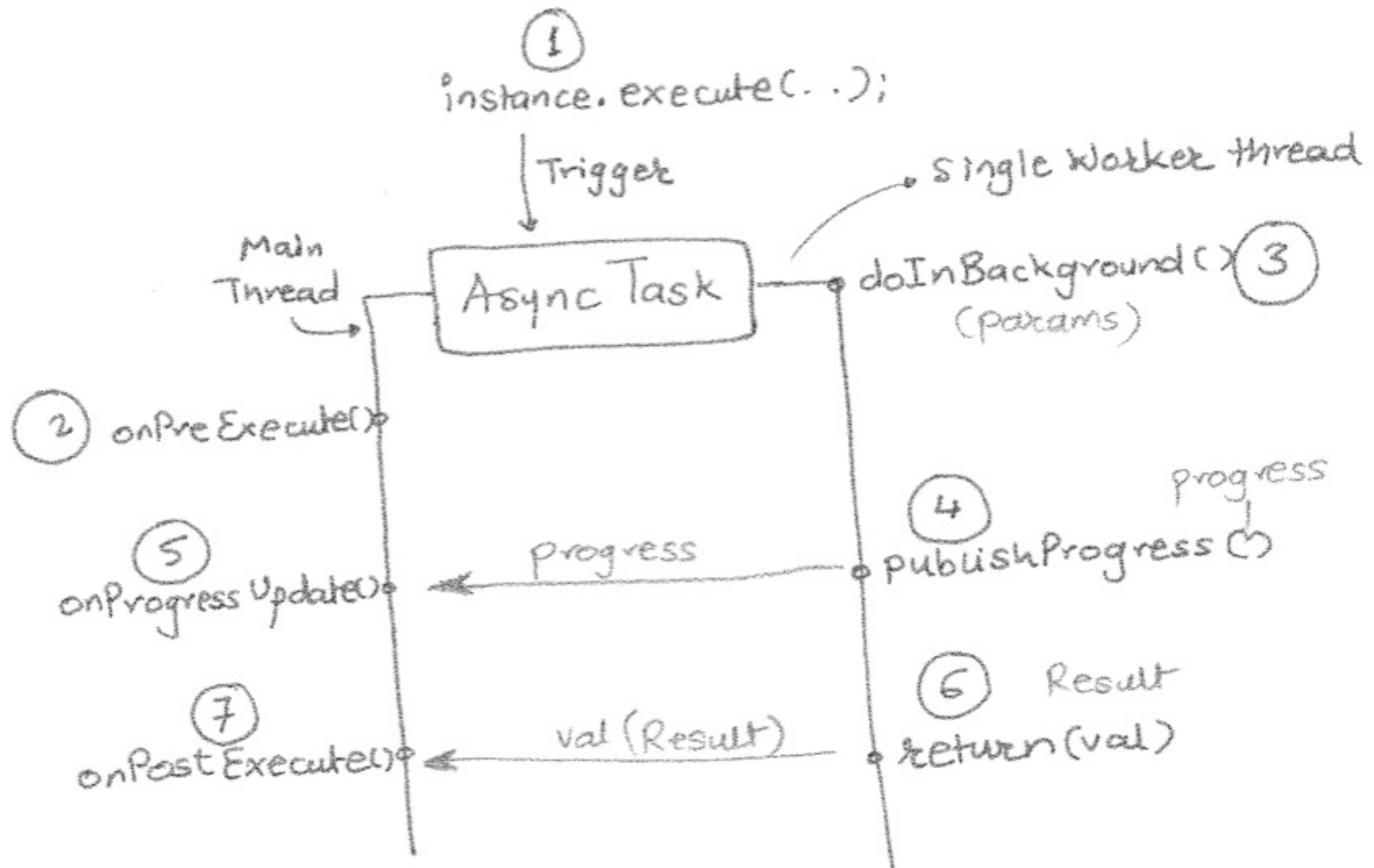
When the user launches your app, Android creates a new Linux process along with an execution thread. This main thread, also known as the UI thread, is responsible for everything that happens onscreen.

Nearly any block of code your app executes is tied to an event callback, such as input, layout inflation, or draw. When something triggers an event, the thread where the event happened pushes the event out of itself, and into the main thread's message queue. The main thread can then service the event.

Some Guidelines for Main Thread

- If the main thread cannot finish executing blocks of work within 16ms, the user may observe hitching, lagging, or a lack of UI responsiveness to input. If the main thread blocks for approximately five seconds, the system displays the Application Not Responding (ANR) dialog, allowing the user to close the app directly.

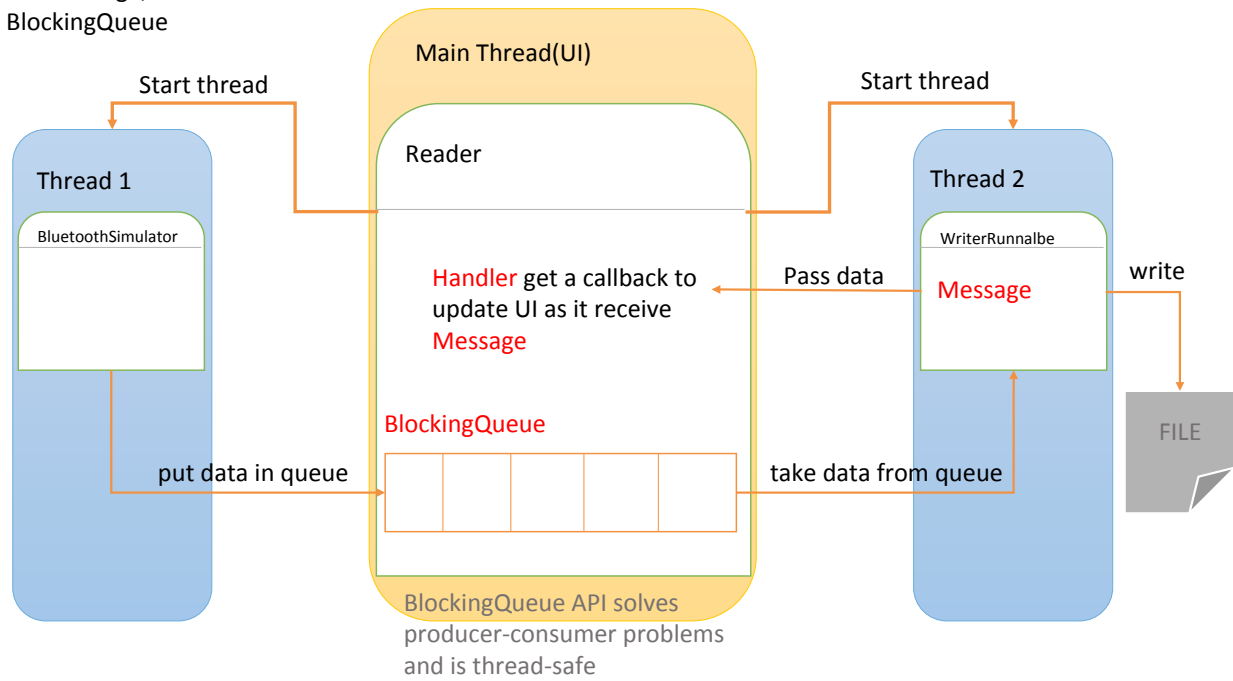
Android: AsyncTask



Challenge

- Classical Producer–consumer problem
 - The producer and the consumer share a common, fixed-size buffer used as a queue.
- Need to:
 - Make sure the producer won't try to add data into the buffer if it's full
 - Make sure the consumer won't try to remove data from an empty buffer
- Solution: Java concurrent API for Producer–consumer problem
 - BlockingQueue interface
 - BlockingQueue implementation is thread-safe
 - put()/take(): the method blocks the current thread until the operation can succeed

NOTE: API is marked as red
Android API: Message, Handler
Java API: BlockingQueue



Helper classes for threading - AsyncTask

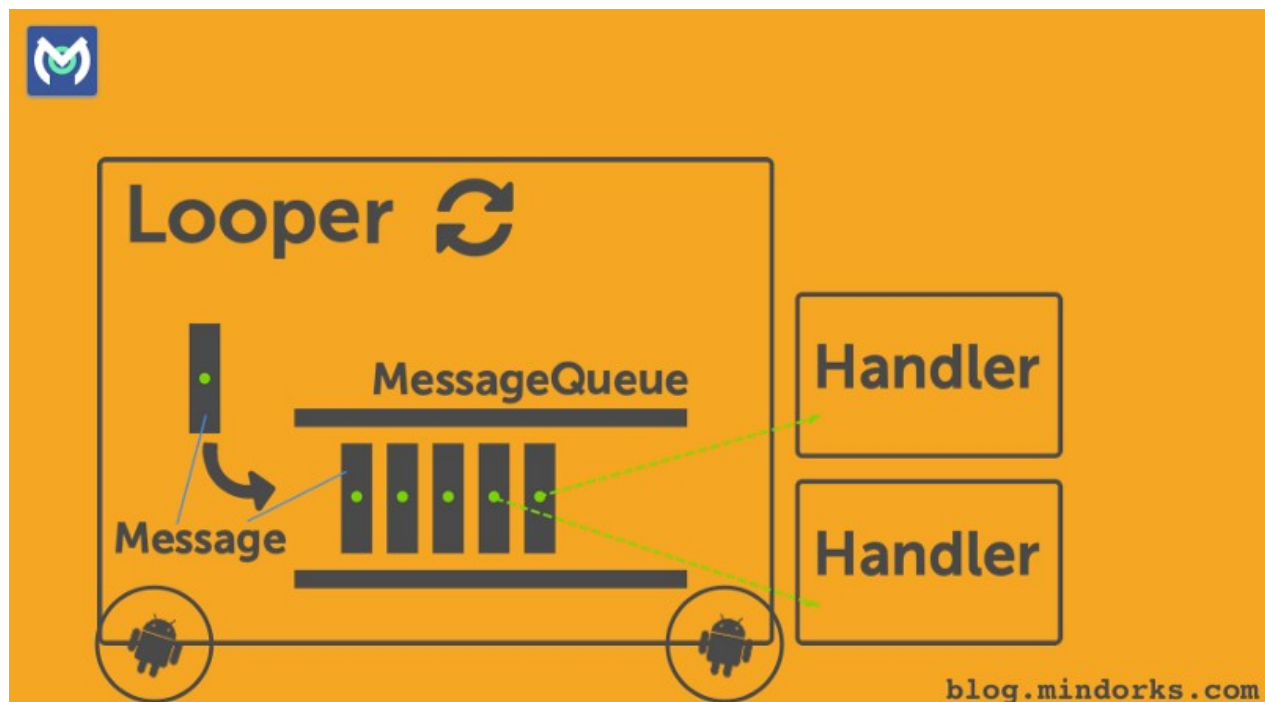
The AsyncTask class is a simple, useful primitive for apps that need to quickly move work from the main thread onto worker threads. For example, an input event might trigger the need to update the UI with a loaded bitmap. An AsyncTask object can offload the bitmap loading and decoding to an alternate thread; once that processing is complete, the AsyncTask object can manage receiving the work back on the main thread to update the UI.

Helper classes for threading – Handler Thread

Consider a common challenge with getting preview frames from your Camera object. When you register for Camera preview frames, you receive them in the `onPreviewFrame()` callback, which is invoked on the event thread it was called from. If this callback were invoked on the UI thread, the task of dealing with the huge pixel arrays would be interfering with rendering and event processing work. This is a situation where a handler thread would be appropriate: A handler thread is effectively a long-running thread that grabs work from a queue, and operates on it.

This Article covers Android Looper, Handler, and HandlerThread.

<https://blog.mindorks.com/android-core-looper-handler-and-handlerthread-bd54d69fe91a>



Terminologies

The above model is implemented in the Android via `Looper` , `Handler` , and `HandlerThread` . The System can be visualized to be a vehicle as in the article's cover.

1. `MessageQueue` is a queue that has tasks called messages which should be processed.
2. `Handler` enqueues task in the `MessageQueue` using `Looper` and also executes them when the task comes out of the `MessageQueue` .
3. `Looper` is a worker that keeps a thread alive, loops through `MessageQueue` and sends messages to the corresponding `handler` to process.
4. Finally `Thread` gets terminated by calling `Looper`'s `quit()` method.

Handler vs HandlerThread

```
class LooperThread extends Thread {  
    public Handler mHandler;  
  
    public void run() {  
        Looper.prepare();  
  
        mHandler = new Handler() {  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
                // this will run in non-ui/background thread  
            }  
        };  
  
        Looper.loop();  
    }  
}
```

```
private class MyHandlerThread extends HandlerThread {  
  
    Handler handler;  
  
    public MyHandlerThread(String name) {  
        super(name);  
    }  
  
    @Override  
    protected void onLooperPrepared() {  
        handler = new Handler(getLooper()) {  
            @Override  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
                // this will run in non-ui/background thread  
            }  
        };  
    }  
}
```

Android has provided `HandlerThread` (subclass of `Thread`) to streamline the process. Internally it does the same things that we have done but in a robust way. So, always use `HandlerThread`.