Android Multi-Threading

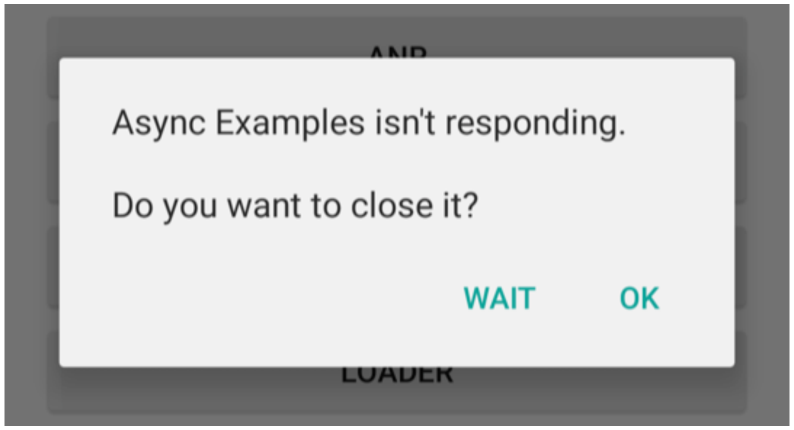
AsyncTask

Plan

- Threads in Android
- AsyncTask
- Advanced AsyncTask

Threads in Android

- Application starts in a new Linux process with a single thread of execution.
- “main” thread or aka **UI thread**
  - By default, all components (Activities, Services, Broadcast Receiver etc) of the same application run in the same process and thread
- It is responsible for the graphical interface, drawing the UI, user interactions
- UI thread **cannot get blocked by “intensive work”**
- Otherwise we lose responsiveness and application can be killed by Android
- i.e. the infamous "application not responding" (ANR) dialog


Async Examples isn't responding.

Do you want to close it?

Threads in Android

- When the app has to performs a potentially lengthy operation, **do not perform it on the UI thread**
  - E.g. downloads, network connections etc.
- Instead create a worker thread and do most of the work there.
- This keeps the UI thread running (responsiveness) and avoid frozen appearance of the app.
- The Andoid **UI toolkit is *not* thread-safe**. ==> you **MUST NOT** manipulate your UI from a worker thread
  - All UI manipulations must be done from the UI thread.

! Rules for Android thread model !

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

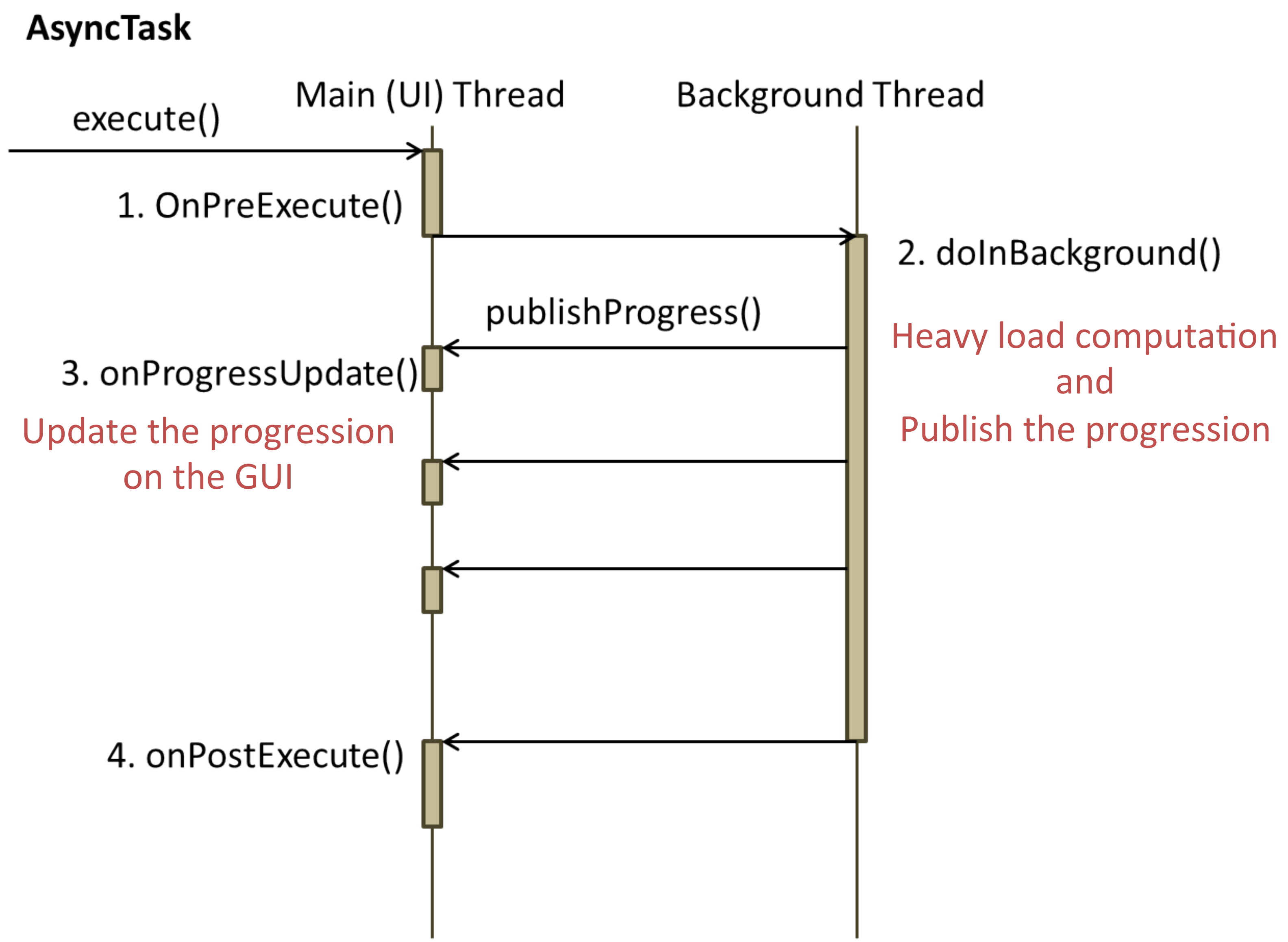
Multi-Threading in Android

- Use a different thread to complete time-consuming tasks
- The threads will run in the same process alternating the execution on the CPU
- Responsiveness is maintained
- Could we use the `Thread` class from Java though?
- It depends... If the thread needs to access the UI to update it, it cannot be done
  - UI objects are NOT thread-safe.
- Android provides a class specifically designed for this --> `AsyncTask`

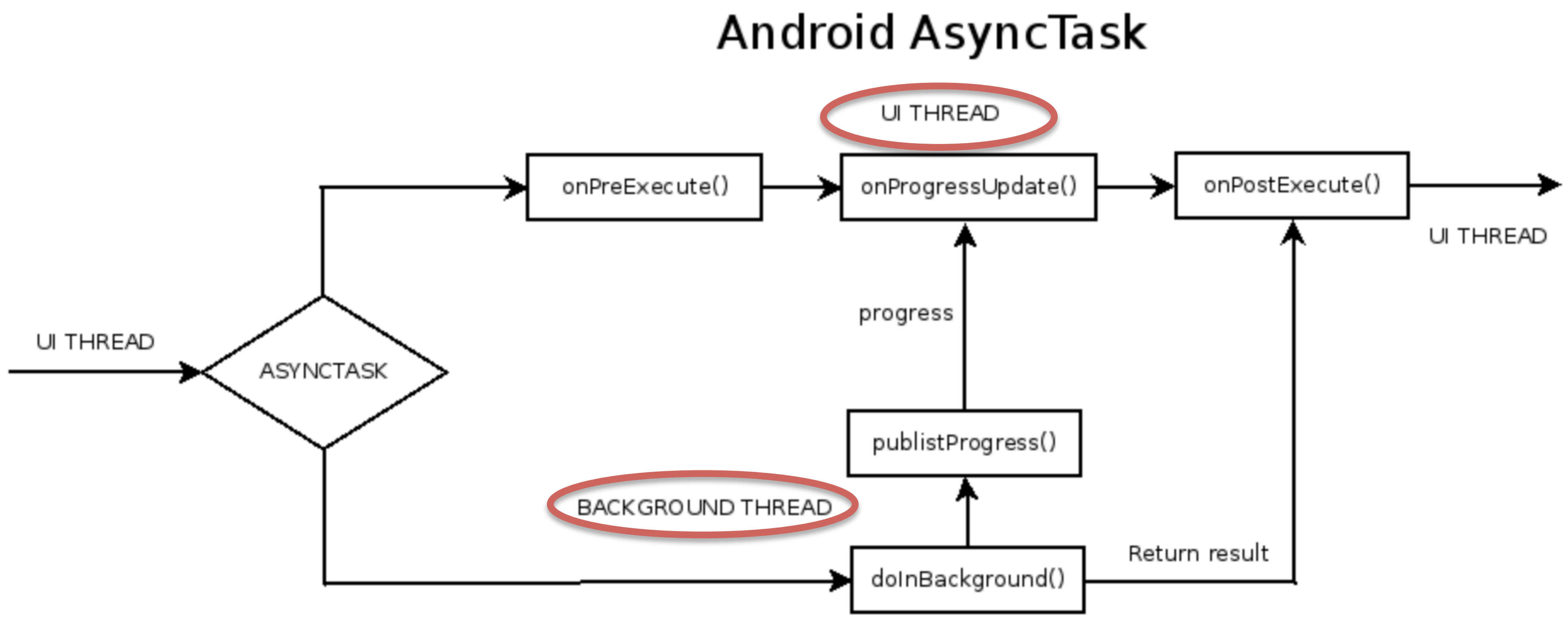
AsyncTask

- AsyncTask enables proper and easy use of the UI thread.
- Allow to perform background operations and publish results on the UI thread without having to manipulate threads.
- Designed to be a helper class around Thread.
- Used for short operations (max few seconds)
- An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

AsyncTask


AsyncTask
Main (UI) Thread
Background Thread
execute()
1. OnPreExecute()
2. doInBackground()
publishProgress()
Heavy load computation
and
Publish the progression
3. onProgressUpdate()
Update the progression
on the GUI
4. onPostExecute()

AsyncTask


Android AsyncTask
UI THREAD
onPreExecute()
onProgressUpdate()
onPostExecute()
UI THREAD
progress
UI THREAD
ASYNCTASK
publistProgress()
BACKGROUND THREAD
doInBackground()
Return result

AsyncTask

```java
private class myATask extends AsyncTask<Integer,Integer, Integer> {

   @Override
   protected void onPreExecute()
   {
      super.onPreExecute();
   }

   @Override
   protected Integer doInBackground(Integer... params)
   {
      publishProgress(50);
      return null;
   }

   @Override
   protected void onProgressUpdate(Integer... values)
   {
      super.onProgressUpdate(values);
   }

   @Override
   protected void onPostExecute(Integer result)
   {
      super.onPostExecute(result);
   }
}
```

AsyncTask

```java
private class myATask extends AsyncTask<Integer,Integer, Integer> {

   @Override
   protected void onPreExecute()
   {
      super.onPreExecute();
   }
```

- invoked **on the UI thread** before the task is executed.
- E.g, setup of the task by showing a progress bar in the user interface.

```java
   @Override
   protected void onProgressUpdate(Integer... values)
   {
      super.onProgressUpdate(values);
   }

   @Override
   protected void onPostExecute(Integer integer)
   {
      super.onPostExecute(integer);
   }
}
```

AsyncTask

```
private class myATask extends AsyncTask<Integer,Integer, Integer> {

    @Override
    protected void onPreExecute()
    {
        super.onPreExecute();
    }

    @Override
    protected Integer doInBackground(Integer... params)
    {
         publishProgress(50)
        return null;
    }
```

- **invoked on the background thread** after onPreExecute() finishes.
- perform background computation that can take a long time.
- publishProgress(Progress...) used to publish one or more units of progress.
- This triggers onProgressUpdate(Progress...).

```
    }
}
```

AsyncTask

```
private class myATask extends AsyncTask<Integer,Integer, Integer> {
```

- invoked **on the UI thread** after publishProgress(Progress...).
- Used to display any form of progress in the UI while the background computation is still executing.
  - animate a progress bar or show logs in a text field.

```
    @Override
    protected void onProgressUpdate(Integer... values)
    {
        super.onProgressUpdate(values);
    }

    @Override
    protected void onPostExecute(Integer integer)
    {
        super.onPostExecute(integer);
    }
}
```

AsyncTask

```java
private class myATask extends AsyncTask<Integer,Integer, Integer> {

    @Override
    protected void onPreExecute()
    {
        super.onPreExecute();
    }
```

- **invoked on the UI thread** after the background computation finishes.
- The result of the background computation is passed to this step as a parameter.

```java
    @Override
    protected void onPostExecute(Integer result)
    {
        super.onPostExecute(result);
    }
}
```

AsyncTask

Threading rules :

- The AsyncTask class must be loaded on the UI thread.
- 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.)

```
// create the task and execute it
MyATask task = new MyATask();
task.execute(10);

// or, equivalently, in just one shot without explicitly creating the object
new MyATask().execute(10);
```

AsyncTask

```
// create the task and execute it
MyATask task = new MyATask();
task.execute(10f);


//                            AsyncTask<Param, Progress, Result>
private class MyATask extends AsyncTask<Float, Integer, String> {

   @Override
   protected void onPreExecute() { super.onPreExecute(); }


   @Override
   protected String doInBackground(Float... params)
   {
      publishProgress(50);
      return “I’m done”;
   }

   @Override
   protected void onProgressUpdate(Integer... values) { super.onProgressUpdate(values); }

   @Override
   protected void onPostExecute(String result) { super.onPostExecute(result); }

}
```

AsyncTask

```java
// create the task and execute it
MyATask task = new MyATask();
task.execute(10f);


//                              AsyncTask<Param, Progress, Result>
private class MyATask extends AsyncTask<Float, Integer, String> {

   @Override
   protected void onPreExecute() { super.onPreExecute(); }


   @Override
   protected String doInBackground(Float... params)
   {
      publishProgress(50);
      return "I'm done";
   }

   @Override
   protected void onProgressUpdate(Integer... values) { super.onProgressUpdate(values); }

   @Override
   protected void onPostExecute(String result) { super.onPostExecute(result); }

}
```

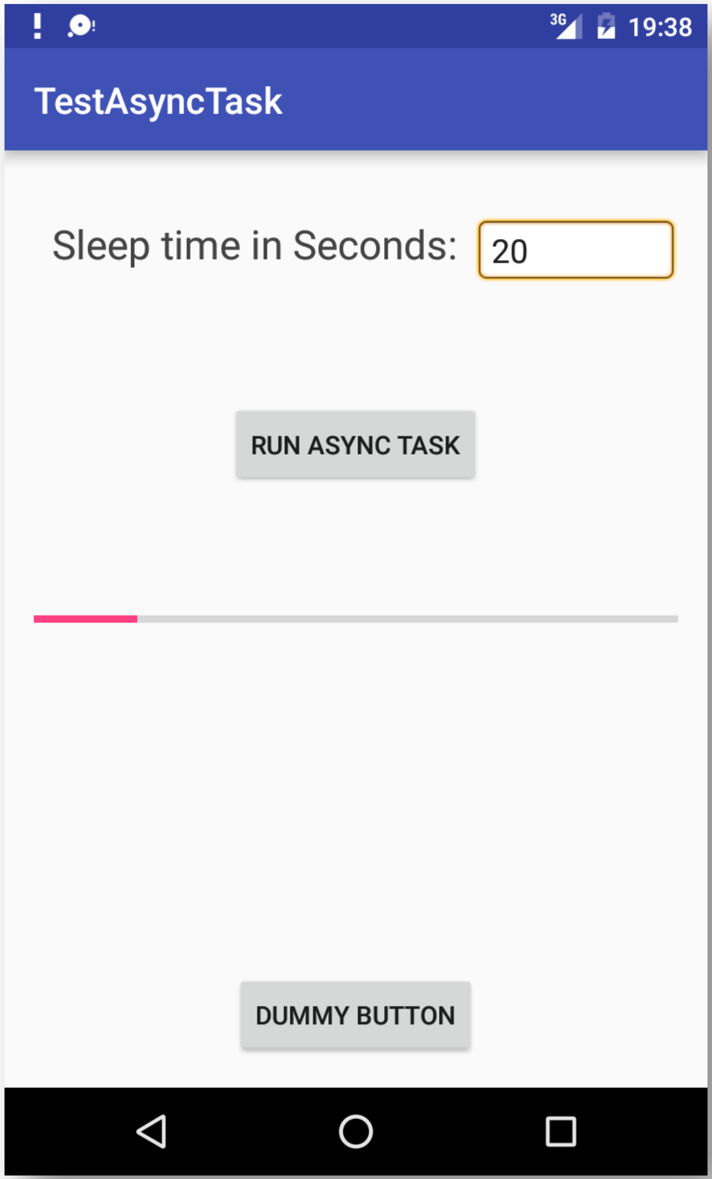
AsyncTask

```
// create the task and execute it
MyATask task = new MyATask();
task.execute(10f);


//                           AsyncTask<Param, Progress, Result>
private class MyATask extends AsyncTask<Float, Integer, String> {

   @Override
   protected void onPreExecute() { super.onPreExecute(); }


   @Override
   protected String doInBackground(Float... params)
   {
      publishProgress(50);
      return “I’m done”;
   }

   @Override
   protected void onProgressUpdate(Integer... values) { super.onProgressUpdate(values); }

   @Override
   protected void onPostExecute(String result) { super.onPostExecute(result); }

}
```

AsyncTask

```
// create the task and execute it
MyATask task = new MyATask();
task.execute(10f);


//                           AsyncTask<Param, Progress, Result>
private class MyATask extends AsyncTask<Float, Integer, String> {

   @Override
   protected void onPreExecute() { super.onPreExecute(); }


   @Override
   protected String doInBackground(Float... params)
   {
      publishProgress(50);
      return "I'm done";
   }

   @Override
   protected void onProgressUpdate(Integer... values) { super.onProgressUpdate(values); }

   @Override
   protected void onPostExecute(String result) { super.onPostExecute(result); }

}
```

AsyncTask - Example



AsyncTask - Example

```java
private final String TAG = "TestAsync";
private final static int PROGRESS_MAX = 100;
private ProgressBar mProgressBar;
private Button mStartButton;
private EditText mTimeInput;

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

   mProgressBar = (ProgressBar) findViewById(R.id.progressBar);
   mProgressBar.setMax(PROGRESS_MAX);

   mTimeInput = (EditText) findViewById(R.id.in_time);

   mStartButton = (Button) findViewById(R.id.btn_run);
   mStartButton.setOnClickListener(new View.OnClickListener()
   {
      @Override
      public void onClick(View v)
      {
          // get the number of second to sleep from the edit text
          int msToSleep = Integer.parseInt(mTimeInput.getText().toString());
          new AsyncTaskRunner().execute(msToSleep);
          Log.d(TAG, "launched runner for sleeping " + msToSleep);
      }
   });

}
```

AsyncTask - Example

```
// AsyncTask<Params, Progress, Result>
private class AsyncTaskRunner extends AsyncTask<Integer, Integer, Void>
{

   @Override
   protected void onPreExecute()
   {
      super.onPreExecute();
      mProgressBar.setProgress(0);
   }

   @Override
   protected Void doInBackground(Integer... params)
   {
      // how many 100ms in given seconds
      // param is given in seconds
      int numLoops = params[0] * 10;
      Log.d(TAG, "AsyncTaskRunner is gonna do " + numLoops + " loops");
      for(int i = 0; i < numLoops; ++i)
      {
          // sleep 100ms
          Thread.sleep(100);

          // publish update as % of the number of loops
          publishProgress((int) PROGRESS_MAX * i / numLoops);

          Log.d(TAG, "AsyncTaskRunner publishing " + ((int) PROGRESS_MAX * i / numLoops) + " progress");
      }
      return null;
   }
```

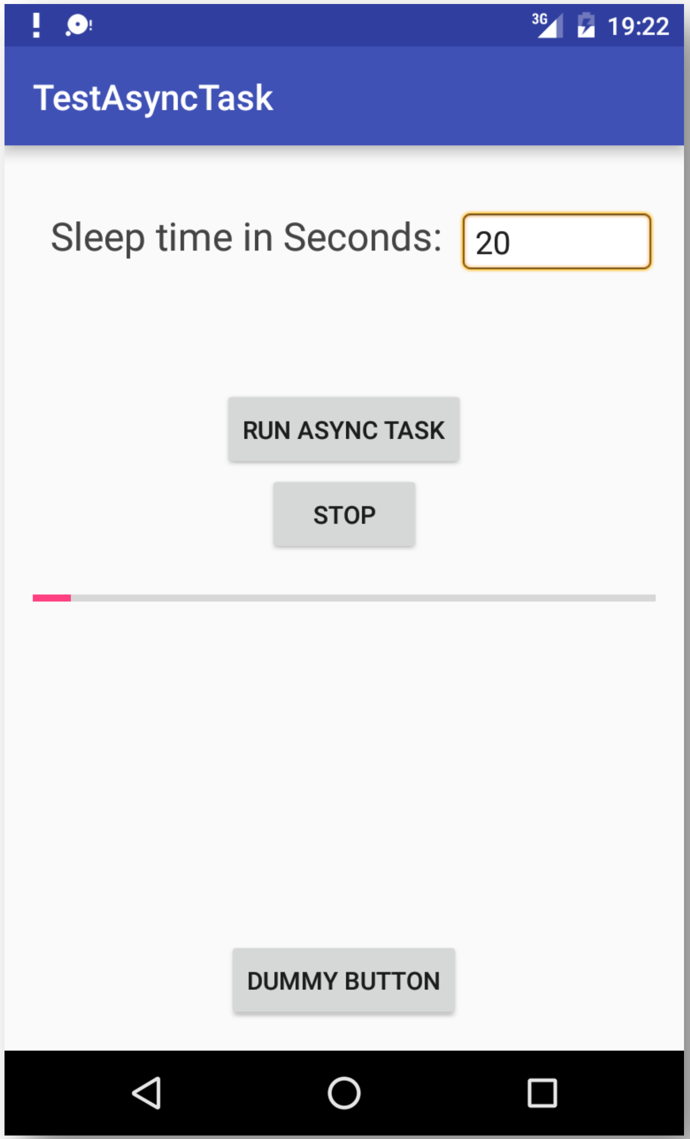
Remember, this will be executed by the UI thread, that's why it is safe to manipulate `mProgressBar`

AsyncTask - Example

```
// AsyncTask<Params, Progress, Result>
private class AsyncTaskRunner extends AsyncTask<Integer, Integer, Void>
{


   @Override
   protected void onProgressUpdate(Integer... values)
   {
      super.onProgressUpdate(values);
      // update the progress bar
      mProgressBar.setProgress(values[0]);

      Log.d(TAG, "AsyncTaskRunner setting progress bar to " + values[0] + " progress");
   }

   @Override
   protected void onPostExecute(Void aVoid)
   {
      super.onPostExecute(aVoid);
      // set the progress bar to full
      mProgressBar.setProgress(PROGRESS_MAX);
   }
}
```

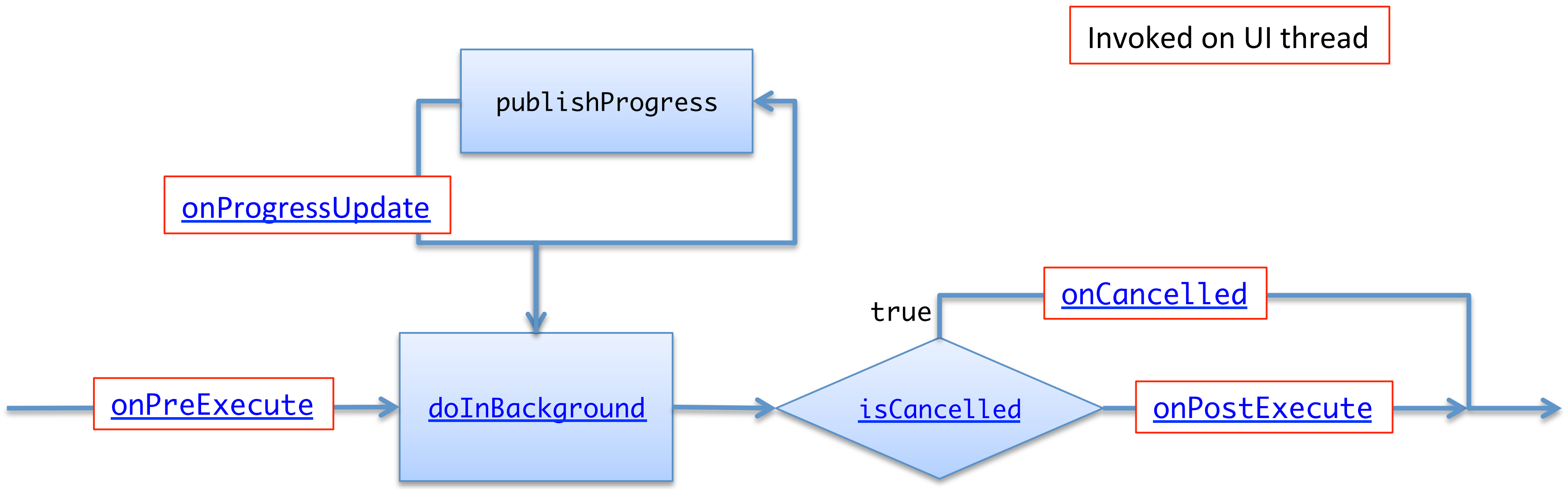
Again, the 2 callbacks will be executed by the UI thread, that’s why it is safe to manipulate `mProgressBar`

AsyncTask – Cancel the task



AsyncTask – Cancel the task

- A task can be cancelled at any time by invoking `cancel(boolean)`.
- Then method `isCancelled()` will return true.
- After `cancel()`, `onCancelled(Object)`, instead of `onPostExecute(Object)` will be **invoked on the UI thread** after `doInBackground(Object[])` returns.
- To cancel as quickly as possible, check the return value of `isCancelled()` periodically from `doInBackground(Object[])`



AsyncTask – Cancel the task

```java
mStopButton = (Button) findViewById(R.id.btn_stop);
// initially the stop button is disabled
mStopButton.setEnabled(false);
mStopButton.setOnClickListener(new View.OnClickListener()
{
   @Override
   public void onClick(View v)
   {
      // if we have already a task
      if (mAsyncTask != null)
      {
         // cancel it and disable the stop button
         mAsyncTask.cancel(true);
         mStopButton.setEnabled(false);
      }
   }
});

mStartButton = (Button) findViewById(R.id.btn_run);
mStartButton.setOnClickListener(new View.OnClickListener()
{
   @Override
   public void onClick(View v)
   {
      ...
```

AsyncTask – Cancel the task

```
mStartButton = (Button) findViewById(R.id.btn_run);
mStartButton.setOnClickListener(new View.OnClickListener()
{
   @Override
   public void onClick(View v)
   {
         int msToSleep = Integer.parseInt(mTimeInput.getText().toString());

         // if we have already a task, cancel it
         if (mAsyncTask != null)
            mAsyncTask.cancel(true);

         // create a new task
         mAsyncTask = new AsyncTaskRunner();
         mAsyncTask.execute(msToSleep);

         // enable the stop button
         mStopButton.setEnabled(true);

         Log.d(TAG, "launched runner for sleeping " + msToSleep);

   }
});
```

AsyncTask – Cancel the task

```
// AsyncTask<Params, Progress, Result>
private class AsyncTaskRunner extends AsyncTask<Integer, Integer, Void>
{

   @Override
   protected Void doInBackground(Integer... params)
   {
      // how many 100ms in given seconds
      // param is given in seconds
      int numLoops = params[0] * 10;

      for(int i = 0; i < numLoops; ++i)
      {
         if(isCancelled())
            return null;

         // sleep 100ms
         Thread.sleep(100);
         // publish update as % of the number of loops
         publishProgress((int) PROGRESS_MAX * i / numLoops);
      }
      return null;
   }
```

AsyncTask – Cancel the task

```
// AsyncTask<Params, Progress, Result>
private class AsyncTaskRunner extends AsyncTask<Integer, Integer, Void>
{


   @Override
   protected void onCancelled(Void aVoid)
   {
      super.onCancelled(aVoid);
      Log.d(TAG, "AsyncTaskRunner has been cancelled :-(");
   }


   @Override
   protected void onPostExecute(Void aVoid)
   {
      super.onPostExecute(aVoid);
      // set the progress bar to full
      mProgressBar.setProgress(PROGRESS_MAX);
      mStopButton.setEnabled(false);
   }
}
```