SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

19ITE310 - MOBILE APPLICATION DEVELOPMENT

III YEAR - VI SEM

UNIT 2 – BUILDING BLOCKS OF MOBILE APPS – I

TOPIC – UI Resources, Activity Resources

UNIT – 2
BUILDING BLOCKS OF MOBILE APPS – I

App user interface designing – Mobile UI resources (Layout, UI elements, Draw-able, Menu) - Activity – States and life cycle - Interaction amongst activities - App functionality beyond user interface – Threads - Async task - Services – states and lifecycle - Notifications.

Lab Experiments:

1. Create an application that takes the name from a text box and shows hello message along with the name entered in text box, when the user clicks the OK button
2. Create a screen that has input boxes for User Name, Password, and Address, Gender (radio buttons
for male and female), Age (numeric), Date of Birth (Date Picket), State (Spinner) and a Submit button.
On clicking the submit button, print all the data below the Submit Button (use any layout)

INTRODUCTION

- “First impression is the last impression,” very aptly fits to the user experience of an app.
- User experience focuses on the overall experience users have while interacting with the app to pursue their task at hand.
- User experience is not limited to the user interface of an app but also includes aspects such as usability, brand value, trust worthiness, usefulness, and accessibility.
- This chapter primarily focuses on user interface aspects of an app that contribute to a rich user experience.
- It starts with the core building block of Android UI – Activity – and further delves into layouts and other UI elements that are required for designing the screen makeup.
- It also drills down into the nitty-gritty of handling user interface across Android devices with varying form factor

ACTIVITY

- An Android app user interface (UI) typically comprises components such as screen layouts, UI controls, art work, and events that may occur during user interaction.
- These components can be broadly categorized into programming and nonprogramming components.
- The advantage with this clear distinction is reusability of components.
- It also brings in separation of concern that makes sure that nonprogramming components can be designed without bothering much about nuances of the app logic.
- Programming components of UI mean app components that have to be programmed using Java such as an Activity and event handling.
- The nonprogramming components of UI typically include resources such as image files, icon files, XML files, screen layout files, and other media elements.
- Android app development framework not only strives to bring in this distinction but also provides Application Programming Interfaces (APIs) that enable access of nonprogramming components inside programming components.

ACTIVITY

- We already had a glimpse of Activity in Chapter 2 in the HelloWorld app.
- Typically, an app may contain one or more Activities based on the UI requirements.
- There might be a possibility that an app may not have any Activity, in case UI is not needed.
- You may recall from Chapter 2 that an app's Activity is created by extending the Activity class.
- The Activity of an app, being a central point to the app UI, hosts the event-handling logic and runtime interaction with the nonprogramming components.
- A user typically begins interacting with an app by starting an Activity.
- Over the course of interaction, the state of the Activity may undergo changes such as being visible, partially visible, or hidden.
- Let us now explore these states, and the associated life-cycle methods that get fired behind the scenes to manage the state change

Activity States

- When the user taps on an app icon, the designated Activity gets launched into the foreground where it has the user focus.
- Pressing the Back key destroys the Activity in focus (current Activity) and displays the previous Activity to the user.
- Pressing the Home key moves the Activity in focus to the background, navigating to the Home screen. In a nut shell, from the end user's perspective, the Activity is either visible or invisible at a given point in time.
- Though, if we put on a developer's hat and try to analyze the behavior of an Activity, we may ponder upon the following questions:
- How does an Activity get started?
- What happens to the Activity when it is invisible?
- When the Activity is invisible, is it still alive?
- Does Activity retain its state when it is visible again?

Activity States

An Activity, at any given point in time, can be in one of the following four states:

1. **Active**: An Activity in this state means it is active and running. It is visible to the user, and the user is able to interact with it. Android runtime treats the Activity in this state with highest priority and never tries to kill it

2. **Paused**: An Activity in this state means that the user can still see the Activity in the background such as behind a transparent window or a dialog box, but cannot interact with it lest he or she is done with the current view. It is still alive and retains its state information. Android runtime usually does not kill an Activity in this state, but may do so in an extreme case of resource crunch.

Activity States

3. Stopped: An Activity in this state means that it is invisible but not yet destroyed. Scenarios such as a new Activity started on top of the current one or user hitting the Home key may bring the Activity to the stopped state. It still retains its state information. Android runtime may kill such an Activity in case of resource crunch.

4. Destroyed: An Activity in this state means that it is destroyed (eligible to go out of memory). This may happen when a user hits Back key or Android runtime decides to reclaim the memory allocated to an Activity that is in the paused or stopped state. In this case, the Activity does not retain its stat

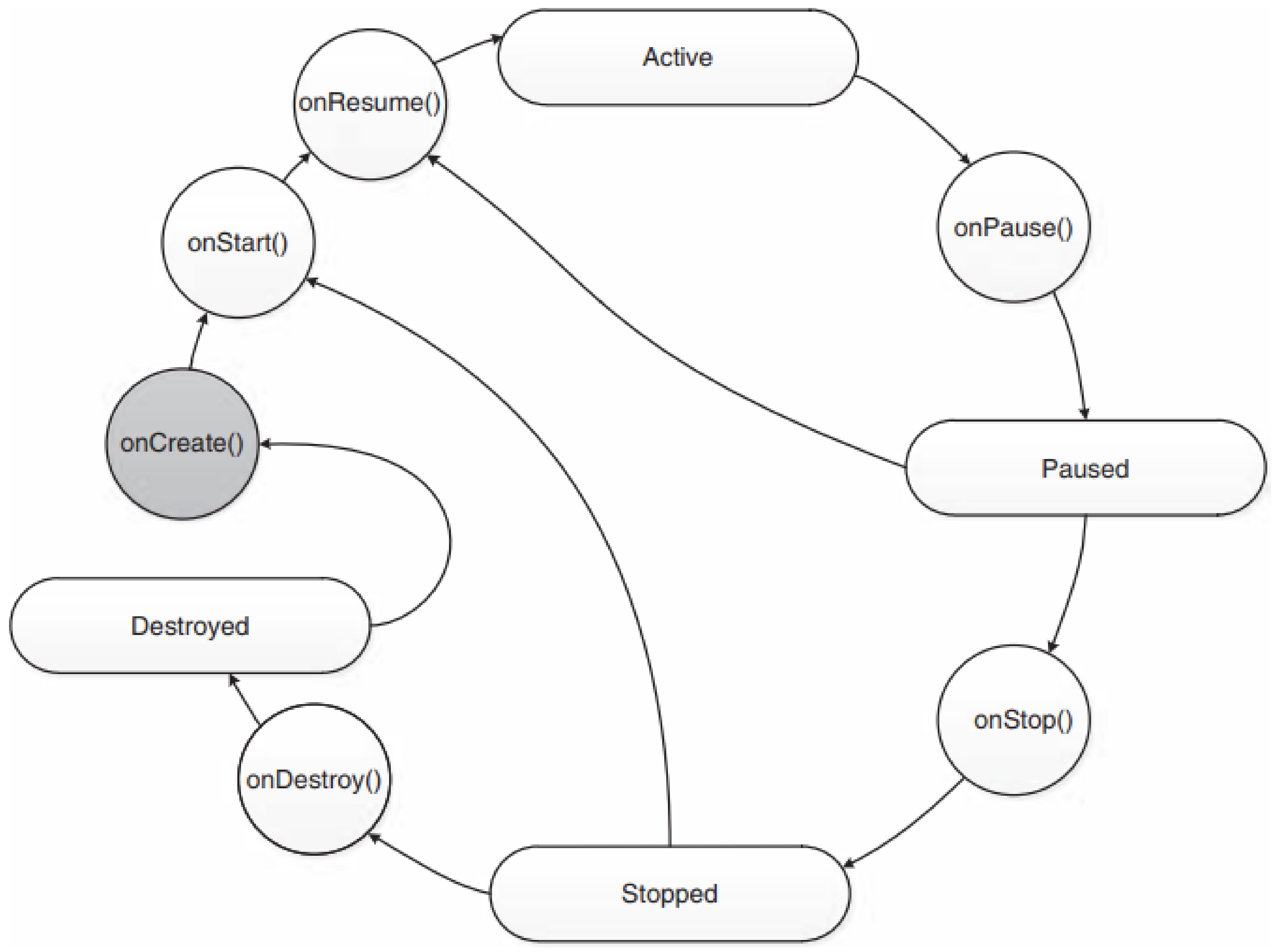
Activity States

- Android runtime manages Activities in a task stack.
- The Activity in active state always sits on the top of the stack.
- As new Activities get started further, they will be pushed on top of the last active Activity.
- And, as this happens, the last active Activity will transition to the paused/stopped state based on the scenarios explained above.
- When the Activity is destroyed, it is popped from the task stack.
- If all Activities in an app are destroyed, the stack is empty.
- An Activity does not have the control over managing its own state.
- It just goes through state transitions either due to user interaction or due to system-generated events.
- This calls for the responsibility on part of a developer to manage the app logic during state transitions, for ensuring the robust behavior of an app.
- Let us now explore the callback methods, also referred to as life-cycle methods, provided by Android app framework, which help developers in achieving this task

Life-Cycle Methods

- As we have noticed in the HelloWorld app in Chapter 2, the MainActivity has an overridden Activity life-cycle method – onCreate().
- There are six such key life-cycle methods of an Activity:
- onCreate(),
- onStart(),
- onResume(),
- onPause(),
- onStop(), and
- onDestroy().
- These life-cycle methods get executed throughout the life cycle of an Activity based on its state transitions, as depicted in Figure.

Activity States



Activity life-cycle methods.

UI Resources

User interface resources are typically the nonprogramming components of UI that help a developer in laying out the visual structure of screens, along with string constants and images that populate these screens.

There are other UI resources such as menu that provide different ways of user interaction and navigation in app

UI Resources

1 Layout Resource

- Layout resource provides a visual blueprint for an Activity and defines how the UI elements in a screen is arranged.
- Layout resources are XML files located in the res\layout folder.
- A layout file may be associated with one or more Activities.
- You may recall the setContentView(R.layout.activity_main) method that associates an Activity with a layout file (activity_main.xml).
- Upon creating a layout resource, Android framework generates a unique id for it in the R.java file present in the gen folder.
- The R.java file contains information about all nonprogramming components, using which a developer can resolve these resources inside programming components.
- For example, if a layout file is named activity_main. xml, it is represented as an integer constant – public static final int activity_main – in the R.java file under a static class – layout.
- This is applicable to all nonprogramming component

UI Resources

2 String Resource

- Android provides a mechanism to maintain string resources in XML files so that they can be reused across the app codebase.
- We could have hard coded the strings but at the cost of reusability.
- If an app developer requires to change a string, instead of fiddling with the code, he or she just needs to change appropriate string resource in the XML file. Usually string resource XML files are kept in res\values folder.
- To create a string resource XML file, right click on values folder, select New → Android XML File, and provide the appropriate values in New Android XML File wizard.
- Editing of this XML file can be done either in a visual XML file editor or a simple text-based editor. Two most commonly used string resources are string constants and string arrays. String constant is a simple key–value pair.
- Similar to a layout resource, a unique id is generated for string constants in the R.java file that may be further used to programmatically access it.

UI Resources

3 Image Resource

- The very first place where a user interacts with an image resource is the app icon itself.
- There are several other places where images can be incorporated such as in image view, layout background, button, and some other UI elements to enhance the visual appeal of an app.
- Image files are placed in res\drawable folders.
- Android provides a mechanism to access these images in app components programmatically or nonprogrammatically.
- Programmatically, Resources class is used to access an image resource using the getDrawable() method
- The return type of this method is Drawable.
- The unique id of image resource (R.drawable.filename) has to be passed as parameter to this method. In this example, ic_launcher is the filename that represents the app icon

1 Drawable drawable=resources.getDrawable(R.drawable.ic_launcher

Processes and threads overview

- When an application component starts and the application doesn't have any other components running, the Android system starts a new Linux process for the application with a single thread of execution.
- By default, all components of the same application run in the same process and thread, called the main thread.
- If an application component starts and there is already a process for that application, because another component from the application already started, then the component starts within that process and uses the same thread of execution.
- However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.
- This document discusses how processes and threads work in an Android application.

Processes and threads overview

Processes

- By default, all of an application's components run in the same process, and most applications don't change this.

- However, if you find that you need to control which process a certain component belongs to, you can do so in the manifest file.

- The manifest entry for each type of component element—
- <activity>, <service>, <receiver>, and <provider>—
- supports an android:process attribute that can specify a process the component runs in.

- You can set this attribute so that each component runs in its own process or so that some components share a process while others don't.

- You can also set android:process so that components of different applications run in the same process, provided that the applications share the same Linux user ID and are signed with the same certificates.
- The <application> element also supports an android:process attribute, which you can use to set a default value that applies to all components.

Processes and threads overview

Processes

- Android might decide to shut down a process at some point, when resources are required by other processes that are more immediately serving the user.
- Application components running in the process that's shut down are consequently destroyed.
- A process is started again for those components when there's work for them to do.
- When deciding which processes to shut down, the Android system weighs their relative importance to the user.
- For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to a process hosting visible activities.
- The decision of whether to terminate a process, therefore, depends on the state of the components running in that process.
- The details of the process lifecycle and its relationship to application states are discussed in Processes and app lifecycle.

Processes and threads overview

Threads

- When an application is launched, the system creates a thread of execution for the application, called the main thread.
- This thread is very important, because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events.
- It is also almost always the thread in which your application interacts with components from the Android UI toolkit's android.widget and android.view packages.
- For this reason, the main thread is sometimes called the UI thread.
- However, under special circumstances, an app's main thread might not be its UI thread.
- The system does not create a separate thread for each instance of a component.
- All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread.
- Consequently, methods that respond to system callbacks—such as onKeyDown() to report user actions, or a lifecycle callback method—always run in the UI thread of the process.

Processes and threads overview

Threads

- For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue. The UI thread dequeues the request and notifies the widget to redraw itself.

- Unless you implement your application properly, this single-thread model can yield poor performance when your app performs intensive work in response to user interaction. Performing long operations in the UI thread, such as network access or database queries, blocks the whole UI. When the thread is blocked, no events can be dispatched, including drawing events.

- From the user's perspective, the application appears to hang. Even worse, if the UI thread is blocked for more than a few seconds, the user is presented with the "application not responding" (ANR) dialog. The user might then decide to quit your application or even uninstall it.

- Bear in mind that the Android UI toolkit is not thread-safe. So, don't manipulate your UI from a worker thread. Do all manipulation to your user interface from the UI thread. There are two rules to Android's single-thread model:

- Don't block the UI thread.
- Don't access the Android UI toolkit from outside the UI thread.

Processes and threads overview

Worker threads

- Because of this single-thread model, it's vital to the responsiveness of your application's UI that you don't block the UI thread. If you have operations to perform that aren't instantaneous, make sure to do them in separate background or worker threads. Just remember that you can't update the UI from any thread other than the UI, or main, thread.

- To help you follow these rules, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:

- Activity.runOnUiThread(Runnable)
- View.post(Runnable)
- View.postDelayed(Runnable, long)

Processes and threads overview

Worker threads

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            // A potentially time consuming task.
            final Bitmap bitmap =
                    processBitMap("image.png");
            imageView.post(new Runnable() {
                public void run() {
                    imageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

Processes and threads overview

Worker threads

- This implementation is thread-safe, because the background operation is done from a separate thread while the ImageView is always manipulated from the UI thread.

- However, as the complexity of the operation grows, this kind of code can get complicated and difficult to maintain. To handle more complex interactions with a worker thread, you might consider using a Handler in your worker thread to process messages delivered from the UI thread. For a full explanation of how to schedule work on background threads and communicate back to the UI thread, see Background Work Overview.

Thread-safe methods

- In some situations, the methods you implement are called from more than one thread, and therefore must be written to be thread-safe.

- This is primarily true for methods that can be called remotely, such as methods in a bound service. When a call on a method implemented in an IBinder originates in the same process in which the IBinder is running, the method is executed in the caller's thread. However, when the call originates in another process, the method executes in a thread chosen from a pool of threads that the system maintains in the same process as the IBinder. It's not executed in the UI thread of the process.

- For example, whereas a service's onBind() method is called from the UI thread of the service's process, methods implemented in the object that onBind() returns, such as a subclass that implements remote procedure call (RPC) methods, are called from threads in the pool. Because a service can have more than one client, more than one pool thread can engage the same IBinder method at the same time, so IBinder methods must be implemented to be thread-safe.

Processes and threads overview

Thread-safe methods

- Similarly, a content provider can receive data requests that originate in other processes. The ContentResolver and ContentProvider classes hide the details of how the interprocess communication (IPC) is managed, but the ContentProvider methods that respond to those requests—the methods query(), insert(), delete(), update(), and getType()—are called from a pool of threads in the content provider's process, not the UI thread for the process. Because these methods might be called from any number of threads at the same time, they too must be implemented to be thread-safe.

Threads, Async task, Services

- Creating multi-thread applications for Android application development is a challenging task for many Android developers. Single and multi-threading approaches are used to create complex Android enterprise mobile apps, as they help to streamline functional operation of the code. But sometimes it is necessary to update the UI from the background thread about the operations performed. If you ever tried to access an UI element from a background thread, you have already noticed that an exception is thrown. This article will explain how to notify activity with the information posted by another thread.
- For creating multi-thread apps, Android by default does not allow the developer to modify the UI outside of the main thread. This problem is faced by many coders and if you still managed to do it you'd be breaking the second rule of the single-threaded model which is "do not access the Android UI toolkit from outside the UI thread", as stated here http://developer.android.com/guide/components/processes-and-threads.html.

Threads, Async task, Services

- Problem
- While creating complex multi-thread functions in an enterprise android app, the information generated from these threads are not notified in the UI resulting in mismatch of app business logic and crashing of the application.
- Solution
- • Implement a Handler class, override method handleMessage() which will read messages from thread queue
- • Next post message using sendMessage() method in worker thread
- There are many situations when it is required to have a thread running in the background and send information to main Activity's UI thread. From the architectural level we can use two different approaches for notifying thread activity.
- 1. Use of Android AsyncTask class
- 2. Start a new thread

Threads, Async task, Services

- Part 1 – Add Handler
- Add an instance of Handler class to your MapActivity instance.
- public class MyMap extends MapActivity {
- . . . .
- public Handler _handler = new Handler() {
- @Override
- public void handleMessage(Message msg) {
- Log.d(TAG, String.format("Handler.handleMessage(): msg=%s", msg));
- // This is where main activity thread receives messages
- // Put here your handling of incoming messages posted by other threads
- super.handleMessage(msg);
- }
- };
- . . . .
- }

Threads, Async task, Services

- Part 2 – Post Message
- In the worker thread post a message to activity main queue whenever you need Add handler class instance to your MapActivity instance.
- /**
- * Performs background job
- */
- class MyThreadRunner implements Runnable {
- // @Override
- public void run() {
- while (!Thread.currentThread().isInterrupted()) {
- // Just dummy message -- real implementation will put some meaningful data in it
- Message msg = Message.obtain();
- msg.what = 999;
- MyMap.this._handler.sendMessage(msg);
- // Dummy code to simulate delay while working with remote server
- try {
- Thread.sleep(5000);
- } catch (InterruptedException e) {
- Thread.currentThread().interrupt();
- }}}}

AsyncTask

- AsyncTask enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.
- AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.
- An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

AsyncTask

- public abstract class AsyncTask
- extends Object
- java.lang.Object
- ↳ android.os.AsyncTask<Params, Progress, Result>
- AsyncTask must be subclassed to be used. The subclass will override at least one method (doInBackground(Params...)), and most often will override a second one (onPostExecute(Result).)
- AsyncTask's generic types
- The three types used by an asynchronous task are the following:
- 1. Params, the type of the parameters sent to the task upon execution.
- 2. Progress, the type of the progress units published during the background computation.
- 3. Result, the type of the result of the background computation.

AsyncTask

- **The 4 steps**
- When an asynchronous task is executed, the task goes through 4 steps:

1. onPreExecute(), invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
2. doInBackground(Params...), invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress...) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress...) step.
3. onProgressUpdate(Progress...), invoked on the UI thread after a call to publishProgress(Progress...). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
4. onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.