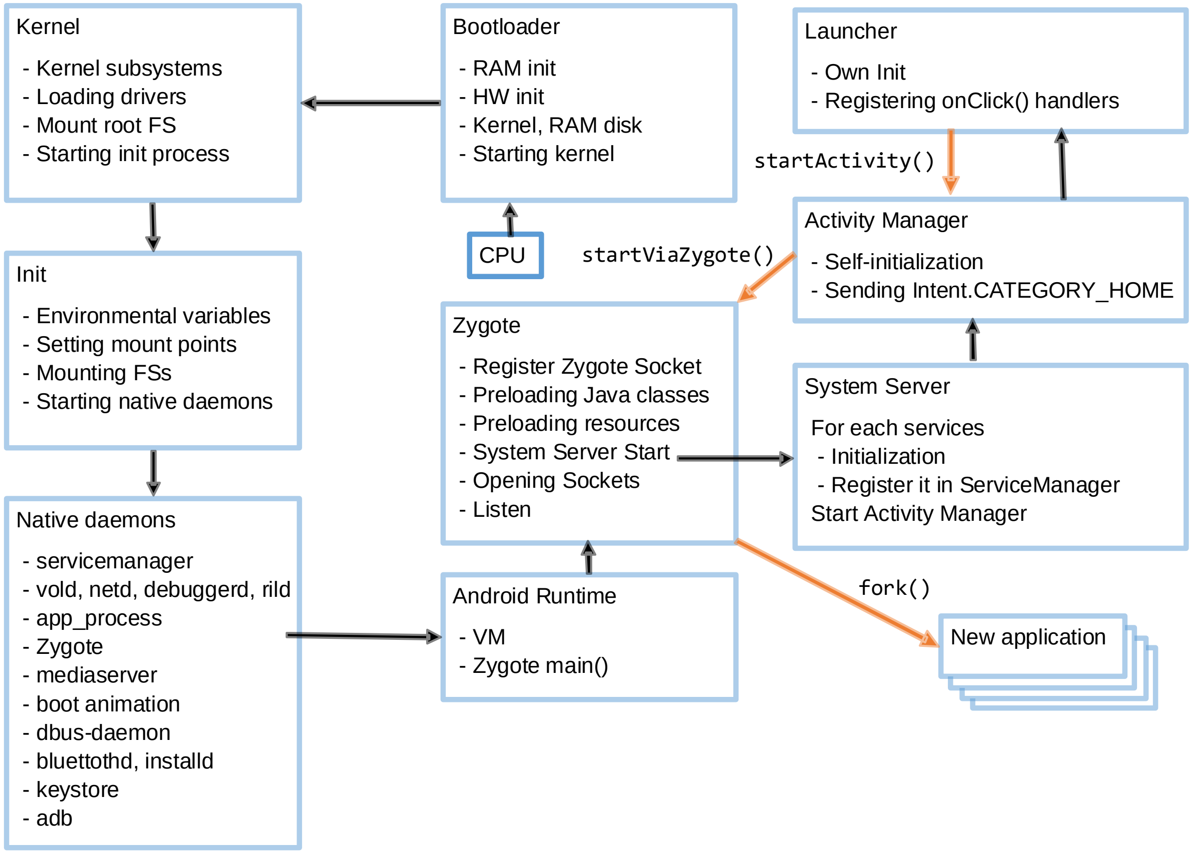
Android Development

Multithreading

App runtime
linux vs java

Boot process



Kernel

- Kernel subsystems
- Loading drivers
- Mount root FS
- Starting init process

Bootloader

- RAM init
- HW init
- Kernel, RAM disk
- Starting kernel

Launcher

- Own Init
- Registering onClick() handlers

CPU

Activity Manager

- Self-initialization
- Sending Intent.CATEGORY_HOME

Init

- Environmental variables
- Setting mount points
- Mounting FSs
- Starting native daemons

Zygote

- Register Zygote Socket
- Preloading Java classes
- Preloading resources
- System Server Start
- Opening Sockets
- Listen

System Server

For each services
 - Initialization
 - Register it in ServiceManager
Start Activity Manager

Native daemons

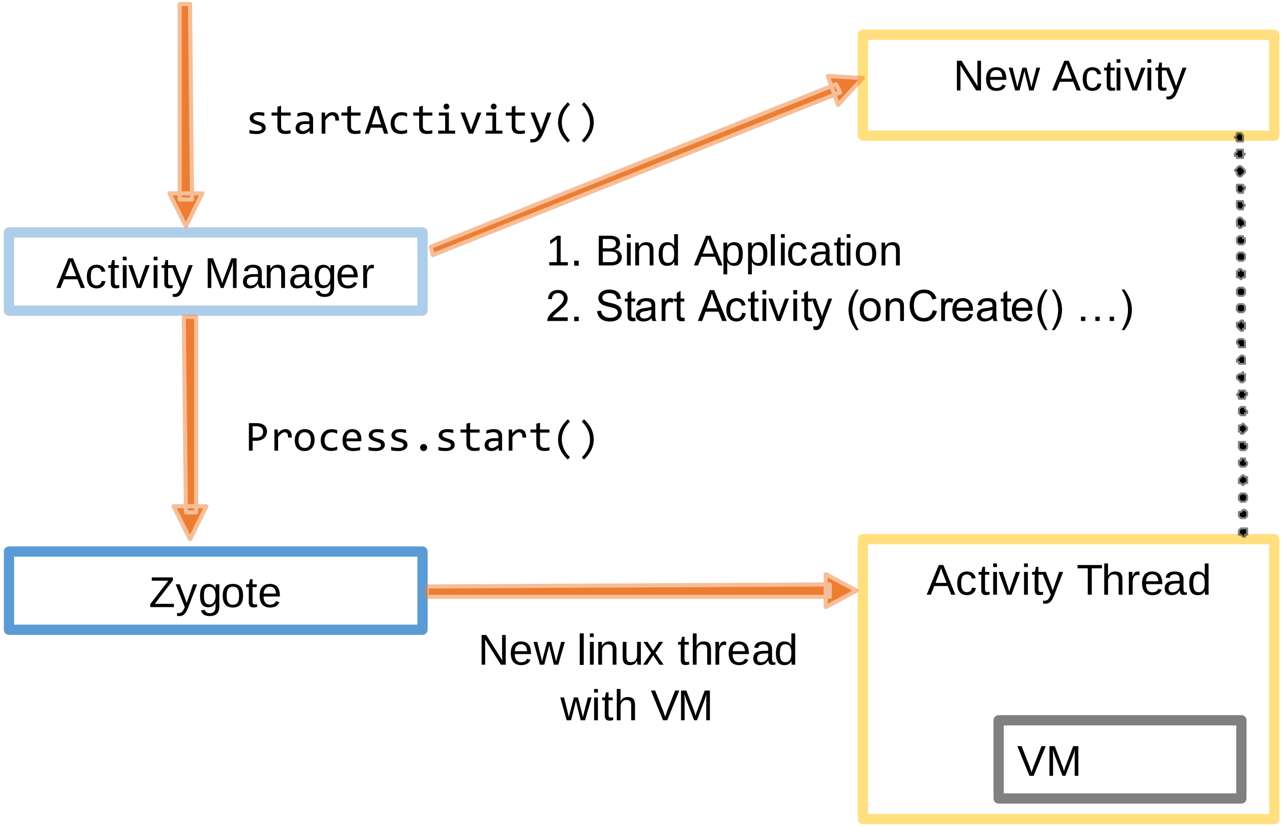
- servicemanager
- vold, netd, debuggerd, rild
- app_process
- Zygote
- mediaserver
- boot animation
- dbus-daemon
- bluettothd, installd
- keystore
- adb

Android Runtime

- VM
- Zygote main()

New application

Application process



Summary

- Following components are executed on the main thread of the application
  - `Activity`
  - `Service`
  - `BroadcastReceiver`
- Executing tasks in Android
  - The `Activity` must be alive, and responsive
  - It is being checked
  - When the reaction time of an `Activity` is more than five seconds, the system suppose that activity should be killed as it does not respond
  - Network tasks cannot be executed on the main thread, as they last longer than five seconds
    - It is ensured by the Android system
- In case of longer calculations or networking, you have to use new threads or services
  - As it has been done in Java

Multithread programs

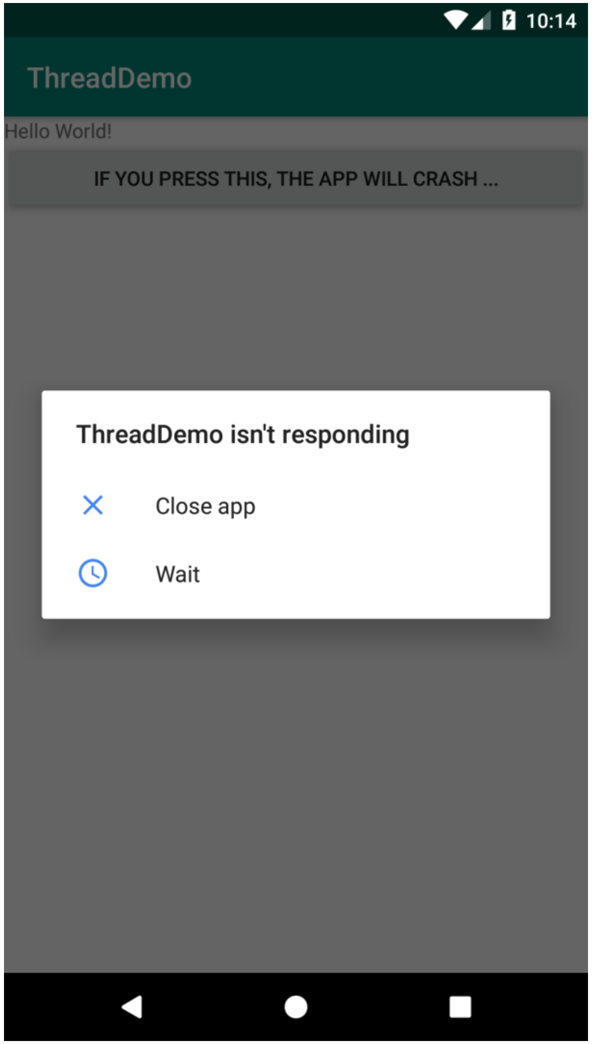
- Two different methods exist to execute background tasks
  - Starting a new `Thread`
    - Multithreading in an Activity or other program Context
    - While the user performing actions on the UI, it is possible to execute tasks in the background
    - It is similar to the method introduced in Java
      - Please be noted, that you have to respect the constraints on execution time and the access of GUI elements
  - Background `Service`
    - New Context for executing specific tasks
    - For tasks, which are not required to implement a user interface
      - However, they can be controlled via activities
    - For tasks should be executed when other applications are in the foreground
    - Examples
      - Downloading data from the web (downloading is continued while another activity is in the foreground)
      - Playing media (while listening to music other activities can be done)
      - Scheduled tasks (checking emails)
      - Complex calculations (Processing HDR+ images)

Main Thread

- The application is started on the main thread
  - It is called UI thread as well
- The components are instantiated by this thread
  - All the event handlers are executed in this thread
  - 1. The user touches a button
    2. UI thread forwards the event to the button
    3. The button refreshes itself
- Principal rules
  - It is not permitted to block the execution of this thread by any calculations
  - It is not allowed to access UI elements from any other threads
    - If it occurs, the system generates an exception, and the program will be killed

Exceptions

- Application Not Responding
  - ANR
- NetworkOnMainThreadException
  - In the case when networking is done on the main thread
- CalledFromWrongThreadException
  - Calling UI functions from any other thread than the main thread



ThreadDemo isn't responding

New Thread

- Example

```
private void methodInAndroidClass() {
     Thread thread = new Thread(doSomething, "In background");
     thread.start();
}

private Runnable doSomething = new Runnable() {
     public void run() { /* do the something here */ }
};
```

UI functions

```
private void methodInAndroidClass() {
      new Thread(new Runnable() {
             @Override
             public void run() {
                    ((TextView) findViewById(R.id.btn))
                          .setText("Wont work");
             }
      }).start();
}
```

- This example is to demonstrate what is not allowed!
  - `CalledFromWrongThreadException` is being thrown.

Returning to the main thread

- There are several solutions
  - `Activity.runOnUiThread(Runnable)`
    - Sending a `Runnable` to the main thread to be executed
    - As it is called on the main thread, it will be executed immediately
    - Otherwise, it is scheduled for execution later (as soon as possible)
  - `View.post(Runnable)`
    - Similar to the previous solution
  - `View.postDelayed(Runnable, long delayInMillis)`
    - Similar to the previous solution
    - Delay can be specified

Example

```
private void methodInAndroidClass() {
  new Thread(new Runnable() {

    @Override
    public void run() {
      ((TextView) findViewById(R.id.btn))
        .post(new Runnable() {

        @Override
        public void run() {
          ((TextView) findViewById(R.id.btn))
          .setText("It works!");
        }
      });
    }
  }).start();
}
```

AsyncTask

AsyncTask

- Allows you to run a task on a background thread while publishing results to the UI thread
- Generic Class
  - Takes parameterized types in its constructor
  - … means that it can be an array
- Three necessary types:
  - `Params`
    - Parameter type sent to the task upon execution
  - `Progress`
    - Type published to update progress during the background computation
  - `Result`
    - The type of the result of the background computation

`AsyncTask<Params, Progress, Result>`

- Functions
  - `onPreExecute()`
    - `doInBackground()` executed before doInBackground()
  - `doInBackground(Params... params)`
    - Tasks to be executed on a new thread, asynchronously
  - `publishProgress(Progress... values)`
    - The progress of calculations can be indicated by this call
  - `onProgressUpdate(Progress... values)`
    - The actual progress can be returned
    - Executed after the `publishProgress()` call
  - `onPostExecute(Result result)`
    - `doInBackground()` executed after the doInBackground call
  - `get()`
    - This function call is blocked, until `doInBackground()` finishes, and retrieves the results
  - `execute(Params... params)`
    - Starting the background task
- To start call `new AsyncTask().execute()`

Example

```
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));
             if (isCancelled()) break;
         }
         return totalSize;
     }

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

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

AsyncTask short

- Or if you do not want to return back to the main Thread

```
AsyncTask.execute(new Runnable() {
      @Override
         public void run() {
          //TODO your background code
      }
    });
```

Loaders

Why Loaders?

- AsyncTasks are tied to the Activities
- Why is that bad?
  - If the Activity goes to the background or stops running (for example on orientation change), the AsyncTask will return back to the old Activity and won't let it stop.
- Loaders from API 13 provide a framework for asynchronous loading of data
  - With the help of the LoaderManager they are not tied to Activities

Loader

- https://developer.android.com/guide/components/loaders.html
- How to create a Loader
  - 1. Create Loader ID
  - 2. Fill-in Loader Callbacks
  - 3. Initialize the loader with LoaderManager
- Loader Types
  - AsyncTaskLoader
    - Similar to AsyncTask
  - CursorLoader
    - Loads the data from a Cursor

Using a CursorLoader

1. Implement LoaderManager.LoaderCallbacks<Cursor>
2. Init/restart the loader
    - getSupportLoaderManager().restartLoader(if, bundle, callback);
3. Callback methods
    - public Loader<Cursor> onCreateLoader(int id, Bundle args)
        - Create the loader on the main thread
        - Define the query for the cursor
    - public void onLoadFinished(Loader<Cursor> loader, Cursor data)
        - Returns the result to the main thread
    - public void onLoaderReset(Loader<Cursor> loader)
        - If the loading failed or reseted for some reason

AsyncTaskLoader

- A parent of CursorLoader
- They are as efficient as CursorLoaders, but takes more code to implement, so I recommend using this over AsyncTasks when
  - The task is big enough
  - Or the independence from the Activity is important
- Some good examples of how to use them:
- https://stackoverflow.com/a/22675607/3162918

Handler

- Using handlers, one can send messages and runnable codes between threads
  - Handler – object
    - Message, Runnable
    - Handler → Thread where it has been created + MessageQueue
  - Can be scheduled to execute the code later
    - Can be sent to the itself
    - Can be sent to another thread
  - A Handler always belongs to a Thread
    - Messages sent from other threads can be processed
    - The incoming messages must be checked and processed

Example

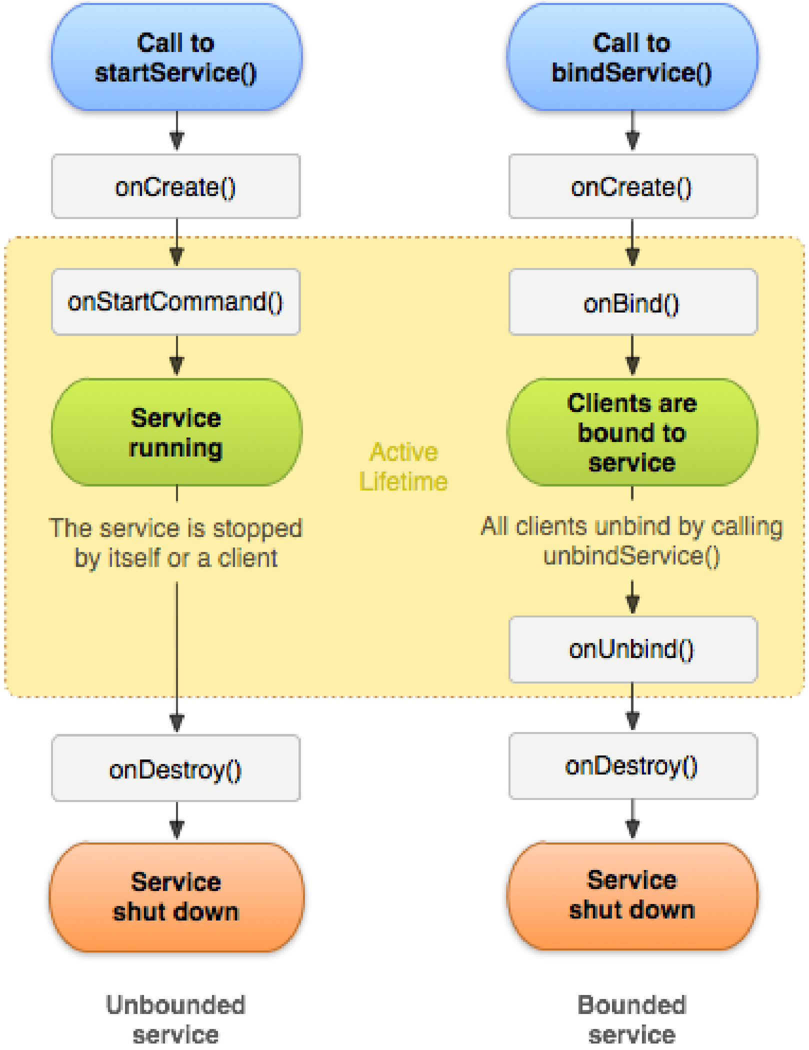
```
public class BackgroundDemos extends Activity {
        Handler handler = new Handler();
        TextView hello;
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                hello = (TextView) this.findViewById(R.id.hellotv);
                hello.setText("testing");
                new Thread() {
                        public void run() {
                                // ...
                                handler.post(doUpdateMaps);
                        }
                }.start();
        }
        Runnable doUpdateMaps = new Runnable() {
                public void run() {
                        hello.setText(maps);
                }
        };
}
```

Service

Service

- Background tasks, which have to be executed even when our Activity is paused
  - There is no UI attached to these tasks
- Starting
  - `startService()` call from an `Activity`
    - The service is independent of its starting `Activity`
    - When `Activity` finishes the execution of the service can be continued
  - `bindService()` is called to join (bind) the service to a specific `Activity`
    - The service will be finished when the starting component is finished
    - This service cooperates with the `Activity`
    - After calling this function the onServiceConnected() will be executed and the binder object is received
      - It can be used to call the functions of the service directly
- Services of other applications can be reached by using the AIDL

Started / Bounded



Types

- Foreground
  - A foreground service performs some operation that is noticeable to the user.
  - Foreground services must display a Notification.
  - Foreground services continue running even when the user isn't interacting with the app.
- Background
  - A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.
- Bound
  - A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC).
  - A bound service runs only as long as another application component is bound to it.

Service

- Service – on which thread?
  - On the main thread of the HOST process
    - local `Service`
    - If you are planning to execute a CPU intensive work, a new thread must be started
      - Playing media
      - Accessing network
      - To avoid ANR or `NetworkOnMainThreadException`
  - `IntentService`
    - Can be started to perform tasks on a new background thread
    - Example: to download a file, which tasks should not be interrupted when users leave the activity
- Think before acting! Do not mix the purpose of services and background threads
  - Executing tasks, when our application is in the background? – Service
  - Long calculations? – AsyncTask
  - Long uninterruptible calculations? – IntentService
  - Continuous, complex tasks in the background, which can be controlled from UI? – Service + Handler

Service

- AndroidManifest.xml

  ```
  <service android:name=".NewService" />
  ```

- Service class

```
public class NewService extends Service {
  public void onCreate() {
  }

  public void onStartCommand(Intent intent, int flags, int startId) {
    // ...
  }

  public IBinder onBind(Intent intent) {
    return null;
  }
}
```

Starting a Service

```
public class MyActivity extends Activity {

  @Override
  public void onCreate(Bundle savedInstanceState) {
    startService(new Intent(this, NewService.class));
    // ...
  }

  @Override
  public void onDestroy() {
    stopService(new Intent(this, NewService.class));
    // ...
  }
}
```

Background Execution Limits since 8.0

- An app is considered to be in the foreground if any of the following is true:
  - It has a visible activity, whether the activity is started or paused.
  - It has a foreground service.
  - Another foreground app is connected to the app, either by binding to one of its services or by making use of one of its content providers.
- For example, the app is in the foreground if another app binds to its:
  - IME
  - Wallpaper service
  - Notification listener
  - Voice or text service
- If none of those conditions is true, the app is considered to be in the background.

Background Execution Limits since 8.0

- While an app is in the foreground, it can create and run both foreground and background services freely.
- When an app goes into the background, it has a window of several minutes in which it is still allowed to create and use services.
- At the end of that window, the app is considered to be idle.
  - At this time, the system stops the app's background services, just as if the app had called the services' Service.stopSelf() methods.

Background Execution Limits since 8.0

- What should we do
  - The app can replace background services with JobScheduler jobs
  - This job is launched periodically, queries the server, then quits.
- As of 9.0
  - Apps using foreground services must request the FOREGROUND_SERVICE permission.

IntentService

- IntentService is a base class for `Service` that handles asynchronous requests (expressed as `Intent`s) on demand.
  - Clients send requests through `startService(Intent)` calls
  - The service is started as needed, handles each Intent, in turn, using a worker thread
  - And stops itself when it runs out of work.

Example:

```
public class SimpleIntentService extends IntentService {
    public SimpleIntentService() {
        super("SimpleIntentService");
    }
    @Override
    protected void onHandleIntent(Intent intent) {…}
}

Intent msgIntent = new Intent(this, SimpleIntentService.class);
startService(msgIntent);
```

BroadcastReciever

- An object which is notified about specific events
  - System-level events
  - For example: receiving SMS
  - This kind of events can be raised by `sendBroadcast()` call
- Registering
  - `Context.registerReceiver()`
  - Or defined in *AndroidManifest.xml* between <receiver>tags
- Unregistering
  - `Context.unregisterReceiver()`
- While the `Activity` is paused it should not receive `Intents`; thus the registering and unregistering should be done in `onResume()` and `onPause()`

BroadcastReciever

A BroadcastReceiver object is only valid (exists) while the onReceive() function call executes

  - Thus no asynchronous operation can be performed
  - And no dialog can be opened
  - And no Service can be bonded

- Example:

```
public class MyReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {…}
}
```

```
<receiver android:name="MyReceiver" >
  <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
</receiver>
```

sendBroadcast()

An Intent is the parameter of the function call

- It can be sent to any object capable of receiving broadcast call
- The Intent has to match
- In that way, we can raise broadcast messages similar to pre-defined system broadcast message

Example

```
public void broadcastIntent(View view) {
  Intent intent = new Intent();
  intent.setAction("com.CUSTOM_INTENT");
  sendBroadcast(intent);
}
<receiver android:name="MyReceiver">
  <intent-filter>
    <action android:name="com.CUSTOM_INTENT">
    </action>
  </intent-filter>
</receiver>
```

Homework

- You need to create an application which compares the multithreading capabilities of Android.
  - You need to do some background work.
- You need to compare the speed of a Service, AsyncTask, AsyncTaskLoader, new Thread.
  - Each implementation is opened by a button on the MainActivity.
  - After each test, you need to present the runtime on the screen.
- While the calculations are running the UI should be responsive (You sould be able to use it normally).
- You also need to add a progress indicator to the UI which is showing the current progress of the task.

Notification quick example

-
  ```
  NotificationManager notificationManager = (NotificationManager)
          getSystemService(NOTIFICATION_SERVICE);
  Intent intent = new Intent(this, NotificationReceiver.class);
  PendingIntent pIntent =
          PendingIntent.getActivity(this, (int)
  System.currentTimeMillis(), intent, 0);
  Notification n = new Notification.Builder(this)
          .setContentTitle("New mail from " + "test@gmail.com")
          .setContentText("Subject")
          .setSmallIcon(R.drawable.icon)
          .setContentIntent(pIntent)
          .setAutoCancel(true)
          .addAction(R.drawable.icon, "Call", pIntent)
          .addAction(R.drawable.icon, "More", pIntent)
          .addAction(R.drawable.icon, "And more", pIntent)
          .build();
  notificationManager.notify(0, n);
  ```

Notification quick example in 8.0

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    CharSequence name = getString(R.string.channel_name);
    String description = getString(R.string.channel_description);
    int importance = NotificationManagerCompat.IMPORTANCE_DEFAULT;
    NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name,
importance);
    channel.setDescription(description);
    // Register the channel with the system
    NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(this);
    notificationManager.createNotificationChannel(channel);
}
```

Notification quick example in 8.0

-
  ```
  // Create an explicit intent for an Activity in your app
  Intent intent = new Intent(this, AlertDetails.class);
  intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
  Intent.FLAG_ACTIVITY_CLEAR_TASK);
  PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);

  NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this,
  CHANNEL_ID)
          .setSmallIcon(R.drawable.notification_icon)
          .setContentTitle("My notification")
          .setContentText("Hello World!")
          .setPriority(NotificationCompat.PRIORITY_DEFAULT)
          // Set the intent that will fire when the user taps the notification
          .setContentIntent(pendingIntent)
          .setAutoCancel(true);
  ```

-
  ```
  NotificationManagerCompat notificationManager =
  NotificationManagerCompat.from(this);

  // notificationId is a unique int for each notification that you must define
  notificationManager.notify(notificationId, mBuilder.build());
  ```
- More on Notifications

Storage – further options

Next week