Mobila applikationer och trådlösa nät, HI1033, HT2012

Today:
- User Interface basics
- View components
- Event driven applications and callbacks
- Menu and Context Menu
- ListView and Adapters
- Android Application life cycle,
- Activity life cycle

Expect this when developing software for limited devices

Hardware-imposed design considerations

- Limited memory capacity and processor speed
- Network: High latency, low speeds
- Small screens, of different sizes
- Different input devices (soft/hard keyboard, five way nav, ….)

Design with this in mind:
Be efficient and be responsive

Android SDK

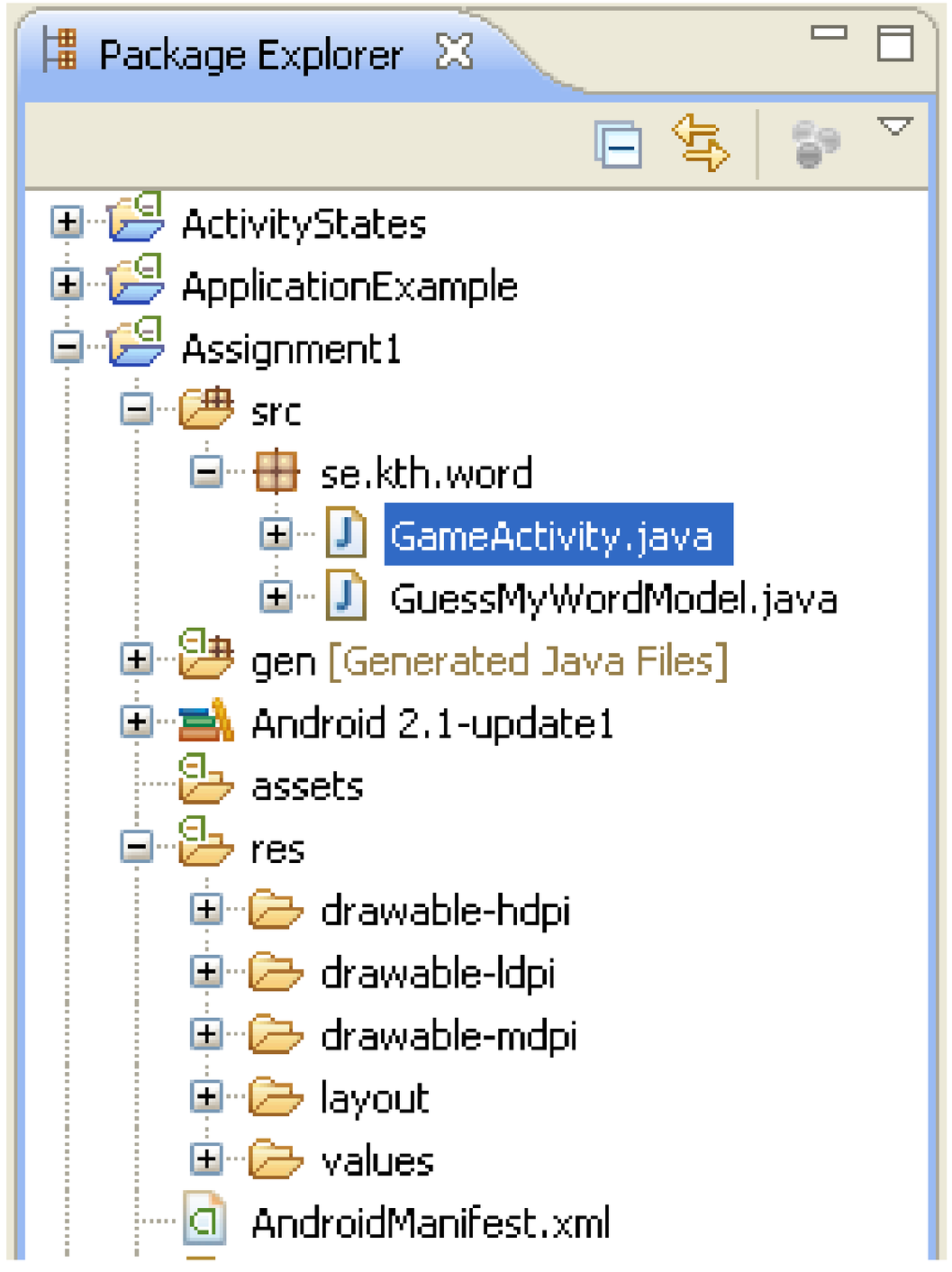
- API, Compiler, Debugger, …
- Android Virtual Device and Emulator
- Deployment tool

- Documentation, sample code, … at http://developer.android.com/

- The Eclipse IDE plugin adds:
  - Project management, project wizard
  - Editors for layouts and other resources
  - Automated building of projects
  - AVD manager
  - Debugger interface (Dalvik Debug Monitor Service, DDMS)

Anatomy of an Android Application

- Classes
- Resources, e.g. strings, layouts, data files, …
- Application Manifest, defining the entry point (e.g. an Activity), and other application settings
- On installation, packaged into an installation bundle, an Android Package (APK) file.
  - META-INF
  - res
  - AndroidManifest.xml
  - classes.dex



The Android Manifest file

- Contains information on application components, entry point, permissions, …

- ```
  <?xml version="1.0" encoding="utf-8"?>
  <manifest . . . >
      <application . . . >
          <activity
            android:name="com.example.project.GameActivity"
              android:icon="@drawable/small_pic.png"
              android:label="@string/myLabel"
              . . .  >
          </activity>
          . . .
      </application>
  </manifest>
  ```
  ```

The Android Manifest file

- Multiple application components

- ```
  <activity
      android:name="com.example.project.GameActivity”
      <intent-filter>
          <action
              android:name="android.intent.action.MAIN" />
          <category
              android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
  . . .  >
  </activity>

  <activity
      android:name="com.example.project.OtherActivity”
  . . .  >
  </activity>
  ```
  ```

The Android Manifest file

- By default, an application has not permission to perform operations considered potentially harmful to other applications, the operating system, or the user.

- Reading or writing another application's files, performing network access, keeping the device awake, etc requires permissions

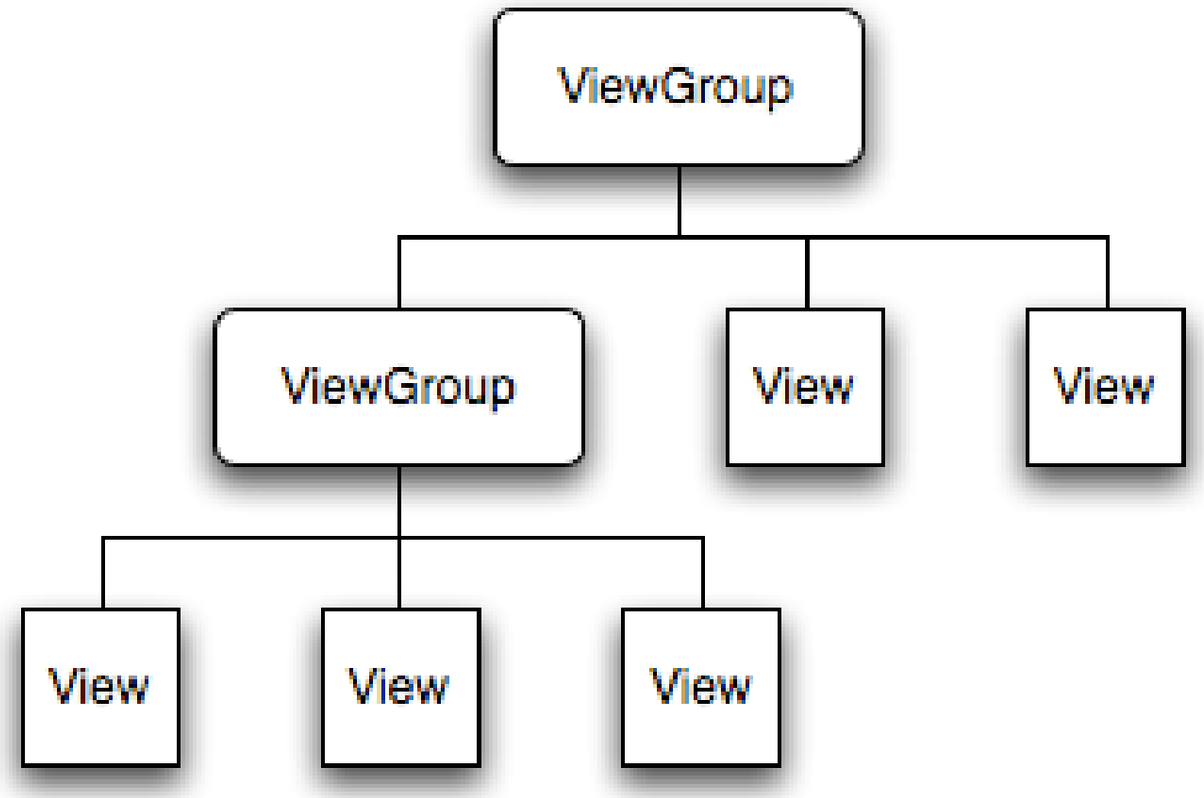
- <manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.android.app.myapp" >
      <uses-permission
          android:name="android.permission.RECEIVE_SMS”
          android:name="android.permission.INTERNET />
      ...
  </manifest>

User Interfaces, View components

- Super class View
  - ViewGroup extends View

- View sub classes (Widgets):
  TextView, EditText, Button,
  RadioButton, … , DatePicker,
  Clock, + custom widgets

- ViewGroup sub classes:
  LinearLayout, GridView, ListView,
  RadioGroup, …



User Interface, View Components

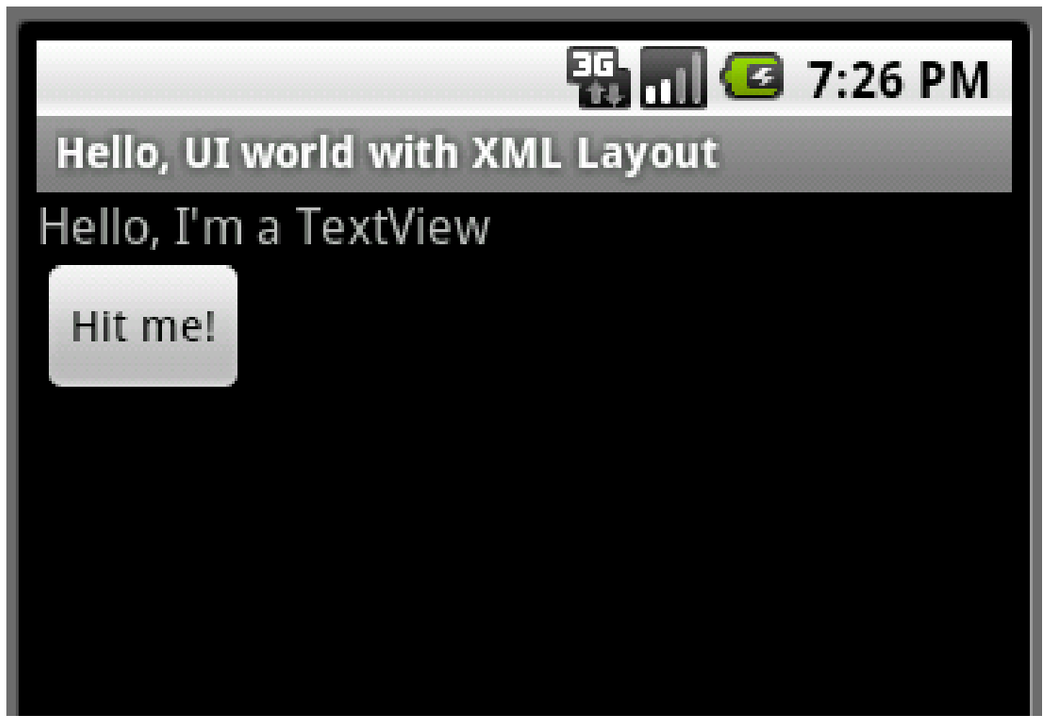
- Occupies a rectangular area of the screen

- Each view component has a set of properties representing layout parameters, text configuration, preferred size, …

- *As an object in the user interface, a View is also a point of interaction for the user and the receiver of the interaction events*

- The Activity must call **`setContentView()`** to attach the root node to the screen

UI example



- A LinearLayout containing a TextView and a Button
- It’s possible define layout and widgets and their properites in the source code, but…
- Instead – use Layout Files

Layout files

- Separate presentation and actions!

- Define layout and widgets andtogether with their properties in an XML file, e.g. res/main.xml

- The file must contain exactly one root component, that may contain child components

- Multiple layout files allowed
  At least one layout file per Activity

Layout file, res/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <TextView android:id="@+id/text"
       android:layout_height="wrap_content"
       android:layout_width="fill_parent"
       android:text="Hello, I'm a TextView"
       android:textSize="16sp">
    </TextView>

    <Button android:id="@+id/button"
          . . . >
    </Button>

</LinearLayout>
```

Layout file

- The corresponding Activity

```
public class UIActivity2 extends Activity {
    private TextView textView;
    private Button button;

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

        setContentView(R.layout.main);

        textView = (TextView) findViewById(R.id.text);
        button = (Button) findViewById(R.id.button);
    }
}
```

- NB: setContentView should be called *before* findViewById

User Interaction

- Event-based application - the flow of the program is determined by events
- In this case, events are user actions; screen touched, key or navigation-key pressed,
- Callbacks: The (Android) system detects such events and calls the appropriate method defined in or registered on the View component
- The application programmer provides the code by defining such Event Handlers or Event Listeners, and their call back methods

Event Listeners interfaces and methods

- onClick() from View.OnClickListener
- onLongClick() from View.OnLongClickListener
- onFocusChange() from View.OnFocusChangeListener
- onKey() from View.OnKeyListener.
  Called when the view has focus and a key is pressed
- onTouch() From View.OnTouchListener – gestures
- Methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction

OnClickListener, inner class

```
• public class UIAvtivity3 extends Activity {
    . . .
    public void onCreate(Bundle savedInstanceState) {
       . . .
       textView = (TextView) findViewById(R.id.text);
       button = (Button) findViewById(R.id.button);
       OnClickListener listener = new OnButtonClickListener();
       button.setOnClickListener(listener);
}

private class OnButtonClickListener implements
  OnClickListener {
    public void onClick(View v) {
      textView.setText("Hit number " + ++clicks);
    }
}
```

Event Handlers

- View components also have their own callback methods for handling events, e.g.

- onKeyDown(int, KeyEvent) - Called when a new key event occurs.

- onKeyUp(int, KeyEvent) - Called when a key up event occurs.

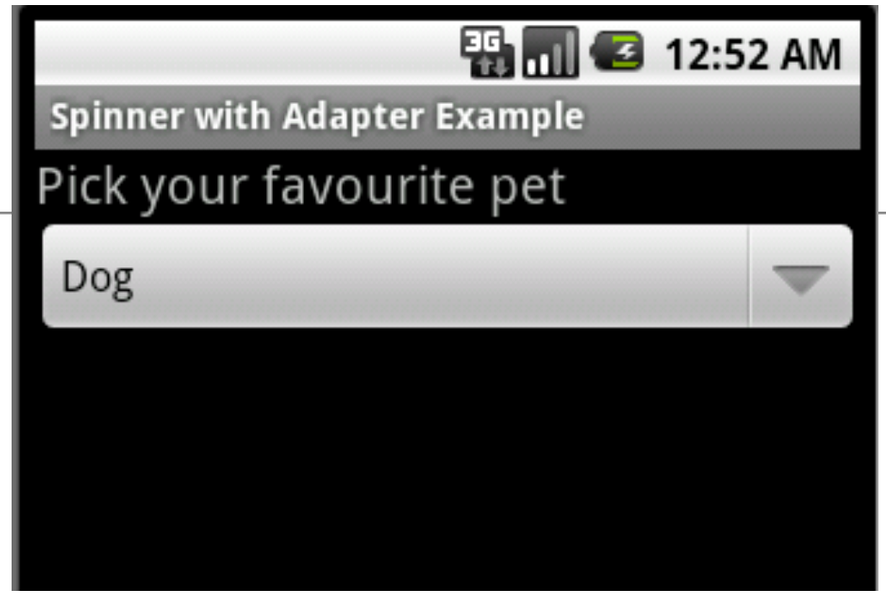
- onTouchEvent(MotionEvent) - Called when a touch screen motion event occurs.

- onFocusChanged(boolean, int, Rect)

- Extend the View class and override the appropriate method

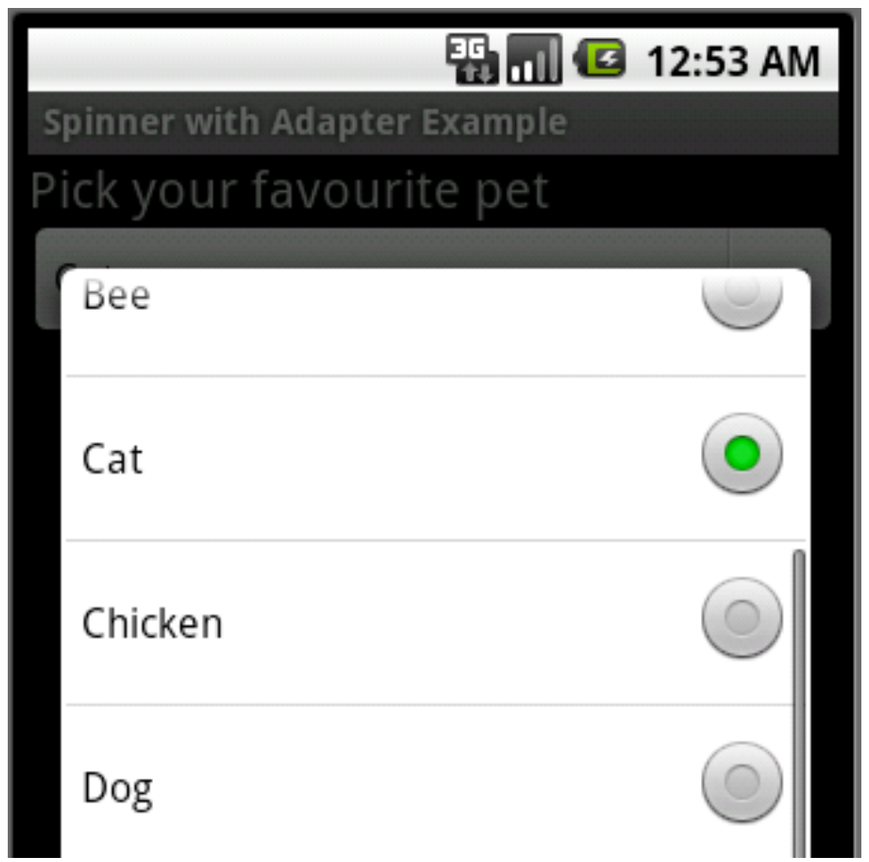
Adapters

- An Adapter represents a bridge between data, such as an array or list, and a View, such as a ListView or a Spinner.



- The Adapter creates the child views representing individual data

- The adapter is automatically updates the view(s) if the underlying data is changed



Adapters

- The Spinner, defined in res/main.xml

```
<LinearLayout . . .
    >
       <TextView android:layout_height="wrap_content"
           . . .
       </TextView>

       <Spinner android:layout_height="wrap_content"
          android:id="@+id/PetSpinner"
          android:layout_width="fill_parent"
          android:drawSelectorOnTop="true">
       </Spinner>
</LinearLayout>
```

- The strings could be defined in res/values/strings.xml

Adapters

- Create a list/array:

```
private static final String[] PETS = { "Aardvark", "Ant", "Bee",… };
```

- In the Activity's onCreate(), bind the array PETS to the Spinner:

```
spinner = (Spinner) findViewById(R.id.PetSpinner);

ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    android.R.layout.simple_spinner_item, PETS);

adapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);

spinner.setAdapter(adapter);
spinner.setOnItemSelectedListener(new SpinnerListener());
```

Adapters

- The listener can access underlying data via the adapter

- ```
  private class SpinnerListener implements OnItemSelectedListener {

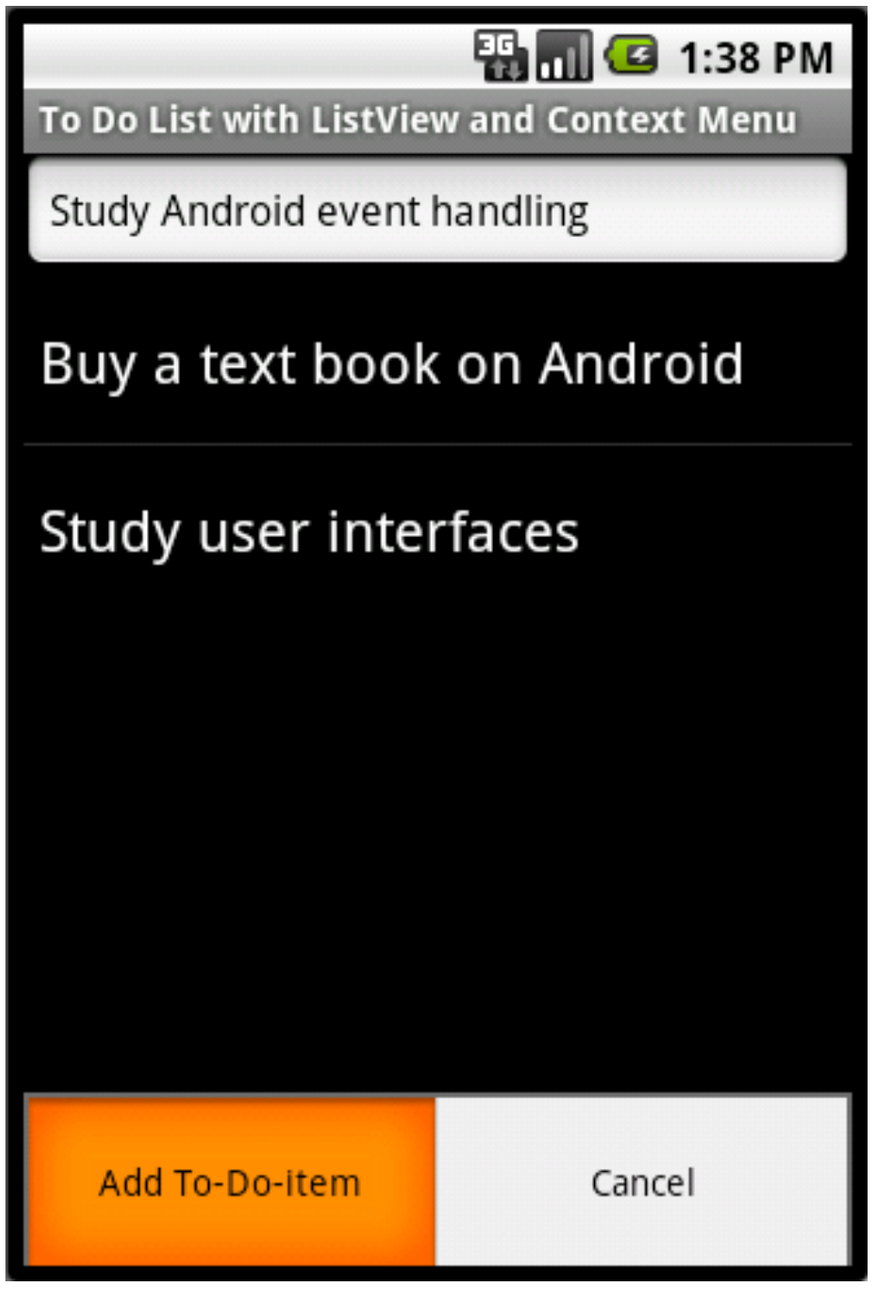
      public void onItemSelected(AdapterView<?> parent, View view, int
                                 pos, long id) {
          String pet = (String) spinner.getSelectedItem();
          showToast("Your favourite pet is a(n) " + pet);
      }

      public void onNothingSelected(AdapterView<?> parent) {}
  }
  ```
  ```

Options Menu

- Activated from the device's menu button

- 3 stages
  - Icon menu. Displayed at bottom of screen, max 6 items
  - Expanded menu. Scrollable. Displayed if "More" selected, can display full text, radio buttons, check boxes
  - Submenus. Displays as Expanded menu.



Options Menu

- Create a menu by overriding the Activity’s onCreateOptionMenu method, or
- Define the menus widgets in a layout file (XML)
- Event Handler/Listener
  - Override Activity’s onOptionsItemSelected, or
  - Create and add an OnMenuItemClickListener – inefficient, discouraged

Option Menu in Activity

```
public class ToDoActivity extends Activity {

  private EditText toDoInput;
  private ListView toDoListView;
  . . .
  private static final int ADD_ITEM = 0, REMOVE_ITEM = 1,
                           CANCEL_ITEM = 2;

  public boolean onCreateOptionsMenu(Menu menu) {
      . . .
  }

  public boolean onOptionsItemSelected(MenuItem item) {
     super.onOptionsItemSelected(item);
     . . .
  }
```

Option Menu, onCreateOptionsMenu

```
public boolean onCreateOptionsMenu(Menu menu) {

    menu.add(0, ADD_ITEM, Menu.NONE, "Add To-Do-item");
    menu.add(0, CANCEL_ITEM, Menu.NONE, "Cancel");

    return true;
    }
```

- menu.add(. . .) arguments:
  - Menu group (for example for radio buttons),
  - unique id for this menu item (an integer),
  - order,
  - title, a text

Options Menu, the Event Handler

```
public boolean onOptionsItemSelected(MenuItem item) {
   super.onOptionsItemSelected(item);

   switch(item.getItemId()) {
     case ADD_ITEM:
       String text = toDoInput.getText().toString();
       toDoItems.add(text);
       adapter.notifyDataSetChanged(); // Update view via adapter
       toDoInput.setText("");
       return true;

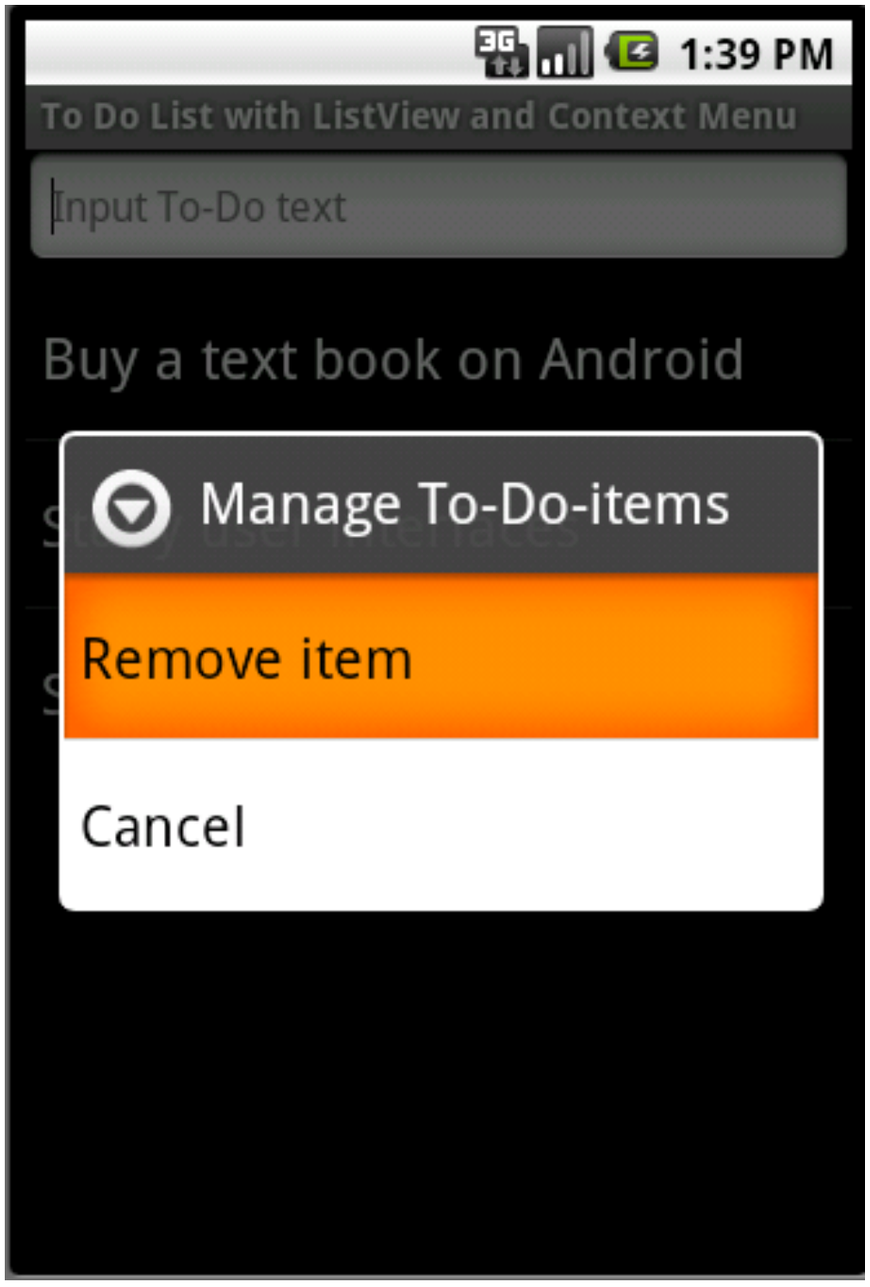
     case CANCEL_ITEM:
       . . .
   }

   return false;
}
```

Context Menu

- Associated with a specific widget, e.g. an item in a list view.

- Triggered on user event
  - long-click on widget
  - pressing the middle D-pad button or trackball (widget must be focused), or



Context menu

- Create a context menu by overriding the *Activity*'s onCreateContextMenu method, or

- Define the menus widgets in a layout file (XML)

- Event Handler/Listener
  - Override Activity's onContextItemSelected, or
  - Create and add an OnMenuItemClickListener – inefficient, discouraged

Context Menu, onCreateContextMenu

- In the Activity, override the onCreateContextMenu method

```
public void onCreateContextMenu(ContextMenu menu, View view,
        ContextMenu.ContextMenuInfo menuInfo) {

    super.onCreateContextMenu(menu, view, menuInfo);
    menu.setHeaderTitle("Manage To-Do-items");
    menu.add(0, REMOVE_ITEM, ContextMenu.NONE, "Remove");
    menu.add(0, CANCEL_ITEM, ContextMenu.NONE, "Cancel");
}
```

Context Menu

- In the Activity's *onCreate* method, register the menu to the view

```
public void onCreate(Bundle savedInstanceState) {
   . . .
   this.registerForContextMenu(toDoListView);
}
```

Context Menu, the Event Handler

- In the Activity, override the onContextItemSelected method

```
public boolean onContextItemSelected(MenuItem item) {
   super.onContextItemSelected(item);

   switch(item.getItemId()) {
   case REMOVE_ITEM:
      . . .
      toDoItems.remove(info);
      adapter.notifyDataSetChanged(); // Update view
      return true;
   case CANCEL_ITEM:
      . . .
   }
   return false;
}
```

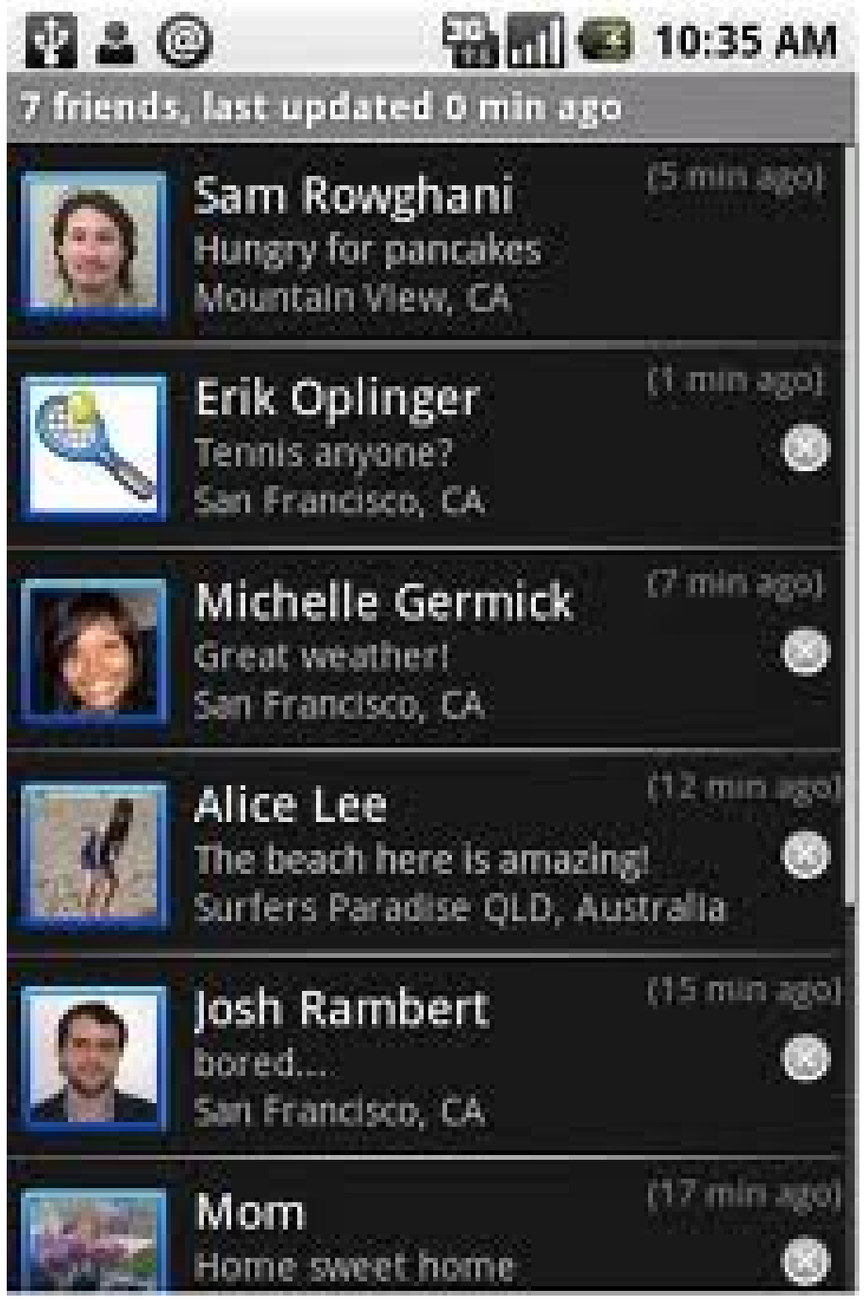
ListView

- Shows items in a vertically scrolling list

- Items come from a ListAdapter (or ArrayAdapter) associated with the list view.

- Simple list items: the data objects toString() method is called, or

- Customized items



ListView, the To-Do list example

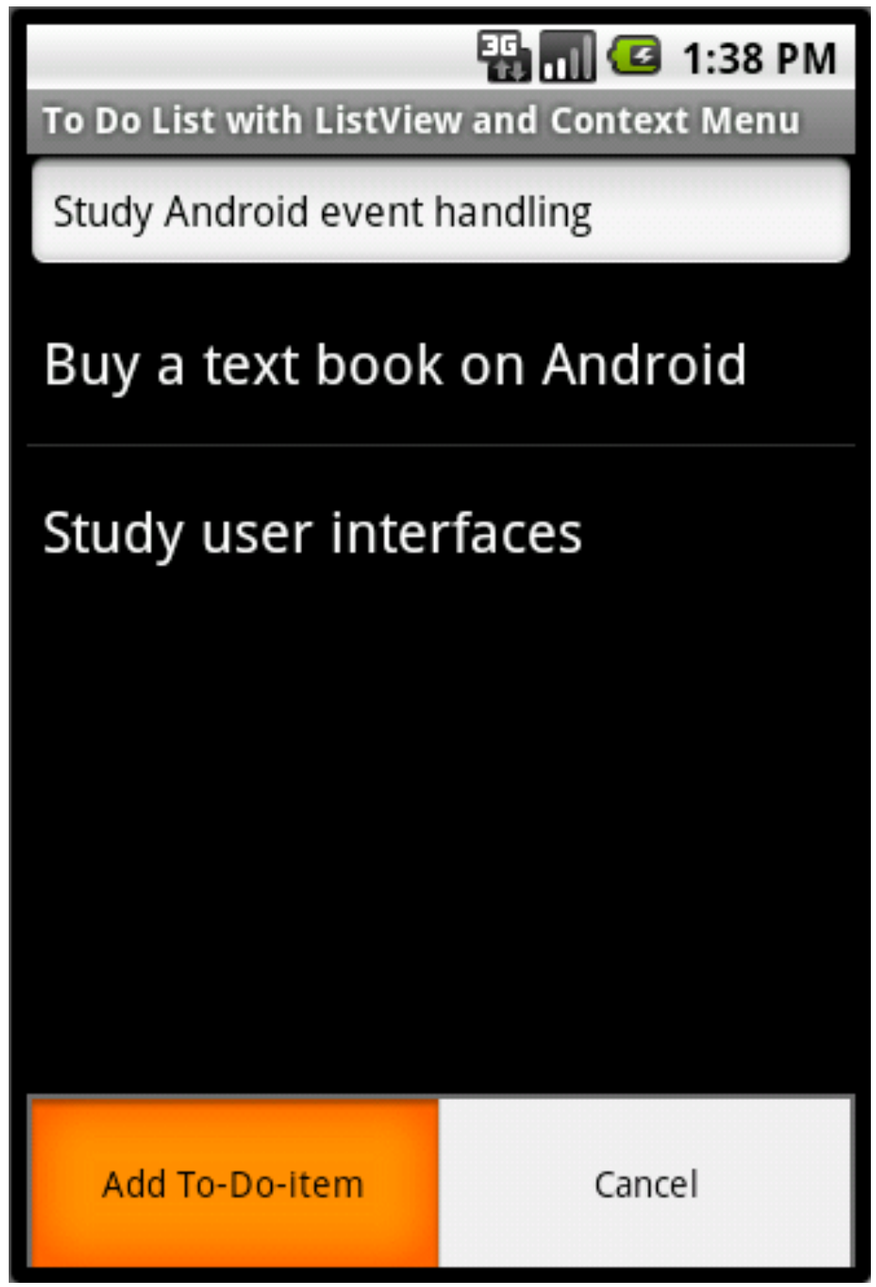
The layout file

```
<LinearLayout . . .
    >

<EditText
        android:id="@+id/InputText"
        . . .
</EditText>

<ListView
        android:id="@+id/ToDoListView"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
</ListView>

</LinearLayout>
```



ListView + Adapter

- Create an ArrayAdapter to bind the ListView to the data

```
public class ToDoActivity extends Activity {

   private ListView toDoListView;

   private ArrayList<String> toDoItems;
   private ArrayAdapter<String> adapter;

   public void onCreate(Bundle savedInstanceState) {
         . . .

         toDoItems = new ArrayList<String>();

         adapter = new ArrayAdapter<String>(
               this, android.R.layout.simple_list_item_1,
               toDoItems);
         toDoListView.setAdapter(adapter);
   }
```

ListView + Adapter

- Changes in the underlying data will automatically be reflected in the corresponding list view, e.g.:

```
public boolean onContextItemSelected(MenuItem item) {
    super.onContextItemSelected(item);
    switch(item.getItemId()) {
    case REMOVE_ITEM:
        AdapterView.AdapterContextMenuInfo menuInfo;
        menuInfo = (AdapterView.AdapterContextMenuInfo)
              item.getMenuInfo();
        int info = menuInfo.position;
        toDoItems.remove(info);
        adapter.notifyDataSetChanged();
        return true;
    case CANCEL_ITEM:
    . . .
}
```

- Complete example: ToDoList.zip!

The Dalvik VM and applications

- Every Android application runs in its own process, with its own instance of the Dalvik virtual machine.
  Dalvik has been written so that a device can run multiple VMs efficiently.

- The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint

- JIT, Just-In-Time compilation enhance performance

- Android starts the process when any of the application's code needs to be executed.
  The process is shut down when it's no longer needed *and system resources are required by other applications(!)*

Android application components

- Android applications don't have a single entry point (no main method) Instead: Consists of essential *components* that the system can instantiate and run as needed

- ***Activities*** *presents a visual user interface (holding View components)*

- **Services** doesn't have a visual user interface, but rather runs in the background

- **Broadcast receivers** receive and react to broadcast announcements, e.g. battery is low

- **Content providers** makes a specific set of the application's data available to other applications

Android Application Life Cycle

- *The process remains running until it is no longer needed and the system needs to reclaim its memory*

- Priorities

  1. A **foreground process**, holding an Activity at the top of the screen
  2. A **visible process**, holding an Activity that is visible
  3. A **service process**, holding a Service that has been started
  4. A **background process**, holding an Activity that is not currently visible to the user
  5. An **empty process**, that doesn't hold any active application components

- http://developer.android.com/videos/index.html#v=fL6gSd4ugSI

Android *Activity* Life Cycle

- An activity has essentially 4 states

1. *Active* or *running* when in the foreground
2. *Paused* if visible but not in focus.
   A paused activity is alive, it maintains all state and member information and remains attached to the window manager
3. *Stopped* if completely obscured by another activity.
   It still retains all state and member information
4. The Activity might be dropped from memory.
   When it is displayed again to the user, it must be completely restarted and restored to its previous state.

- *If an activity is paused or stopped, it's "killable".*
  The system can drop the activity from memory by either asking it to finish, or simply killing its process

Activity Life Cycle Call Back

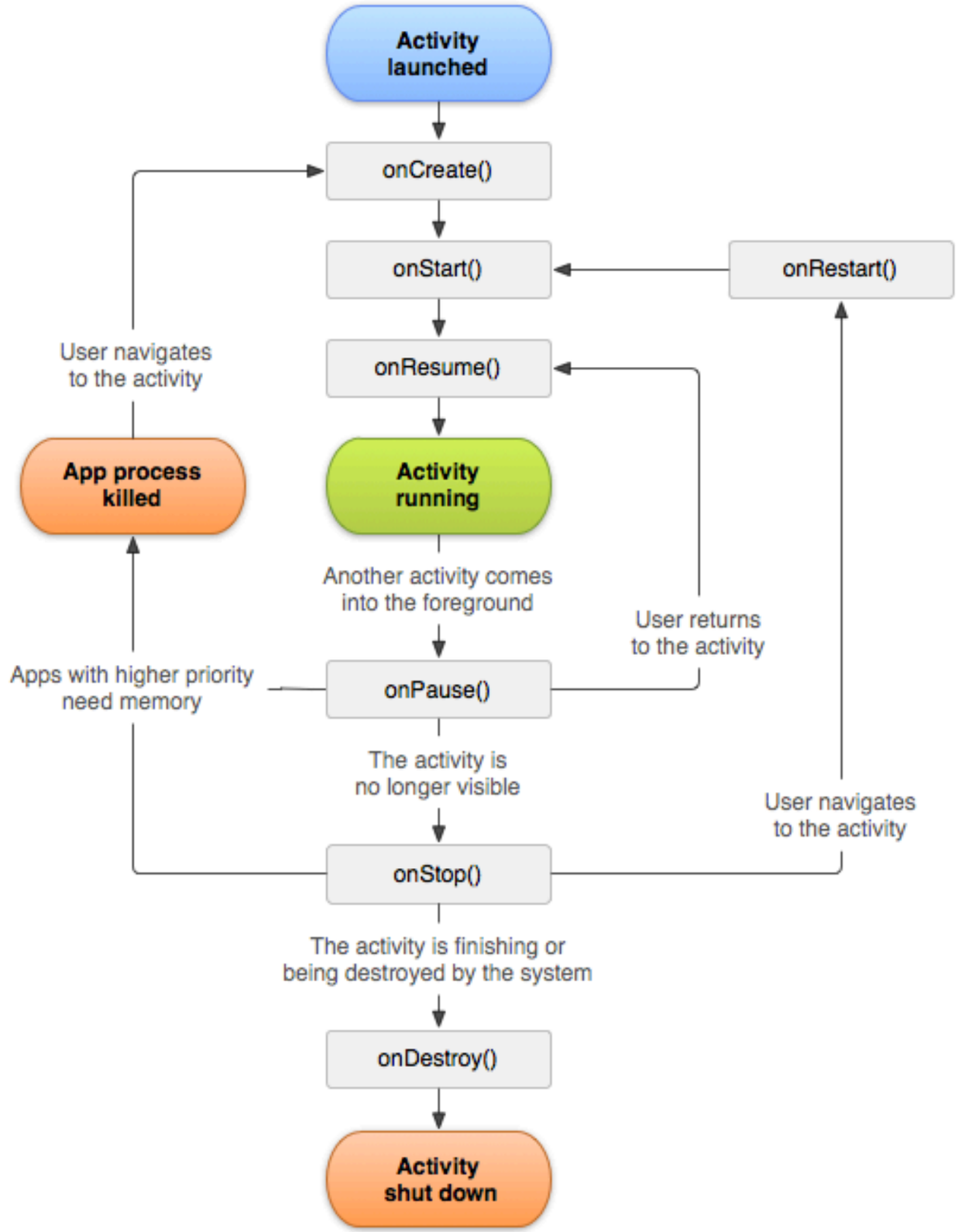
The activity is

- Visible between onStart and onStop

- In the foreground , and can receive user input events, between onPause and onResume

- Killable after onPause

Override the appropriate life cycle methods to handle state changes



Activity Life Cycle

- Start application
  - onCreate
  - onStart
  - onResume
- Other application started (e.g. a phone call)
  - onPause
  - onStop

  (then onRestart, onStart and onResume)
- Press home key
  - onPause
  - onStop
- Press back key (!)
  - onPause
  - onStop
  - onDestroy

  i.e. object deallocated

Activity Life Cycle Call back methods

Full lifetime

- onCreate – override to initialize your Activity, e.g. the user interface, create services and threads, connect to database,…
- onDestroy – override to free resources created in onCreate, close external connections, …

Visible lifetime

- onStart - override to resume/restart the above below
- onStop – override to pause animations, threads and services used to update the UI, sensor listeners, GPS lookups, ….

Activity Life Cycle Call back methods

Active lifetime (Activity in foreground)

- onPause - override to save UI state, e.g. uncommited user input in an EditText (Activity killable after this call)
- onResume - override to restore UI state (or better, do this in onCreate)

Example: SaveUIStateExample.zip

Saving UI state using Life Cycle methods

- SharedPreferences can store primitive data, and Strings, in key-value pairs

- Use SharedPreferences.Editor to store data

- UI state or other preferences:
  - save in onPause, after the Activity is “killable”

- - restore in onCreate(!)
  - avoid unnecessary tasks in onResume (Activity is in Activity Stack until destroyed)

Saving UI state using Life Cycle methods

- Storing state, in the Activity's onPause

```
protected void onPause() {
    super.onPause();

    String currentInput = textInput.getText().toString();

    // Save UI state; get a SharedPreferences editor
    SharedPreferences settings =
         getSharedPreferences(PREFS, MODE_PRIVATE);
    SharedPreferences.Editor editor = settings.edit();
    editor.putString("text_input", currentInput);
    // Commit the edits
    editor.commit();
}
```

Saving UI state using Life Cycle methods

- Restoring UI state

```
public class SaveActivity extends Activity {

    public static final String PREFS = "SaveActivityPrefs";
    private EditText textInput;

    public void onCreate(Bundle savedInstanceState) {
      ...
      textInput = (EditText) findViewById(R.id.InputText);

      // Restore UI state
      SharedPreferences settings =
        getSharedPreferences(PREFS, MODE_PRIVATE);
      String currentInput =
        settings.getString("text_input", "");
      textInput.setText(currentInput);
}
```

Saving UI state using Life Cycle methods

- Alternative to SharedPreferences:
  - onSaveInstanceState(Bundle), called before onPause
  - onCreate(Bundle)

- Data stored with onSaveInstanceState will only be held in memory until the application is closed

- Data stored using SharedPreferences is written to a database on the device and is available all the time

Where to go from here?

- Introduction to User Interfaces
  http://developer.android.com/guide/topics/ui/index.html

- Application resources and application anatomy
  http://developer.android.com/guide/topics/resources/index.html

- The API documentation
  http://developer.android.com/reference/packages.html

- The textbook(!)