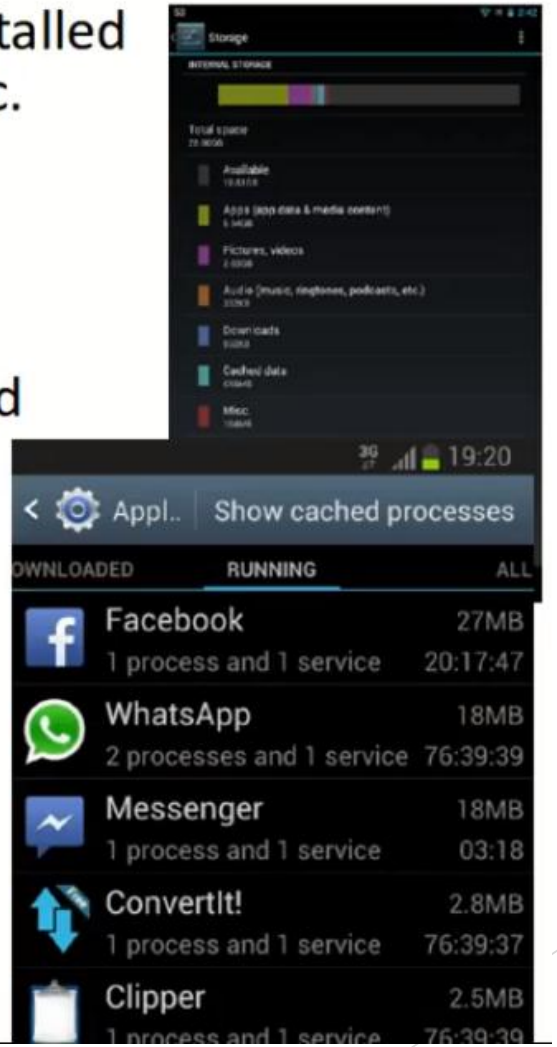
Android Week-5

Activity Life Cycle, State, Preferences
Adding External Library

The Activity Life-Cycle

- **storage**: Your device has apps and files installed and stored on its internal disk, SD card, etc.
  - **Settings → Storage**

- **memory**: Some subset of apps might be currently loaded into the device's RAM and are either running or ready to be run.
  - When the user loads an app, it is loaded from storage into memory.
  - When the user exits an app, it might be cleared from memory, or might remain in memory so you can go back to it later.
  - See which apps are in memory:
    - **Settings → Apps → Running**

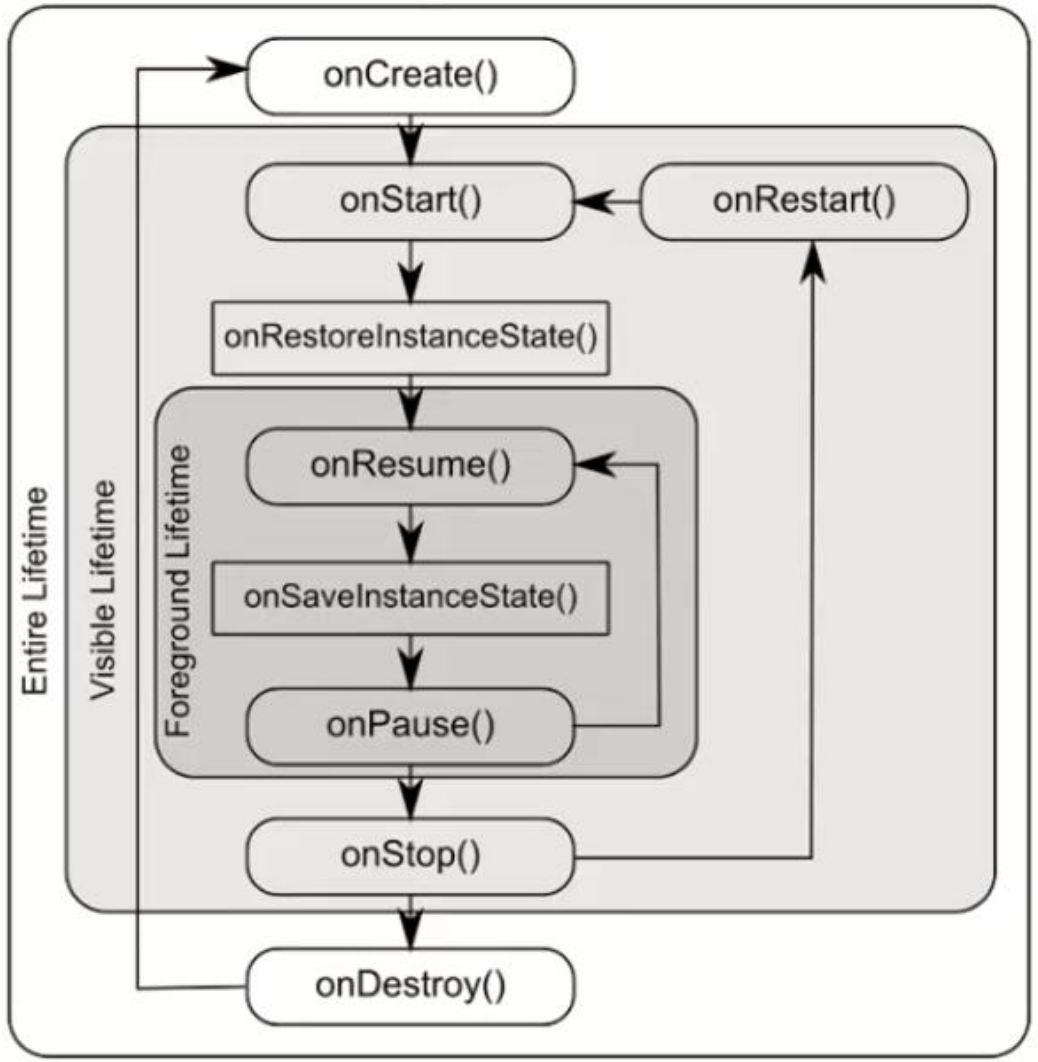


Activity State

- An activity can be thought of as being in one of several states:
  - **starting**: In process of loading up, but not fully loaded.
  - **running**: Done loading and now visible on the screen.
  - **paused**: Partially obscured or out of focus, but not shut down.
  - **stopped**: No longer active, but still in the device's active memory.
  - **destroyed**: Shut down and no longer currently loaded in memory.

- Transitions between these states are represented by **events** that you can listen to in your activity code.
  - onCreate, onPause, onResume, onStop, onDestroy, ...

Activity Lifecycle



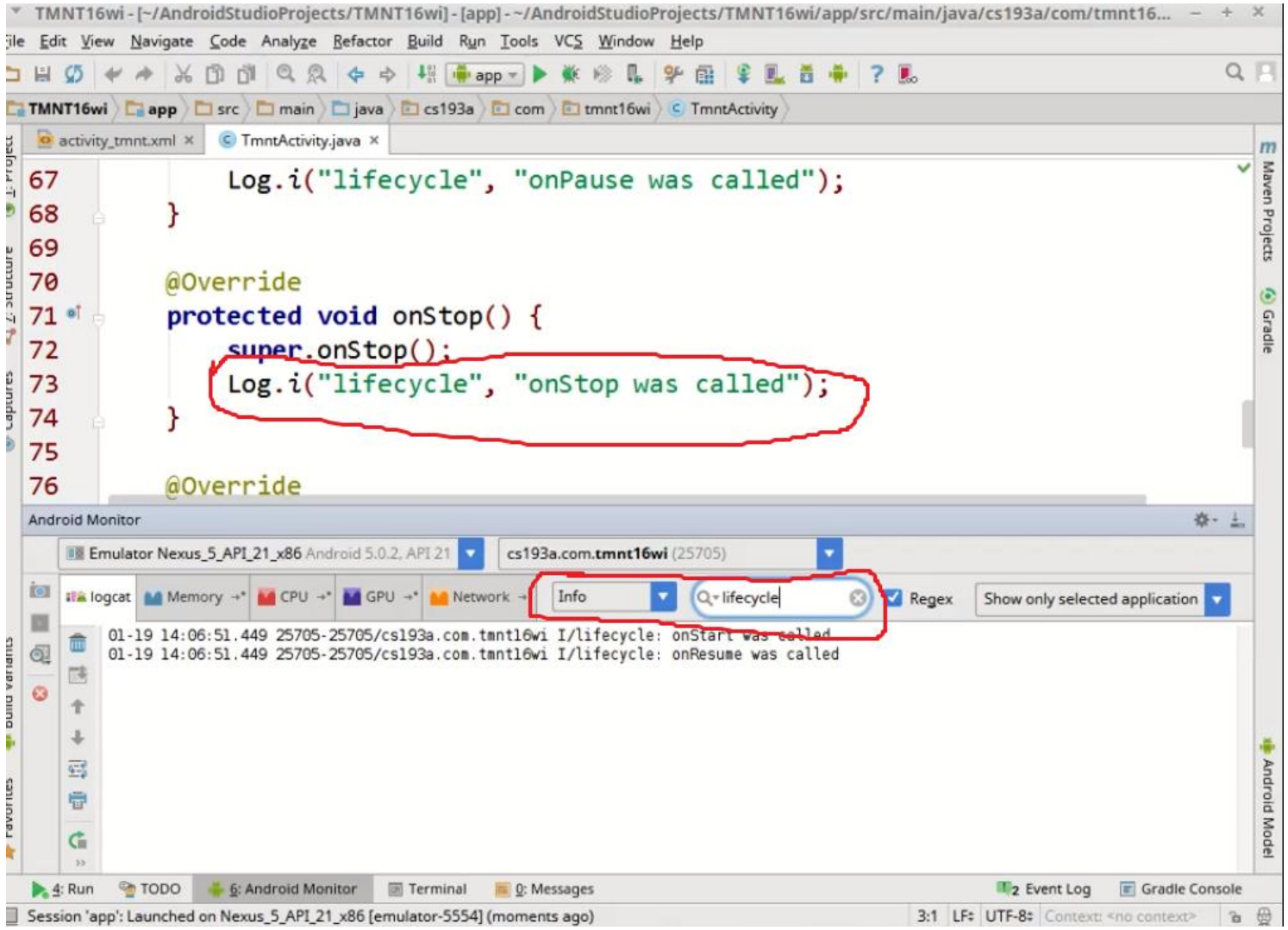
Log Methods

Method	Description
`Log.d("tag", "message");`	debug message (for debugging)
`Log.e("tag", "message");`	error message (fatal error)
`Log.i("tag", "message");`	info message (low-urgency FYI)
`Log.v("tag", "message");`	verbose message (rarely shown)
`Log.w("tag", "message");`	warning message (non-fatal error)
`Log.wtf("tag", exception);`	log stack trace of an exception

- Each method can also accept an optional exception argument:

```
try { someCode(); }
catch (Exception ex) {
    Log.e("error4", "something went wrong", ex);
}
```

Easy way to follow log message



onCreate Method

- In **onCreate**, you create and set up the activity object, load any static resources like images, layouts, set up menus etc.
  - after this, the Activity object exists
  - think of this as the "constructor" of the activity

```
public class FooActivity extends Activity {
    ...
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);    // always call super
        setContentView(R.layout.activity_foo); // set up layout
        any other initialization code;   // anything else you need
    }
}
```

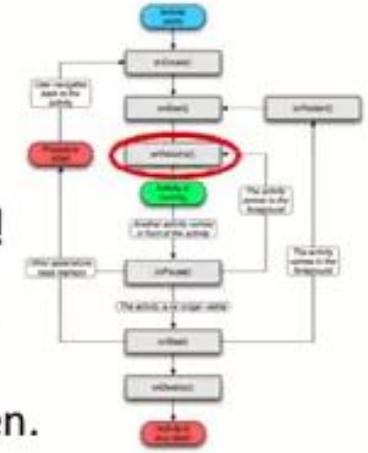
onPause

- When **onPause** is called, your activity is still partially visible.
- May be temporary, or on way to termination.
  - **Stop animations** or other actions that consume CPU.
  - **Commit unsaved changes** (e.g. draft email).
  - **Release system resources** that affect battery life.

```
public void onPause() {
    super.onPause();            // always call super
    if (myConnection != null) {
        myConnection.close();    // release resources
        myConnection = null;
    }
}
```

onResume

- When **onResume** is called, your activity is coming out of the Paused state and into the Running state again.
- Also called when activity is first created/loaded!
  - **Initialize resources** that you will release in onPause.
  - **Start/resume animations** or other ongoing actions that should only run when activity is visible on screen.



```
public void onResume() {
    super.onPause();            // always call super
    if (myConnection == null) {
        myConnection = new ExampleConnect();  // init.resources
        myConnection.connect();
    }
```

onStart and onRestart

- **onStart** is called every time the activity begins.
- **onRestart** is called when activity *was* stopped but is started again later (all but the first start).
  - Not as commonly used; favor onResume.
  - Re-open any resources that onStop closed.

```
public void onStart() {
    super.onStart();            // always call super
    ...
}
public void onRestart() {
    super.onRestart();          // always call super
    ...
}
```

onDestroy

- When **onDestroy** is called, your entire app is being shut down and unloaded from memory.
  - Unpredictable exactly when/if it will be called.
  - Can be called whenever the system wants to reclaim the memory used by your app.
  - Generally favor onPause or onStop because they are called in a predictable and timely manner.

```
public void onDestroy() {
    super.onDestroy();          // always call super
    ...
}
```

State and Preferences

Activity State Instance

- **instance state**: Current state of an activity.
  - Which boxes are checked
  - Any text typed into text boxes
  - Values of any private fields
  - ...

- Example: In the app at right, the instance state is that the Don checkbox is checked, and the Don image is showing.

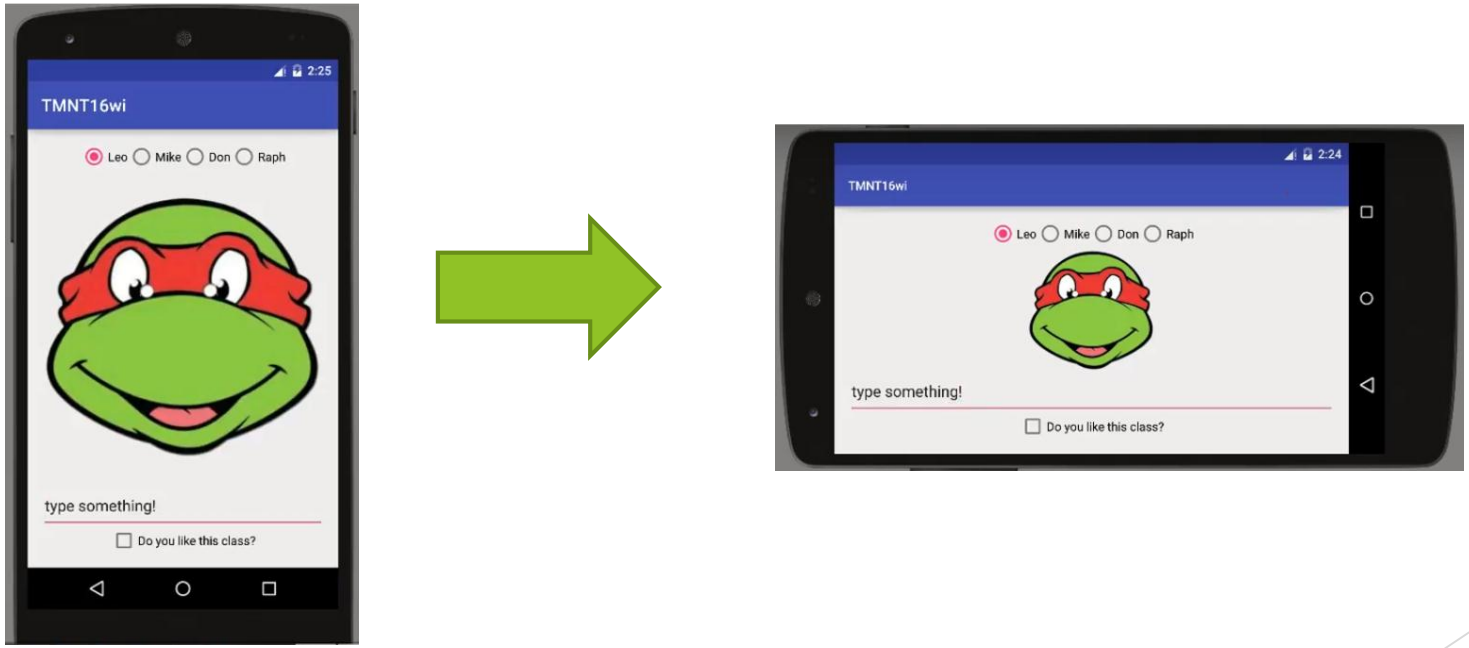
All of the private field variable is constituting the state of the activity

Lost Activity State

- Several actions can cause your activity state to be lost:
  - When you go from one **activity** to another and back, within same app
  - When you launch another **app** and then come back
  - When you rotate the device's **orientation** from portrait to landscape
  - ...

Hot key for Rotating

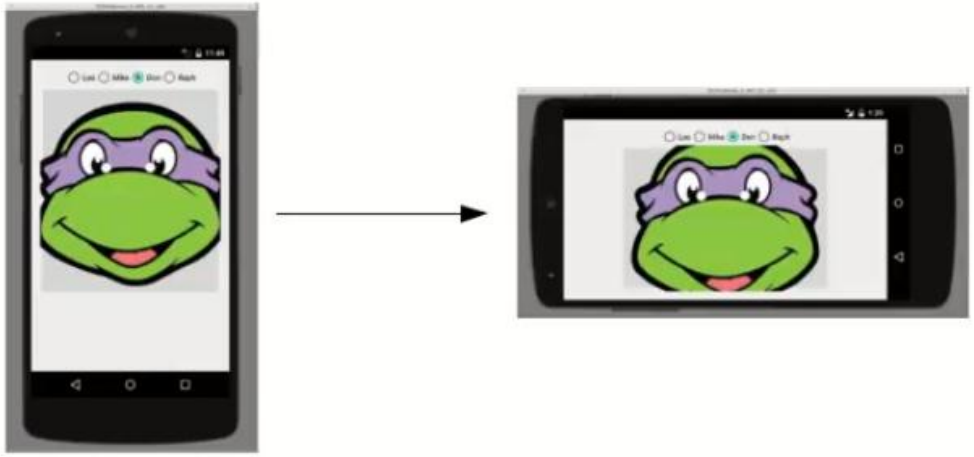
- Ctrl + F11
  - To change the orientation of the emulator



Handling Rotation

- A quick way to retain your activity's GUI state on rotation is to set the **configChanges** attribute of the activity in **AndroidManifest.xml**.
  - This doesn't solve the other cases like loading other apps/activities.

```
<activity android:name=".MainActivity"
    android:configChanges="orientation|screenSize"
    ...>
```



Another way to solve problem with JAVA code

- When an activity is being destroyed, the event method **onSaveInstanceState** is also called.
  - This method should save any "non-persistent" state of the app.
  - **non-persistent state**: Stays for now, but lost on shutdown/reboot.
- Accepts a **Bundle** parameter storing key/value pairs.
  - Bundle is passed back to activity if it is recreated later.
  - Superclass version saves state of any Views that have an ID.

```
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("name", value);
    outState.putString("name", value);
    ...
}
```

Code

```java
@Override
protected void onSaveInstanceState(Bundle bundle) {
    super.onSaveInstanceState(bundle);
    Log.i("lifecycle", "onSaveInstanceState was called");
    bundle.putInt("clicks", clicks);
}

@Override
protected void onRestoreInstanceState(Bundle bundle) {
    super.onRestoreInstanceState(bundle);
    Log.i("lifecycle", "onRestoreInstanceState was called");
    clicks = bundle.getInt("clicks");
}
```

What about if we close the app?

- What about if we close the application completely.
  - All data will be gone?
  - How to save it in the way of Android programming?
  - How to save application depended variables such as username and password, auto login flag, remember-user flag etc.
  - How to save user settings for your application?
    - Sound settings,
    - Screen settings,
    - Vibration is enabled?
    - Etc..
- The answer is: Use Preferences

Shared Preferences

- The shared preferences information is stored in an XML file on the device.
  - Typically in/data/data/<Your Application's package name>/shared_prefs
- SharedPreferences can be associated with the entire application, or to a specific activity.
- Use the getSharedPrefernces() method to get access to the preferences

Shared Preferences Example

- Usage:
  - SharedPrefernces prefs = this.getSharedPrefernces('myPrefs, MODE_PRIVATE');
- If the preferences XML file exist, it is opened, otherwise it is created.
- To Control **access permission to the file:**
  - MODE_PRIVATE:private only to the application
  - MODE_WORLD_READABLE:all application can read XML file
  - MODE_WORLD_WRITABLE:all application can write XML file

Shared Preferences

- To add Shared preferences, first an editor object is needed
  - Editor prefsEditor = prefs.edit();
- Then,use the put() method to add the key-value pairs
  - prefsEditor.putString(“username”,”D-Link”);
  - prefsEditor.putString(“password”,”vlsi#1@2”);
  - prefsEditor.putint(“times-login”,1);
  - prefsEditor.commit();

Shared Preferences

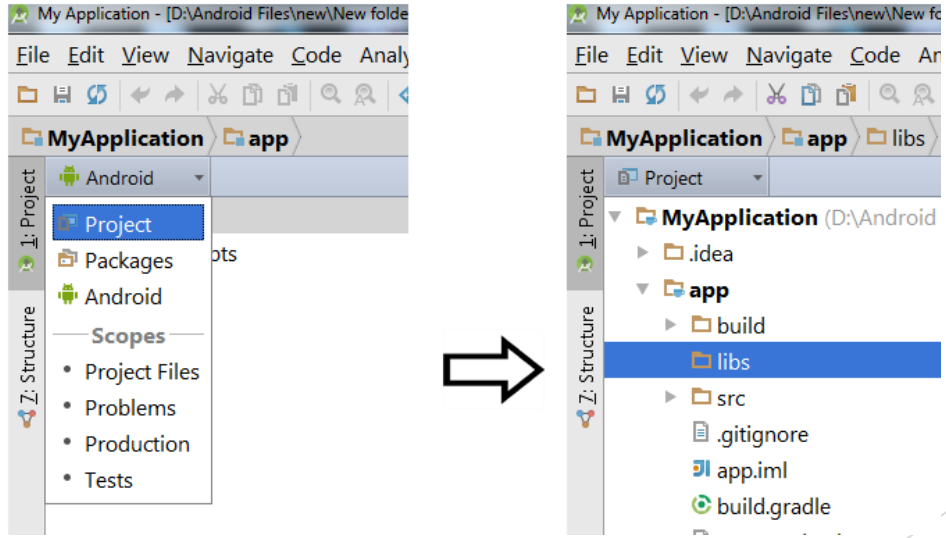
- To retrieve shared preferences data:
  - String username = prefsEditor.getString(“username”.” ”);
  - String password = prefsEditor.getString(“password”.” ”);
- If you are using SharedPrefernces for **specific activity**, then use **getPreferences()** method
  - No need to specify the name of the preferences file

Add External Lib.

External Libraries

How to add external libs.

- Create «libs» directory under your project if it does not exist.
- Add external library jar file under this directory.
- From your IDE, add this library to the build path.
- Use it in you Java code.



Adding Jsoup library

- Class Example with codes…
- Class Example with Gradle ….

Thanks for your attention…