CS 478: Software Development for Mobile Platforms

Set 3: Activities and User Interfaces

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

January 24, 2019

Android activities

- Activity = Single, focused thing that a user can do
- Main means of interaction with app user
- Each activity should support a single action
- When designing your apps, think in terms of the single actions that app should support and define one activity for each action
  - Examples of actions: Writing an email message, viewing a list of SMS messages, writing an SMS, dialing a phone number, etc.
- See http://developer.android.com/reference/android/app/Activity.html

1

Tasks and the backstack

- *Task*: A sequence of activities for a given user scenario
- *Task backstack*: Stack of activities capturing user interactions
  - Activities are popped and pushed following user commands
- Multiple backstacks possibly active simultaneously
  - Jump between backstacks by going back to home screen and selecting app on top of stack
  - Or use “recents” button
- See http://developer.android.com/reference/android/app/Activity.html

2

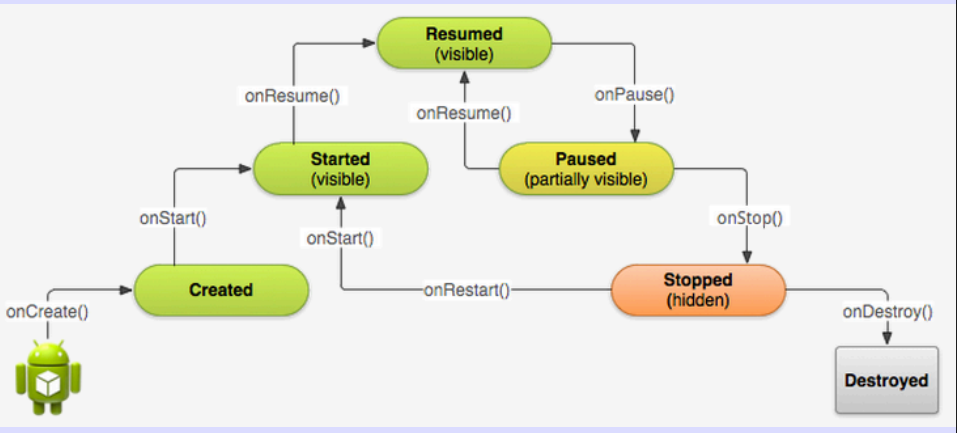
Activity lifecycle

- When app started, its main activity is usually pushed on top of stack
  - New activity is visible and in focus
  - In Java: New instance of activity’s class is created
- Activity can be deleted, e.g., because one of the following happens
  - User pushes “back” button on the device
  - Activity kills itself by calling *finish()* OS method
  - OS kills it to reclaim resources held by the activity
- If activity in focus deleted, previous activity in backstack regains focus
  - This activity is popped off stack and deleted
- Activities go through various states and state transitions in their lifetime (see next)

3

State diagram of an activity

- Source: http://developer.android.com/training/basics/activity-lifecycle/starting.html



4

Activity states

- *Created* – Java object controlling activity has been created and initialized (defining content and user interactions), but not visible yet
- *Started* – Activity is visible, but not in focus (not interacting with user)
- *Resumed* (aka *running*) – Activity is visible and in focus (interacting)
- *Paused* – Visible, but not interacting (e.g., partially covered)
- *Stopped* – Not visible, but in backstack
- *Destroyed* – Activity object has been deleted

Caveat: *Created and Started* are transient states, the other 4 are permanent states

Activity could be in one of the permanent states for an indefinite time

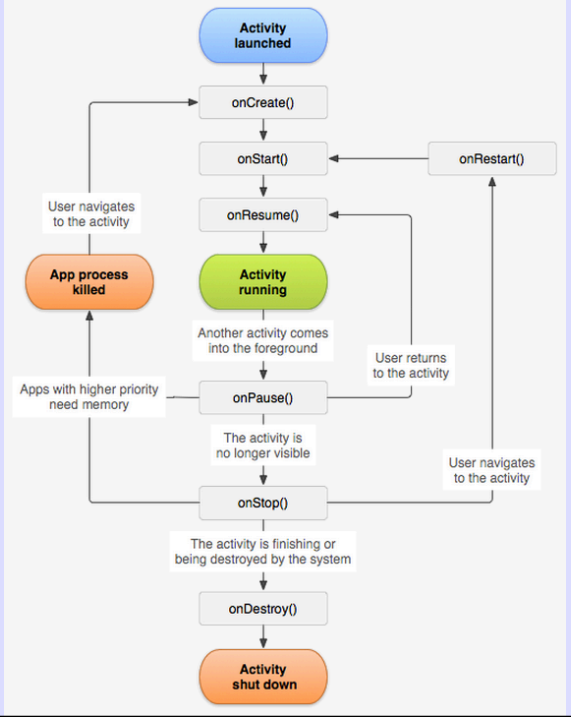
5

Activity states

- Android's app execution interleaves OS code with app code
  - App is not a standalone module with a *main()*
  - OS calls programmer-defined code when appropriate
- For activities, OS calls app code whenever a state transition occurs
- Programmer responds to state changes by defining callback methods in activity class invoked automatically by OS, e.g.,
  - *onCreate()* – Called when activity transitions to *Created* state
  - *onStart()* – Called when activity transitions to *Started* state
  - *onResume()* – etc.
  - *onPause()*
  - *onStop()*
  - *onDestroy()*

6

Typical lifecycle of an activity



Source: http://developer.android.com/reference/android/app/Activity.html

Activity lifecycle—The callbacks

- *onCreate()*—Initialize the new activity object
  - Think of it as an additional activity "constructor"
  - Input parameter: a *Bundle* with saved instance state (See later)
- *onStart()*—Activity about to become visible
  - Load persistent state, update content, update locations, etc.
- *onResume()*—Activity about to gain focus (interact with user)
  - Start focus-related activities (e.g., start animations, background sounds)
- *onPause()*—Activity about to lose focus
  - Save persistent state, stop animations

8

Activity lifecycle—The callbacks (cont'd)

- *onStop()*—Activity about to become invisible (may not be called in Gingerbread and earlier OS)
  - Cache state
- *onDestroy()*—Activity about to go away for good
  - Caveat: *onDestroy()* may not be called at all, if OS kills activity
  - Release allocated resources
- Typically, you will not write all these methods but only those needed for your app to run
  - *onCreate()* is mandatory
  - *onPause()* is used quite frequently
  - Other callbacks are used less frequently

9

Additional activity callbacks

- *onRestart()*—Called when stopped activity is started again
  - Part of activity lifecycle
  - Usually unnecessary because always followed by *onStart()*
  - Very technical, probably will never use
- *onSaveInstanceState()*—Called when OS about to kill activity because memory is low or configuration change occurred
  - Not part of lifecycle, not guaranteed to get called
  - Main goal: Create a *Bundle* instance that will be passed to *onCreate()* (and possibly *onRestoreInstanceState()*) when activity created again
  - http://developer.android.com/training/basics/activity-lifecycle/recreating.html

10

Defining an activity

1. Create your own class by extending framework class *Activity*
   - In Java:
     - `public class MyActivity extends Activity ...`
   - In Kotlin:
     - `class MyActivity : Activity ...`
2. Define appropriate fields
3. Define appropriate callback methods:
   - `public void onCreate(Bundle savedInstanceState) { ...`
   - `public override fun onCreate(savedInstanceState: Bundle?) {`
4. Define additional methods as needed

11

Writing method *onCreate()*

- Goal: Set up data structures + screen configuration + user interactivity
- Java source template
  1. Call *super.onCreate()*
  2. Set layout configuration for activity (call *setContentView()* on appropriate layout file)
  3. Get references to views, e.g., *findViewById(R.id.viewName)*
  4. Check saved instance state if expected, act upon it
  5. Set up listeners for interactive views (e.g., buttons, menus, etc.)
     - Listener = Code block invoked when user interacts with a view
  6. Configure views and do other initialization actions

12

"Bundles"

- Groups of *(key, value)* pairs
  - Key: String
  - Value: A *Parcelable* object
- Popular methods
  - Setter methods take a *String* (the key) and value (*int, String...*): *putInt(), putString(), putDouble(), putIntArray(), putBundle(),* etc.
  - Getter methods take a string, return a value: *getBoolean(), getChar(), getByte(), getCharArray()*
- https://developer.android.com/reference/android/os/Bundle.html
- https://developer.android.com/guide/components/activities/parcelables-and-bundles.html

13

Parcelable objects

- Objects meant to be passed across process boundaries
- (1) *marshalled* (flattened) into parcels, (2) shipped to another process, and (3) *unmarshalled* into their original form
- *Parcel:* Container for flattened (marshaled) data that be passed between processes through binder mechanism
- Parcelable is a Java *interface*—Familiarity with interfaces assumed
- Examples of *Parcelable* objects
  - All primitive types and primitive arrays (predefined *Parcelable*)
  - Instances of classes that implement *Parcelable* Java interface
  - You'll have to implement *Parcelable* interface for classes whose instances are passed between processes; we'll see how later on
- https://developer.android.com/reference/android/os/Parcelable.html

14

A disclaimer: Activity killed by OS

- Android can kill a running process if system is low on resources
- If process being killed has activities, the activities will also be destroyed without running any more methods
  - Pre-Honeycomb (e.g., Gingerbread, Froyo, etc.)—Paused and stopped activities can be destroyed without calling activity's *onStop()* and *onDestroy()* methods
  - Honeycomb and later (e.g., ICS, JB, KK, etc.)—Only stopped activities can be destroyed
- Friendly advice: Don't count on your *onDestroy()* method being executed (*onStop()* too, if app can run on Gingerbread, etc.)
  - Save critical state in *onPause()* and *onStop()*

15

Another disclaimer: Configuration changes

- **Current activity generally killed when configuration change occurs**
  - Device rotated (e.g., from portrait to landscape mode)
  - Language change, input mode change, etc.
- Why killed? Accommodate for (possible) new layout, etc.
- This means that *onPause(), onStop(), onDestroy(), onCreate(), onStart(), onResume()* will be called in sequence on old and new activity instances
  - Stopped activities destroyed and recreated when in focus again
- Two ways to avoid total destruction: (1) Use *fragments*, or (2) override callback *onConfigurationChanged()* in activity class
  - See next…

16

Handling configuration changes

1. Ad-hoc methods for saving and restoring activity instance state
   - *onSaveInstanceState()* called by OS before it kills an activity
     - Save state in argument bundle
   - *onRestoreInstanceState()* called after *onStart()* when activity restarted
     - Your bundle passed back to it
     - The same bundle is also passed to *onCreate()*
2. Redefine *onConfigurationChanged()* – Not recommended
   - More complex – Programmer must handle configuration change
   - Declare type of change handled in manifest file
   - https://developer.android.com/guide/topics/resources/runtime-changes.html

17

Starting an activity

Two ways to start an activity

1. *startActivity(Intent)*

   Must first create intent, describing activity to be started

   New activity created, started and put on focus, covering calling activity

2. *startActivityForResult(Intent, int)*

   Calling activity expects result (e.g., success vs. failure) back

   Add an integer identifying the call

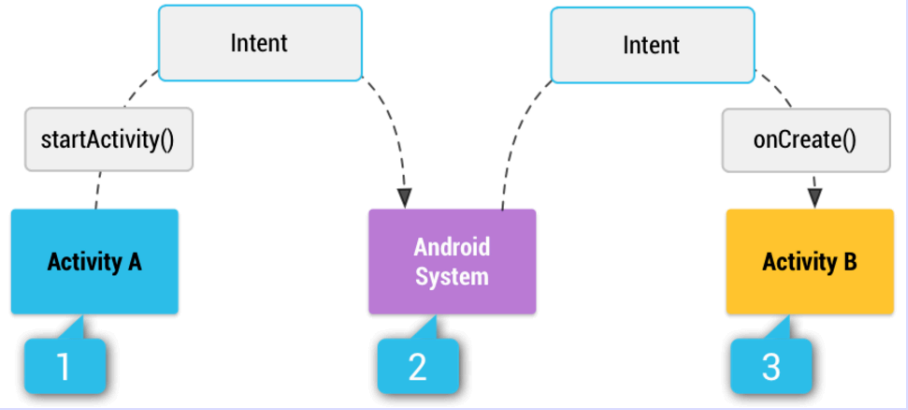
   Called activity retrieves intent with *getIntent()*—When done, it calls *setResult(int)*

   Calling activity gets callback *onActivityResult(int, int, Intent)*; second arg is result code

18

Flow of information of *startActivity()*

- Methods *startActivity()* and *startActivityForResult()* are part of Android framework (declared abstract in *Context* class and defined in *Activity*)



19

About *startActivityForResult()*

The protocol of starting an activity with a result

1. Parent activity calls *startActivityForResult()* – Child activity gets focus
   - Intent argument *intent* specifies what child activity to start
   - Integer argument *requestCode* (usu symbolic) identifies call
2. Child activity sets result code (also symbolic) with *setResult(int)* or *setResult(int,Intent)* before finishing
   - Integer argument *resultCode* reports on success/failure of child activity
3. OS calls parent activity's callback *onActivityResult(int, int, Intent)*
   - Argument *requestCode* is original code identifying call
   - Argument *resultCode* is result set by child
   - Argument *intent* is original intent (presumably with new extras)

20

About *onActivityResult()*

- Called right before *onResume()* in parent activity
- Intent *extras* can return additional information from child to parent activity
- Common symbolic names for integer *resultCode*: RESULT_OK, RESULT_CANCEL (public static fields of framework class *Activity*)
- If child activity did not call *setResult(),* default value RESULT_CANCEL passed to *onActivityResult()*
- Custom user codes can be defined too, e.g., RESULT_FIRST_USER (a predefined custom user code)
- Integer argument *requestCode* useful because parent activity could start multiple child activities depending on situation

21

Example of *startActivityForResult()*

jellybean/packages/apps/Contacts/src/com/android/contacts/activities/PeopleActivity.java:1093-1102

- Activity showing contact list starts activity to edit a contact

```
public void onEditContactAction(Uri contactLookupUri) {
  Intent intent = new Intent(Intent.ACTION_EDIT, contactLookupUri);
  Bundle extras = getIntent().getExtras();
  if (extras != null) {
    intent.putExtras(extras);
  }
  intent.putExtra("finishActivityOnSaveCompleted", true);
  startActivityForResult(intent, SUBACTIVITY_EDIT_CONTACT);
}
```

Create new intent with action and data to start contact editor activity.

Add extra information to new intent.

Return when user saves contact info.

Launch new activity.

22

Example of started activity

- Contacts app manifest file declares *ContactEditorActivity* to listen for *ACTION_EDIT* intents
- This activity will be launched in response to previous intent
- Activity uses intent extra to set result and return

```
public void onSaveFinished(Intent resultIntent) {
  if (mFinishActivityOnSaveCompleted) {
    setResult(resultIntent == null ? RESULT_CANCELED : RESULT_OK, resultIntent);
  } else if (resultIntent != null) {
    startActivity(resultIntent);
  }
  finish();
}
```

Called when user saves contact.

Field set based on intent's extra.

ContactEditorActivity sets result code.

ContactEditorActivity terminates by calling *Activity.finish()*

23

Example of getting activity result

- *onActivityResult()* will first check type of request triggering this result, then check result code, finally take appropriate action
- PeopleActivity.java:1565-1587 (JB V4.1.1)

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
 switch (requestCode) {
  case SUBACTIVITY_ACCOUNT_FILTER: {
    …
    break;
  }

  case SUBACTIVITY_NEW_CONTACT:
  case SUBACTIVITY_EDIT_CONTACT: {
    if (resultCode == RESULT_OK && … ) {
      …
    }
    break;
  }
```

First check request type.

Specify this method in parent activity.

Now check the result set by called activity.

Specify actions in response to returned code and data.

24

Starting activities with intents

- Intent: A general-purpose message used for starting activities, notifying apps of some condition (e.g., low battery), starting a service, etc.
  - Goal: Allow component to signal *intention* to get something done
  - Target(s) could be an activity, a (set of) broadcast receivers or a service
  - Typically an argument to such methods as *startActivity(), broadcastIntent(),* or *startService()*
- Allow inter-app interactions
- Intent lifetime:
  - Created and sent by component (i.e., activity, service, broadcast receiver) that wants an operation done (e.g., starting an activity)
  - Received by interest parties (e.g., by activity to be started) via OS

25

Two kinds of intents

1. **Explicit**

   Intent specifies exactly the component (package and class name) to be called (i.e., an activity, a receiver, or a service)

2. **Implicit**

   Intent just declares the kind of functionality expected of the target component, does not name component explicitly

   Examples of desired functionality when activity is target

   - Able to dial a phone number
   - Able to view a web page
   - Able to compose and edit an email message, etc.

- https://developer.android.com/training/basics/intents/index.html

26

Intent class

- Intents are instances of framework class *Intent*
- Many fields used to pass different kinds of information from sender to receiver(s)
  - Action
  - Data
  - Category
  - Type
  - Component
  - Extras
  - Flags

27

Implicit intents: *Action* and *data* fields

- Primary fields: Characterize implicit intents
- Action: String conveying operation to be performed
- Data: Operation data
- Many actions, e.g.,
  - ACTION_DIAL: Show phone number specified as data in phone dialer (part of built-in *Phone* app)
  - ACTION_CALL: Show dialer and call number specified as data
  - ACTION_VIEW: View some specified data (e.g., text file, web page...)
  - ACTION_DIAL, ACTION_VIEW, etc. are public, static, final *String* constants defined in *Intent* class

28

Setting intent *action*

Two main ways to set an intent's action field

1. Directly when creating *Intent* instance (e.g., done by constructor)

```
Intent anIntent = new Intent(Intent.ACTION_MAIN) ;
```

2. Calling setter method after instance created, e.g.,

```
Intent anIntent = new Intent() ;
anIntent.setAction(Intent.ACTION_MAIN) ;
```

- Source code:

  frameworks/base/core/java/android/content/Intent.java

- Documentation:

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

29

Most popular *Action* fields

- Many action strings, public static final fields in *Intent*
  - ACTION_MAIN: Start main activity of an app – No data expected
  - ACTION_VIEW: View specified data – Data is *Uri* instance specifying what to view
  - ACTION_EDIT: Edit data specified as *Uri*
  - ACTION_DIAL: Show dialer with phone number specified as data
  - ACTION_CALL: Call phone number specified as data (restricted)
  - ACTION_SEND: Send a message or an email to someone else – Data specifies recipient (*Uri* type determines app)
  - ACTION_SYNC: Synchronize device with server
  - ACTION_PICK: Allow use to select an item – Data is directory of choices

30

Intent *data*

- Expressed as Uniform Resource Identifier (URI), generalization of URL (Uniform Resource Locator) concept
- URIs have several formats, called *schemes*
- Each scheme has a tag ending in colon, and additional data
  - *http, https*—For browser data
  - *geo*—Address information (coordinates and/or address)
  - *tel*—Telephone number
  - *mailto*—Email information
  - *content*—For arbitrary content (e.g., a document file)
- Examples: "tel:(555) 555-1212" and "tel:+1-555-555-1212"

31

Setting intent *data*

- Again, can use *Intent* constructor with action and data args, e.g.,

```
Intent i = new Intent(Intent.ACTION_DIAL, Uri.parse("tel:+15555551212") ;
```

  - Note: *Uri.parse()* is static *Uri* method returning new *Uri* instance
- Another way: Calling setter method after instance created, e.g.,

```
Intent anIntent = new Intent() ;
anIntent.setAction(Intent.ACTION_DIAL) ;
anIntent.setData(Uri.parse("tel:+15555551212")) ;
```

- Source code:

  frameworks/base/core/java/android/content/Intent.java
- Doc: http://developer.android.com/reference/android/content/Intent.html

32

Action + *Data* combo

- Action and data are often enough to define intent target
- Assume ACTION_VIEW, then
  - *http, https* data schemes → open a browser activity
  - *geo* scheme → open a map view activity
  - *tel* scheme → open the phone dialer activity
  - *content* scheme → open a file viewer

33

Intent category (cont'd)

- String giving more info about kind of activity that should handle an intent
  - Refine an intent's action
- Examples:
  - CATEGORY_LAUNCHER – Activity can be called from launcher (it appears in the launcher screen)
  - CATEGORY_BROWSABLE – Activity can be called by browser
  - CATEGORY_DEFAULT – Activity can be called by *startActivity()* even though the outstanding intent does not include a category
    - Use in intent filter (see later)
- Doc: http://developer.android.com/reference/android/content/Intent.html

34

Intent category (cont'd)

- Additional category examples, asking for specific app (Combine with ACTION_MAIN):
  - CATEGORY_APP_BROWSER
  - CATEGORY_APP_CALCULATOR
  - CATEGORY_APP_CALENDAR
  - CATEGORY_APP_CONTACTS
  - CATEGORY_APP_EMAIL
  - CATEGORY_APP_GALLERY
  - CATEGORY_APP_MAPS
  - CATEGORY_APP_MARKET
  - CATEGORY_APP_MESSAGING
  - CATEGORY_APP_MUSIC
- Others CATEGORY_CAR_DOCK, CATEGORY_DESK_DOCK, etc.

35

Intent MIME type

- Intent's *type* field specifies type of data in intent (a MIME type)
- Especially useful when *data* field not given
- Usually derived from intent's *data* field; specifying *type* field overrides type implicitly contained in *data* field
- Retrieved with *getType(): String*
- Set with *setType(String): Intent*
- Examples:
  - text/plain
  - text/html
  - image/jpg
  - etc.

36

Explicit intents: Intent component

- Use for **explicit intents**
- Defines specific class defining target component for this intent
  - Class could belong, e.g., to an activity or service
- If set, all other intent fields are overruled
- Typically set with convenience constructor *Intent(Context, Class)*
  - Context (Global app where target class resides)
  - Actual class defining activity to be launched
- Set also from methods *setClass(Context, Class)* and *setComponent(ComponentName)*
- Example: `Intent i = new Intent(MainActivity.this, ChildActivity.class) ;`

37

Intent extras

- Additional information contained in an intent
- Extras = *Bundle* instance
  - Map of (key, value) pairs, where keys must be strings
- Values can be primitive types, primitive arrays, or bundles
- Getter method *getExtras()* retrieves entire map
- Getter methods *getIntExtra(String), getIntArrayExtra(String)*, etc. retrieve value associated with arg *String* key
- Setter methods *putExtra(String, <value-type>)* places an extra in intent
- Example: Extra for adding email addresses directed to a compose message activity

38

Example of useful type

- Source: https://developer.android.com/training/basics/intents/sending

Intent for sending email message with attachment.

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
// The intent does not have a URI, so declare the "text/plain" MIME type

emailIntent.setType(HTTP.PLAIN_TEXT_TYPE);
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[] {"jon@example.com"}); // recipients
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");
emailIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse(
                        "content://path/to/email/attachment"));
```

Recipient is an extra

Text +subject extras.

Specify attachment as an extra.

39

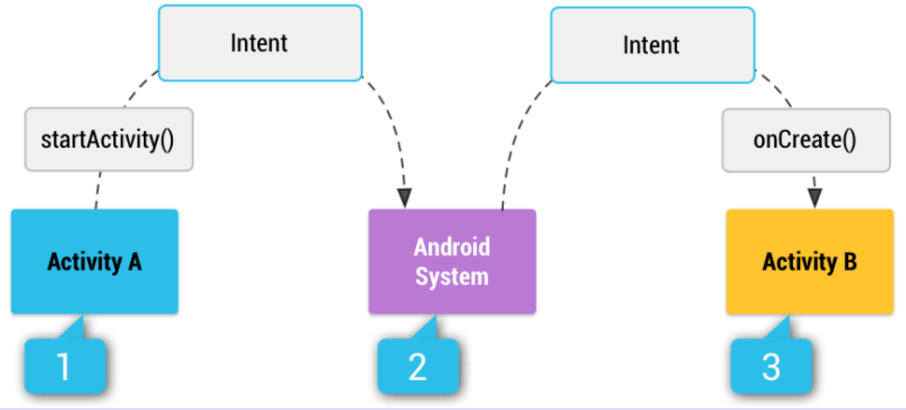
Intent flags

- Metadata specifying how an intent is to be handled by OS
- Example: FLAG_ACTIVITY_NO_HISTORY, if set, will not put the child activity in the backstack
- Doc: http://developer.android.com/reference/android/content/Intent.html - setFlags(int)

40

Defining intents that start an activity

- Recall *startActivity()* or *startActivityForResult()*
  - Both require intent specifying activity to be launched



41

Intent resolution process

- *Intent resolution process* determines components that respond to intent
- Intent resolution works in two ways:
  1. **Explicit**

     Intent specifies component field → Explicit call to specified activity or service or broadcast receiver
  2. **Implicit**

     Intent does not specify component → Android's intent resolution process decides what activity to launch based on intent's action, data, category and type

Doc: https://developer.android.com/guide/components/intents-filters.html

42

Explicitly calling an activity

- Calling activity creates intent that stores a *Context* object and a *Class* object
  - *Context* object specifies global application shared among application components
  - *Class* object specifies class to be instantiated for new activity
- Example from *SimpleCounter4*, starting tip calculator activity

```
private void switchToStateList() {
  Intent i = new Intent(SimpleCounterActivity2.this, StateListActivity.class) ;
  startActivity(i) ;
```

- Caveat: Explicit intents are much more secure than implicit intents, but implicit intents support platform extensibility

43

Implicitly calling an activity

- Android looks for an activity matching that operation
- Intent *resolution*: Procedure for finding activity matching an intent
  - Activities interested in answering some intent must declare *filters* specifying intent actions
  - Activities can declare intent filters *statically* in manifest file or *programmatically* through Java statements (using class *IntentFilter)*
- OS resolves implicit intent based on 3 intent fields
  1. Action – Answering component must include this action
  2. Data – Uri data used to deduce type if intent does not have type field
  3. Categories – Answering component must include all categories in intent

44

Intent filters

- *Intent filters*: Specify what intents a component should respond to
  - Components: activities, broadcast receivers and services
- Rules for matching
  - Action matches if *any* of the values in filter matches intent action
  - Data type matches if *any* of the values matches intent data
  - Data scheme matches if *any* of the values match intent's data scheme
  - Categories match if all categories in intent are contained in filter (filter may contain additional not mentioned in intent)
  - Etc.
- http://developer.android.com/reference/android/content/IntentFilter.html
- https://developer.android.com/guide/components/intents-filters.html

45

Example of intent filter

- *Phone* app declares two intent filters for emergency dialing activity
- Source: jellybean/packages/apps/Phone/AndroidManifest.xml:103—116

```
<activity android:name="EmergencyDialer"
        android:label="@string/emergencyDialerIconLabel"
        android:theme="@style/EmergencyDialerTheme"
        android:screenOrientation="nosensor">
        <intent-filter>
          <action android:name="com.android.phone.EmergencyDialer.DIAL" />
          <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
        <intent-filter>
          <action android:name="com.android.phone.EmergencyDialer.DIAL" />
          <category android:name="android.intent.category.DEFAULT" />
          <data android:scheme="tel" />
        </intent-filter>
    </activity>
```

Activity tag declares activity.

First intent filter declares *DIAL* action, *DEFAULT* category.

Second intent filter declares also URI with scheme *tel.*

46

Example of implicitly starting an activity

- Phone app declares activity for emergency dialing, with two intent filters
- Source: jellybean/packages/apps/Settings/src/com/android/settings/CryptKeeper.java

```
private void launchEmergencyDialer() {
    final Intent intent = new Intent(ACTION_EMERGENCY_DIAL);
    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
            | Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
    startActivity(intent);
}
```

Create intent with emergency dial action.

Start new task in history stack.

Now start activity.

Exclude activity from list of recently-started activities.

47

Android tasks

- Task = Collection of related activities that support a user's workflow:
  - Activities may belong to different apps
  - Example:

    Launcher → Email activity showing message list

    → Activity showing selected message

    → Compose message activity to reply to message
- Each task is stored in its own backstack
- Task started in launcher (when user selects app) or by activity
- Task can be resumed from home screen when user selects again app that started the task, or through the "recents" menu
- https://developer.android.com/guide/components/tasks-and-back-stack.html

48

Activity *singleTop* start mode

- Activity can specify a launch mode, e.g.,,

```
<activity …
    …
  launchMode=[“standard” | “singleTop” | “singleTask” | “singleInstance” ]
  … />
```

- Default is *standard; singleTop* means reuse the same instance if that instance is at top of current backstack task (both launched with *startActivity()*)
- Modes *singleTask* and *singleInstance* always begin a new task
  - *singleInstance* excludes other activity instances from task; using *startActivity()* will begin new task

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

49

Android display objects

- Classes defining icons (aka UI elements, widgets…) that populate activities
- Subclasses of special *View* class
- Each instance associated with **rectangular** portion of display
- Main responsibilities of *View* and its subclasses:
  - Drawing themselves
  - Knowing their position in the window displaying them
  - Intercepting + responding to user interactions (e.g, button press)
  - Adding/accessing/removing subviews (for container views)
- Source: jellybean/frameworks/base/core/java/android/view/View.java
- Doc: http://developer.android.com/reference/android/view/View.html

50

The *View* class hierarchy

- *View* is a subclass of Java *Object*
- Subclasses include
  - *TextView*
  - *ImageView*
  - *KeyboardView*
  - *ViewGroup* (a set of views—see *Composite* pattern)
- *ViewGroup* subclasses include most layouts (e.g., *LinearLayout*, *RelativeLayout, DrawerLayout*, etc.) as well as compound views such as lists, galleries, drop-down menus (aka spinners), etc.
- Straightforward implementation of *Composite* design pattern…

51

Composite pattern

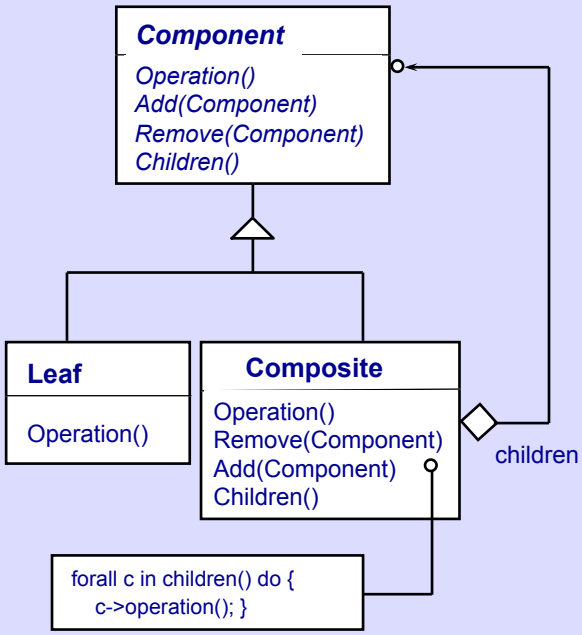
- Motivation:

  Support *recursive composition* in such a way that a client need not know the difference between a single and a composite object (as with *Views*)
- APIs of all display objects is declared in abstract superclass – All concrete subclasses must support that functionality
- Applicability:

  Dealing with hierarchically-organized objects (e.g., columns containing rows containing words …)
- Structure

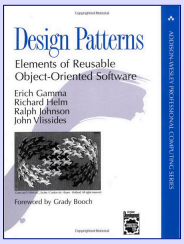
  See class diagram next

52



Class diagram of *Composite*

Source:
E. Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*



53

Composite pattern: *Component* responsibilities

Minimally, a *Component* instance should support these APIs:

1. Know how to draw itself
2. Know and be able to change its position and size
3. If *Component* is a *Composite*, manage its children
4. If interactive component, respond to user interactions

54

Composite pattern

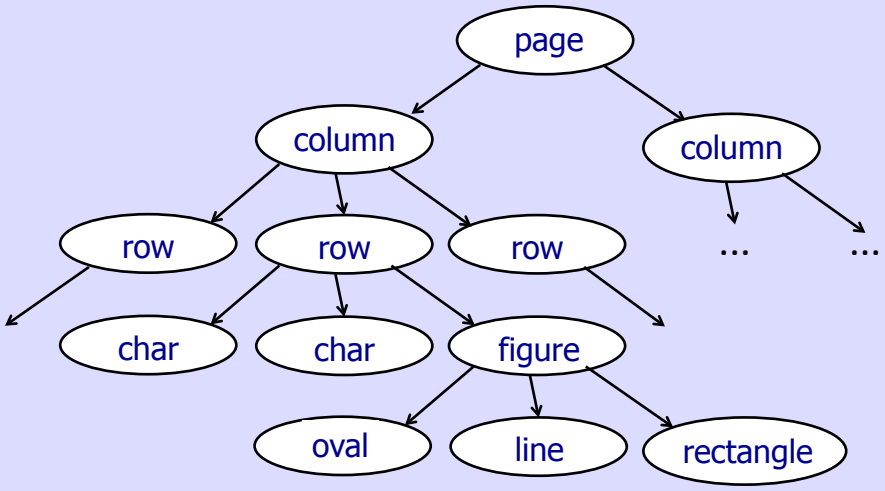
Consequences:

1. Define class hierarchies consisting of simple and composite objects
2. Simplify clients
3. Extensibility (clients need not be modified if new components added)
4. Too general a pattern?

   (makes it difficult to to restrict functionality of concrete leaf subclasses)
5. Implementation issue: Where do you define *children* field?
   - In *Component* superclass: Leaf subclasses don't use this field
   - In *Composite* subclass: Loss of uniformity (e.g., concerning child maintenance API)

55

Composite pattern: An example object diagram

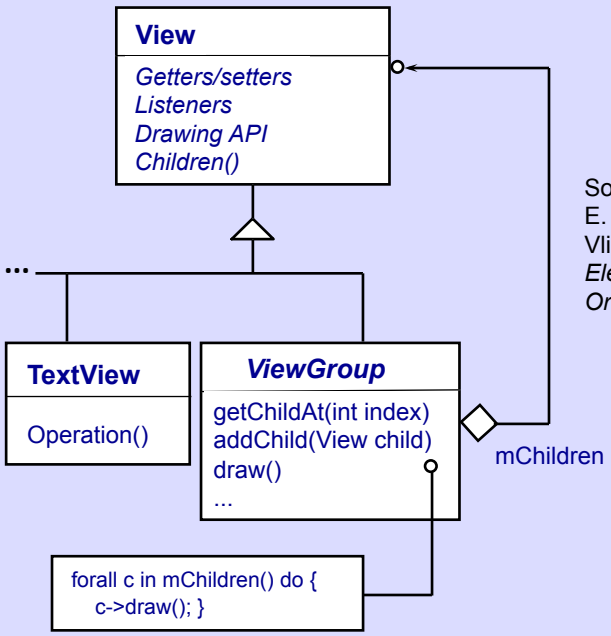


56

The *Composite* pattern and Android views

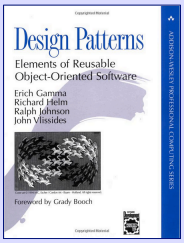
- How Android views implement *Composite*
  - *View* class matches *Component*
  - *ViewGroup* class matches *Composite*
  - Many leaf classes, e.g., *TextView* and its subclasses including *Button*
- *ViewGroup* has lots of direct and indirect subclasses
  - E.g., *AdapterView<T>, LinearLayout, RelativeLayout, FrameLayout, Gallery, Toolbar, SlidingDrawer, TvView, ViewPager, Spinner,* etc.

57



Android's take on *Composite*

Source:
E. Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*



58

View's APIs

- Position API – Position determined by four factors: (1) Distance from top of parent, (2) Distance from left margin of parent, (3) height, and (4) width
- All measures in pixels (px)
- Getter methods (all no-arg + final + returning integer)
  - *getTop()*
  - *getLeft()*
  - *getHeight()*
  - *getWidth()*
- Setter methods
  - *setLeft(int)*
  - *setTop(int)*

59

View's APIs (cont'd)

- Event handling APIs – Determine how *View* instances respond to user events
- Android's approach: Use Java interfaces nested in *View* class
- Each Java interface declares different abstract methods, e.g.,
  - *View.OnClickListener* → *onClick(View)*
  - *View.OnLongClickListener* → *onLongClick(View)*
  - *View.OnFocusChangeListener* → *onFocusChange(View, boolean)*
  - *View.OnKeyListener* → *onKey(View, int, KeyEvent)*
  - *View.OnLayoutChangeListener* → *onLayoutChange(View, …)*
  - *View.OnScrollChangeListener* → *onScrollChange(View, ...)*
- Handle events by creating appropriate instance of nested interface and associating instance with *View object*

60

Example of listener definition

- From *SimpleCounter2* app, activity *SimpleCounterActivity2*

```
public View.OnClickListener upListener = new View.OnClickListener() {

    // Called when up button is selected
    @Override
    public void onClick(View v) {
        incrementCount() ;
    }
} ;
…
public void onCreate(Bundle savedInstanceState) {

  super.onCreate(savedInstanceState);
  …
  up = (Button) findViewById(R.id.upButton);
  up.setOnClickListener(upListener) ;
```

upListener = instance of anonymous class implementing Java interface *View.OnClickListener.*

Anonymous class must implement abstract method *onClick().*

onCreate() associates *upListener* with *Button* view.

61

Comments on code

- Java interface names are legitimate type identifiers
  - Class implementing the interface can be *anonymous*
  - How defined? Give implementation (code) of abstract methods on-the-fly when declaring instance
    - Anonymous class is created automatically, along with an instance of the class
  - See earlier example
- Very convenient feature enhancing advantages of polymorphism in Java
- https://developer.android.com/reference/android/view/View
- https://developer.android.com/reference/android/view/ViewGroup

62

View's APIs: Drawing views

- Three stage process
  1. Figure out dimensions (width and height) of view object
  2. Position view in container
  3. Actually draw view
- APIs
  - *onMeasure()*
  - *onLayout()*
  - *onDraw()*
- Predefined *View* subclasses have methods above; programmer must define the methods for new views

  See https://developer.android.com/guide/topics/ui/index.html

63

View's additional APIs

- General properties of *View* instances have getter and setter methods
  - Visibility – *getVisibility(), setVisibility()*
  - Selected state – *isSelected(), setSelected()*
  - Clickable – *isClickable(), setClickable()*
  - Focus – *hasFocus(), requestFocus()*
  - … and many, many more

  See http://developer.android.com/reference/android/view/View.html

64

Examples of popular Android views

- Button
- SeekBar
- TextView
- CheckBox
- ListView
- WebView
- etc.

65

Android text views

- Root of hierarchy is class *TextView*
  - Main responsibility: Display text for user's benefit
  - Default behavior is no edit; either change default (declare attribute *android:editable="true"*) or use subclass *EditText* for editable text view
- Rich API allows user to define various fonts, color, sizes, backgrounds for the text box
- Key methods:
  - *getText()*—Returns a *CharSequence*
  - *setText()*—Overloaded, typically use with *CharSequence* arg
  - And all methods inherited from *View...*
- Doc: http://developer.android.com/reference/android/widget/TextView.html

66

Declaring a *TextView* in layout xml file

- Caveat: No need to edit xml directly, can use GUI builder in SDK to set properties

```
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
    android:orientation="vertical"
    ...

    <TextView    android:id="@+id/text"
                 android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="This is a text view!" />

    ...
</LinearLayout>
```

Arrange views vertically in linear layout.

id attribute allows Java to find this *View*.

Declare TextView and its attributes.

Done with linear layout tag.

67

Specifying an icon's width and height

- Can give exact size in density-independent pixel units (e.g., *7dp*) or use special keywords
  - *wrap_content*—size view to dimensions required by content
  - *match_parent*—as big as parent view (which directly contains this view)
- Doc: http://developer.android.com/guide/topics/ui/declaring-layout.html

68

Class *TextView*'s hierarchy

- Key direct subclasses
  - *EditText* – Similar to *TextView* but editable by default
  - *TextClock* – An animated view to display time
  - *Chronometer* – A timer class counting up by default (API: *start(), stop()*)
  - *Button* – Root of button hierarchy
- Doc: http://developer.android.com/reference/android/widget/TextView.html

69

Class *TextView*'s hierarchy (cont'd)

- Key indirect subclasses
  - *AutoCompleteTextView* – An *EditText* that shows completion suggestions automatically
  - *AppCompatTextView* – *EditText* with backward compatibility to older versions of Android
  - *CompoundButton* – Abstract *Button* subclass with "unchecked" and "checked" states (implements *Checkable* interface and *toggle()* method)
  - *CheckBox* – Concrete *CompoundButton* subclass showing check mark
  - *ToggleButton* – Concrete *CompoundButton* subclass with light indicator
  - *Switch* – Concrete *CompoundButton* subclass sliding right and left
  - Many "app compatibility" versions of *TextView* subclasses

70

Android buttons

- Text and/or icon reacting to a "press" events
- Three kinds of buttons: (1) text, (2) icon, (3) text + icon
- Create text button with <*Button*> tag – Specify text with *android:text* attribute, a *String* (e.g., *android:text* = *@string/string_name*)
- Create icon button with <*ImageButton*> tag – Specifiy icon with *android:src* attribute, a picture file (e.g., *android:src* = *@drawable/icon_file_name*)
- Create text + image button with <*Button*> tag – Specify text with *android:text* attribute and image with *android:drawableLeft* attribute (e.g., *android:drawableLeft* = *@drawable/drawable_file_name*)
- Doc: http://developer.android.com/guide/topics/ui/controls/button.html

71

Android buttons

- Must be associated with a listener to respond to clicks
  - Done programmatically or through GUI editor
  - Button listeners are instances of (classes implementing) *static* Java interface *View.OnClickListener* nested in *View* class
  - Respond to long clicks by instantiating *View.OnLongClickListener*
- *Listener*: A special kind of class that lets you specify actions in response to a user interaction with a view
- Doc: http://developer.android.com/reference/android/view/View.OnClickListener.html

  http://developer.android.com/reference/android/widget/Button.html

72

Setting button's short click listener

- **Programmatically**: Implement Java interface *View.OnClickListener* and create an instance of the resulting concrete class
  - Resulting class can be anonymous—Used only for that instance
  - Implementation must define just one method—*onClick()*
- Use *Button* setter method *setOnClickListener()* to set listener
- **Statically**: Set listener in layout file (avoid pain of anonymous class…)
  - In this case, specify attribute *android:onClick* in corresponding XML layout file with name of method that will be called, e.g.,

```
<Button …
    android:onClick="someMethodName"
    …/>
```

- Doc: http://developer.android.com/reference/android/view/View.OnClickListener.html

73

SimpleCounter4 button example

- Define listener (anywhere in *SimpleCounterActivity4* class)

Note *upListener*'s data type is an interface, not a class.

```
// Listener for the up button
 public View.OnClickListener upListener = new View.OnClickListener() {

   // Called when up button is selected
        @Override
        public void onClick(View v) {
             incrementCount() ;
        }
   } ;
```

Anonymous class must implement abstract method *onClick()* declared by Java interface *OnClickListener*.

Listener for up button just increments current count.

74

Declaring a button

- Done in appropriate location in */res/layout/layout-file.xml* file
- Use <Button> tag for button displaying text or text and an image
- This can be done interactively by dragging button from palette in IDE
- Main attributes: id (code identifier), *layout_width, layout_height, text* (the text displayed in the button), e.g.

```
<Button
     android:id="@+id/upButton"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/upString"
/>
```

- Doc: http://developer.android.com/guide/topics/ui/controls/button.html

75

Adding a picture to a button

- Use one of attributes *drawableRight, drawableLeft, drawableTop*, and *drawableBottom* with appropriate drawable value in <Button> tag

```
<Button
    android:id="@+id/upButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/upString"
    android:drawableLeft="@drawable/my_pic"
    android:drawablePadding="4dp"
/>
```

- Can also change button's background by specifying a given color or xml file containing <*selector*> tag (for default, focused and pressed state of the button)

76

Image buttons

- Use <ImageButton> tag for button displaying just an image
- Include file from which button will be retrieved with attribute *android:src*, e.g.,

```
<ImageButton
    android:id="@+id/upButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/upArrow"
/>
```

- Doc: http://developer.android.com/guide/topics/ui/controls/button.html

77

CheckBox

- Class *CheckBox* defines a check box view
- Subclass of *Button* by way of *CompoundButton*
- Define a two-state button: checked and unchecked, with some text to go with it
- Again, define checkbox in layout file and let instance be created when layout is inflated, e.g.,

```
<CheckBox xmlns:android="http:// … "
      android:id=@+id/aCheckBox
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text=@string/anInitialString
/>
```

78

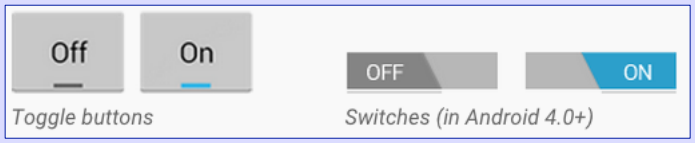
Interacting with check boxes

- Abstract superclass *CompoundButton* defines **nested static interface** *CompoundButton.OnCheckedChangeListener* and concrete method *setOnCheckedChangedListener()*
- *OnCheckedChangeListener* declares just one abstract method, *onCheckedChanged(CompoundButton, boolean)*
  - Method called when state of button has changed, passed the button in question and the new state of the button
- Additional relevant methods
  - *isChecked() : boolean*
  - *setChecked(Boolean) : void*
  - *toggle() : void*
- http://developer.android.com/reference/android/widget/CompoundButton.html

79

CheckBox variations

- Classes *ToggleButton* and *Switch* also define two-state buttons
- Both are subclasses of *CompoundButton*
- See http://developer.android.com/guide/topics/ui/controls/togglebutton.html



80

ToggleButton class

- Automatically displays text On/Off and fills area at bottom
- Toggle button responds to *onClick* event
  - Define attribute *android:onClick* in *ToggleButton* tag in layout
  - Callback method will take a *View* (the icon that was clicked), must be *public* and return *void*
- Toggle button can also use *OnCheckedChangedListener* in way similar to *CheckButton*

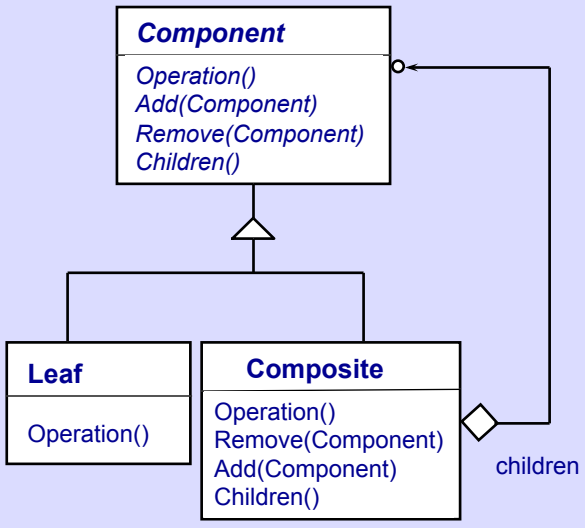
81

Class *ViewGroup*

- Ways to organize sets of icons
- Abstract subclass of *View*
- Main direct subclasses
  - Most layouts: *LinearLayout, RelativeLayout, ConstraintLayout, FrameLayout* (with subclass *ScrollView*), *GridLayout, DrawerLayout*
  - *AdapterView<T>*
  - *SlidingDrawer*
- Main indirect subclasses
  - *CalendarView, GridView, ListView, Spinner, TableLayout, TableRow, WebView, RadioGroup, TimePicker, DatePicker,* etc.
- https://developer.android.com/reference/android/view/ViewGroup.html

82

Recall *Composite* pattern…



83

ViewGroup in the *Composite* pattern

- Class *ViewGroup* plays the role of the *Composite* in the pattern
- See frameworks/base/core/java/android/view/ViewGroup.java:82,377—378

ViewGroup inherits from *View* just as *Composite* inherits from *Component*.

```
…
public abstract class ViewGroup extends View
                    implements ViewParent, ViewManager {
  …
 // Child views of this ViewGroup
  private View[] mChildren;  }
```

ViewGroup also inherites from interfaces *ViewParent* and *ViewManager*.

ViewGroup instances can contain multiple *View* instances.

84

ViewGroup's child maintenance API

- *addView(View* child) : **void**
- *addView(View* child*, int* index) : **void**
- *addView(View* child*, int* index, *ViewGroup.LayoutParams* params) : **void**
- *getChildAt(int* index) : *View*
- *getChildCount()* : **int**
- *indexOfChild(View* view*)* : **int**
- *removeView(View* view*)* : **void**
- *removeAllViews()* : **void**

85

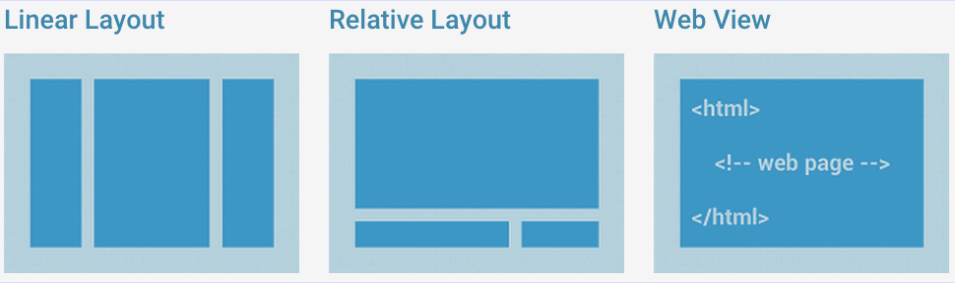
Layouts

- Ways to organize sets of icons
- Many alternatives predefined in latest versions of Android
  - ➢ Use for this class
- Source code: Different layout classes are subclasses of *ViewGroup*, a subclass of *View*
- Parent-child relationship between a layout object and the views it contains

86

Different Layouts

- Linear layout: Items arranged in a row (horizontally or vertically)
- Relative layout: Items arranged relative to each other
- Web view: Use for web pages (may also defer to browser app)
- Source: http://developer.android.com/guide/topics/ui/declaring-layout.html



87

Linear layouts

- Class *LinearLayout,* a subclass of *ViewGroup*
- Typically defined interactively using GUI builder, saved in layout.xml file (find it in "Layouts" section of palette)
  - Delimited by tags <*LinearLayout*> … </*LinearLayout*>
- Arrange items horizontally or vertically
  - Specified in *android:orientation* attribute
  - Values are "horizontal" and "vertical"
  - Can be changed programmatically with *setOrientation()*, using static fields *HORIZONTAL* and *VERTICAL*
- http://developer.android.com/reference/android/widget/LinearLayout.html

88

Layout weights

- Amount to allocate to a given layout child relative to the other children
- Sometimes described as a percentage— if total weights add up to 100
  - If not, Android still computes the relative proportion of each element with respect to the total
- In this case, specify *android:height="0dp"* and *android:weight="xx"* attributes for child contained in vertical linear layout
  - Use *android:width* attribute for horizontal layout
- Can mix and match weight, *wrap_content*, and *dp* specs
  - Weights use spaces remaining after other specs accounted for
- https://developer.android.com/guide/topics/ui/layout/linear.html

89

Gravity

- *Gravity*—Define position where each child will be placed in container
- Default case (no gravity specified): Top and left aligned
- Used both to define gravity of child within layout and contents of a view (e.g., the text within a text view)
- Set statically with *android:gravity* attribute in layout file or programmatically (dynamically) using *setGravity()* method, e.g.,
  - *android:gravity="bottom"* (start from bottom of linear layout)
  - *android:gravity="right|bottom"* (start from bottom right),
  - *android:gravity="center_horizontal|center_vertical"*, etc.
- Doc: http://developer.android.com/reference/android/view/Gravity.html https://developer.android.com/guide/topics/ui/layout/linear.html

90

Relative layouts

- "A Layout where the positions of the children can be described in relation to each other or to the parent" [http://developer.android.com/reference/android/widget/RelativeLayout.html]
- Class *RelativeLayout*, a subclass of *ViewGroup*
- Efficient alternative to nesting *LinearLayout* views
- Positions views in relative layout are specified using boolean attributes (see next)

91

Positions relative to parent

- Popular attributes (set to *"true"* or *"false"*):
  - android:layout_alignParentTop—Align with top of container
  - android:layout_alignParentBottom—Align with bottom of container
  - android:layout_alignParentLeft—Align with left side of container
  - android:layout_alignParentRight—Align with right side of container
  - android:layout_centerHorizontal—Center in container (hor.)
  - android:layout_centerVertical—Center in container (vert.)
  - android:layout_fillHorizontal—Fill container's width
  - android:layout_fillVertical—Fill container's height

92

Positions relative to siblings

- Popular attributes (use view (id) reference as value):
  - android:layout_above—Place above referenced icon
  - android:layout_below—Place below referenced icon
  - android:layout_toLeftOf—Align with left side of icon
  - android:layout_toRightOf—Align with right side of icon
  - android:layout_alignTop—Align with top of referenced icon
  - android:layout_alignBottom—Align with bottom of referenced icon
  - android:layout_alignLeft—Align left sides
  - android:layout_alignRight—Align right sides
- Example: *android:layout_alignBottom="@+id/name_of_other_view"*

93

Other useful layouts

- *TableLayout*—Use extends beyond tables; ability to use cell coordinates for item placement
- *ScrollView*—XML Wrapper that provides vertical scroll bar for views that are too tall to fit on screen
  - Do not use with *TextView* or *ListView*; these views already contain scrollbars
- *HorizontalScrollView*—Similar to *ScrollView* but provides horizontal scroll bar if needed
- *ConstraintLayout*—Similar to relative layout but gives system greater flexibility in arranging UI elements within layout
  - Similar to iOS approach to layout configurations
  - https://developer.android.com/training/constraint-layout/

94

Constraint layouts

- "Flat" structure of display views, similar to relative layouts
- Goal: Capture typical designs produced with IDE's Layout Editor
- View positioning determined by constraints
  - Specify at least one horizontal and one vertical constraint per view
  - Constraints specify "connections" to other views, parent view or invisible guidelines
  - Use "handles" on each side of the view to define constraints relative to other views or parent container

95

Constraint kinds

- Examples of constraints
  - Relative positioning
  - Margins
  - Center positioning
  - Etc.
- https://developer.android.com/reference/android/support/constraint/ConstraintLayout

96

Class *AdapterView*

- View group whose children are determined by an *Adapter*
- Examples of adapter views: Lists, grids, galleries
- Adapter's main responsibilities:
  1. Store data to be displayed in view group
  2. Creating a view for each item in view group, using special layout file
- *AdapterView*'s main responsibility is to display data provided by adapter

97

Class *AdapterView*

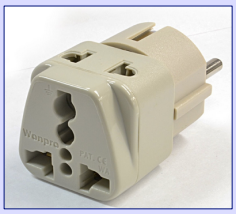
- Source code: *AdapterView* is an abstract template subclass of *ViewGroup*
  - frameworks/base/core/java/android/widget/AdapterView.java:49

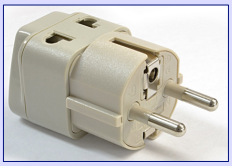
    public abstract class AdapterView<T extends Adapter> extends ViewGroup
- Popular subclasses: *Spinner* (dropdown menu), *ListView, GridView,* and *Gallery*
- "Adapter" is a popular pattern from the Gang-of-Four system (see next…)

98

Adapter pattern

- Goal: manage incompatible interfaces between classes
- Say that client expects certain APIs of a given class *C*…
- … but that class *C* offers different methods in its API
- Adapter pattern creates a new class whose job is to fit the client's calls into the methods exposed by class *C*
  - This is similar to the adapter you use in electrical outlets when you travel abroad

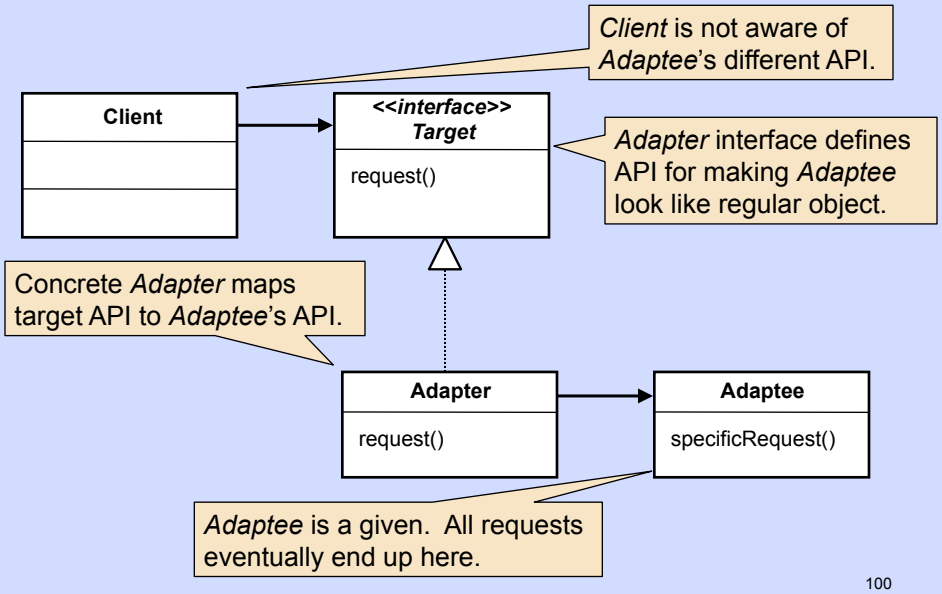




Pictures of Schuko adapter from www.amazon.com.

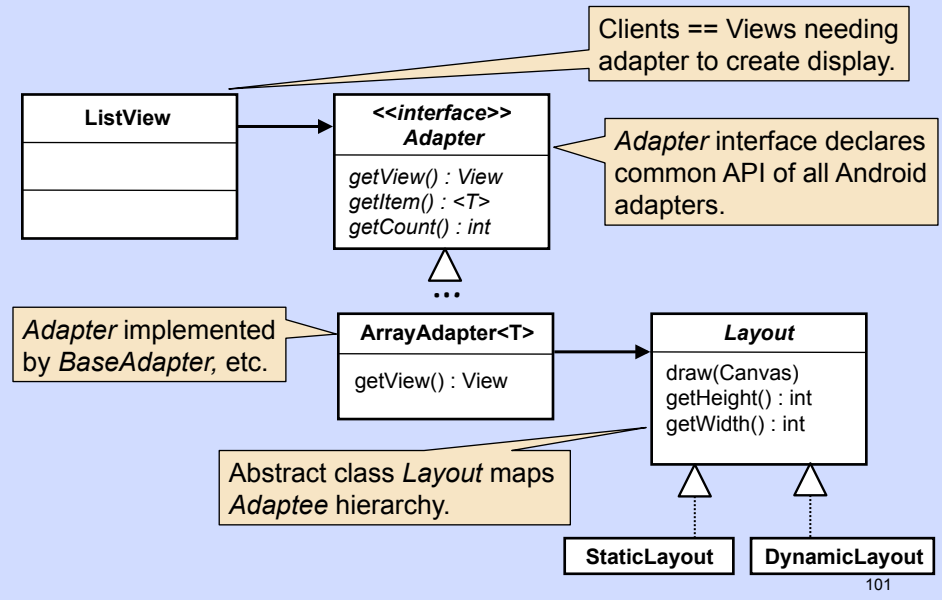
99

Class diagram of *Adapter*



100

Android's implementation of *Adapter* pattern



101

Creating instance of built-in adapter

- First, define layout file of each item to be shown in container
- Next, create appropriate adapter to fit each item's information into the item's format defined in a layout file in activity with *AdapterView*
- Typical actions (variations possible):
  1. Declare an adapter field, *mAdapter* in activity class
  2. In activity's *onCreate()* method instantiate *Adapter* subclass, e.g., *ArrayAdapter,* bind instance to *mAdapter*
  3. Attach adapter to view requiring it (e.g., a *ListView*), e.g., by calling *setListAdapter()*
- Key adapter method: *getView()*

102

Recall adapter's two main responsibilities...

- Adapter's main responsibilities:
  1. Store data to be displayed in view group
  2. Creating a view for each item in view group, using special layout file
- *AdapterView*'s main responsibility is to display data provided by adapter

103

ArrayAdapter

- Perhaps the simplest built-in adapter, convert array elements to strings (using Java's *toString()* method)
- *ArrayAdapter* converts list elements in an array to strings
- Commonly used *ArrayAdapter* constructor takes 3 args:
  1. Context (e.g., *Activity* instance)
  2. Layout resource used to inflate each list item
  3. Array of items (possibly coming from the *Strings.xml* resource)
- Normally, adapter instance holds list of items to be displayed (either as a list of resource references, or the actual items)
- http://developer.android.com/reference/android/widget/ArrayAdapter.html

104

Creating a *ListView*

- Key points:
  1. Define a list widget
     - Either include a *ListView* in some layout (e.g., *LinearLayout*), or
     - Define activity as *ListActivity* subclass, no need for layout file in this case (but could still be included)
       - *ListActivity* = *Activity* subclass that hosts a *ListView* by default
       - Use *getListView()* to find *ListView* object in this case
  2. Create adapter (e.g., an *ArrayAdapter*), attach to list by calling *setListAdapter(mAdapter)*
- http://developer.android.com/reference/android/app/ListActivity.html
  http://developer.android.com/reference/android/widget/ListView.html

105

Listening to list item selections

- Class *AdapterView* declares static interface *OnListItemClickListener*
- *OnListItemClickListener* declares **public void** method *onItemClick()*
  - Method signature:
    1. *AdapterView<?> parent*—Adapter view containing item clicked
    2. *View view*—Selected item
    3. *int position*—Position of item in list (starting from 0)
    4. *long id*—Resource identifier of the selected item
- Doc: http://developer.android.com/reference/android/widget/AdapterView.OnItemClickListener.html

106

Java interface *AdapterView.OnItemClickListener*

1. Implement interface (perhaps using anonymous class) by defining method *onItemClick()*
2. Create class instance
3. Set listener in *ListView* instance by calling *setOnItemClickListener()*

Create new instance of interface *OnItemClickListener.*

```
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    public void onItemClick(AdapterView<?> parent,
                            View view, int position, long id) {

      // specify actions in response to click
    }
  }) ;
```

Must implement abstract method *onItemClick().*

107

Listening to list item selections in *ListActivity*

- Class *ListActivity* defines method *onListItemClick()*
  - Parameters (similar to *onItemClick()* above):
    1. *ListView* parent—List view containing item selected
    2. *View* view—Selected item
    3. *int* position—Position of item in list (starting from 0)
    4. *long* id—Id of the selected item
- This method is an alternative to defining a listener for the *ListView*
  - No more need to call *setOnItemClickListener()* on *ListView* object
- Doc: http://developer.android.com/reference/android/app/ListActivity.html

108

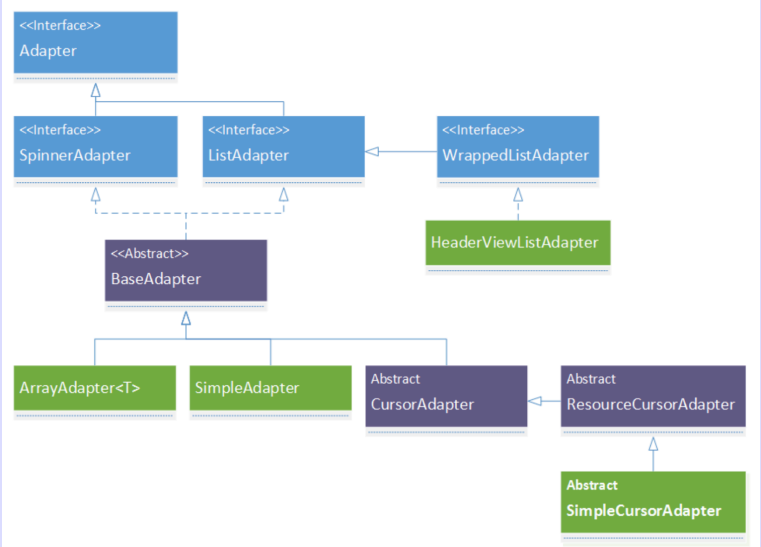
Creating your own adapters

- Sometimes predefined adapters don't work well for your list items
  - Grid view example: Arrange pictures in rows and columns
  - Adapter must "squeeze" a set of image files into grid tiles
- Defining your adapter, e.g., *MyAdapter*
  1. Define *MyAdapter* to extend abstract superclass *BaseAdapter*
  2. Define following methods (at least):
     - *getView()*: Creates (or Recycles) and fills view for a given grid item
     - *getItemId()*: Return the resource reference for item at position
     - *getCount()*: Returns the number of items managed by adapter
- Doc: https://developer.android.com/guide/topics/ui/binding
  https://developer.android.com/reference/android/widget/Adapter

109

Predefined adapter hierarchy

https://www.intertech.com/Blog/wp-content/uploads/2014/06/HeirarchyOfAdapter.png



110

Creating your own adapters: Details

- Pictures specified as resources
  - Place in a *drawable* folder
  - Customizable for pixel density of hosting device (e.g., *drawable-hdpi, drawable-xhdpi, drawable-xxhdpi*, etc.)
  - Default: *drawable-nodpi*
- *OnCreate()* method: Place resources in *ArrayList<Integer>*,
  - Pass list to adapter instance which will save list in appropriate field
  - Attach adapter to *GridView* instance with *setAdapter()*
  - Define listener (similar to *ListView*)
- Doc: http://developer.android.com/reference/android/widget/GridLayout.html

111

Fine points about image adapters

- Use *ImageView* widget class to display picture
  - Size of picture and *ImageView* widget may differ
  - Enumeration *ImageView.ScaleType* defines options for scaling picture in *ImageView* to size of *ImageView*
  - Popular values (many more available):
    - CENTER (center picture w/o scaling)
    - CENTER_CROP (scale picture until height and width as large or larger than view while maintaining aspect ratio and center into view)
    - CENTER_INSIDE (scale until smaller than view maintaining aspect ratio and center into view)
- Doc: http://developer.android.com/reference/android/widget/ImageView.html

112

Specifying *GridView* resource

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

   <GridView
      android:id="@+id/gridview"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:columnWidth="80dp"
      android:gravity="center"
      android:horizontalSpacing="10dp"
      android:verticalSpacing="10dp"
      android:numColumns="auto_fit"
      android:stretchMode="columnWidth" />
</LinearLayout>
```

Use *GridView* tag.

The standard attributes.

Important: Specify column width for grid.

Space between view items.

Let grid determine number of columns.

113

Main activity class

Usual declaration of an activity class.

Define images to be displayed.

```
public class GridLayoutActivity extends Activity {
  …
  private ArrayList<Integer> mThumbIdsFlowers = new ArrayList<Integer>(
      Arrays.asList(R.drawable.image1, R.drawable.image2, …));

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    GridView gridview = (GridView) findViewById(R.id.gridview);
    gridview.setAdapter(new ImageAdapter(this, mThumbIdsFlowers));
    gridview.setOnItemClickListener(new OnItemClickListener() {
      public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
        Intent intent = new Intent(GridLayoutActivity.this,
            ImageViewActivity.class);
        intent.putExtra(EXTRA_RES_ID, (int) id);
        startActivity(intent);  }
    });
  } }
```

Usual *onCreate()* code.

Now instantiate special adapter.

Implement *OnItemClickListener* interface. *onItemClick()* starts new activity, passing image id.

114

ImageAdapter class

Special adapter must inherit from *BaseAdapter.*

```
public class ImageAdapter extends BaseAdapter {
    private static final int PADDING = 8;
    private static final int WIDTH = 250;
    private static final int HEIGHT = 250;
    private Context mContext;
    private List<Integer> mThumbIds;

    // Store the list of image IDs
    public ImageAdapter(Context c, List<Integer> ids) {
        mContext = c;
        this.mThumbIds = ids;   }

    // Return the number of items in the Adapter
    public int getCount() {
        return mThumbIds.size();      }

    // Will get called to provide the ID that
    // is passed to OnItemClickListener.onItemClick()
    public long getItemId(int position) {
        return mThumbIds.get(position);  }
```

Constructor gets context and list of image identifiers.

Return resource identifier of i-th list item.

115

ImageAdapter class (cont'd)

```
public class ImageAdapter extends BaseAdapter {
    …
    // Return an ImageView for each item referenced by the Adapter
    public View getView(int position, View convertView, ViewGroup parent) {

        ImageView imageView = (ImageView) convertView;

        // if convertView's not recycled, initialize some attributes
        if (imageView == null) {
            imageView = new ImageView(mContext);
            imageView.setLayoutParams(new GridView.LayoutParams(WIDTH, HEIGHT));
            imageView.setPadding(PADDING, PADDING, PADDING, PADDING);
            imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
        }
        imageView.setImageResource(mThumbIds.get(position));
        return imageView;
    }
}
```

Method *getView()* returns view to be displayed for grid item.

Create new *ImageView.*

Set view's params.

Crop image around center for thumbnail display.

116

ImageViewActivity class

```
public class ImageViewActivity extends Activity {

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Get the Intent used to start this Activity
    Intent intent = getIntent();

    // Make a new ImageView
    ImageView imageView = new ImageView(getApplicationContext());

    // Get the ID of the image to display and set it as the image for this ImageView
    imageView.setImageResource(intent.getIntExtra(GridLayoutActivity.EXTRA_RES_ID, 0));

    setContentView(imageView);
  }
}
```

This activity is called when selected picture is displayed

Create new *ImageView.*

Set image for new *ImageView.*

setContentView() uses new *ImageView* to populate screen.

117

Menus

- We consider three kinds of Android menus
  - Options menus

    Display global options for an entire activity

    User brings up by pressing "menu" soft key
  - Context menus

    Floating displays of choices pertaining to a specific view

    Usually brought up by "long click" on the view

    Example: Bring up list of options for a contact in the Contact's *PeopleActivity* listing a bunch of contacts
  - Submenus

    Menus appearing after an item was selected in previous menu
- Doc: http://developer.android.com/reference/android/view/Menu.html

118

Defining menus in XML

- Use special *res* subfolder named *menu*
  - Define one XML file for each options or context menu
  - Support for different locales, configurations, devices, etc.
  - Top-level tag: <menu> … </menu>
    - When inflated, this creates an instance of *Menu* class
  - Within <menu> tag, use <item> and <item> … </item> tags to define menu options
    - When inflated, these create *MenuItem* instances
  - <menu> tag nested in <item> tag defines submenu
  - <group> … </group> tag groups common items (invisibly) sharing common properties (e.g., active state or visibility)

119

Defining menus in XML (cont'd)

- Key attributes of menu items:
  - android:id — The usual
  - android:icon — Reference to drawable that is the item's icon
  - android:title — Reference to string that is the item's display text
  - android:showAsAction — Specifies whether item should appear in action bar (for options menu only)

    By default all items displayed in overflow area unless

    android:showAsAction="ifRoom" or
    android:showAsAction="always" or
    android:showAsAction="never"
- See example from http://developer.android.com/guide/topics/ui/menus.html

120

Example of menu declaration: A file menu

menu tag declares main menu.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/file"
      android:title="@string/file" >
    <!-- "file" submenu -->
    <menu>
      <item android:id="@+id/create_new"
          android:title="@string/create_new" />
      <item android:id="@+id/open"
          android:title="@string/open" />
    </menu>
  </item>
</menu>
```

item tag declares item in main menu and item's attributes.

Nested menu tag creates submenu.

Items in submenu.

Source: https://developer.android.com/guide/topics/ui/menus.html

121

Options menus

- Brought up (expanded) when user clicks on menu icon
  - Was a hardware button up to Gingerbread (Android 2.3)
  - Since Honeycomb (Android 3.0) different places (often bottom left or right of display, also action bar with overflow area on top right)
  - Global options for an activity, e.g., settings, search, etc.
- Doc: http://developer.android.com/guide/topics/ui/menus.html

122

Defining menus in XML (cont'd)

- Options menu support in *Action Bar*
- Source: http://developer.android.com/guide/topics/ui/menus.html

Legend:
1. App icon and name
2. Action buttons
3. Overflow area

Tabs allowing user to switch views.

123

Options menus (cont'd)

Three key methods in life time of options menu

1. *onCreateOptionsMenu()* –

   Set up options menu (called once only by OS)

2. *onPrepareOptionsMenu()* –

   Initialization actions (e.g., enable vs. disable items)

   Called by OS after *invalidateOptionsMenu()* called

3. *onOptionsItemSelected()* –

   Handle user interactions

   Called by OS whenever user selects a menu item

- Doc: http://developer.android.com/reference/android/app/Activity.html - onCreateOptionsMenu(android.view.Menu)

124

Options menus (cont'd)

- Define an option menu by overriding *onCreateOptionsMenu()*
  - *onCreateOptionsMenu(Menu)* : *boolean*
    - — Defined in *Activity* class
    - — Create view displaying content of options menu using *inflater*
    - — Return value *true* to indicate that menu must be displayed now
    - — Called only first time menu called
  - *onPrepareOptionsMenu(Menu)* : *boolean*

    Typical *onPrepareOptionsMenu()* actions: Enable vs. disable item options, add/remove items, etc.
- Doc: http://developer.android.com/reference/android/app/Activity.html - onCreateOptionsMenu(android.view.Menu)

125

Options menus (cont'd)

- Key methods (cont'd)
  - *onOptionsItemSelected(MenuItem)* : *boolean*

    Callback also defined in *Activity* class, can also be used in fragments

    Typical actions: (1) get *MenuItem*'s id and (2) switch on the id of selected item

    Default action returns *false* to let normal processing to continue, e.g., to call the *MenuItem*'s *Runnable*

    Return *true to* indicate that menu processing is done, no need to propagate further
- Doc: https://developer.android.com/guide/topics/ui/menus.html#RespondingOptionsMenu

126

Creating an options menu

- Typical action for *onCreateOptionsMenu()*: Inflate XML spec
- Inflater = Android module that maps XML layout spec into a Java *View* object and displays the object
- Source: http://developer.android.com/guide/topics/ui/menus.html

onCreateOptionsMenu() callback is passed a *Menu* instance.

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

Return *MenuInflater* instance.

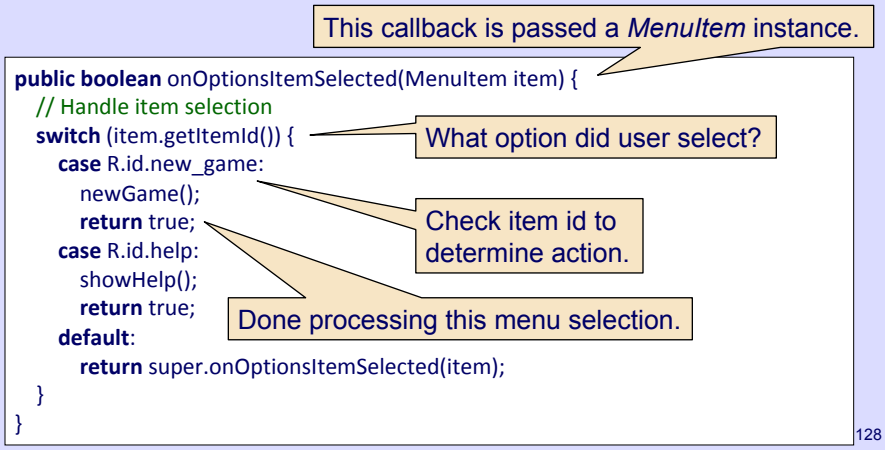
Inflate resource XML spec into *Menu* instance.

Display menu in host activity now.

127

Handling options selections

- Example code for *onOptionsItemSelected()*: Figure out what to do…
- Source: http://developer.android.com/guide/topics/ui/menus.html




```
public boolean onOptionsItemSelected(MenuItem item) {
   // Handle item selection
   switch (item.getItemId()) {
      case R.id.new_game:
         newGame();
         return true;
      case R.id.help:
         showHelp();
         return true;
      default:
         return super.onOptionsItemSelected(item);
   }
}
```

128

Options menus (cont'd)

- Supporting resource
  - Use special folder *menu* in project's *res* directory
  - Define an XML file for each menu in application
  - Use <menu> … </menu> tag
  - Within tag, use <item> or <item> … </item> tag to specify each menu item
  - Each item uses attributes *id, icon* (a drawable), and *title* (a string)
  - Specify submenus by nesting new <menu> … </menu> tag within an <item> … </item> tag

129

Options menus (cont'd)

- Changing enabled status of menu items
  - Send message *getItem(int index)* or *findItem(int id)* to options menu to get *MenuItem* instance
  - Send message *setEnabled(boolean)* to *MenuItem* instance to change enabled status

130

Context menus

- Brought up (expanded) by long click on a specific icon in a display
- Behavior is very similar to option menus, however:
  - Associate specific view with a context menu by calling *Activity* method

    ```
    registerForContextMenu(aView) ;
    ```

  - Methods are now called *onCreateContextMenu()* and *onContextItemSelected()*
  - *onCreateContextMenu()* now takes a menu, the view selected, and additional information about the menu
  - See *HelloAndroidWithMenus* app from Porter's Coursera MOOC

131

Creating a context menu

- Callback *onCreateContextMenu()* inflates approprite XML spec
- Source: http://developer.android.com/guide/topics/ui/menus.html

onCreateContextMenu() callback is passed a *Menu* instance, the *View* object associated with the menu, some extra info.

```
public void onCreateContextMenu(ContextMenu menu, View v,
                    ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);
  MenuInflater inflater = getMenuInflater();
  inflater.inflate(R.menu.context_menu, menu);
}
```

Get *MenuInflater* instance.

Inflate appropriate XML menu resource into *Menu* instance.

132

Handling context selections

- Example code for *onContextItemSelected()*: Figure out what to do…
- Source: http://developer.android.com/guide/topics/ui/menus.html

This callback is passed a *MenuItem* instance.

```
public boolean onContextItemSelected(MenuItem item) {
  AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
  switch (item.getItemId()) {
    case R.id.edit:
      editNote(info.id);
      return true;
    case R.id.delete:
      deleteNote(info.id);
      return true;
    default:
      return super.onContextItemSelected(item);
  }
}
```

Check item id and perform appropriate action.

133

The *Action Bar*

- Dedicated area on top of an app's display
- Three main goals:
  1. Identify the app (e.g., by app name + icon)
  2. Access frequently-used actions, perhaps in overflow area
  3. Support navigation tabs
- Aka the *app bar*
- Doc: http://developer.android.com/training/appbar/index.html

134

The *Action Bar* (cont'd)

- Recall action bar's picture below
- Source: http://developer.android.com/guide/topics/ui/menus.html

Legend:
1. App icon and name
2. Action buttons
3. Overflow area

Tabs allowing user to switch views.

135

Setting up the *Action Bar*

- Traditionally managed by class *ActionBar*, constantly evolving
- Now use class *Toolbar* instead for compatibility with previous versions
- Action bar introduced in Honeycomb (V 3.0) – API level 11
- Library *v7 appcompat* and class *AppCompatActivity* provide support down to Éclair (V2.1) – API level 7
- Constant evolution through addition of new features, e.g.,

  Material design experience introduced in Lollypop
  - See http://developer.android.com/design/material/index.html
  - Beyond our scope…
- Doc: http://developer.android.com/training/appbar/setting-up.html

136

Steps in setting up *Action Bar*

1. Add *v7 appcompat* library to Studio project
   - Download *Android Support Library* and *Android Support Repository* in SDK (under the *SDK Tools* tab in SDK Manager window)
   - Add *maven* repository in project *build.gradle* file

```
allprojects {
   repositories {
        google()
        // if you're using a version of Gradle lower than 4.1, you must
        // instead use
        // maven {
        //     url "https://maven.google.com"
        // }
   }
}
```

   - See http://developer.android.com/tools/support-library/setup.html

137

Steps in setting up *Action Bar*

2. Add support library to *dependencies* section

```
dependencies {
   ...
   implementation "com.android.support:support-core-utils::28.0.0"
}
```

138

Steps in setting up *Action Bar*

3. Define your activity as subclass of *AppCompatActivity* instead of *Activity*
   - *AppCompatActivity* == *Activity* subclass supporting latest app bar
   - Import *android.support.v7.app.ActionBar* if (7 ≤ min API ≤ 10)
   - Import *android.app.ActionBar* if (min API ≥ 11)
   - See http://developer.android.com/tools/support-library/setup.html

139

Steps in setting up *Action Bar* (cont'd)

4. Manifest file: Set <application> tag to include a *NoActionBar* theme
   - Prevent use of native *ActionBar* class in device

```
<application
    android:theme="@style/Theme.AppCompat.Light.NoActionBar"
    />
```

5. Add toolbar to activity's layout file

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar_1"
    android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />
```

   - See http://android-developers.blogspot.com/2014/10/appcompat-v21-material-design-for-pre.html

140

Steps in setting up *Action Bar* (cont'd)

6. In activity's *onCreate()* method, find toolbar view and set toolbar's content to be that view
   - Instruct toolbar to act as action bar with call to *setSupportActionBar()*

```
protected void onCreate() {
    super.onCreate() ;
    …
    Toolbar myToolbar = (Toolbar) findViewById(R.id.toolbar_1) ;
    setSupportActionBar(myToolbar) ;
    …
    }
```

7. Now use options menu to add shortcuts to app bar and to populate overflow area (see details in next slides…)

141

Action Bar caveat

- If app contains multiple activities that must display the same action bar, avoid code duplication (tedious + error prone)
  - Define the toolbar in its own layout file, e.g., *res/layout/toolbar_1.xml*

```
<android.support.v7.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:iosched="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar_actionbar"
    iosched:popupTheme="style/ActionBarPopupThemeOverlay"
    ioshed:theme="style/ActionBarPopupThemeOverlay"
    … />
```

  - Include layout file *toolbar_1.xml* in all layouts that want it

```
<LinearLayout …
    <include layout="@layout/toolbar_1" />
```

142

Using the *Action Bar*

- Define activity's name as android:label attribute of *activity* tag (a string)
- Use *options menu* to define both action buttons and overflow area
- By default, items in *options menu* will be displayed in overflow area
  - Change by using attribute android:showAsAction

    android:showAsAction="ifRoom" or…

    android:showAsAction="never"

    Other options for attribute value: "withText" (to add text to icon), "always" (use carefully), and "never"
- Use *onOptionsItemSelected(menuItem) : boolean* to respond to item selection by user

  Doc: http://developer.android.com/training/appbar/actions.html

143

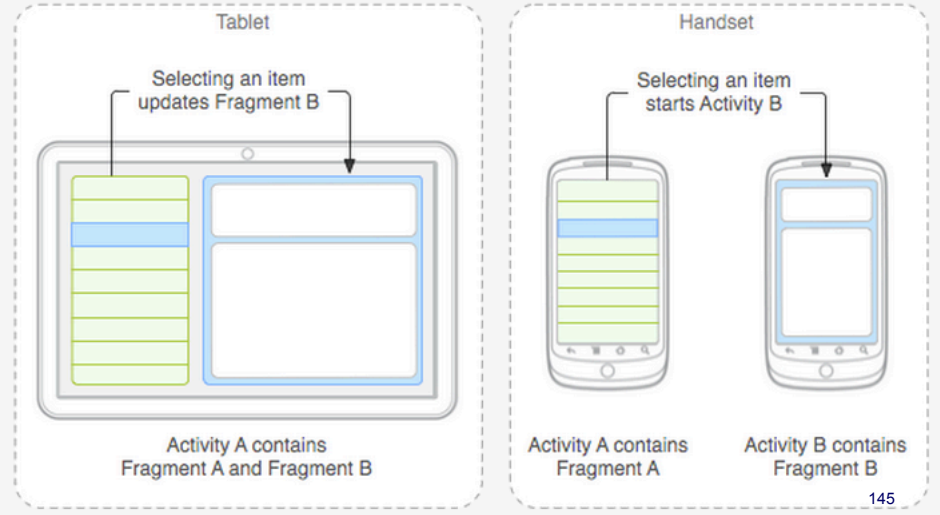
Fragments

- Support for dividing activity's display into multiple panes
- Introduced in Honeycomb V3.0 (API level 11)
  - Specifically intended to support larger tablet displays
  - Still working with "1 activity = 1 screen" metaphor
- Much more screen real-estate on a table than on a phone
  - Use tablet's display more efficiently (avoid empty space)
  - Display multiple things at once, thereby avoiding needless navigation between activities

144

Supporting tablets and phones with fragments

- Source: http://developer.android.com/guide/components/fragments.html



145

Fragments

- Fragment = Portion of an activity
  - Fragment must be hosted in activity in order to be displayed
  - Fragment defines its own layout and behavior
  - Compose multiple fragments to form an activity
  - Same fragment can be used in multiple activities
  - Simultaneous support for tablets and phones
- Doc: http://developer.android.com/guide/components/fragments.html

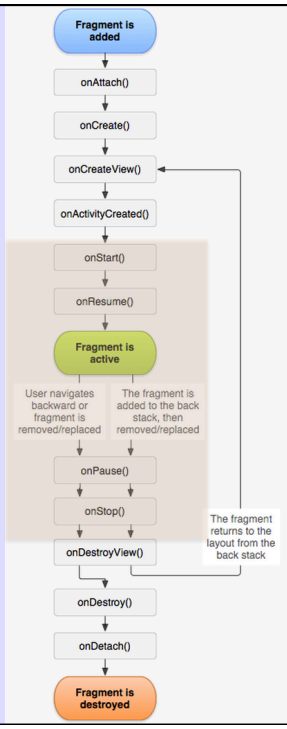
146

Fragments

- Fragment is tied to a *containing activity*
  - Before it can be displayed, fragment must be *attached* to hosting activity
  - Lifetime of fragment is similar to an activity's, but there are additional events, when fragment is first created, attached to an activity or detached from the activity
  - When activity paused (stopped, resumed, etc.), all fragments it contains are also paused (stopped, resumed, etc.)
  - While activity running, fragments can be dynamically added and removed, etc.
  - Fragments can be added to back stack to allow navigation with device's *back* button

[http://developer.android.com/guide/components/fragments.html – Lifecycle]

147



Fragment lifecycle

Mirror lifecycle events of hosting activity

Fragment will execute *onStart()*, *onResume()*, *onPause()*, *onStop()* when hosting activity does

Five additional methods:

- *onAttach()*
- *onCreateView()*
- *onActivityCreated()*
- *onDestroyView()*
- *onDetach()*

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

148

Fragment-specific callbacks

- *onAttach(Context activity)* — Called when fragment is attached to activity (record activity object)
- *onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState): View* – Inflate content view displayed in fragment
- *onActivityCreated(Bundle savedInstanceState)* — Called when containing activity completes *onCreate()* method
- *onDestroyView()* — Called when previously created view hierarchy is deleted
- *onDetach()* — Called when fragment is no longer attached to its activity
- *onSaveInstanceState(Bundle savedState)* — Saves state in *Bundle* that will be passed to *onCreate(), onCreateView()* and *onActivityCreated()*

[http://developer.android.com/reference/android/app/Fragment.html]

149

Programming fragments

Minimally, provide following callbacks

- *onAttach()*—Save the containing activity in a fragment field
- *onCreate()*—Android calls when fragment created (a constructor?)

  Initialize fragment components that must be retained when fragment is stopped, and then resumed
- *onCreateView()*—Called when fragment must generate its visual display for the first time (return a *View*)

  Gets passed a *LayoutInflater*, a *ViewGroup* (the container), and a *Bundle* (*savedInstanceState*)

  Uses inflater to create the view
- *onPause()*—Commit changes that must be saved

150

Order of method execution

- Creation methods are called on containing activity first, followed by fragments
  - E.g., *onCreate(), onStart(), onResume()*
- Deletion methods are called on fragments first, followed by containing activity
  - E.g., *onPause(), onStop(), onDestroy()*

151

Example order of method execution

- From fragment example we'll see later on

```
I/QuoteViewerActivity: entered onCreate()
I/TitlesFragment: entered onAttach()
I/TitlesFragment: entered onCreate()
I/TitlesFragment: entered onCreateView()
I/QuoteViewerActivity: entered onStart()
I/TitlesFragment: entered onStart()
I/QuoteViewerActivity: entered onResume()
I/TitlesFragment: entered onResume()
I/TitlesFragment: entered onPause()
I/QuoteViewerActivity: entered onPause()
I/TitlesFragment: entered onStop()
I/QuoteViewerActivity: entered onStop()
I/TitlesFragment: entered onDestroyView()
I/TitlesFragment: entered onDestroy()
I/TitlesFragment: entered onDetach()
I/QuoteViewerActivity: entered onDestroy()
```

Activity does "creation" callbacks first

Now activity is visible and interacting.

Fragment does "deletion" callbacks first.

152

Java and fragments

- Fragments are subclasses of app framework's class *Fragment*
  - Fragment's code: Define fragment class as a subclass of *Fragment*

    Similar to activities being subclasses of *Activity*
  - Defining fragment's instance either
    1. Statically – In XML layout file of containing activity
    2. Programmatically – Java code of containing activity

153

Static fragment definition

- Defining fragment instance *statically*:
  - Activity's layout file defines fragments explicitly
  - Use <fragment … /> tag, e.g., in linear layout
  - Attribute android:name identifies class defining fragment, e.g.,
    - android:name="edu.uic.cs478.examples.MyFragment"
  - Also, need android:id to identify fragment programmatically
  - Can differentiate between *layout, layout-land, layout-large*, etc.
  - See http://developer.android.com/guide/practices/screens_support.html
  - Example is next…

154

Example of XML fragment specification

- Example XML layout file for activity containing fragments

  From http://developer.android.com/guide/practices/screens_support.html

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <fragment android:name="com.example.news.ArticleListFragment"
       android:id="@+id/list"
       android:layout_weight="1"
       android:layout_width="0dp"
       android:layout_height="match_parent" />
  <fragment android:name="com.example.news.ArticleReaderFragment"
       android:id="@+id/viewer"
       android:layout_weight="2"
       android:layout_width="0dp"
       android:layout_height="match_parent" />
</LinearLayout>
```

Usual *LinearLayout* attributes.

Width defined by weights.

The class name

The second fragment in this layout.

155

Dynamic fragment definition

- Define fragment programmatically
  - Activity hosting a fragment must make room for it
  - Typically reserve some space with a *FrameLayout* in containing activity
  - Then, use *fragment transaction* to create, add and remove fragment instance dynamically

    http://developer.android.com/reference/android/widget/FrameLayout.html:

    > FrameLayout is designed to block out an area on the screen to display a single item. Generally, FrameLayout should be used to hold a single child view, because it can be difficult to organize child views in a way that's scalable to different screen sizes without the children overlapping each other. …
    >
    > Child views are drawn in a stack, with the most recently added child on top.

156

XML layout of activity with dynamic fragments

```
<LinearLayout xmlns:android="blah! blah! blah!"
   android:id="@+id/someName"
   android:orientation="horizontal"
   etc. >
   <FrameLayout
      android:id="@+id/someFrame"
      android:width="0dp"
      android:height="matchParent"
      android:weight="1">
   </FrameLayout>
   <FrameLayout
      android:id="@+id/someOtherFrame"
      android:width="0dp"
      android:height="matchParent"
      android:weight="3">
   </FrameLayout>
</LinearLayout>
```

Usual spec of a *LinearLayout*.

This *FrameLayout* takes 25% of *LinearLayout*'s width.

This *FrameLayout* takes remaining 75% of *LinearLayout*'s width.

157

Defining a fragment's layout

- Once more, define fragment layouts either:
  1. Statically – Use XML layout file for fragment
  2. Programmatically – Java code in *onCreateView()* callback
- Setup fragment transactions (e.g., add to an activity) for inclusion in back stack by calling *addToBackStack()* whenever fragment is added to view

  Activity events are included in back stack automatically, fragments events are not

158

Generating fragment's display

- Done in fragment callback *onCreateView()*, which returns root *View*
- *inflate()* takes: (1) layout file, (2) parent view (optional), and (3) boolean indicating whether returned view is attached to parent (use *false*)

  See http://developer.android.com/reference/android/view/LayoutInflater.html

Method must create view that contains fragment's visual display.

```
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container, Bundle savedState)  {

  return inflater.inflate(R.layout.myFragment, container, false) ;

}
```

Assume fragment spec in file res/layout/myFragment.xml.

Do not attach to parent *ViewGroup*.

159

Fragment and activity communication

- No direct communication between fragments
  - Intended receiver (activity or fragment) may or may not exist
  - Containing activity mediates communication
  - Fragment gets containing activity by calling *getActivity()*, after fragment attached, or from *onAttach()*
  - Activity gets fragment reference with *fragment manager* and fragment's ID
- Examples
  - Button aButton = (Button) getActivity.findViewById(R.id.upButton) ;
  - MyFragment aFragment = getFragmentManager().findFragmentById(R.id…) ;

160

Fragment and activity communication (cont'd)

- Fragment may enforce communication with activity via a *Java interface*
  - Fragment defines nested interface with named method(s)
  - Activity containing fragment must *implement* interface
  - Fragment calls method(s) in activity to share information with activity

Fragment class definition.

```
public class MyFragment extends Fragment {
    …
    // Containing activity will implement this interface
    public interface OnSomeEventListener {
        public void onSomeEvent(…) ;
    }
    …
};
```

Nested interface declares method that containing activity must implement.

161

Fragment and activity communication (cont'd)

- Activity must implement interface nested in fragment for all this to work

Containing activity must implement Java interface *OnSomeEventListener.*

```
public class MyActivity extends FragmentActivity
                        implements OnSomeEventListener {
    …
    // Implement methods declared by OnSomeEventLlistener
    public void OnSomeEvent ( … ) {

    }
    …
};
```

Activity gives code for method declared by *OnSomeEventListener* in order to implement *OnSomeEventListener.*

162

Fragment and activity communication (cont'd)

- Fragment wants to make sure that activity has implemented the interface
  - But how? Fragment defines reference to activity **but uses interface type in declaration**
  - Declaration works only if activity implements interface

```
public class MyFragment extends Fragment {
    OnSomeEventListener mContainingActivity ;
    public void onAttach(Context activity) {
        try (
            mContainingActivity = (OnSomeEventListener) activity ;
        } catch (ClassCastException ex) {
            …
        } …
```

This activity must respond to fragment events, implement *OnSomeEventListener.*

This cast will succeed only if activity implements interface.

Catch block executed if cast fails.

163

Structure of fragment static layout app

- One activity, two fragments (titles and quotes)
- Titles fragment is subclass of *ListFragment* (implicitly contains a *ListView*)
- Three layout resources: One each for activity and quotes fragment, one for list item (just a simple *TextView,* no layout needed for *ListFragment*, similar to *ListActivity*)
- Activity layout statically defines two fragments
- Three Java source files, one each for activity and two fragments
- Communication between fragment and activity:
  1. Fragment declares Java interface that activity must implement, and
  2. Fragment sets its event listener to be that activity
- Source: Porter's Coursera MOOC, example app *FragmentsStaticLayout*

164

Static layout app: *QuoteViewerActivity*

- *QuoteViewerActivity* acts as conduit between between *TitlesFragment* and *QuotesFragment*
  - Declares static arrays with titles and quotes
  - Obtains reference to quotes fragment through *FragmentManager*
  - Implements Java interface *ListSelectionListener* by defining function *onListSelection()* that will be called by titles fragment whenever user selects an item in the list
    - Java interface defined in fragment class *TitlesFragment*
  - Calls *getShownIndex()* and *showQuoteAtIndex()* in details fragment to find out what quote currently displayed and to change quote

165

Static layout app: the titles fragment

- Subclass of *ListFragment*
- Declares interface *ListSelectionListener* implemented by quotes viewer activity (method *onListSelection()*)
- Declares listener field of type *ListSelectionListener*
- Defines method *onListItemClick()* (part of *ListFragment* API)
  - This method calls *onListSelection()* on listener (i.e., containing activity)
- *onAttach()* method sets listener to containing activity (passed as arg)
- *onActivityCreated()* sets array adapter, using resource file and quotes array

166

Static layout app: the quotes fragment

- Subclass of *Fragment*
- Method *onCreateView()* method inflates layout resource
- Method *onActivityCreated()* gets quote view defined in fragment's layout
  - Must wait for activity to be created in order to get view object
- Define methods *getShownIndex()* and *showViewAtIndex()*, called by quote viewer activity

167

Defining fragments programmatically: Summary

Use following steps in *onCreate()* of containing activity :

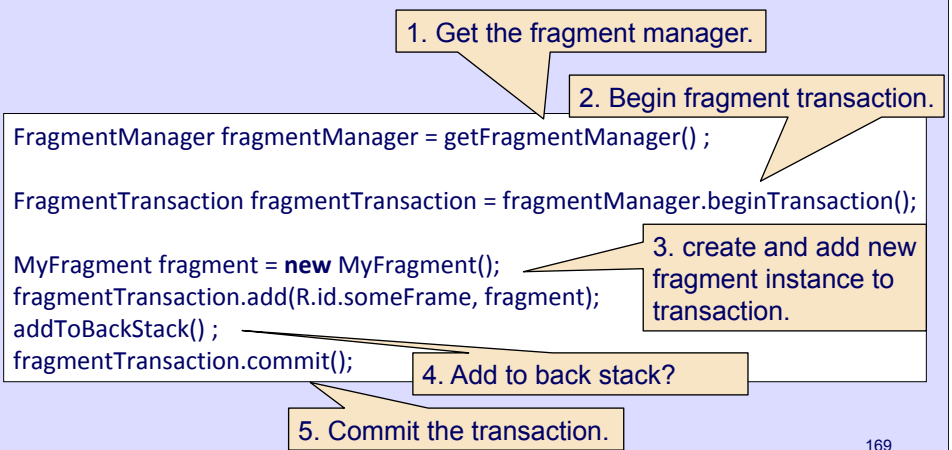
1. Retrieve the *fragment manager* – Call method *getFragmentManager()*
2. Begin a *fragment transaction – fragmentManager.beginTransaction()*
3. Add/Remove/Replace fragment wrt container view –
   transaction.add(container_id, fragment)
   transaction.replace(container_id, fragment)
   transaction.remove(container_id, fragment)
4. (optional) Add this transaction to the back stack –
   transaction.addToBackStack(null)
5. Commit the transaction – *transaction.commit()*
6. (optional) Force transaction to be performed now – Call
   executePendingTransactions()

168

Defining fragments programmatically

- Example code

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



169

Why this complexity?

- Two complications:
  1. Must call fragment manager to add and remove fragments
  2. Fragment operations must be part of a fragment transaction
- A plausible reason?
  - Different threads could execute fragment operations simultaneously
  - Coordination among threads needed to avoid conflicting changes
  - In general, only UI thread can modify interface
  - *commit()* executed by UI thread regardless of requesting thread
  - Fragment transactions enforce *atomicity*
- *FragmentTransaction* class to provide atomicity?
- Doc: https://developer.android.com/guide/components/fragments

170

Defining fragments programmatically

- Fragment manager can usefully "find" fragments in a display, e.g., if a fragment was defined statically in layout file

  getFragmentManager().findFragmentById(R.id.theFragment)
- Call *addToBackStack()* multiple times during same transaction to add different actions to the back stack

  See Porter's *FragmentsProgrammaticLayout* example

171

Fragments programmatic layout

- Dynamic fragment creation alone does not improve user experience
  - Reconfigure fragments programmatically to improve experience
- Activity's layout file defines 2 frame layouts instead of 2 fragments
- Activity's *onCreate()* method adds titles fragment and quotes fragment in single fragment manager transaction
- Nothing else changed; user experience is identical to static layout case

172

Adding fragments dynamically to an activity

- Suppose fragment *F* should be added and removed dynamically from an activity
- Activity can use these steps
  1. Activity creates fragment instance, does not add to display
  2. Activity uses method *isAdded()* to check status of fragment
  3. Activity uses fragment manager's transactions to add and remove fragment instance from display
  4. Activity can set up an *OnBackStackChangedListener* to reset layout programmatically when user navigates back from fragment action

173

OnBackStackChangedListener

- Static interface nested in class *FragmentManager*
- One abstract no-arg method, *onBackStackChanged() : void*
- Associate instance with fragment manager by sending message *addBackStackChangedListener()*

  May use anonymous class when instantiating interface, e.g.,

Send message adding listener to fragment manager.

```
// Add a OnBackStackChangedListener to reset the layout when the back stack changes
mFragmentManager.addOnBackStackChangedListener(
        new FragmentManager.OnBackStackChangedListener() {
                public void onBackStackChanged() {
                    // Blah! Blah! Blah!
                }
}) ;
```

Create anonymous class and its (only) instance.

Don't forget to define method *onBackStackChanged()*.

174

executePendingTransactions()

- Force the fragment manager to executed current transactions immediately
- Call after you commit a transaction
- Prevent fragment manager from "optimizing" screen updates

175

Fragments dynamic layout

- Activity's layout file defines frame layouts without weights as width will be determined dynamically
  - Width is `match_parent` for first frame layout and `0dp` for second frame layout
  - Titles fragment's *TextView* width is `match_parent`, quotes fragment's is `wrap_content`
- Activity's *onCreate()* method adds only titles fragment, sets up back stack changed listener to call *setLayout()*
- Method *setLayout()* checks whether quotes fragment added; sets layout configurations programmatically
- *onListSelection()* listener adds quotes fragment (if not added), adds transaction to back stack (triggering *setLayout()*)

176

Reusing fragments across activities

- Avoid recreating fragment when an activity is destroyed
- Potential savings:
  - Deleting and creating the fragment objects (e.g., less garbage)
  - Fragment's *onDestroy()* and *onCreate()* methods
  - Creating the fragment's view
- To keep fragment around, call *setRetainInstance(true)* in fragment's class *onCreate()* method
- Retrieve retained fragment through fragment manager (See next…)

177

Finding retained fragment

- https://developer.android.com/guide/topics/resources/runtime-changes.html#RetainingAnObject

```
private RetainedFragment dataFragment;

public void onCreate(Bundle savedInstanceState) {
    ....
    // find the retained fragment on activity restarts
    FragmentManager fm = getFragmentManager();
    dataFragment = (DataFragment) fm.findFragmentByTag("data");

    // create the fragment and data the first time
    if (dataFragment == null) {
       // add the fragment
       dataFragment = new DataFragment();
       fm.beginTransaction().add(dataFragment, "data").commit();
       // load the data from the web
       dataFragment.setData(loadMyData());
    }
```

Field holding fragment reference.

Does fragment manager have retained instance?

No. Then create new fragment instance and add to display.

178

Fragment static config layout

- New version with distinct layouts for phone/tablet
- Default layout is similar to programmatic layout (2 frame layouts)
  - Now used for portrait mode
  - Titles reduced to one line in portrait mode
  - Font size reduced to 24sp from 32sp
- Layout-land redefines layout of both fragments to use bigger 32sp font size
- Default layout uses ellipses for titles items
- Java code similar to previous static layout example
- Source: Porter's Coursera MOOC

179