CS 478: Software Development for Mobile Platforms

Set 4: Permissions and Broadcast Receivers

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

February 28, 2019

Android's security model

- Hard to secure OS with open source
  - Open source is great for teaching, extensibility…
  - ... but not so great for security
  - "Hackers" can look at source code and find security holes much more easily than the case of closed-source OSes (e.g., iOS)

    Just diff latest version of Android with previous version to identify security patches, then go after devices running the older version...

1

Android's security model

- Key Android principle: Each app runs as its process with distinct identity + user name
- Main security components:
  - The app *Sandbox* (restrict access to data, h/w and s/w components)
  - Secure IPC (communication between processes/apps managed by OS)
  - App signing (identify app authors)
  - Permission structure (strict rules when app operates out of sandbox)

2

Android's security model (cont'd)

- App operates in sandbox (with its own process id)
  - Each app has its own PID
  - No direct access to data (e.g., files) owned by another application, unless sharing explicitly allowed

    (Use attribute *android:sharedUserId* if id sharing among apps desired)

    If desired, app can make file world readable/writable upon creation
  - No default access to hardware resources (e.g., GPS receiver, camera, microphone, network, etc.)
  - Sandbox implemented at Linux kernel level: Entire OS stack covered!
    - Native C/C++ code run by an app covered as well
- Doc: https://source.android.com/security/

3

Why the sandbox?

- Apps can do sensitive things on their hosting devices, e.g.,
  - Read and write personal data —

    Contacts, phone history, calendar, phone book, browsing history, etc.
  - Connect to the internet —

    Browsing, email, etc.
  - Take pictures and video, record conversations
  - Get/disseminate the device's location
  - Make calls, send texts
- All these things can be abused by malicious apps
  - Exfiltrate personal or location information, access pictures, etc.

4

Android permissions

- **Permissions allow an app to operate outside its sandbox**
- Permissions control access to sensitive resources, e.g.,
  1. Sensitive services — Phone app, sms/mms, etc.
  2. Private user information — Calendar, email, contacts, file system, bookmarks, browsing histories, etc.
  3. Hardware resources — Camera, vibrator, battery, microphone, etc.
- Android's philosophy: User must grant an app *permission* to do sensitive things on the user's device
  - User must judge whether app is entitled to certain actions or not
  - Application must declare permissions it needs in manifest files
- http://developer.android.com/guide/topics/security/permissions.html

5

Android permissions

- Permissions specified as strings in manifest files
- Long list of predefined permissions

  http://developer.android.com/reference/android/Manifest.permission.html
- Programmer can define new permissions

6

Permission *risk levels*

- Risk level associated with a permission
  - *Normal* – Associated with "low risk" operations; declared in manifest but **automatically granted** without user input
    - Examples: Check Wi-Fi status, use Internet (?!), vibrate device
    - https://developer.android.com/guide/topics/security/normal-permissions.html
  - *Dangerous* – Permission covers key resource; **user must grant**
  - *Signature* – Requesting app must be signed with same certificate as app that defined the permission
    - **Automatically granted** if apps have same signature
    - Useful for ad-hoc components (e.g., database clients and server)
- Default is normal

7

Dangerous permissions

Different process for granting depending permissions depending on OS version

- New model (MM and later):
  1. Permission declared in AndroidManifest.xml file, and
  2. Running app requests permission when operation needing permission is about to take place
- New model used when:
  1. Hosting (user) device runs MM or later OS (Nougat, Oreo, Pie); and
  2. App compiled with targetSdkVersion ≥ 23

8

Permission granting process

- Old model: Permission requested when app installed on device, never asked again
- Old model used either if:
  1. Device runs Lollypop or earlier version, or
  2. App's targetSdkVersion < 23

Observations

- Pre-MM permissions cannot be revoked, MM+ permission can be revoked with *Settings→Apps*
- Pre-MM permissions are all-or-nothing, all granted or app not installed
- Post-MM permissions are selective
  - ➢ User may grant some but not all permissions requested by app

9

Permission granting process (cont'd)

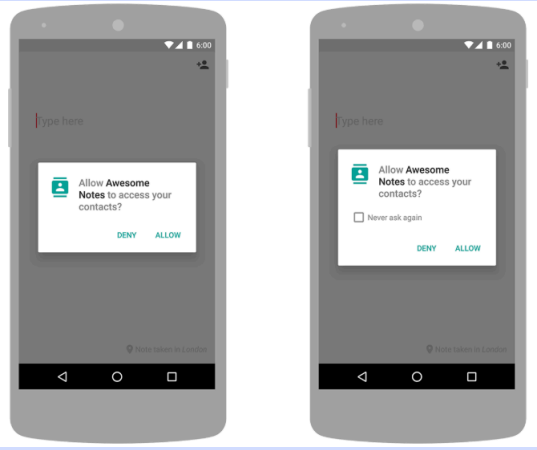
- New model is similar to iOS permission model
- Greater programming effort involved in new model
  - For each action requiring a dangerous permission must code:
    1. User request dialog,
    2. Processing of user response, and
    3. Different actions depending on user response
- Moral of the story: Make life better for app user, more difficult for hard-working Android developer…

10

Post-MM permission dialogs

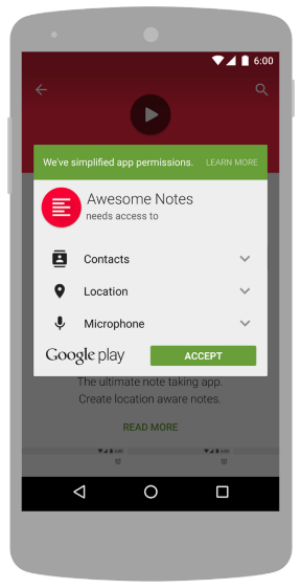
- Secondary dialog displayed if user previously denied permission to app

  Source: https://developer.android.com/guide/topics/permissions/overview



11

Pre-MM permission dialogs



Dialog displayed at app install time

App installed only if user grants all requested permissions

Source: https://developer.android.com/guide/topics/permissions/overview

12

Permission groups

- Permissions are normally fine-grained, e.g.,
  - READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE are distinct permissions
- *Permission groups*: Avoid too many authorization requests by grouping permissions, working with groups instead of individual permissions
  - E.g., Read/write external storage permissions belong to same group
- Consequence 1 (Marshmallow + target ≥ 23): App will request permission *P* only if it does not already have a permission in the same group as *P*
- Consequence 2 (version ≤ Lollypop) or (target ≤ 22): App will request only permission groups at install time
- Doc: http://developer.android.com/guide/topics/security/permissions.html

13

Examples of dangerous permission groups

- https://developer.android.com/reference/android/Manifest.permission.html

GROUP	INDIVIDUAL PERMISSIONS
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE, CALL_PHONE, READ_CALL_LOG, USE_SIP, WRITE_CALL_LOG, ADD_VOICEMAIL, PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS, RECEIVE_SMS, READ_SMS, RECEIVE_MMS, RECEIVE_WAP_PUSH
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

14

Listing of normal and dangerous permissions

Normal:
https://developer.android.com/guide/topics/permissions/normal-permissions.html

Dangerous:
https://developer.android.com/guide/topics/permissions/requesting.html -normal-dangerous

15

Permission enforcement pre-Marshmallow

1. Application that needs permissions declares them in manifest file
2. User grants declared permissions (hopefully) when app downloaded
3. OS checks permissions automatically, e.g., when…
   - app makes a system call
   - app wants to launch an activity
   - app wants to send a broadcast intent (as determined by receiver)
   - broadcast receiver receives intent (as determined by intent sender)
   - an app accesses a content provider
   - an app starts a service or binds to a service
4. Permission failure generally throws *SecurityException* at point of call (except, e.g., when sending broadcast)

16

Permission enforcement pre-Marshmallow (cont'd)

- Pre-Marshmallow permissions are not revocable
  - Permissions granted by user when app installed
  - Must uninstall app to remove permission
- Permissions in Marshmallow and later versions are revocable
  - Use *Settings* app, choose *Apps* menu
- Pre-MM permissions still used most often
  - Recall MM+ permissions used only if:
    1. App running on MM+ device, and
    2. App was compiled with target-SDK ≥ 23 (MM)

17

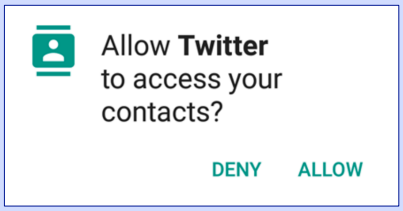
Permissions model in Marshmallow and later OSs

- User not prompted for permissions when installing app
- User prompted for permissions the first time app does operation requiring permission
  - User may or may not grant each of requested permissions
  - Operation performed only if user grants needed permissions
- App "remembers" permission groups it was granted
  - E.g., if *READ_CONTACTS* granted, *WRITE_CONTACTS* assumed by default, no need to ask user
  - Not asked again unless user revokes permission in *Settings* app
- This is a similar model to iOS

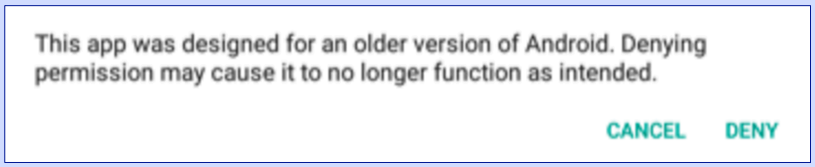
18

Permission dialogs for post-MM and pre-Mm apps

Post-MM dialog, post-MM app



Post-MM dialog for pre-MM app



19

Permission enforcement post-Marshmallow

1. Application that needs permissions declares them in manifest file
2. Before critical operation, Java code must check explicitly whether needed permission(s) were granted
3. If granted, proceed with operation
4. If not granted yet, do:
   - (Optional) explain why permission needed
   - Request permission(s)
5. Examine request result
6. If granted, do operation and remember permission
7. If not, skip operation

- Doc: http://developer.android.com/guide/topics/security/permissions.html

20

Automatic permission adjustments

- New permissions are added as Android OS evolves to new versions
  - Operations that did not require permission in earlier versions may require permission in later versions of OS
  - Example: READ|WRITE_CALL_LOG permissions added in API 16
  - An older application could break because one of its operations now requires a permission
- Android adds permission requests automatically to prevent older apps from breaking when deployed on new OS versions
  - Done even if app does not actually need the new permissions
  - Keep your apps up to date (with `targetSdkVersion`) to prevent this
- https://developer.android.com/reference/android/os/Build.VERSION_CODES.html
- https://support.google.com/googleplay/answer/6014972?hl=en

21

Specifying Android permissions

- Permissions are declared as strings
- In *AndroidManifest.xml* file an app $\mathcal{A}$ can do any of the following:
  1. Declare permissions that $\mathcal{A}$ wishes to use (must be granted by device user when installing app on device)
  2. Declare permissions that other components/apps or user devices must have in order to use $\mathcal{A}$ or $\mathcal{A}$ components
  3. Create new (programmer-defined) permissions
- This is done inside <manifest> tag, before <application> tag

22

1. Declaring permissions an app wants to use

- App **must** declare permissions to perform operations out of the sandbox
- Permission declared in <uses-permission> tag in app's *AndroidManifest.xml* file, e.g.,
  - <uses-permission android:name="android.permission.SEND_SMS" />
  - Tag appears inside <manifest> … </manifest> tags but before <application> tag
- User must allow declared permissions when downloading app (pre-MM)
  - No way for an app to acquire or lose a permission dynamically until MM
  - See code example next…

23

Example of permission declaration

- Suppose app wants monitor incoming SMS messages
- http://developer.android.com/guide/topics/security/permissions.html

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.android.app.myapp" >
   <uses-permission android:name="android.permission.RECEIVE_SMS" />
   ...
</manifest>
```

<uses-permission> tag declares permissions that this app needs.

<uses-permission> tag appears after <manifest> tag but before <application> tag.

24

Real example of Android permissions

- Some of the permissions in the *Contacts* application...
- From jellybean/packages/apps/Contacts/AndroidManifest.xml:23-35

```
<uses-permission android:name="android.permission.CALL_PRIVILEGED" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
<uses-permission android:name="android.permission.READ_CALL_LOG" />
<uses-permission android:name="android.permission.WRITE_CALL_LOG" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.READ_PROFILE" />
<uses-permission android:name="android.permission.WRITE_PROFILE" />
<uses-permission android:name="android.permission.READ_SOCIAL_STREAM" />
<uses-permission android:name="android.permission.INTERNET" />
...
```

25

2. Requiring permissions of other apps or users

- Permissions can be associated with an entire application, or with application components with *android:permission* attribute in tags <application>, <activity>, <receiver>, <provider>, <service>
- Restrict access to entire app or just some app components
  - Activity can limit apps that can launch that activity only to those holding a certain permission
  - Receiver can limit apps that can call that receiver …
  - Service can limit apps that can start, terminate or bind to that service …
  - Provider can restrict access to its information ...
  - Application can limit access to all its components …
- Syntax: use android:permission attribute inside appropriate tag (e.g., <application>, <activity>, etc.)

26

Example of activity requiring permission of clients

- Some of the permissions in the *Phone* application
- From jellybean/packages/apps/Phone/AndroidManifest.xml:141—152

To use this activity, an app must have this permission.

```
<activity android:name="OutgoingCallBroadcaster"
        android:theme="@style/OutgoingCallBroadcasterTheme"
        android:permission="android.permission.CALL_PHONE"
        android:screenOrientation="portrait"
        android:configChanges="orientation|screenSize|keyboardHidden">
      <!-- CALL action intent filters, for the various ways
          of initiating an outgoing call. -->
      <intent-filter>
        <action android:name="android.intent.action.CALL" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="tel" />
      </intent-filter>
…
```

27

3. Defining new permissions

- App may define custom permissions by <permission> tag in <manifest> tag contained in *AndroidManifest.xml* file
- Permission components (i.e., attributes in <permission> tag):
  - **Name**: String identifying the new permission, e.g.,

    android:name="edu.uic.cs478.myApp.CALL_RANDOM_NUMBERS"
  - **Label**: Short description of new permission, e.g.,

    android:label="Directly call phone numbers"
  - **Description**: String identifying the new permission, e.g.,

    android:description="Allows the application to call phone numbers without your intervention. Malicious applications may cause unexpected calls on your phone bill. Note that this does not allow the application to call emergency numbers."

28

3. Defining new permissions (cont'd)

- Permission components (cont'd):
  - **Protection level**: String identifying protection level, e.g.,

    android:protectionLevel="dangerous"
  - **Permission group** (optional): String identifying permission group, e.g.,

    android:permissionGroup="android.permission-group.COST_MONEY"

https://developer.android.com/guide/topics/manifest/permission-element.html

29

Example of new permission definition

- Doc: http://developer.android.com/guide/topics/security/permissions.html

<permission> tag defines new permission.

```
<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>
```

Tag attributes define features of new permission.

30

Coding Permission enforcement post-MM

Recall 7 steps of post-MM model...

1. Application that needs permissions declares them in manifest file
2. Before critical operation, Java code must check explicitly whether needed permission(s) were granted
3. If granted, proceed with operation
4. If not granted, do:
   - (Optional) explain why permission needed
   - Request permission(s)
5. Examine request result
6. If granted, do operation and remember permission
7. If not, skip operation

- Doc: http://developer.android.com/guide/topics/security/permissions.html

31

Post-MM Model: Requesting permissions at R-T

- **Step 2: Check permission with *checkSelfPermission()***
- Method returns immediately either PERMISSION_GRANTED or PERMISSION_DENIED
- See https://developer.android.com/training/permissions/requesting.html

```
// Assume thisActivity is the current activity
int permissionCheck = ContextCompat.checkSelfPermission(thisActivity,
        Manifest.permission.WRITE_CALENDAR);
if (PackageManager.PERMISSION_GRANTED == permissionCheck) {
    /* Do operation */ }
else {
    /* Prompt user for permission now */ }
```

Call method first.

Handle two possible outcomes.

32

Requesting permissions at R-T (cont'd)

- **Step 4: Ask for permission with Context method *requestPermissions()***
  - If *checkSelfPermission()* returns PERMISSION_DENIED, you should ask for the permission at R-T
- Signature:

  requestPermissions(Activity anActivity, String[] permissions, **int** reqCode): **int**
- See https://developer.android.com/training/permissions/requesting.html

33

Requesting permissions at R-T (cont'd)

- Example code
- See https://developer.android.com/training/permissions/requesting.html

```
if (PackageManager.PERMISSION_DENIED == permissionCheck) {
   ActivityCompat.requestPermissions( thisActivity,
       new String[]{ Manifest.permission.READ_CONTACTS },
       MY_PERMISSIONS_REQUEST_READ_CONTACTS);
}
```

If permission was denied, request it.

Requested permissions are put in a string array.

MY_PERMISSIONS_REQUEST_READ_CONTACTS is a local identifier denoting an integer constant passed to callback *onRequestPermissionsResult()*. Use it to distinguish different permissions requests.

34

Requesting permissions at R-T (cont'd)

- Steps 5-7: **Manage request results with *onRequestPermissionsResult()***
- Signature: onRequestPermissionsResults(**int** reqCode, String[] permissions, **int**[] results) : **void**
- See https://developer.android.com/training/permissions/requesting.html

```
public void onRequestPermissionsResult(int requestCode,
                                       String permissions[], int[] grantResults) {
  switch (requestCode) {
    case MY_PERMISSIONS_REQUEST_READ_CONTACTS: {
      if (grantResults.length > 0
         && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        // permission was granted, yay! Do the
        // contacts-related task you need to do.
      } else {
        // permission denied, boo! Disable the
        // functionality that depends on this permission.
    } }
```

If request canceled, result array is empty.

Don't attempt operation to avoid security exception.

35

iClickers questions

Questions

1. *Signature level* permissions are more secure than *dangerous level*
2. *Signature level* permissions require user approval

Answers

A. 1 true – 2 true

B. 1 true – 2 false

C. 1 false – 2 true

D. 1 false – 2 false

E. I hate Android

36

Broadcast receivers

- One of four kinds of components in an Android app
- Components that listen and react to events
- Broadcast receivers use an *intent filter* to select events of interest
  - We saw intent filters for starting new activity implicitly
  - Filters typically use (1) intent *action*, (2) *data*, and (3) *category* to determine
- Receiver concept based on *Observer* design pattern

37

Observer pattern

- Useful when one or more *observer* (aka *dependent*) objects must be notified of changes in a *master* (aka the *subject*) object
- Key point 1: Master does not know number and types of dependents
- Key point 2: Master and dependents are not related by inheritance
- Key point 3: Dependents (observers) can be added and removed dynamically
- Also known as *Publish-Subscribe* pattern

38

Observer pattern (cont'd)

- Magazine subscriber model
  1. Subscribe to the magazine
  2. Magazine produces a new issue
  3. You receive a copy of each new issue, like all other subscribers
  4. When no longer interested in magazine, unsubscribe
  5. You will no longer receive copies of new issues

39

High-level design of *Observer*

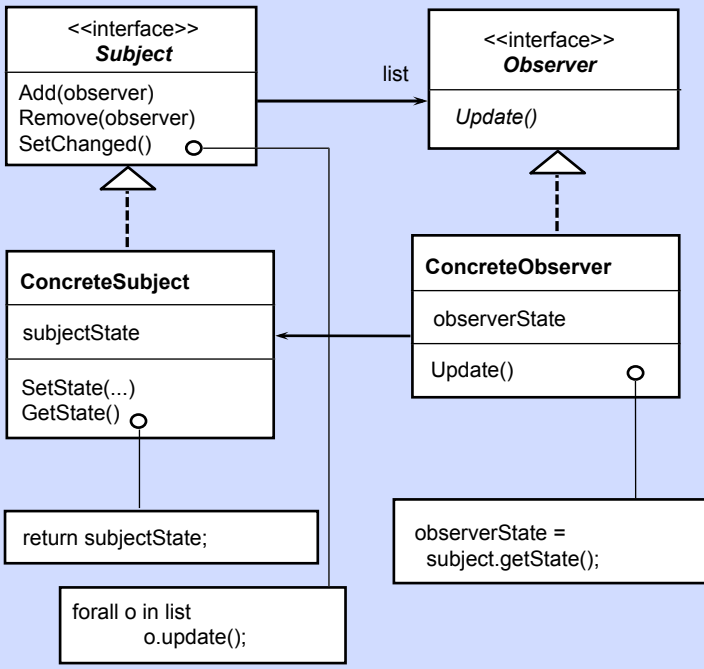
- Two kinds of objects:
  1. Magazine-type objects produce new information, inform subscribers
  2. Subscriber-type objects receive and process the information
- Type (1) objects are the *masters* or *subjects*
- Type (2) objects are the *dependents* or *observers*
- Example: Apps running on mobile device may want to know when the battery level becomes low (e.g., to suspend non-essential tasks)

40

Solution details

- Subjects have two kinds of APIs
  1. Dependents API—Maintain dependents list
     - Methods *addDependent(), removeDependent(), dependents()*
  2. State changed API—Know when to notify dependents
     - Method *setChanged()* advises subject that it has changed
- Dependents have one kind of API
  1. Updating API—Subject uses this API to inform dependent that subject has changed
     - Method *update()*
- Residual design issue: Communicate new subject state to dependents

41



Class diagram of *Observer*

42

Observer pattern

1. Observer registers itself with a subject by calling *addObserver()*

   subject.addObserver(this);

2. When a subject has changed, it is notified…

   subject.setChanged();

   - …or the subject can notify itself

     this.setChanged();

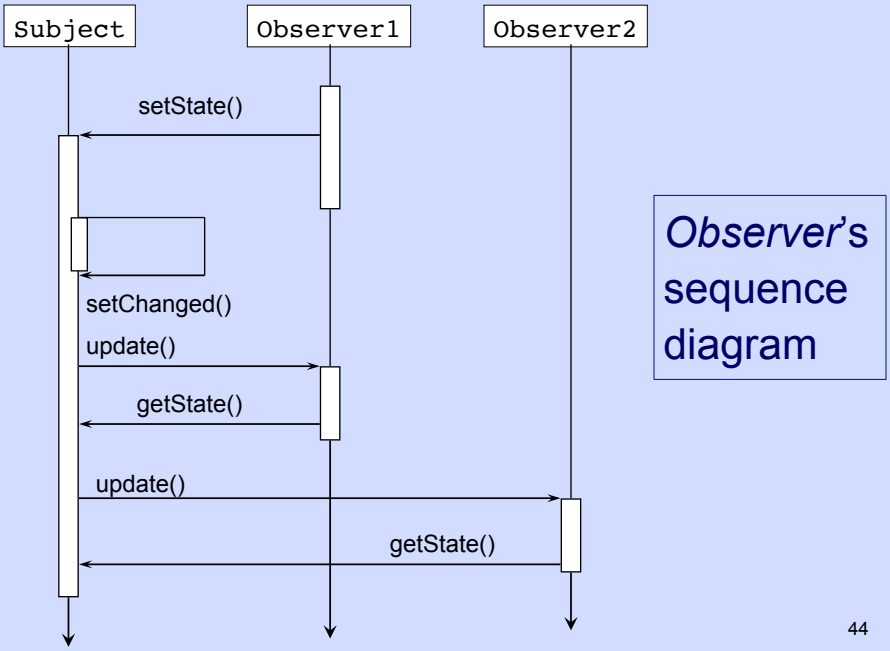
3. When a subject executes *setChanged()*, it notifies its observers

   for (Observer o : observers) { o.update() ; }

4. When an observer no longer wishes to watch the subject:

   subject.removeObserver(this);

43



Observer's sequence diagram

44

Discussing observer pattern

- **Loose coupling** between objects: Happens when objects know little about each other
  - Observers can be added and removed dynamically while the program is running
  - New types of observers can be added without modifying the subject
  - Observers do not have to belong to the same class hierarchy (e.g., they need not have a common superclass) or the same hierarchy as the subject
  - Changes to the behavior of subjects (or observers) will not propagate to observers (or subjects)

45

Pros and cons of *Observer*

- Loose coupling (cont'd)
  - Subjects unaware of number, location and type of dependents
  - Subjects and observers can belong to different components and class hierarchies
  - Support for Model-View-Controller paradigm
- Support for broadcasting
- Run-time penalty
  - Subject cannot control observer computations
  - Especially expensive if update calls on observers trigger new subject changes

46

Residual design issues of *Observer*

- Storing dependents lists
  - Obvious solution: With subject
  - Alternative solution: Global data structure (e.g., a hash table, internal Android registries)
- Observing multiple subjects
  - Add parameter to *update()*, subject sends self to inform observers what subject has changed
- Who triggers update?
  - Subject: Possibly too many updates
  - Observers: Beware of forgotten notification

47

Residual design issues of *Observer* (cont'd)

- Communication between subject and observers
  - *Push model*: Subject always sends all information that observers may ever need

    Disadvantage: May couple subject with observers (by forcing a given observer interface)
  - *Pull model*: Subject sends no information, observer will query subject

    Disadvantage: Can be inefficient

48

Model-View-Controller paradigm

- MVC paradigm = A kind of software architecture that divides overall application design into 3 components
- Popularized with onset of Graphical User Interfaces (GUIs)
  - Model: Algorithms and data structures performing computations in application's domain (i.e., the "business logic")
  - View: Classes/objects that display information for benefit of human user
  - Controller: Classes/objects that control interactions between user and GUI
- Supported by such frameworks as *Smalltalk-80, Rails, Spring, Play, ASP.net,* etc.

49

Android and the MVC paradigm

- Apps framework realizes MVC paradigm
  - Model: Depends on application domain (typically include broadcast receivers, services, files databases and content providers)
  - View: *View* and its subclasses (e.g., *TextView, ViewGroup, ListView, Button, LinearLayout,* etc.)
  - Controller: Activities and fragments

50

Broadcast receivers

- Broadcast receivers: App components that listen and react to *broadcast intents* in mobile device
- Receivers must register themselves with Android system in order to receive intents
- When intent is broadcast, Android dispatches intent to receivers that registered for that kind of intent
- Two kinds of "actors" involved in broadcasting intents
  - Intent sender—Builds intent and broadcasts it to system
  - Broadcast receiver(s)—Executes *onReceive()* method when matching intent is broadcast

51

Java code for broadcast receivers

- Receivers = subclasses of application framework class *BroadcastReceiver*
- *Broadcast intents*: Similar to intents for starting activities, but:
  - Called with *sendBroadcast()* instead of *startActivity()*
  - Handled differently and separately from intents that start activities
  - Useful to signal global events: Battery low, device plugged/unplugged to power source, airplane mode on/off, Wi-Fi connected, etc.
- *Phone* app needs listening to changes in bluetooth connection state
  - *BluetoothHeadset* class will broadcast intent with action *BluetoothHeadset.ACTION_AUDIO_STATE_CHANGED*
  - Phone app uses broadcast receiver to react to intent (see next…)

52

Example of receiver Java code definition

Source: jellybean/packages/apps/Phone/src/com/android/phone/PhoneApp.java:1595—1609

Declare broadcast receiver class.

onReceive() checks kind of intent received.

React to change in bluetooth connection.

```
/**
 * Receiver for misc intent broadcasts the Phone app cares about.
 */
private class PhoneAppBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (action.equals(Intent.ACTION_AIRPLANE_MODE_CHANGED)) {
            boolean enabled = System.getInt(getContentResolver(),
                    System.AIRPLANE_MODE_ON, 0) == 0;
            phone.setRadioPower(enabled);
        } else if (action.equals(BluetoothHeadset.ACTION_CONNECTION_STATE_CHANGED)) {
            mBluetoothHeadsetState = intent.getIntExtra(BluetoothHeadset.EXTRA_STATE,
                                                        BluetoothHeadset.STATE_DISCONNECTED);
            ...
            updateBluetoothIndication(true);  // Also update any visible UI if necessary
```

53

More on broadcast receivers

- Similar to start activity intents, broadcast intents can be implicit or explicit
  - Explicit intents must specify a *BroadcastReceiver* subclass as target
  - Implicit intents will specify action, data, category, etc.

54

Kinds of broadcast receivers

- Two kinds of broadcast receivers
  - App-local receivers: Listen only to broadcast intents originating in the same app as the broadcast receiver

    Managed by class *LocalBroadcastManager*
  - Global receivers: Listen to intents broadcast from everywhere

    Managed by class *Context*

https://developer.android.com/guide/components/broadcasts.html

https://developer.android.com/reference/android/content/BroadcastReceiver.html

55

Typical use case of broadcast receiver

1. Broadcast receiver registers itself (statically or programmatically) with Android system, usually with an intent filter
2. An application broadcasts an intent matching the receiver's filter
   - App sending broadcast can be different from app receiving broadcast
3. Android delivers intent to *all* broadcast receivers with matching filters
   - Receivers must have specified permission (if any), or else not called
   - Order is arbitrary unless *ordered broadcast* sent
4. Receivers wake up, execute *onReceive()* method
   - If receiver's app is not running, it is started to run *onReceive()* in the main thread of the app (pre-Oreo)

56

Defining and registering broadcast receivers

Static vs. programmatic definitions

- Static: Declare broadcast receiver in app's *AndroidManifest.xml* file
- Nest <manifest>, <application>, <receiver> and <intent-filter> tags inside one another
- Intent filter is customary, specifies what events this receiver will react to
  - If unspecified, receiver can only be activated with explicit intent, kind of defeating purpose of broadcasting
- Receiver created and registered at boot time, or when containing app is installed on device

See: http://developer.android.com/guide/topics/manifest/receiver-element.html

Also https://developer.android.com/about/versions/oreo/android-8.0-changes

57

Main attributes of receiver tag

Typical attributes in static receiver definition:

- android:exported="true|false" specifies whether local or global receiver
  - Local receiver listens only to broadcasts received from same app
  - Default is true if receiver specifies at least one filter
- android:permission="string" specifies permission that intent sender must have to trigger this receiver
- android:enabled="true|false" specifies whether receiver is initially enabled or not (default true)
- android:name specifies class implementing receiver
  - E.g., android:name="edu.uic.cs478.MyReceiver"

58

Example of static receiver definition

```
<manifest xmlns:android="blah, blah, blah"
   package="edu.uic.cs478.ReceiverExample"
   … >
  <application android:name="…" android:icon="…">
     <activity android:name="ActivityName" android:icon="…" >
        …
     </activity>
     <receiver
        android:label="Random Receiver"
        android:enabled="true"
        android:name="MyReceiver"
        android:exported="false">
        <intent-filter android:priority="10">
           <action android:name="edu.uic.cs478.ReceiverExample.showToast"  >
           </action>
        </intent-filter>
     </receiver>
  </application>
</manifest>
```

Attribute *android:name* specifies name of class defining receiver.

Attribute *android:exported* specifies whether local or global receiver.

Intent filter specifies events that this receiver will react to and receiver's priority.

59

Programmatic receiver definitions

- Few steps
  1. (optional) Get instance of *LocalBroadcastManager* (a singleton)
  2. Create intent filter and add information as needed
  3. Create instance of receiver class (a *BroadcastReceiver* subclass)
  4. Register receiver with intent filter
     - Use *registerReceiver(BroadcastReceiver,IntentFilter)*
     - Method defined in *LocalBroadcastManager* and *Context* classes
     - Receiver registration's lifetime tied to object that registered it
     - Undo registration explicitly with *unregisterReceiver()*
  5. (optional) Unregister receiver symmetrically with registration

60

Example of programmatic receiver registration

Source: jellybean/packages/apps/Phone/src/com/android/phone/PhoneApp.java:580—596

Create intent filter and add actions.

```
// Register for misc other intent broadcasts.
IntentFilter intentFilter =
      new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
intentFilter.addAction(BluetoothHeadset.ACTION_CONNECTION_STATE_CHANGED);
intentFilter.addAction(BluetoothHeadset.ACTION_AUDIO_STATE_CHANGED);
intentFilter.addAction(
                  TelephonyIntents.ACTION_ANY_DATA_CONNECTION_STATE_CHANGED);
intentFilter.addAction(Intent.ACTION_HEADSET_PLUG);
intentFilter.addAction(Intent.ACTION_DOCK_EVENT);
intentFilter.addAction(TelephonyIntents.ACTION_SIM_STATE_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_RADIO_TECHNOLOGY_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_SERVICE_STATE_CHANGED);
intentFilter.addAction(TelephonyIntents.ACTION_EMERGENCY_CALLBACK_MODE_CHANGED)
;
 …
 intentFilter.addAction(AudioManager.RINGER_MODE_CHANGED_ACTION);
 registerReceiver(mReceiver, intentFilter);
```

Now register receiver.

61

Using *LocalBroadcastReceiver* class

- Use when registering **local** receiver **programmatically**
- Singleton class, get/create unique instance:

  LocalBroadcastManager lbm =
  LocalBroadcastManager.getInstance(getApplicationContext()) ;
- Create receiver instance and intent filter instance programmatically (same for global receiver :

  lbm.registerReceiver(myReceiverInstance, myFilter)
- Remove receiver registration when done
  - *lbm.unregisterReceiver(myReceiverInstance) ;*
- Add compatibility library as dependency in *build.gradle* file:
  - compile **'com.android.support:appcompat-v7:23.2.1'**

62

Broadcasting an intent

- Create intent matching receiver's filter and call one of these methods
  - *sendBroadcast(Intent)*—Unordered broadcast, no receiver permission
  - *sendBroadcast(Intent, String)*—Specify permission that receiver must have in order to handle this intent
  - *sendOrderedBroadcast(Intent, String)*—Broadcast receivers executed sequentially based on their priorities (the higher, the sooner receiver executed)
  - *sendStickyBroadcast(Intent)*—Intent "sticks around" for receivers created after intent was sent
- **All calls are non-blocking**
  - See http://developer.android.com/reference/android/content/Context.html

63

Sticky broadcasts

- Useful for propagating information about system
- Can trigger receiver even if receiver comes to life (created, enabled, part of newly-installed app, etc.) **after** intent was broadcast
- Broadcast with method *sendStickyBroadcast(Intent)*
- Deleted from system with *removeStickyBroadcast(Intent)*
- Deprecated in API 21 because of lack of security, but still…
- Must have permission android.permission.BROADCAST_STICKY (normal level) to send or remove sticky broadcast

  See http://developer.android.com/reference/android/content/Context.html and http://developer.android.com/reference/android/Manifest.permission.html

64

Receiving a broadcast intent

- Broadcast receiver defines method *onReceive(context, intent)*
  - *context* = context in which receiver is running
  - *intent* = intent sent by broadcaster
- Beware:
  - *onReceive()* runs in process of receiving app, not process of app that broadcast intent
  - If receiving app not running, process created to run *onReceive()*
  - If so, process deleted as soon as *onReceive()* completes (pre-Oreo for implicit intents)
  - Don't do long-term operations in *onReceive()*

65

onReceive() dos and don'ts

- *onReceive()* caveats:
  1. Method *onReceive()* runs in UI thread of receiving app – Do not block with long running operations!
  2. App's process running *onReceive()* can be terminated when method returns – Do not start operations that expect a result back
- Can do: Start an activity, display a toast message, start a service, etc.
  - But beware of starting multiple activities from different receivers
- Don't do: Start asynchronous operations that require a result back to caller (e.g., *startActivityForResult(),* request a permission, etc.)

66

Running *onReceive()*

- Process running *onReceive()* has high priority
- If multiple receivers respond to broadcast, this could affect device performance
  - A known disadvantage of *Observer* pattern
- Again: Make sure that *onReceive()* runs quickly
  - If substantial work needs to be done, use service

67

Twist in Oreo

- Statically registered receivers no longer respond to implicit intents
- A severe limitation because
  - Broadcasting only works with implicit intents
  - Receiver responds to implicit intents only if registered programmatically
  - Receiver is automatically unregistered when registering component is destroyed
- No longer possible to wake up an app through a broadcast
- Why is this bad?
  - Example app that synchronizes with a server only when Wifi connected
  - If app not running, receiver for responding to changes in Wifi status will not respond to broadcast

68

Sending ordered broadcasts

- Plain broadcasts: Execute receivers in arbitrary (non-deterministic) order
  - Receivers could even run simultaneously on multi-core hardware
- Ordered broadcasts: Execute receivers sequentially based on their priority
  - *sendOrderedBroadcast(Intent, String)*—Broadcast receivers will be ordered according to priority in their filter
  - Permission string specifies permission to be held by receiver

69

Caveat on ordered broadcasts

- Ordered broadcast can be aborted by a broadcast receiver, preventing receivers down the line from receiving that broadcast
  - *isOrderedReceiver()* returns boolean indicating whether receiver is processing an ordered broadcast
  - *abortBroadcast()* sets flag indicating that current broadcast is aborted (No further receivers called—Must be an ordered broadcast)

70

Typical use case for aborting a broadcast

- First check whether broadcast is ordered, then abort
- Cannot abort ordinary broadcast

```
…
onReceive(Context context, Intent i) {
  …
  if (isOrderedBroadcast()) {
     …
     abortBroadcast() ;
     …
     }
}
```

First, check whether broadcast is ordered.

Next, prevent downstream receivers from responding.

71

Getting result from broadcast intent

- Sometimes useful to get a result back from the chain of receivers
- Special version of *sendOrderedBroadcast()* allows sender to set up its own receiver, that will be called when all receivers are handling the original broadcast
  - Information is passed in the form of a new intent
  - *sendOrderedBroadcast(intent, permissionString, resultReceiver, scheduler, initialCode, initialData, initialExtras)*

    Sets up receiver allowing sender to receive result back
  - *setResultData(), getResultData()* allow receivers to get and set current result data

72

Additional readings

- Android security tips
- Using Transport Layer Security (TLS) – Buyers beware: SSL v3 is broken (see https://en.wikipedia.org/wiki/POODLE)
- Best practices for permissions and user data

73