Android Development

Lecture 8
Android & Concurrency

Lecture Summary

- Concurrency
- Concurrency & Java
- Concurrency & User Interface
- Concurrency & Android
  - Handler
  - AsyncTask
- Status Bar Notification

Concurrency

- Concurrency is the ability to run several parts of a program or several programs in parallel.
- Concurrency can highly improve the throughput of a program if certain tasks can be performed asynchronously or in parallel.
- Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.
- Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as concurrent software.
- Almost every computer nowadays has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

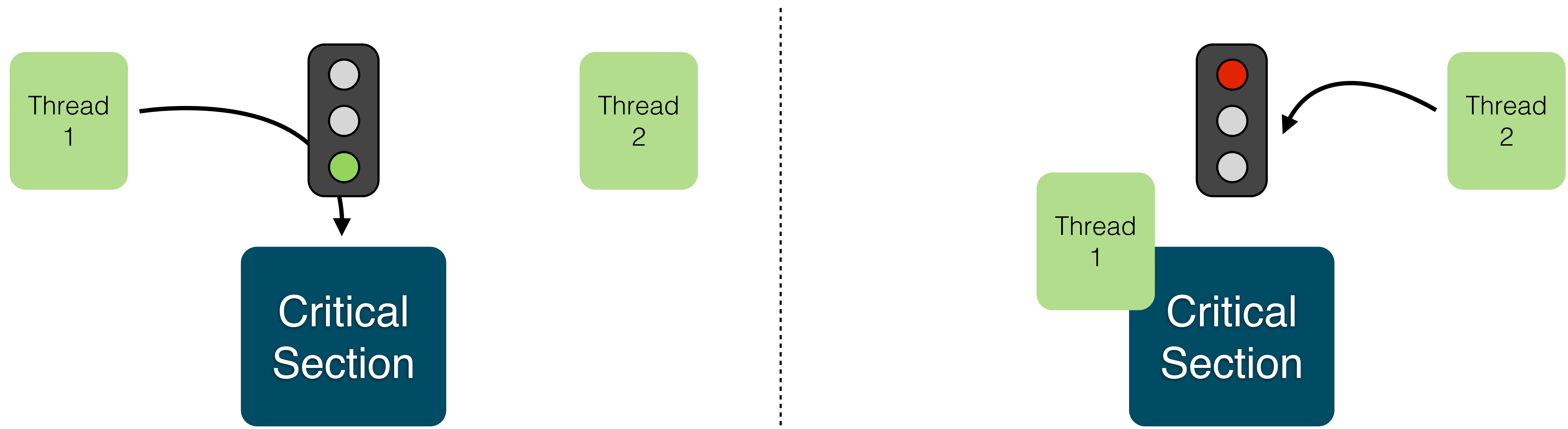
http://www.vogella.com/articles/JavaConcurrency/article.html

Concurrency Approaches

- Shared memory Communication
  - Concurrent components communicate by concurrently working on the same contents of the shared memory locations (e.g: Java and C#).
  - Usually requires the application of some form of locking (e.g., mutexes, semaphores, or monitors) to coordinate between threads working on the same critical sections.
- Message passing Communication
  - Concurrent components communicate by exchanging messages.
  - The exchange of messages may be carried out asynchronously, or may use a rendezvous style in which the sender blocks until the message is received.
  - Asynchronous message passing may be reliable or unreliable (sometimes referred to as "send and pray").
  - Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming.

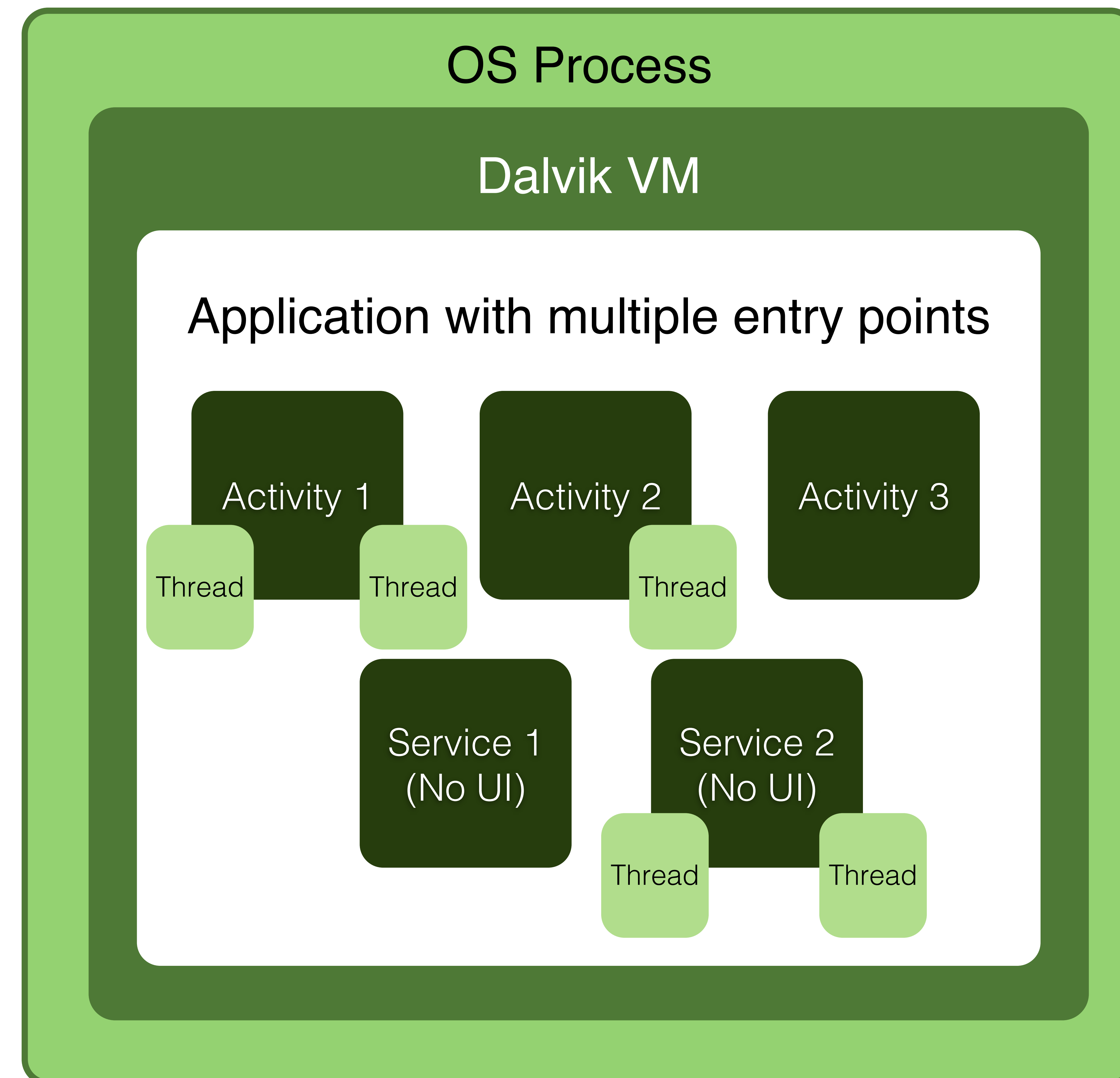
Critical Section

- A critical section is a portion of code that accesses a shared resource (e.g. data structure or device) that must not be accessed at the same time by more than one thread.
- Usually it terminates in fixed time, and a thread, task, or process must wait for a fixed time to enter it.
- Synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.



Process & Threads

- **Process**: A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.
- **Threads**: threads are so called lightweight processes which have their own call stack but an access shared data. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache.



Concurrency Issue

- When two or more threads have access to the same set of variables, it's possible for the threads to modify those variables in a way that can produce data corruption and break application's logic.
- We can have two basic problems visibility and access problems.
- A visibility problem occurs if one thread reads shared data which is later changed by other thread and if thread A does not see this change.
- A access problem occurs if several thread trying to access and change the same shared data at the same time.
- Visibility and access problem can lead to
  - Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.
  - Safety failure: The program creates incorrect data.

Concurrency & Java

- Java supports threads as part of the Java language. Java 1.5 also provides improved support for concurrency with the in the package java.util.concurrent.
- Java also provides locks to protect certain parts of the coding to be executed by several threads at the same time. The simplest way of locking a certain method or Java class is to use the keyword "**synchronized**" in a method or class declaration.
- The synchronized keyword in Java ensures:
  - that only a single thread can execute a block of code at the same time
  - ensures that each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock
- One simple rule for avoiding thread safety violation in Java is: *When two different threads access the same mutable state (variable) all access to that state must be performed holding a single lock*.

Synchronized

- The synchronized keyword in Java ensures:
  - that only a single thread can execute a block of code at the same time
  - ensures that each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock
- Synchronization is necessary for mutual exclusive access of code blocks and for reliable communication between threads.
- Synchronized can be used in three contexts:
  - To create a block (in this case the keyword take as argument a reference to an object to be used as a semaphore. Primitive types cannot be used as semaphore, but any object can)
  - on a dynamic method
  - on a static method

Synchronized

```
public class SynchronizationExample
{
    public synchronized void firstSynchMethod()
    {
        ...
    }

    public synchronized void secondSynchMethod()
    {
        synchronized(this){
            ...
        }
    }

    private Object objLock = new Object();

    public synchronized void thirdSynchMethod()
    {
        synchronized(objLock){
            ...
        }
    }
}
```

A thread executing this method holds the lock on “this”. Any other thread attempting to use this or any other method synchronized on “this” will be queued until this thread releases the lock.

This is equals to using the synchronized keyword in the method def.

A thread executing this method holds the lock on “objLock”, not this. Threads attempting to seize “this” may succeed. Only those attempting to seize “objLock” will be blocked.

Synchronized, wait & notify

- Synchronized is absolutely very convenient, but must be used with caution.
- For example a complex class that has multiple high-use methods and synchronizes them in this way may be setting itself up for lock contention.
- If several external threads are attempting to access unrelated pieces of data simultaneously, it is best to protect those pieces of data with separate locks.
- The class java.lang.Object defines the methods wait() and notify() as a part of the lock protocol that is part of every object.
  - **wait**: Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
  - **notify**: Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.
  - **notifyAll**: Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

http://www.slideshare.net/koolhits/java-performance-threading-and-concurrent-data-structures

Concurrency & User Interface

- Careful use of concurrency is particularly important to an application with User Interface.
- A well-written UI program uses concurrency to create a user interface that never "freezes" and that allows the application to be always responsive to user interaction, no matter what it's doing.
- Traditionally GUI runs on a single thread dedicated to manage both the input (mouse, touch screen, keyboard/keypad, etc ...) and output devices (display, etc ...) and executes requests from each, sequentially, usually in the order they were received.
- **Users demands responsive application, for this reason time-intensive operations (like networking, data storage and loading) should not block the main UI thread.**
- **When timeouts, large amounts of data or additional processing (such as data parsing or processing) is added to your application you should move these time-intensive operations off the main UI thread.**

Concurrency in Android

- The Android Platform supports Background Processing/Activities in 4 different ways:
  - **Threads**: Android supports the usage of the Threads class to perform asynchronous processing. Android also supplies the java.util.concurrent package to perform something in the background, e.g. using the ThreadPools and Executor classes. **Only the user interface face is allow to update the user interface. If you need to update the user interface from another Thread, you need to synchronize with this user interface Threads or you can use the "android.os.Handler" or "AsyncTasks" classes.**
  - **Handler**: The Handler class can update the user interface. A Handler provides methods for receiving instances of the Message or Runnable class.
  - **AsyncTask**: Is a special class for Android development that encapsulate background processing and facilitates the communication and updating of the application's UI.
  - **Service**: A Service allows an application to implement longer-running background operations. An application controls when its service runs by explicitly starting and stopping the service.

Handler

- The Android SDK provides a helper class that allow the developer to run code on another thread. The Handler class can allow a piece of code to run on a target thread - the thread that the Handler was instantiated in.

- The Handler class allows for example to update the application UI from a different Thread is doing a background activity such as downloading an Image from a remote WebSite.

- To use a handler you have to subclass it and override the handleMessage() to process messages. To process a Runnable you can use the post() method. You only need one instance of a Handler in your Activity.

- You can instantiate an Handler in your class declaration using for example the following code:

```
public final Handler handler = new Handler();
```

Handler & Image Download Example

```
Thread imageDownload = new Thread(new Runnable() {
    @Override
    public void run() {
        Drawable d = null;
        try {
            InputStream is = (InputStream) new URL("http://farm8.staticflickr.com/7149/6526427657_3c790c2af7_b.jpg"+image).getContent();
            d = Drawable.createFromStream(is, "image");
        } catch (Exception e) {
            e.printStackTrace();
        }

        if(d != null)
        {
            final Drawable dFinal = d;
            handler.post(new Runnable() {

                @Override
                public void run() {
                    ImageView imageView = (ImageView)findViewById(R.id.imageView);
                    imageView.setImageDrawable(dFinal);
                }
            });
        }
    }
});
imageDownload.start();
```

AsyncTask

- AsyncTask is an abstract helper class for managing background operations that eventually post back to the application's User Interface.
- It creates a simpler interface for asynchronous operations than manually creating a Java Thread class.
- Instead of creating your Thread you can subclass the AsyncTask implementing the appropriate event methods.
- 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

- When an asynchronous task is executed, the task goes through 4 steps:
  - onPreExecute(), invoked on the UI thread immediately after the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
  - 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.
  - 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.
  - 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.

AsyncTask

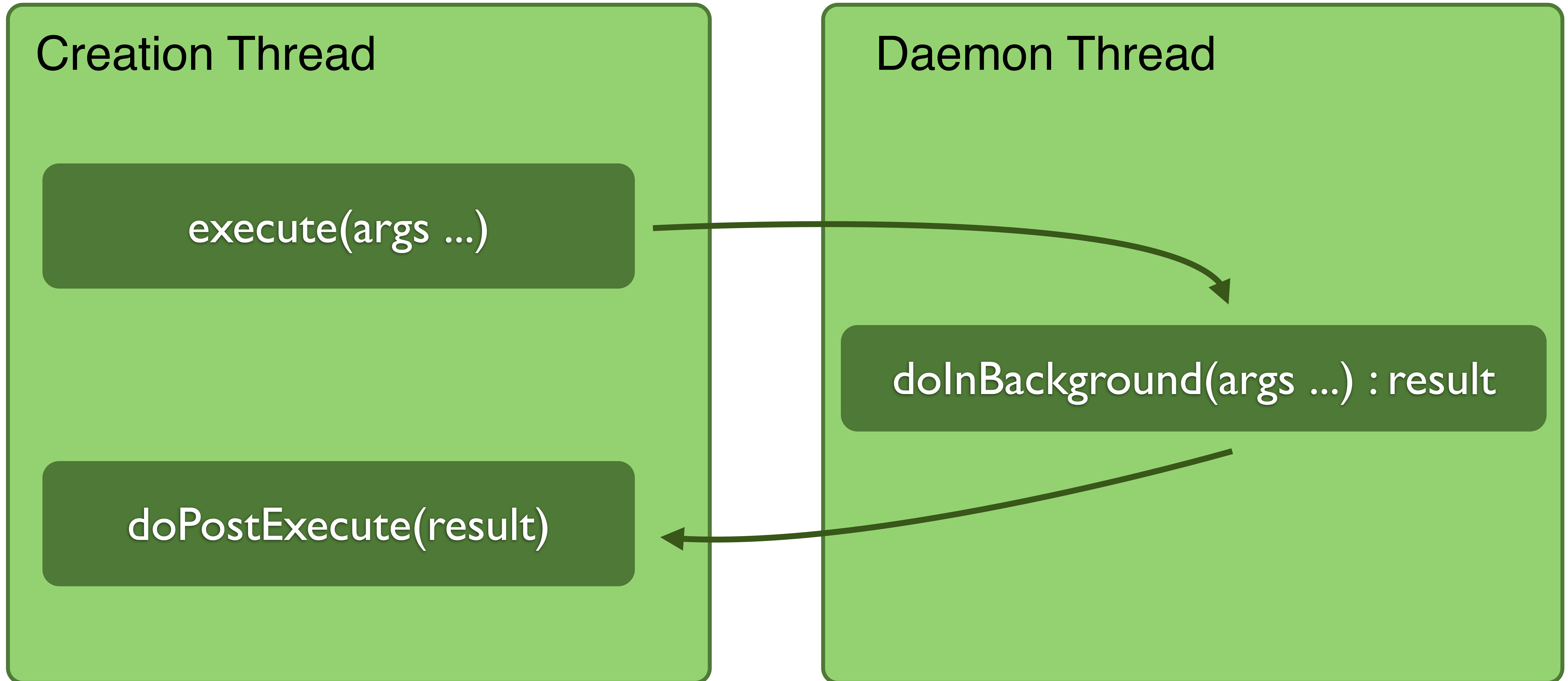
- A task can be cancelled at any time by invoking cancel(boolean). Invoking this method will cause subsequent calls to isCancelled() to return true. After invoking this method, onCancelled(Object), instead of onPostExecute(Object) will be invoked after doInBackground(Object[]) returns. To ensure that a task is cancelled as quickly as possible, you should always check the return value of isCancelled() periodically from doInBackground(Object[]), if possible (inside a loop for instance.)
- There are a few threading rules that must be followed for this class to work properly:
  - The task instance must be created on the UI thread.
  - execute(Params...) must be invoked on the UI thread.
  - Do not call onPreExecute(), onPostExecute(Result), doInBackground(Params...), onProgressUpdate(Progress...) manually.
  - The task can be executed only once (an exception will be thrown if a second execution is attempted.)

AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

```
new DownloadFilesTask().execute(url1, url2, url3);
```

AsyncTask



AsyncTask

- If AsyncTask simplifies the implementation of concurrent processing at the same time imposes strong constraints that cannot be verified automatically.
- The violation of these constraints will cause concurrency bugs that are very difficult to find.
- The most important of these constraints is that the doInBackground method it is executed on a different Thread. For this reason it must make only thread-safe references to variables inherited into its scope.
- You should avoid that one or more variables are accessed from two different threads without synchronization.
- The best solution to this problem is to make the arguments to AsyncTask are immutable. If they can't be changed they are thread-safe and need no further care.

Status Bar Notifications

- A status bar notification adds an icon to the system's status bar (with an optional ticker-text message) and a notification message in the notifications window. When the user selects the notification, Android fires an Intent that is defined by the Notification (usually to launch an Activity). You can also configure the notification to alert the user with a sound, a vibration, and flashing lights on the device.

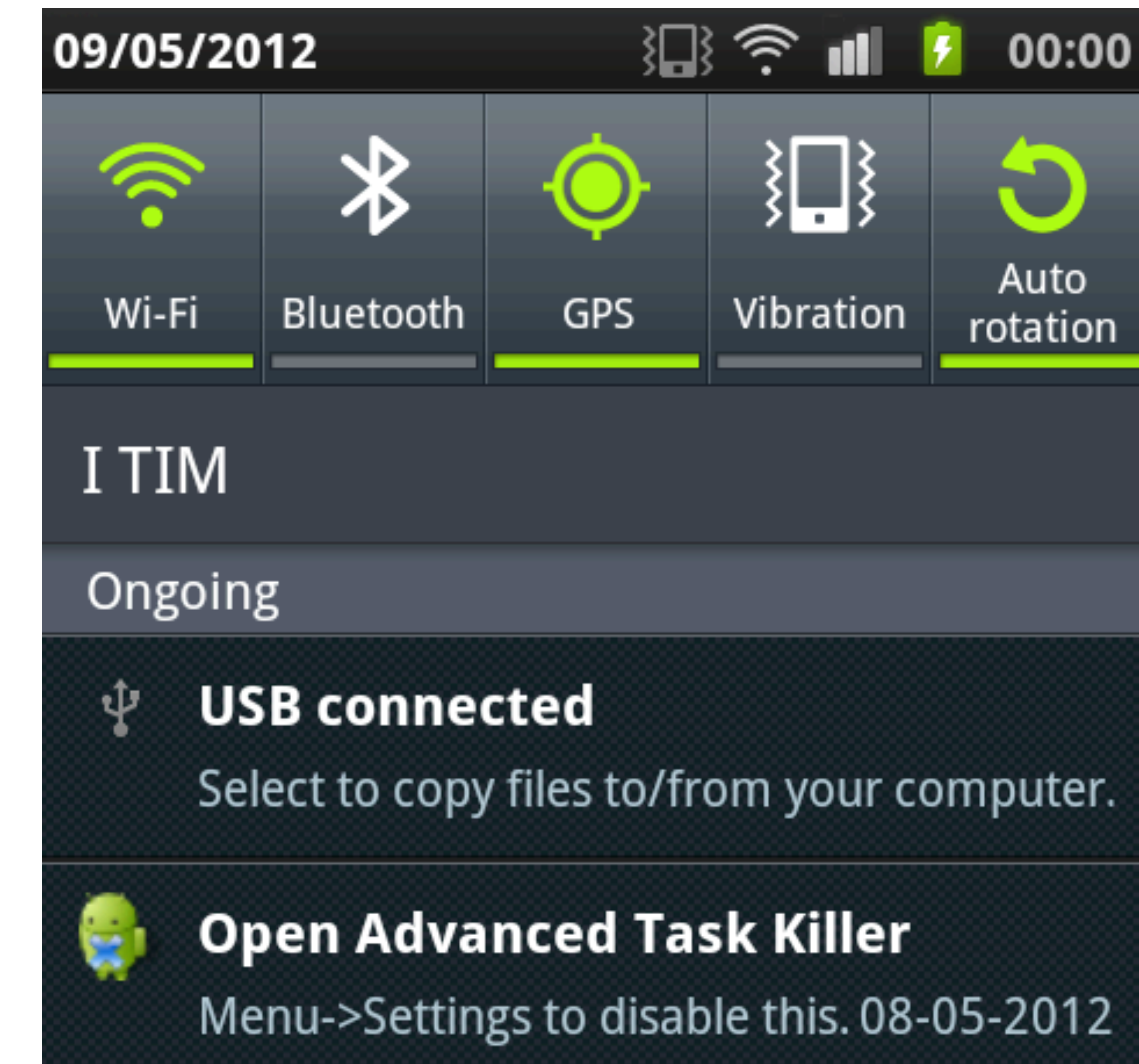
- **A status bar notification should be used for any case in which a background service needs to alert the user about an event that requires a response. A background service should never launch an activity on its own in order to receive user interaction. The service should instead create a status bar notification that will launch the activity when selected by the user.**

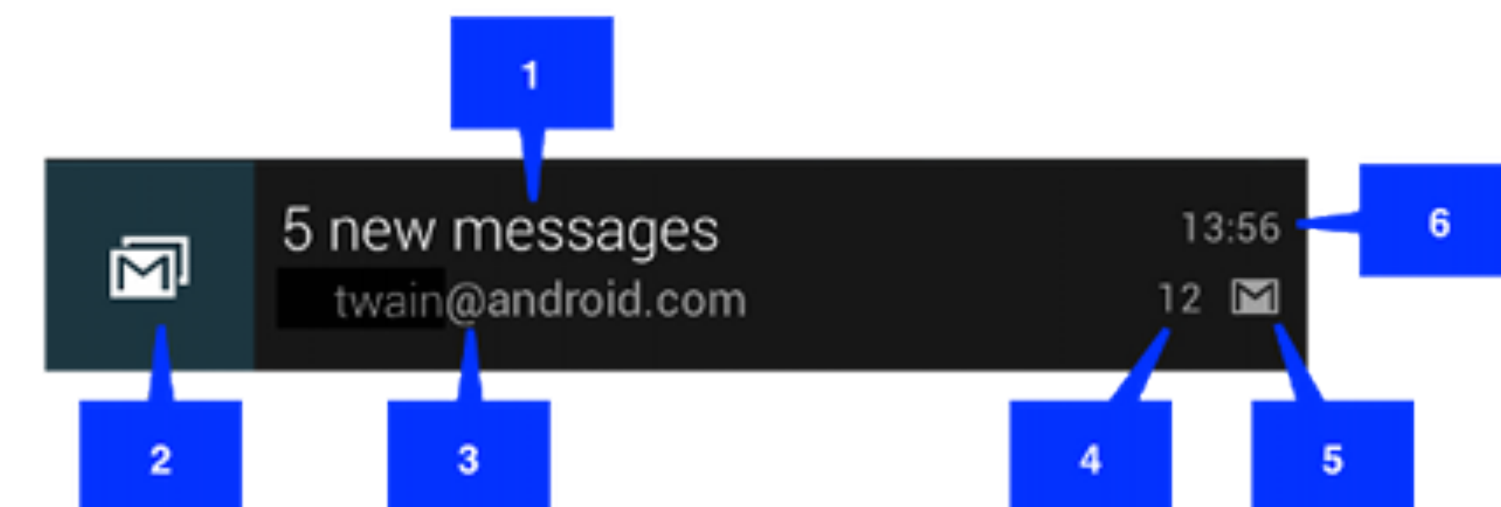
- An Activity or Service can initiate a status bar notification. Because an activity can perform actions only while it is running in the foreground and its window has focus, you will usually create status bar notifications from a service. This way, the notification can be created from the background, while the user is using another application or while the device is asleep. Classes dedicate to the creation and management of notifications are Notification and NotificationManager.

http://developer.android.com/guide/topics/ui/notifiers/notifications.html

Status Bar Notifications

- A Notification object defines the details of the notification message that is displayed in the status bar and notifications window, and any other alert settings, such as sounds and blinking lights.
- A status bar notification requires all of the following:
  - An icon for the status bar
  - A title and message, unless you define a custom notification layout
  - A PendingIntent, to be fired when the notification is selected
- Optional settings for the status bar notification include:
  - A ticker-text message for the status bar
  - An alert sound
  - A vibrate setting
  - A flashing LED setting





http://developer.android.com/guide/topics/ui/notifiers/notifications.html

Status Bar Notifications

- The NotificationManager is an Android system service that executes and manages all status bar notifications. You do not instantiate the NotificationManager directly. In order to give it your Notification, you must retrieve a reference to theNotificationManager with getSystemService() and then, when you want to notify the user, pass it your Notification with notify().

```
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager mNotificationManager = (NotificationManager) getSystemService(ns);
```

- When you want to deliver your status bar notification, pass the Notification to the NotificationManager with notify(int, Notification). The first parameter is the unique ID for the notification and the second is theNotification object. The ID uniquely identifies the notification from within your application. The ID is necessary if you need to update the notification or (if your application manages different kinds of notifications) select the appropriate action when the user returns to your application via the intent defined in the notification.

- To clear the status bar notification when the user selects it from the notifications window, add the "FLAG_AUTO_CANCEL" flag to your Notification. You can also clear it manually with cancel(int), passing it the notification ID, or clear all your notifications with cancelAll().

Status Bar Notifications

- Normally Android considers all activities within an application to be part of that application's UI flow, so simply launching the activity like this can cause it to be mixed with your normal application back stack in undesired ways.
- To make it behave correctly, in the manifest declaration for the activity the attributes android:launchMode="singleTask", android:taskAffinity="" and android:excludeFromRecents="true" must be set. The full activity declaration for this sample is:

```
<activity
android:name=".app.IncomingMessageInterstitial"
        android:label="You have messages"
        android:theme="@style/ThemeHoloDialog"
        android:launchMode="singleTask"
        android:taskAffinity=""
        android:excludeFromRecents="true">
</activity>
```

Status Bar Notifications

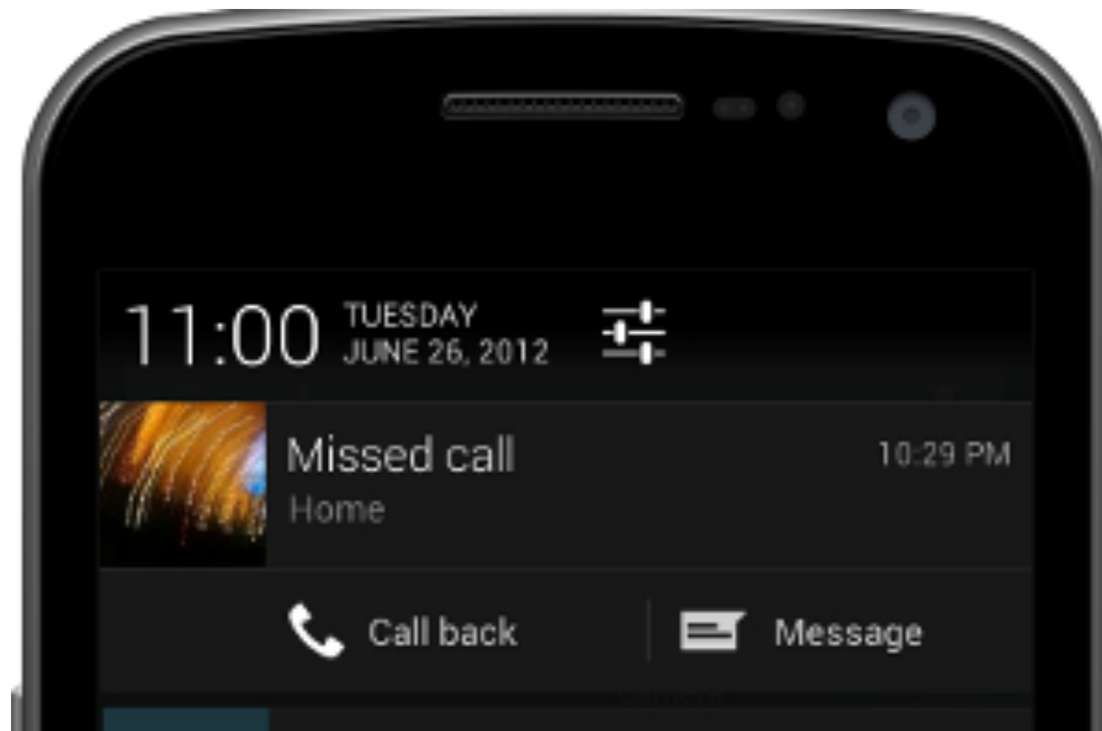
Compatibility using support library v4. NotificationCompat.Builder instead of the traditional Notification.Builder

```
// Prepare the intent triggered if the notification is selected
Intent intent = new Intent(this, MainActivity.class);
PendingIntent pIntent = PendingIntent.getActivity(this, 0, intent, 0);

// Build the notification
// Use NotificationCompat.Builder instead of just Notification.Builder to support older Android versions
Notification n  = new NotificationCompat.Builder(this)
        .setContentTitle("MobDev - YouTube App")
        .setContentText("New Video List Available")
        .setSmallIcon(R.drawable.icon_notification_exclamation_mark)
        .setContentIntent(pIntent)
        .setAutoCancel(true).build();


NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
notificationManager.notify(0, n);
```

Status Bar Notifications (Previous Version)

```
int icon = R.drawable.icon_notification_exclamation_mark;   // icon from resources
CharSequence tickerText = "New Video List Available";       // ticker-text
long when = System.currentTimeMillis();         // notification time
CharSequence contentTitle = "ADSource 6 - YouTube";  // message title
CharSequence contentText = "New Video List Available";      // message text

Intent notificationIntent = new Intent(this, MainActivity.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

// the next two lines initialize the Notification, using the configurations above
Notification notification = new Notification(icon, tickerText, when);
notification.setLatestEventInfo(this, contentTitle, contentText, contentIntent);
notification.flags = Notification.DEFAULT_LIGHTS | Notification.FLAG_AUTO_CANCEL;

String ns = Context.NOTIFICATION_SERVICE;
NotificationManager mNotificationManager = (NotificationManager) getSystemService(ns);
mNotificationManager.notify(NOTIFICATION_ID, notification);
```

Status Bar Notifications + Actions

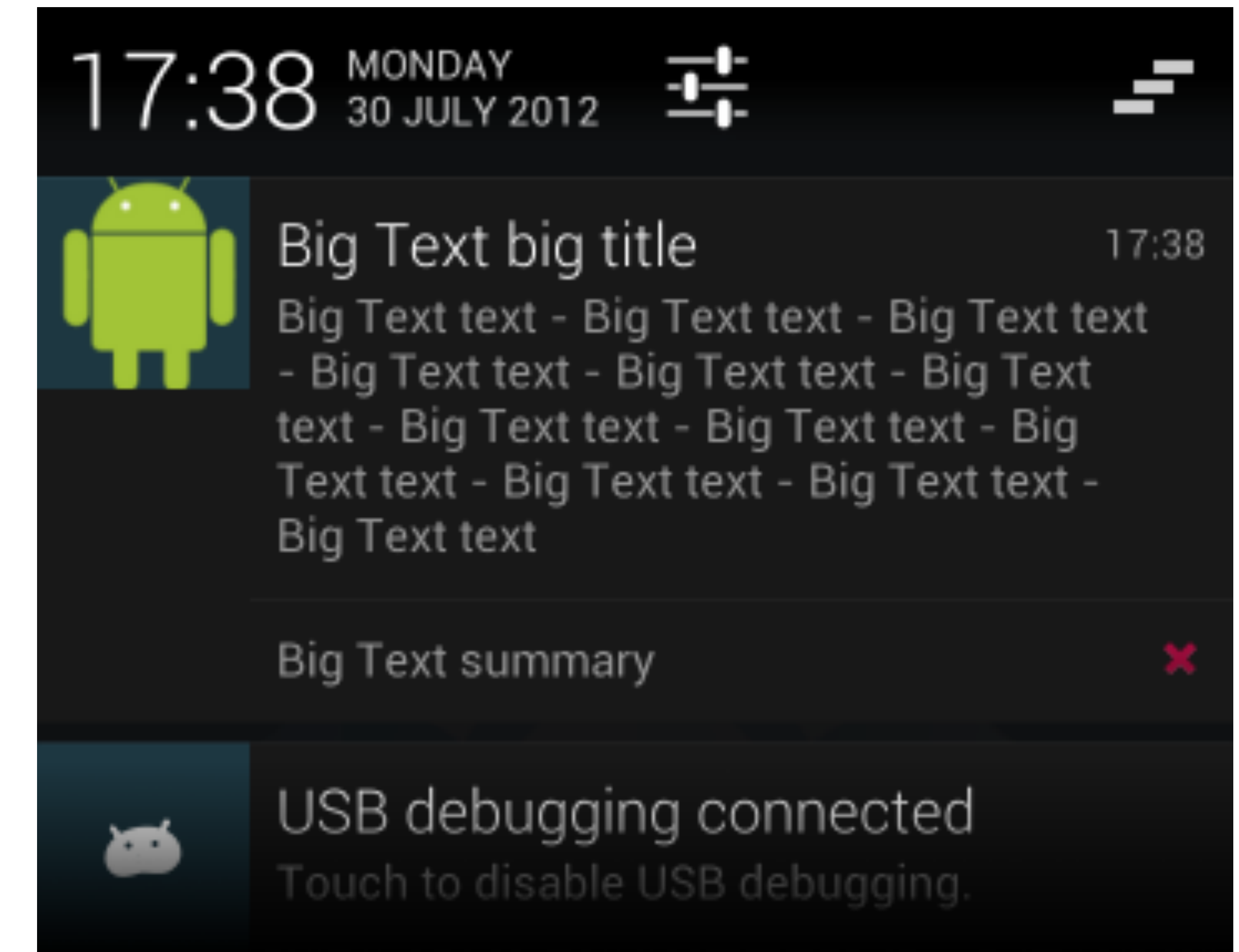



```
Notification n  = new Notification.Builder(this)
        .setContentTitle("Missed Call")
        .setContentText("Home")
        .setSmallIcon(R.drawable.user__icon)
        .setContentIntent(pIntent)
        .setAutoCancel(true)
        .addAction(R.drawable.reply_icon, "Call back", callBackIntent)
        .addAction(R.drawable.delete_icon, "Message", messageIntent).build();
```

You can add specific actions to your notification. An action allows the user to go directly from the incoming notification to an Activity (defined through an Intent) in your application, where they can look at one or more events or do further work.

Notification Style

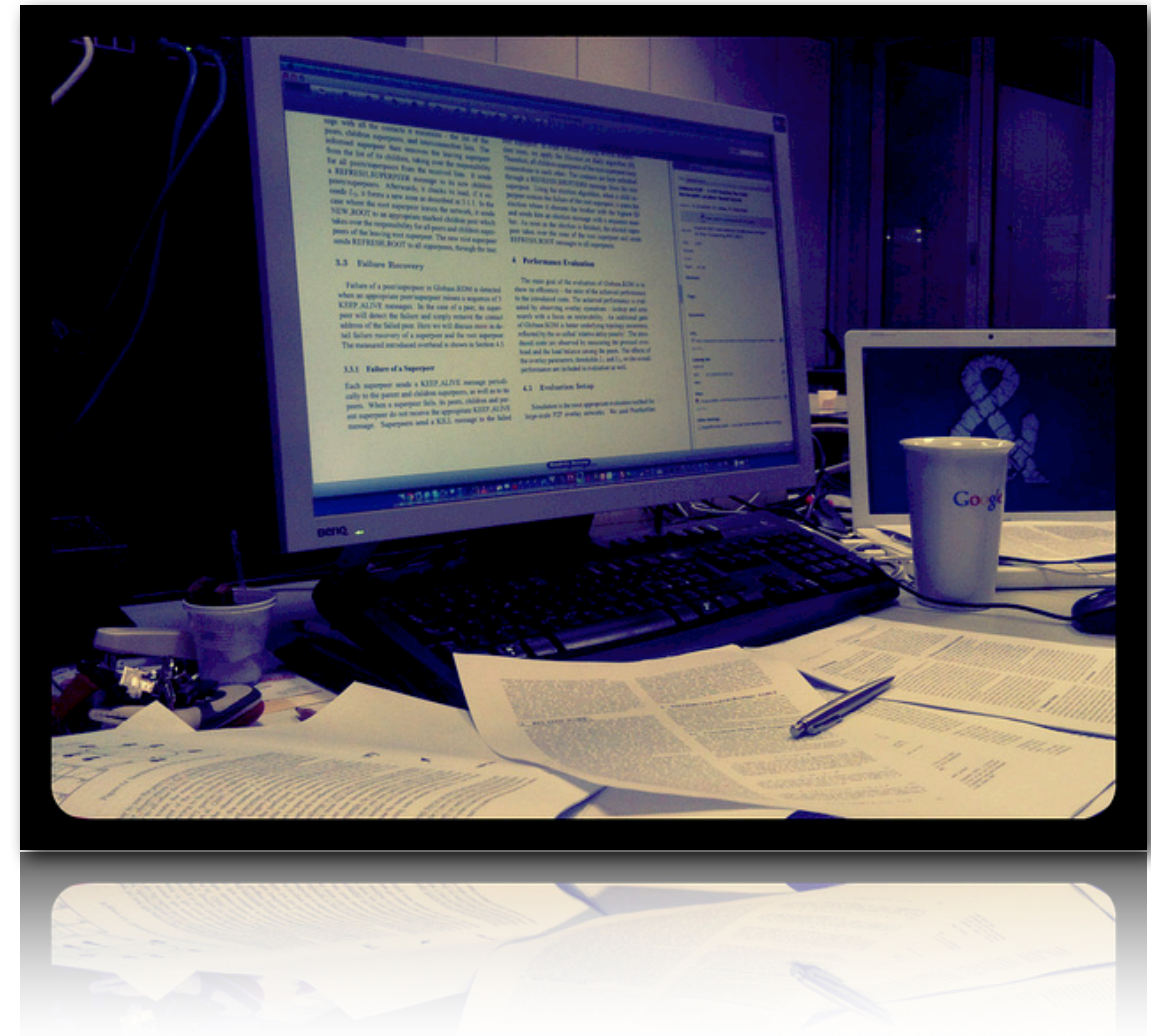
```
String longText = "bla bla bla bla ... :) ";

Notification noti = new Notification.Builder(this).
[...]
.setStyle(new Notification.BigTextStyle().bigText(longText))
```



Starting from Android 4.1 in addition to normal notification view it is also possible to define a big view which gets shown when notification is expanded. There are three styles to be used with the big view: **big picture style, big text style, Inbox style**.

Coming Up

- Next Lecture
  - Android Services
- Homework
  - Review Concurrency Problems in Java and Android (Threads & AsyncTask).



Android Development

Lecture 8
Android & Concurrency