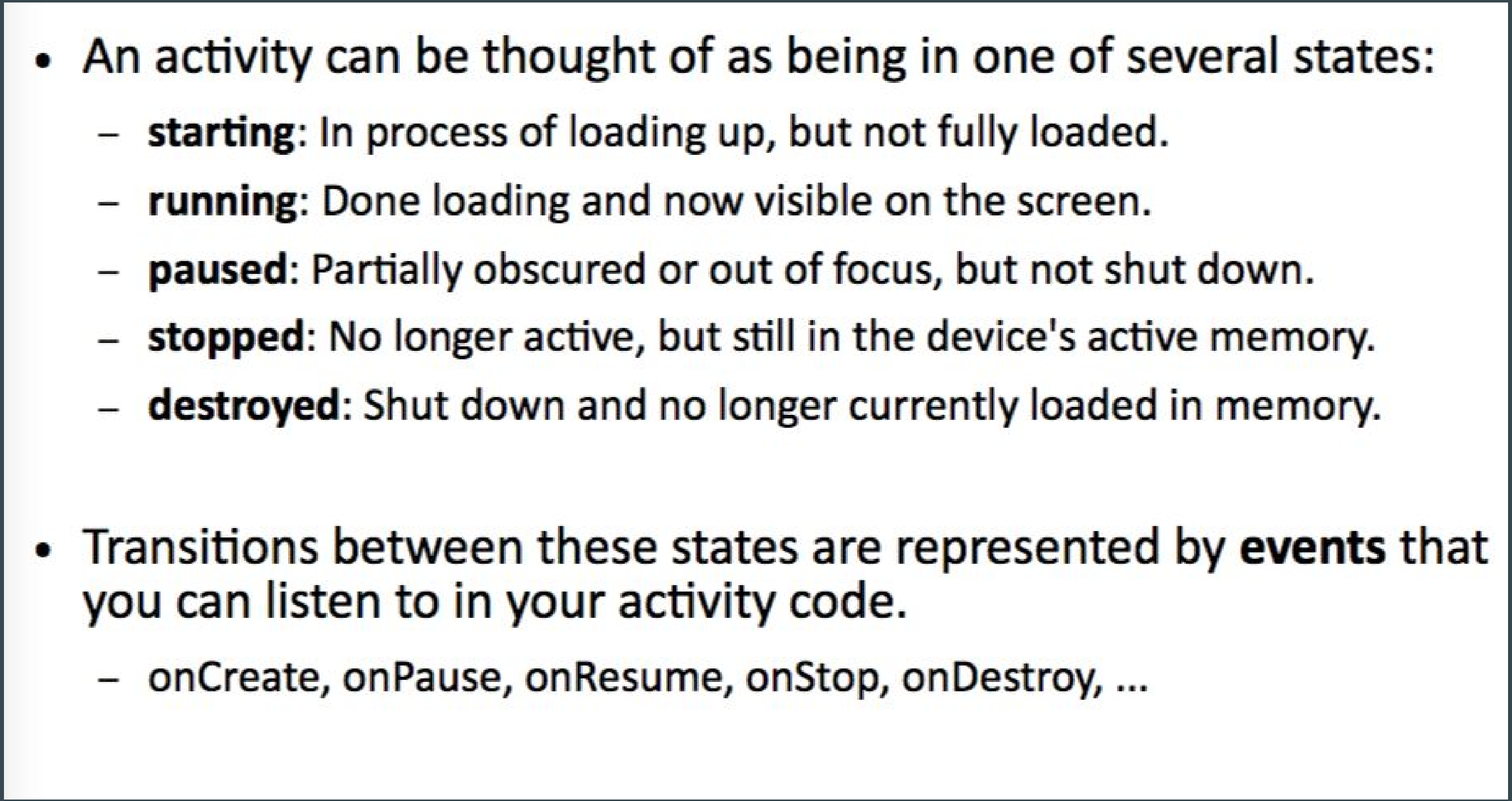
Activity Lifecycle

•••

Lecture 7

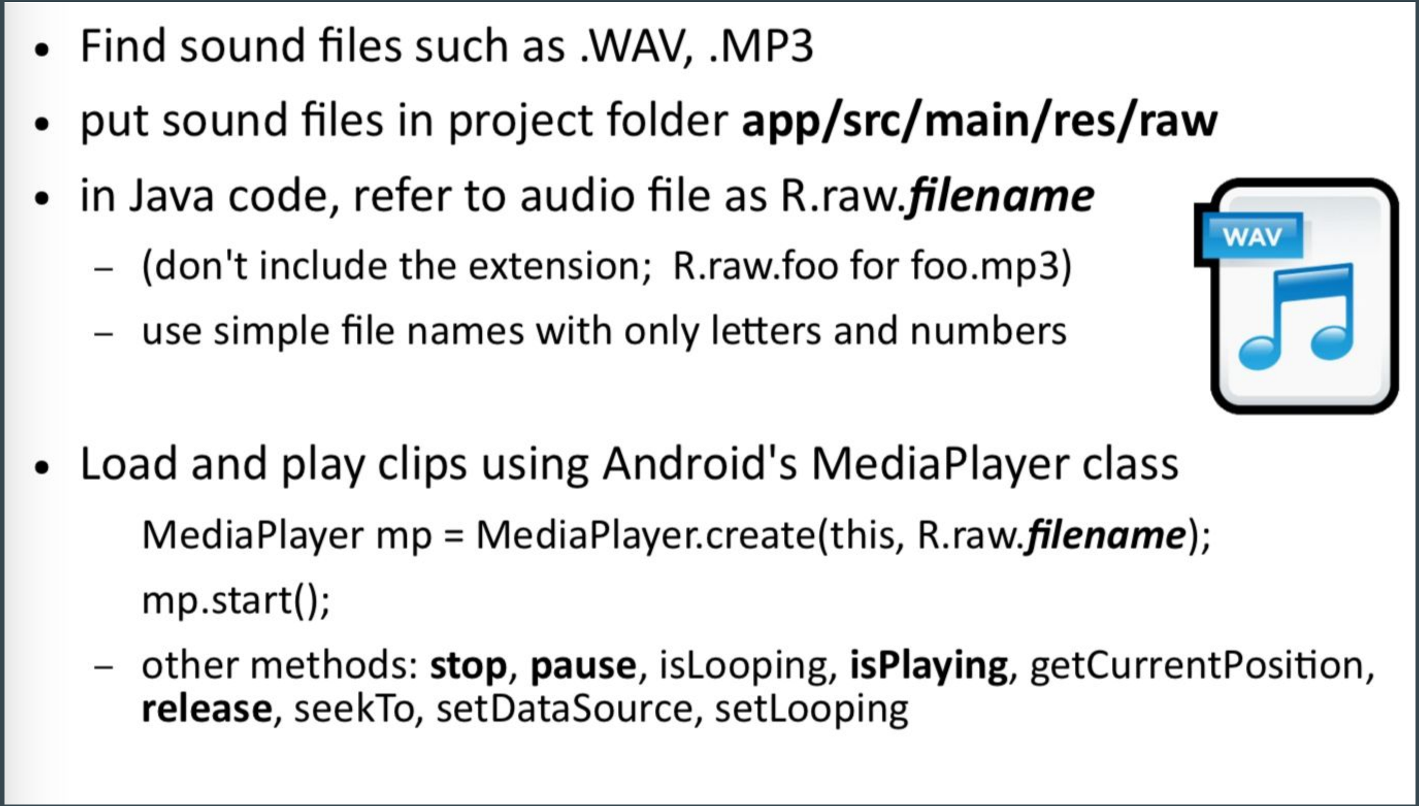
Activity state



- 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  - **stopped**: No longer active, but still in the device's active memory.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, ...

Playing sound effects



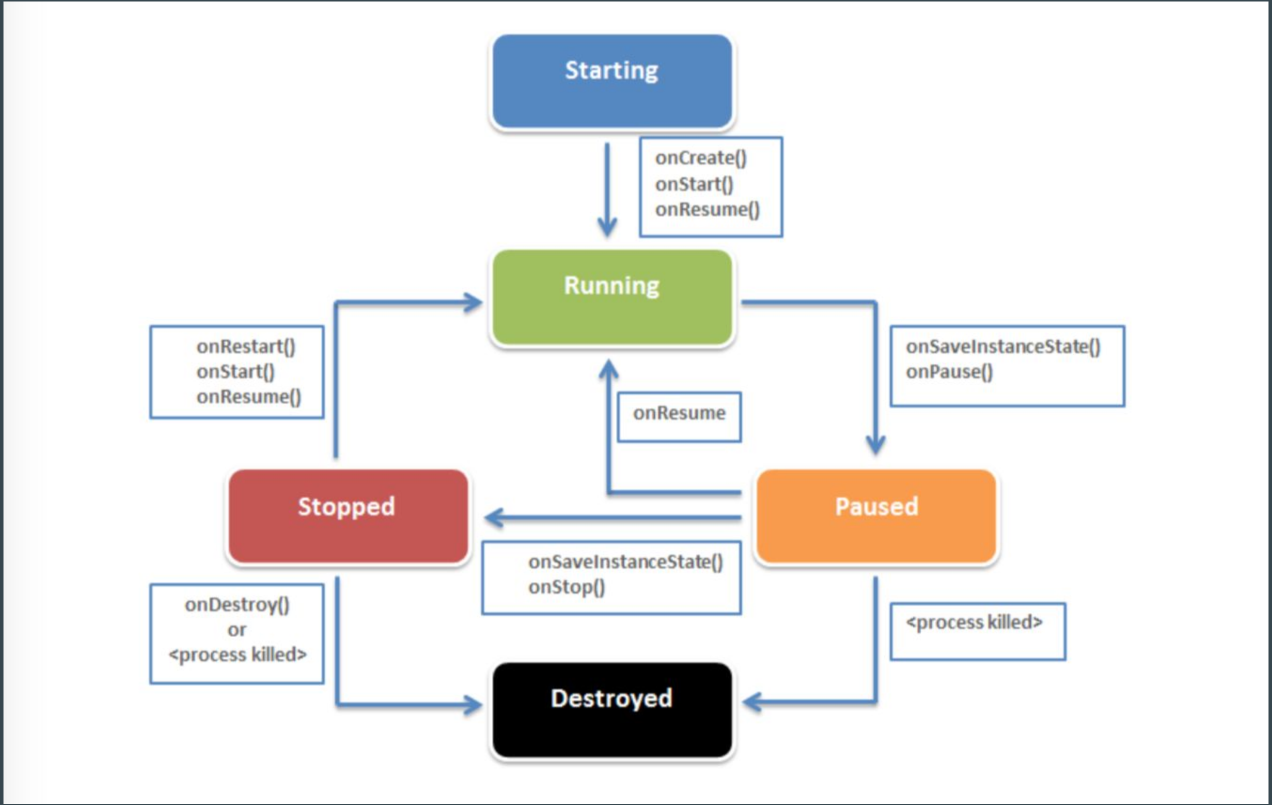
- Find sound files such as .WAV, .MP3
- put sound files in project folder **app/src/main/res/raw**
- in Java code, refer to audio file as R.raw.***filename***
  - (don't include the extension; R.raw.foo for foo.mp3)
  - use simple file names with only letters and numbers
- Load and play clips using Android's MediaPlayer class

  MediaPlayer mp = MediaPlayer.create(this, R.raw.***filename***);

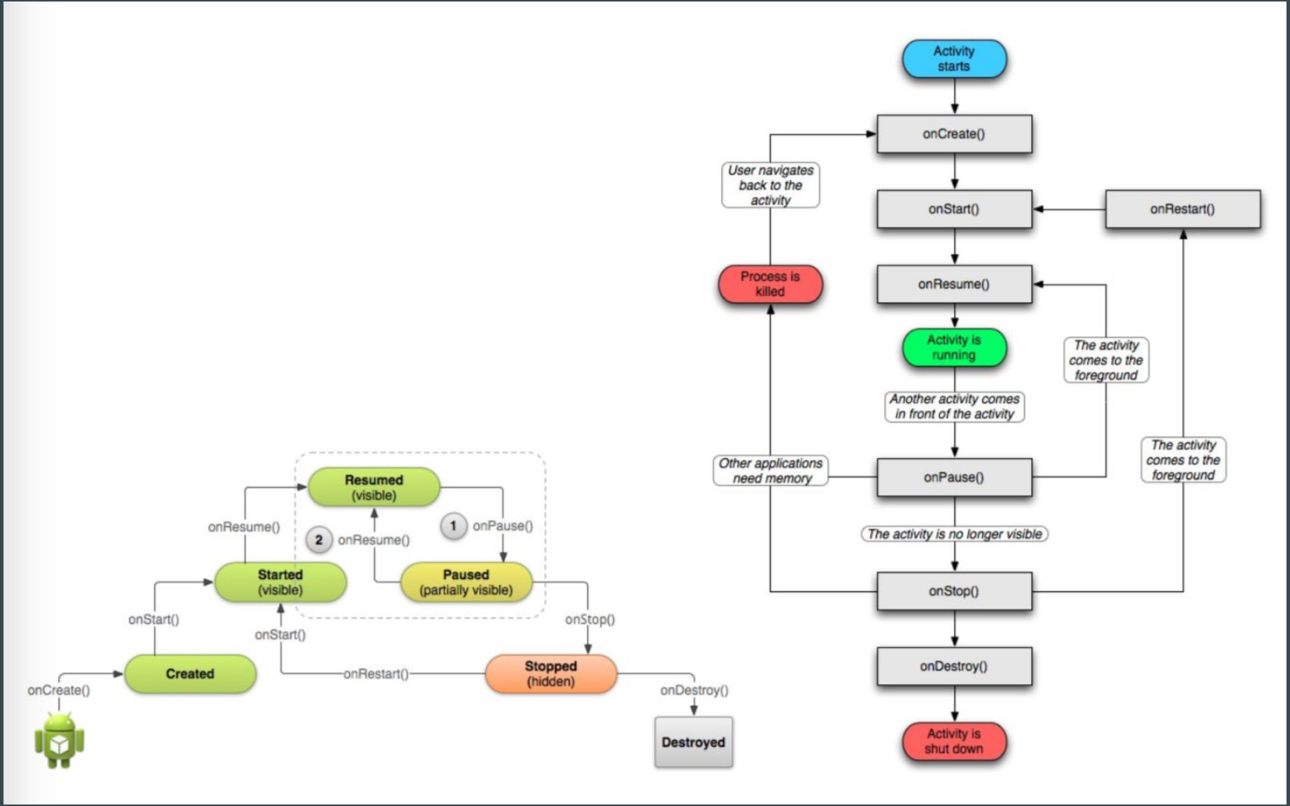
  mp.start();

  - other methods: **stop**, **pause**, isLooping, **isPlaying**, getCurrentPosition, **release**, seekTo, setDataSource, setLooping

Activity Lifecycle

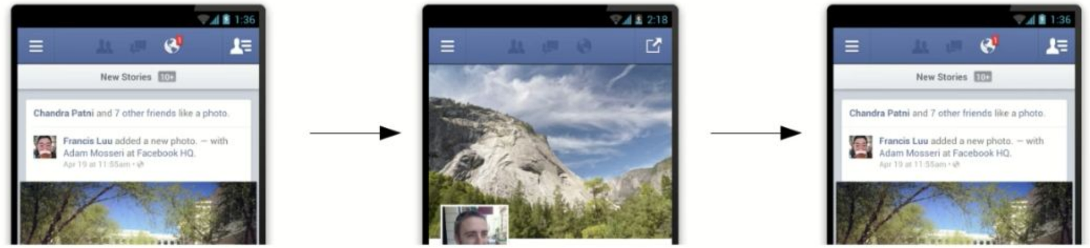


Other Diagrams

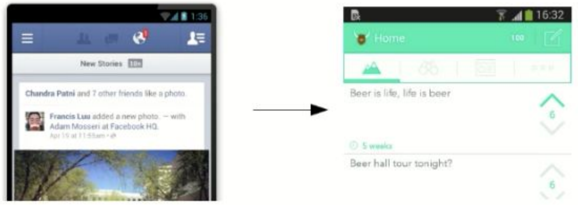


Activity State transitions

- jump between activities in the same app: onPause/onResume


New Stories
Chandra Patni and 7 other friends like a photo.
Francis Luu added a new photo. — with Adam Mosseri at Facebook HQ.
New Stories
Chandra Patni and 7 other friends like a photo.
Francis Luu added a new photo. — with Adam Mosseri at Facebook HQ.

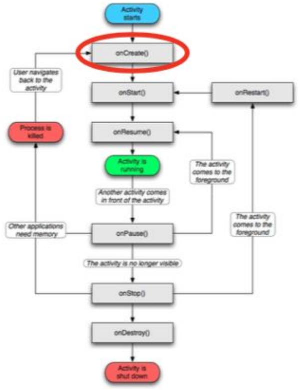
- jump between two apps that are in memory: onStop/onStart


New Stories
Chandra Patni and 7 other friends like a photo.
Francis Luu added a new photo. — with Adam Mosseri at Facebook HQ.
16:32
Home
Beer is life, life is beer
Beer hall tour tonight?

- app loaded/unloaded from memory: onDestroy/onCreate

The onCreate method

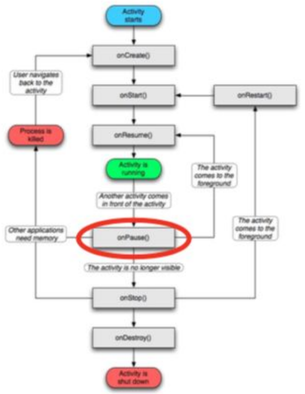
- In **onCreate**, you create and set up the activity object, load any static resources like images, layouts, set up menus etc.
  - after this, the Activity object exists
  - think of this as the "constructor" of the activity


Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

```
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
    }
}
```

The onPause method

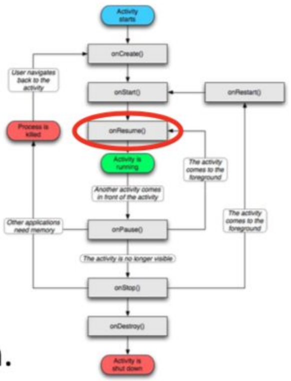
- When **onPause** is called, your activity is still partially visible.
- May be temporary, or on way to termination.
  - **Stop animations** or other actions that consume CPU.
  - **Commit unsaved changes** (e.g. draft email).
  - **Release system resources** that affect battery life.


Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

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

THe onResume method

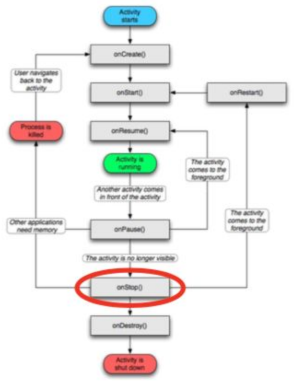
- When **onResume** is called, your activity is coming out of the Paused state and into the Running state again.
- Also called when activity is first created/loaded!
  - **Initialize resources** that you will release in onPause.
  - **Start/resume animations** or other ongoing actions that should only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


Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

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

The onStop method

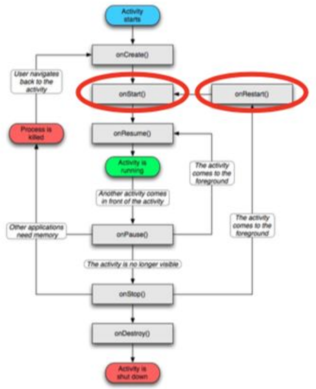
- When **onStop** is called, your activity is no longer visible on the screen:
  - User chose another app from **Recent Apps** window.
  - User starts a **different activity** in your app.
  - User receives a **phone call** while in your app.
- Your app might still be running, but that activity is not.
  - onPause is always called before onStop.
  - onStop performs heavy-duty shutdown tasks like writing to a database.


Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

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

onStart and onRestart

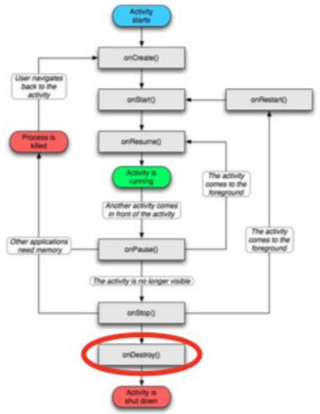
- **onStart** is called every time the activity begins.
- **onRestart** is called when activity *was* stopped but is started again later (all but the first start).
  - Not as commonly used; favor onResume.
  - Re-open any resources that onStop closed.


Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

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

The onDestroy method

- 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.

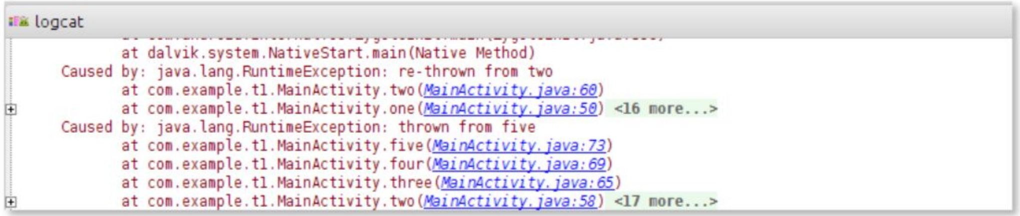

Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

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

Testing activity states

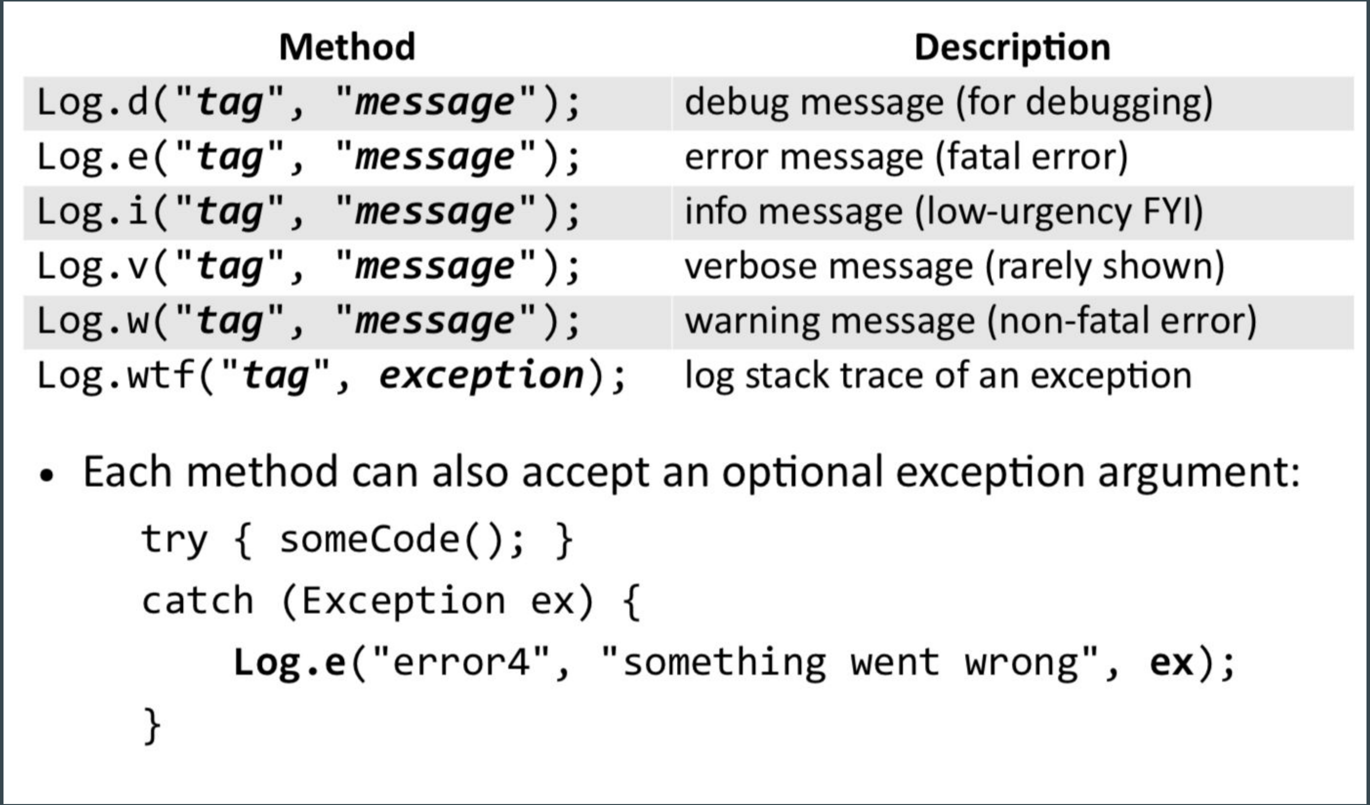
- Use the LogCat system for logging messages when your app changes states:
  - analogous to System.out.println debugging for Android apps
  - appears in the LogCat console in Android Studio

```
public void onStart() {
    super.onStart();
    Log.v("testing", "onStart was called!");
}
```



```
logcat
        at dalvik.system.NativeStart.main(Native Method)
    Caused by: java.lang.RuntimeException: re-thrown from two
        at com.example.t1.MainActivity.two(MainActivity.java:60)
        at com.example.t1.MainActivity.one(MainActivity.java:50) <16 more...>
    Caused by: java.lang.RuntimeException: thrown from five
        at com.example.t1.MainActivity.five(MainActivity.java:73)
        at com.example.t1.MainActivity.four(MainActivity.java:69)
        at com.example.t1.MainActivity.three(MainActivity.java:65)
        at com.example.t1.MainActivity.two(MainActivity.java:58) <17 more...>
```

Log methods



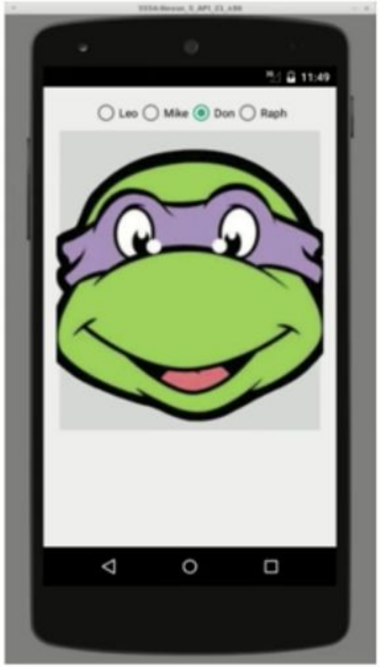
Method	Description
`Log.d("`***tag***`", "`***message***`");`	debug message (for debugging)
`Log.e("`***tag***`", "`***message***`");`	error message (fatal error)
`Log.i("`***tag***`", "`***message***`");`	info message (low-urgency FYI)
`Log.v("`***tag***`", "`***message***`");`	verbose message (rarely 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);
}
```

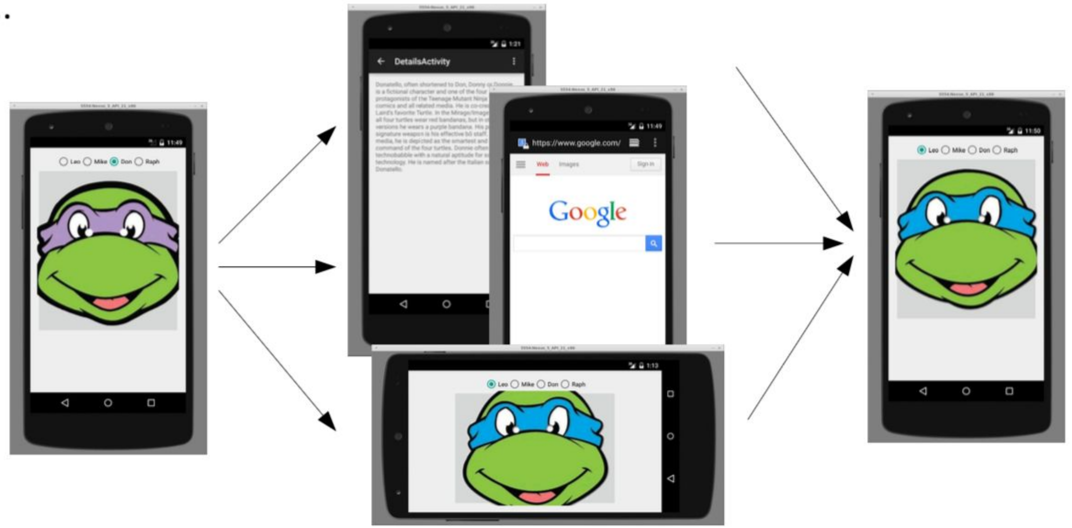
Activity instance state

- **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.



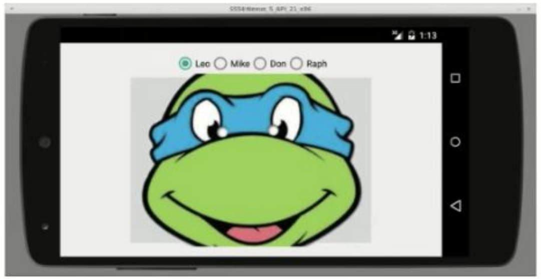
Lost Activity State

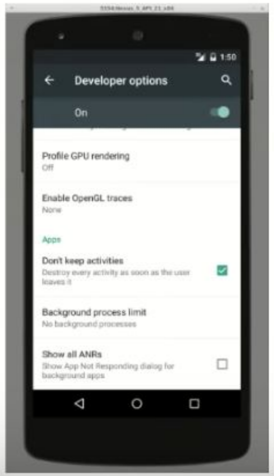
- Several actions can cause your activity state to be lost:
  - When you go from one **activity** to another and back, within same app
  - When you launch another **app** and then come back
  - When you rotate the device's **orientation** from portrait to landscape
  - ...



Simulate state change

- Testing orientation change: press **Ctrl-F11**
- Testing activity shutdown (`onDestroy`):
  - Settings → Developer options → Don't keep activities
  - Developer options → Background process limit → No bg processes





Profile GPU rendering
Off

Enable OpenGL traces
None

Apps

Don't keep activities [✓]
Destroy every activity as soon as the user leaves it

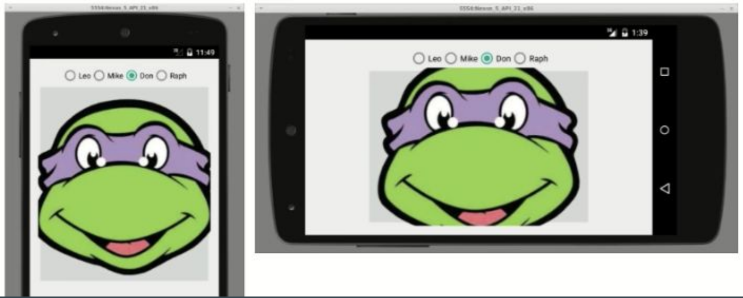
Background process limit
No background processes

Show all ANRs [ ]
Show App Not Responding dialog for background apps

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.

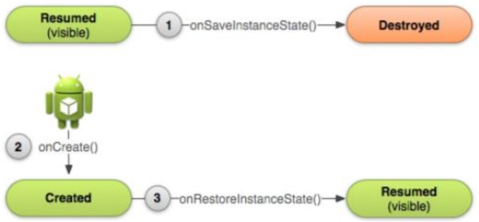
```xml
<!-- AndroidManifest.xml -->
<activity android:name=".MainActivity"
    android:configChanges="orientation|screenSize"
    ...>
```



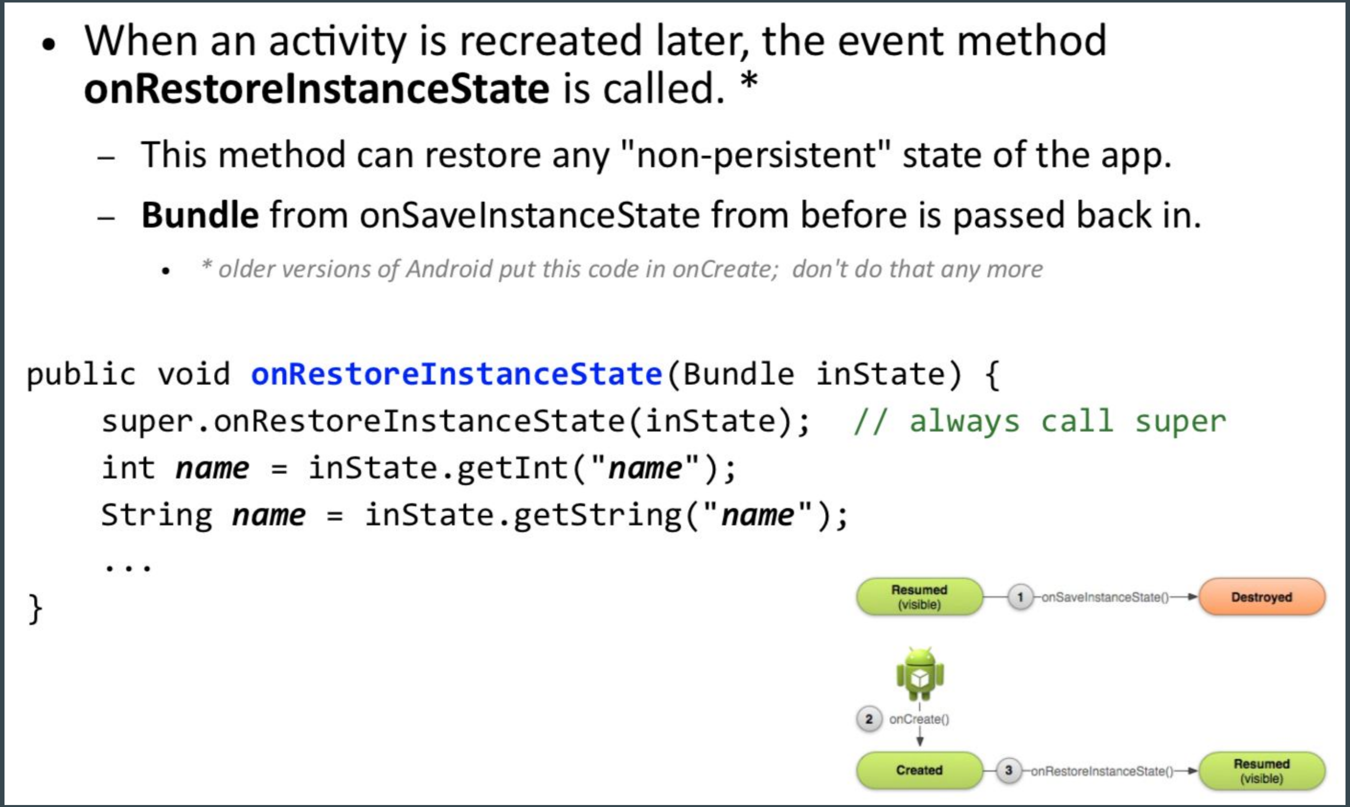
onSaveInstanceState method

- 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.

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

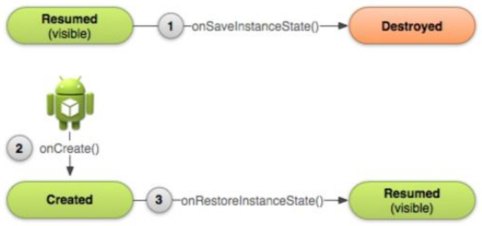


onRestoreInstanceState method

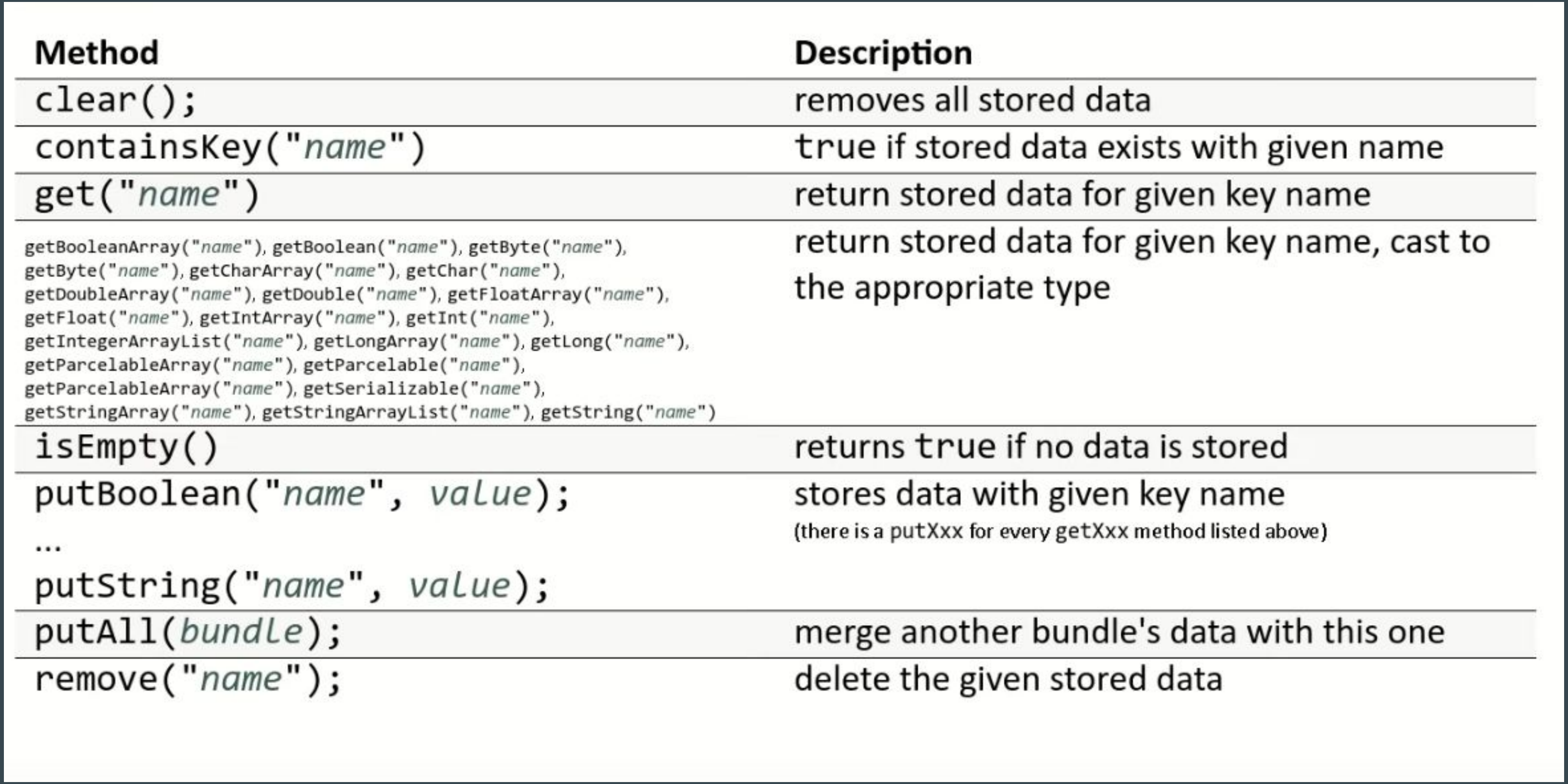


- When an activity is recreated later, the event method **onRestoreInstanceState** is called. *
  - This method can restore any "non-persistent" state of the app.
  - **Bundle** from onSaveInstanceState from before is passed back in.
    - ** older versions of Android put this code in onCreate; don't do that any more*

```
public void onRestoreInstanceState(Bundle inState) {
    super.onRestoreInstanceState(inState);  // always call super
    int name = inState.getInt("name");
    String name = inState.getString("name");
    ...
}
```



Bundle methods

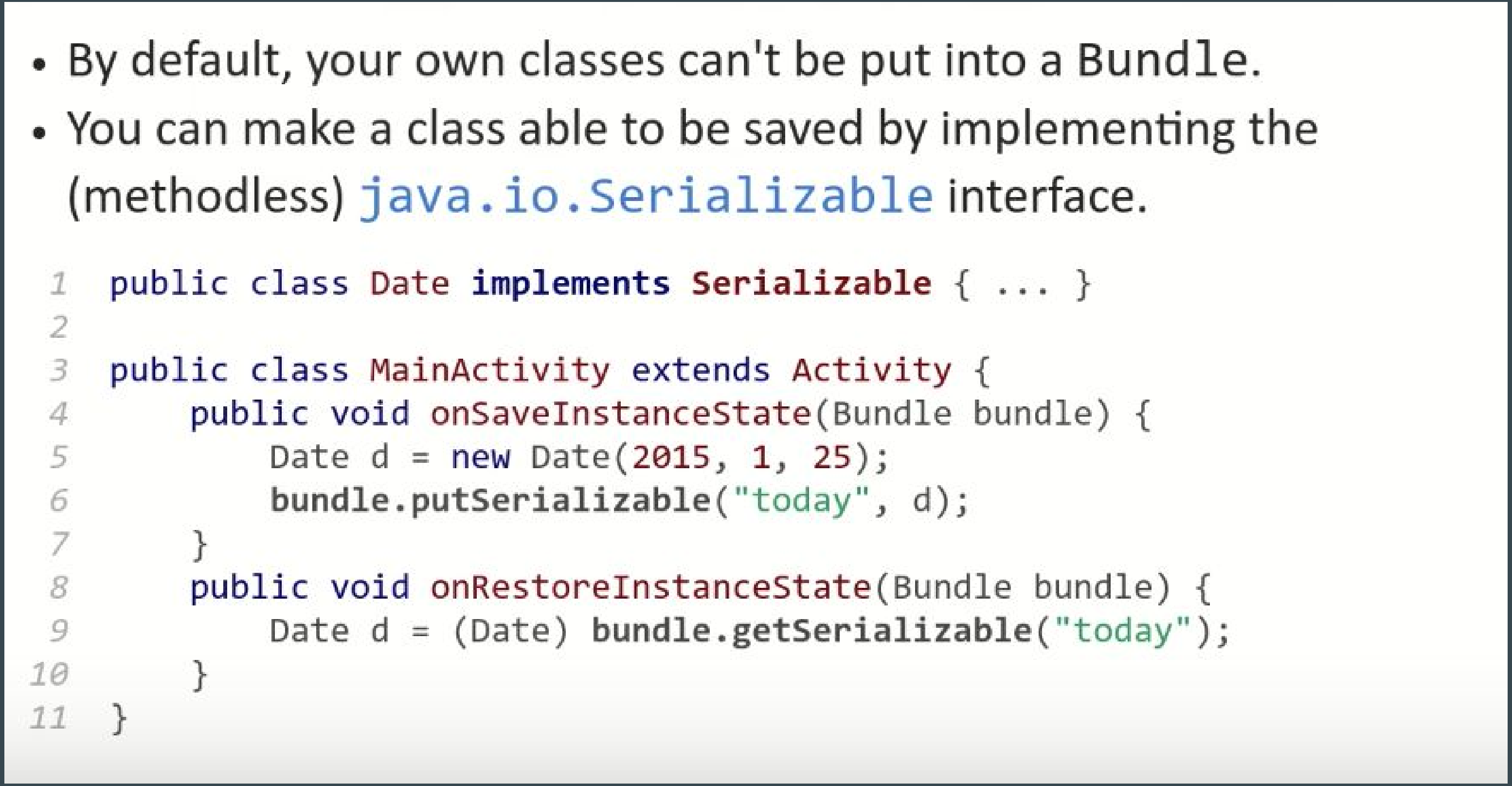


Method	Description
`clear();`	removes all stored data
`containsKey("name")`	`true` if stored data exists with given name
`get("name")`	return stored data for given key name
`getBooleanArray("name")`, `getBoolean("name")`, `getByte("name")`, `getByte("name")`, `getCharArray("name")`, `getChar("name")`, `getDoubleArray("name")`, `getDouble("name")`, `getFloatArray("name")`, `getFloat("name")`, `getIntArray("name")`, `getInt("name")`, `getIntegerArrayList("name")`, `getLongArray("name")`, `getLong("name")`, `getParcelableArray("name")`, `getParcelable("name")`, `getParcelableArray("name")`, `getSerializable("name")`, `getStringArray("name")`, `getStringArrayList("name")`, `getString("name")`	return stored data for given key name, cast to the appropriate type
`isEmpty()`	returns `true` if no data is stored

`putBoolean("name", value);`
 ...
 `putString("name", value);`
	stores data with given key name
	 (there is a `putXxx` for every `getXxx` method listed above)

`putAll(bundle);`	merge another bundle's data with this one
`remove("name");`	delete the given stored data

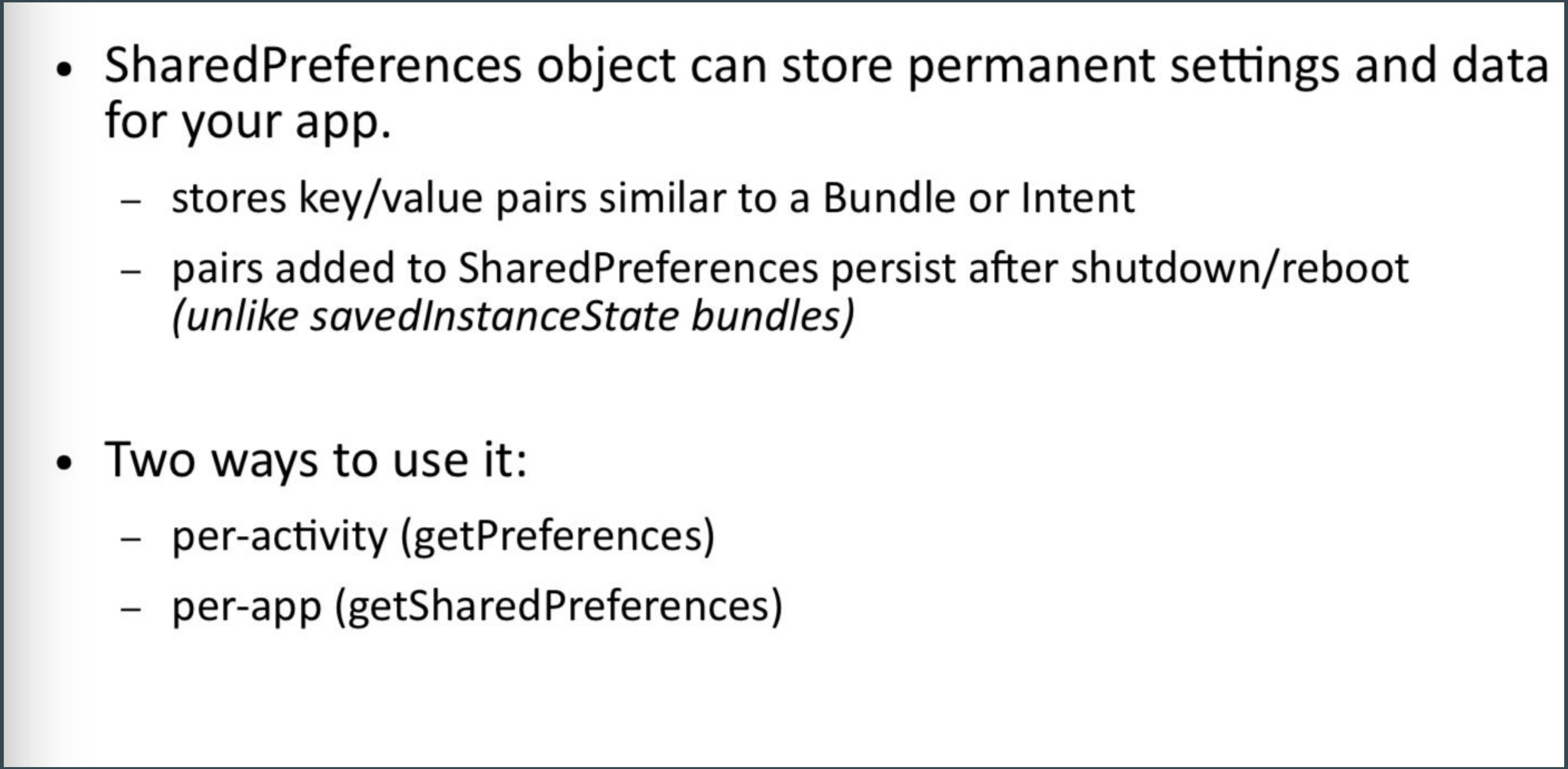
Saving your own classes



- By default, your own classes can't be put into a `Bundle`.
- You can make a class able to be saved by implementing the (methodless) `java.io.Serializable` interface.

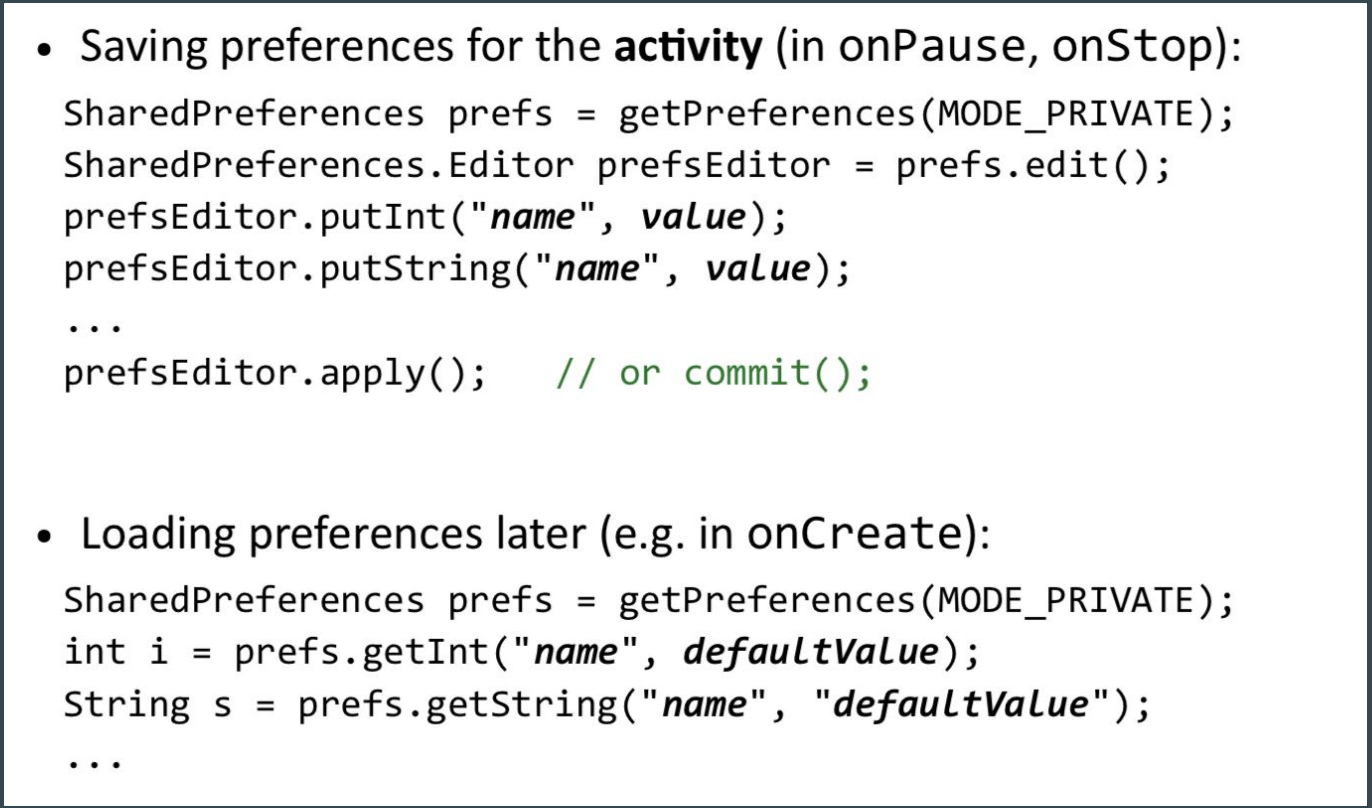
```java
public class Date implements Serializable { ... }

public class MainActivity extends Activity {
    public void onSaveInstanceState(Bundle bundle) {
        Date d = new Date(2015, 1, 25);
        bundle.putSerializable("today", d);
    }
    public void onRestoreInstanceState(Bundle bundle) {
        Date d = (Date) bundle.getSerializable("today");
    }
}
```

Preferences



- SharedPreferences object can store permanent settings and data for your app.
  - stores key/value pairs similar to a Bundle or Intent
  - pairs added to SharedPreferences persist after shutdown/reboot *(unlike savedInstanceState bundles)*
- Two ways to use it:
  - per-activity (getPreferences)
  - per-app (getSharedPreferences)

SharedPreferences example



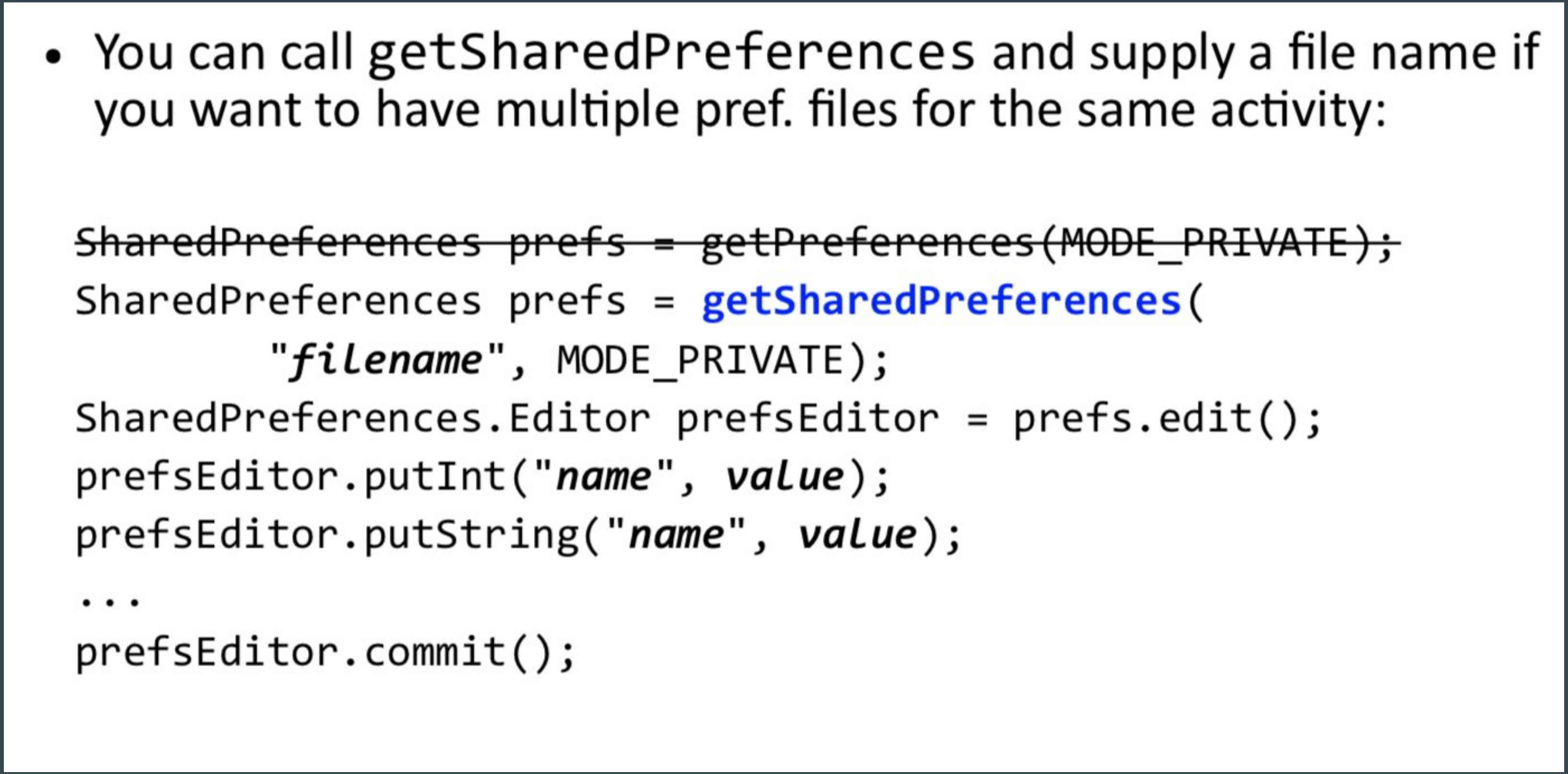
- Saving preferences for the **activity** (in onPause, onStop):

```
SharedPreferences prefs = getPreferences(MODE_PRIVATE);
SharedPreferences.Editor prefsEditor = prefs.edit();
prefsEditor.putInt("name", value);
prefsEditor.putString("name", value);
...
prefsEditor.apply();   // or commit();
```

- Loading preferences later (e.g. in onCreate):

```
SharedPreferences prefs = getPreferences(MODE_PRIVATE);
int i = prefs.getInt("name", defaultValue);
String s = prefs.getString("name", "defaultValue");
...
```

Multiple preference files



- You can call getSharedPreferences and supply a file name if you want to have multiple pref. files for the same activity:

```
~~SharedPreferences prefs = getPreferences(MODE_PRIVATE);~~
SharedPreferences prefs = getSharedPreferences(
        "filename", MODE_PRIVATE);
SharedPreferences.Editor prefsEditor = prefs.edit();
prefsEditor.putInt("name", value);
prefsEditor.putString("name", value);
...
prefsEditor.commit();
```