

CPSC475/575

Threads

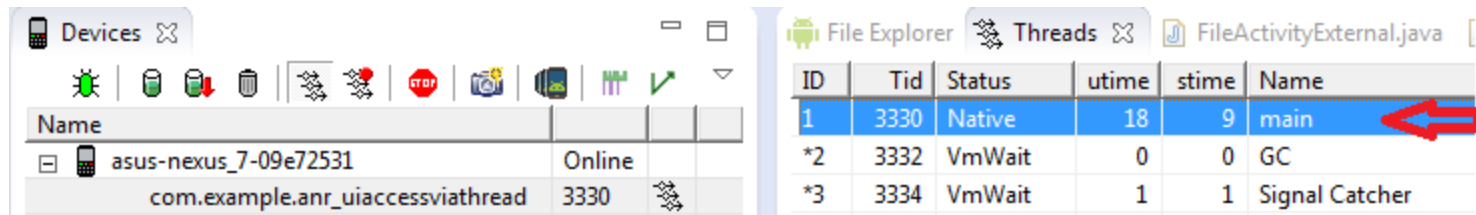
Today

- The 2 rules
- Updating UI with `AsyncTask`
- Handling Rotations
- No Synchronization between Threads Yet

The 2 Rules

- **DO NOT BLOCK THE UI THREAD**
 - Long-running code in main thread will make GUI controls nonresponsive and sometimes generate an ANR.
- **ONLY THE UI THREAD CAN ACCESS UI ELEMENTS**
 - Background threads are prohibited from updating UI.

what's the UI Thread? Its called main



The screenshot shows the Android Studio interface. On the left, the 'Devices' tab is active, displaying a list of virtual devices. The device 'asus-nexus_7-09e72531' is selected, and its details are shown below. On the right, the 'Threads' tab is active, displaying a table of threads. The 'main' thread is highlighted in blue, and a red arrow points to it. The table has columns for ID, Tid, Status, utime, stime, and Name.

ID	Tid	Status	utime	stime	Name
1	3330	Native	18	9	main
*2	3332	VmWait	0	0	GC
*3	3334	VmWait	1	1	Signal Catcher

Nonresponsive GUI Controls

- **Solution**

- *Move time-consuming operations (network access, file access, database access, image manipulation or any long running task) to other threads*
- **Runnables**— most granular, hardest to get right, useful for small tasks requiring 1 thread
- **ExecutorService** — A framework to manage threadpools, lots of flexibility, much easier to get right
- **AsyncTask** — Android specific wrapper around runnable
 - Very useful for task that are run off the UI thread that need to interact with UI Thread elements
 - Methods for starting and stopping, UI updating and returning a result




Threads Cannot Update UI

- **Solutions (alternatives)**
 - Wait until all threads are done, then update UI
 - When multithreading improves performance, but total wait time is small - If 1 thread then use runnable, if many use ExecutorService (not addressed here)
 - Use AsyncTask to divide tasks between background and UI threads
 - Especially when developing for Android from beginning and you have a lot of incremental GUI updates.

AsyncTask

- **Scenario**

- Total wait time might be large, so you want to show intermediate results (progressbar) 
- You are designing code to divide the work between GUI and non-GUI code

- **Approach (4 steps)**

- onPreExecute
- **doInBackground**
- **onProgressUpdate**
 - publishProgress
- onPostExecute or onCancelled

AsyncTask: Quick Example

- **Task itself**

```
private class ImageDownloadTask extends AsyncTask<String, Void, View> {  
    public View doInBackground(String... urls) {  
        //return view  
    }  
  
    public void onPostExecute(View viewToAdd) {  
        //  
    }  
}
```

- **Invoking task**

```
String imageAddress = "http://...";  
ImageDownloadTask task = new ImageDownloadTask();  
task.execute(imageAddress);
```

AsyncTask Details: Constructor

- **Class is genericized with three arguments**
 - `AsyncTask<ParamType, ProgressType, ResultType>`
- **Interpretation**
 - **ParamType**
 - This is the type you pass to execute, which in turn is the type that is send to `doInBackground`. Both methods use varargs, so you can send any number of params.
 - **ProgressType**
 - This is the type that you pass to `publishProgress`, which in turn is passed to `onProgressUpdate` (which is called in UI thread). Use `Void` if you do not need to display intermediate progress.
 - **ResultType**
 - This is the type that you should return from `doInBackground`, which in turn is passed to `onPostExecute` (which is called in UI thread).

AsyncTask Details: doInBackground

- **Idea**

- This is the code that gets executed in the background. It **must not** update the UI.
- It takes as arguments whatever was passed to execute
- It returns a result that will be later passed to onPostExecute in the UI thread.

- **Code**

```
private class SomeTask extends AsyncTask<Type1, Void, Type2> {  
    public Type2 doInBackground(Type1... params) {  
        return(doNonUiStuffWith(params));  
    } ...  
}  
...  
new SomeTask().execute(type1VarA, type1VarB);
```

The ... in the doInBackground declaration is actually part of the Java syntax, indicating varargs.

AsyncTask Details: onPostExecute

- **Idea**

- This is the code that gets **executed on the UI thread**. It can update the UI.
- It takes as argument whatever was returned by doInBackground

- **Code**

```
private class SomeTask extends AsyncTask<Type1, Void, Type2> {  
    public Type2 doInBackground(Type1... params) {  
        return(doNonUiStuffWith(params));  
    }  
    public void onPostExecute(Type2 result) { doUiStuff(result); }  
}  
...  
new SomeTask(). execute(type1VarA, type1VarB);
```

AsyncTask Details: Other Methods

- **onPreExecute**
 - **Invoked by the UI thread** before doInBackground starts
- **publishProgress**
 - Sends an intermediate update value to onProgressUpdate. From background thread. You call this from code that is in doInBackground. The type is the middle value of the class declaration.
- **onProgressUpdate**
 - **Invoked by the UI thread.** Takes as input whatever was passed to publishProgress.
- **Note**
 - All of these methods can be omitted.

AsyncTask Details: Cancel()

- **Idea**
 - Call `myAsyncTask.cancel(true);`
 - Sets internal canceled flag
 - Periodically check `isCanceled()` in `doInBackground` Code
 - If canceled, `onCancelled()` is called verses `onPostExecute`

**What happens when the
phone rotates?**

AsyncTask: Configuration changes

- **Problem**

- Start an AsyncTask and then phone rotates
- Activity is destroyed and restarted
- AsyncTask however is still running
- What about all the references the AsyncTask has to original activity?
- Use `onRetainCustomNonConfigurationInstance()` when activity being destroyed to save ref to thread
- Recapture thread in new Activities `onCreate()`..

AsyncTask: Configuration changes – the old way

```
private UpdateTask myUpdateTask=null;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_async_task);

    bStart = (Button) findViewById(R.id.buttonStart);
    bStop = (Button) findViewById(R.id.buttonStop);
    textViewMessage = (TextView) findViewById(R.id.textView2);
    pBar = (ProgressBar) findViewById(R.id.progressBar1);

    //what is the max value
    pBar.setMax(P_BAR_MAX);

    //lets see if the device rotated and we need to regrab it
    myUpdateTask = (UpdateTask) getLastNonConfigurationInstance();

    //if a thread was retained then grab it
    if (myUpdateTask != null) {
        myUpdateTask.attach( activity: this);
        pBar.setProgress(myUpdateTask.progress);
    }
}

/** Called by the system, as part of destroying an ...*/
@Override
public Object onRetainNonConfigurationInstance() {
    if (myUpdateTask != null) {
        Log.d(TAG, "onRetainNonConfigurationInstance");
        myUpdateTask.detach();
        return (myUpdateTask);
    } else
        return super.onRetainNonConfigurationInstance();
}
```

Reattach any running
asynctask on next
Activity create

Still works but
deprecated in
Android R (11.0)

Return Async task
and let Android
Manage it

AsyncTask: Configuration changes

- **Problem**
 - `onRetainCustomNonConfigurationInstance()` deprecated in Android R

```
@Nullable
@Override
public Object onRetainCustomNonConfigurationInstance() {
    return super.onRetainCustomNonConfigurationInstance();
}
```


AsyncTask: Configuration changes

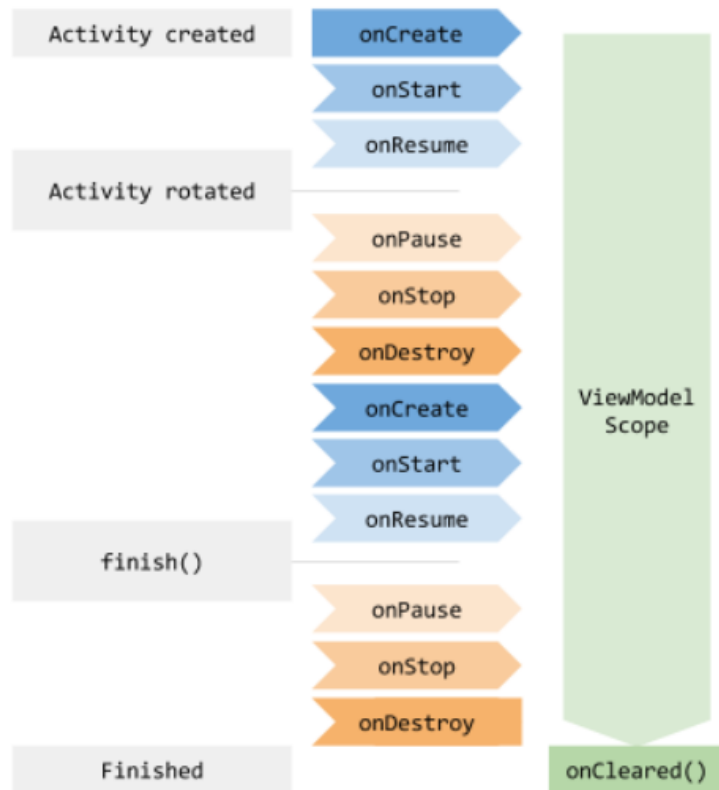
- **Problem**
 - `onRetainCustomNonConfigurationInstance()` deprecated in Android R

```
@Nullable
@Override
public Object onRetainCustomNonConfigurationInstance() {
    return super.onRetainCustomNonConfigurationInstance();
}
```

```
*
* @deprecated Use a {@link androidx.lifecycle.ViewModel} to store non config state.
*/
```

AsyncTask: Configuration changes- Use a ViewModel

- ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way.



- Notice ViewModel is created in onCreate
- Persists through Activity construction/ /destruction cycles
- Is finally destroyed when app is destroyed

AsyncTask: Configuration changes- Use a ViewModel

- ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    tv = (TextView)findViewById(R.id.textView2);
    butStart = (Button)findViewById(R.id.bStart);
    butCancel = (Button)findViewById(R.id.bCancel);
    pBar = (ProgressBar) findViewById(R.id.progressBar1);
    pBar.setMax(P_BAR_MAX);

    // Create a ViewModel the first time the system calls an activity's
    // onCreate() method. Re-created activities receive the same
    // MyViewModel instance created by the first activity.
    myVM = new ViewModelProvider( owner: this).get(DataVM.class);

    //if we have a thread running then attach this activity
    if (myVM.myTask != null) {
        myVM.myTask.set(new WeakReference<MainActivity>( referent: this));

        //a thread is running have the UI show that
        setUIState(false);
    }
}
```

```
public class DataVM extends ViewModel {
    AsyncTask myTask;

    @Override
    protected void onCleared() {
        super.onCleared();
        myTask.cancel( mayInterruptIfRunning: true);
    }
}
```

Some of the ViewModel
Its AsyncTask is a static inner class

← In Activity- get/create a ViewModel

← If there is a running AsyncTask then
attach it to this activity by
WeakReference

AsyncTask: Configuration changes – WeakReference?

- **Problem:** What if AsyncTask is holding a reference to an activity that has been destroyed/recreated (device rotates, phone call...)?
- If AsyncTask dereferences the destroyed Activity, you will get a null pointer exception.
- Worse, as long as AsyncTask holds this reference, Activity (and all its views and resources) cannot be Garbage Collected
- **Solution:** Hold a weak reference to the Activity!
- When activity destroyed the only ref to it will be the weakRef.
- If JVM detects an object with only weak references (i.e. no strong or soft references linked to it), this object will be marked for garbage collection.

AsyncTask: Configuration changes – WeakReference?

```
public static class AddTask extends AsyncTask<Integer,Integer,String> {  
    // if an object can only be reached by a weak reference then its  
    // eligible for garbage collection. So on a configurationchanged  
    // event when the activity is destroyed, it can be GCed even  
    // though ma has a weak reference to it  
    private WeakReference<MainActivity> ma; ← My WeakReference  
    public AddTask(WeakReference<MainActivity> ma) {  
        set(ma);  
    }  
    public void set(WeakReference<MainActivity> ma) {  
        //hold onto this for activity manip  
        this.ma = ma; ← Holding it  
    }  
  
    //set the UI  
    if (ma.get() != null) { ← Verifying it  
        ma.get().setUIState( b: false, s: "La  
    }
```

AsyncTask: Configuration changes

- **One last bit: to use the view model you need to include some libraries in build.gradle (app) see ViewModel Overview on Course website for details.**

ViewModel Overview |  Part of [Android Jetpack](#).

The `ViewModel` class is designed to store and manage UI-related data in a lifecycle conscious way. The `ViewModel` class allows data to survive configuration changes such as screen rotations.

★ **Note:** To import `ViewModel` into your Android project, see the instructions for declaring dependencies in the [Lifecycle release notes](#).



The screenshot shows the `build.gradle (app)` file in an IDE. The `dependencies` block is expanded, showing several implementation dependencies. A red rounded rectangle highlights the configuration for the `ViewModel` dependency, which is added in a separate block below the main dependencies list.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.lifecycle:lifecycle-viewmodel-savedstate:1.0.0-alpha01'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'com.google.android.material:material:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.navigation:navigation-fragment:2.0.0'
    implementation 'androidx.navigation:navigation-ui:2.0.0'

    def lifecycle_version = "2.2.0"
    def arch_version = "2.1.0"
    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
}
```

AsyncTask: Configuration changes

- Demo AsyncTask_simple

AsyncTask: One last bit

- **Standard implementation will execute 1 AsyncTask at a time (even if you try to run many at once)**

```
UpdateTask myTask = new UpdateTask();  
myTask.execute();
```

- **To do more than 1 at a time**

```
UpdateTask myTask = new UpdateTask();  
myTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
```


Summary

- **Update UI incrementally with AsyncTask**
 - One update per task, but several updates per group of tasks
 - How to setup - genericized with three arguments
`AsyncTask<ParamType, ProgressType, ResultType>`
 - How to Run - `myUpdateTask.execute()`
 - What is run in separate thread `doInBackground` and `publishProgress`
 - How to cancel - `myUpdateTask.cancel(true)`
 - **Results** `onPostExecute` and `onCanceled`
 - How to recover from orientation changes, keyboard slideouts etc
 - How to communicate? Not yet just `publishProgress` and `onProgressUpdate`

Reading

- **JavaDoc**
 - AsyncTask
 - <http://developer.android.com/reference/android/os/AsyncTask.html>
- **Tutorial: Processes and Threads**
 - <http://www.javamex.com/>