

Lesson 7.1: AsyncTask

Introduction

A *thread* is an independent path of execution in a running program. When an Android program is launched, the system creates a *main thread*, which is also called the *UI thread*. This UI thread is how your app interacts with components from the Android UI toolkit.

Sometimes an app needs to perform resource-intensive tasks such as downloading files, making database queries, playing media, or computing complex analytics. This type of intensive work can block the UI thread so that the app doesn't respond to user input or draw on the screen. Users may get frustrated and uninstall your app.

To keep the user experience (UX) running smoothly, the Android framework provides a helper class called [AsyncTask](#), which processes work off of the UI thread. Using AsyncTask to move intensive processing onto a separate thread means that the UI thread can stay responsive.

Because the separate thread is not synchronized with the calling thread, it's called an *asynchronous thread*. An AsyncTask also contains a callback that allows you to display the results of the computation back in the UI thread.

In this practical, you learn how to add a background task to your Android app using an AsyncTask.

What you should already know

You should be able to:

- Create an Activity.
- Add a TextView to the layout for the Activity.
- Programmatically get the id for the TextView and set its content.
- Use Button views and their onClick functionality.

*This work is licensed under a [Creative Commons Attribution 4.0 International License](#).
This PDF is a one-time snapshot. See developer.android.com/courses/fundamentals-training/toc-v2
for the latest updates.*

What you'll learn

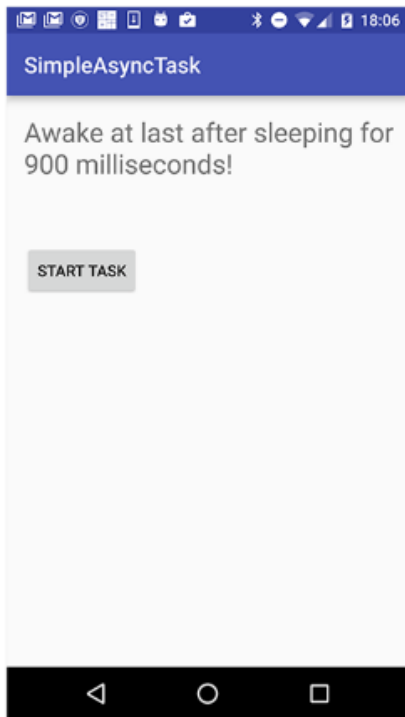
- How to add an AsyncTask to your app in order to run a task in the background of your app.
- The drawbacks of using AsyncTask for background tasks.

What you'll do

- Create a simple app that executes a background task using an AsyncTask.
- Run the app and see what happens when you rotate the device.
- Implement activity instance state to retain the state of a TextView message.

App overview

You will build an app that has one TextView and one Button. When the user clicks the Button, the app sleeps for a random amount of time, and then displays a message in the TextView when it wakes up. Here's what the finished app looks like:



Task 1: Set up the SimpleAsyncTask project

The SimpleAsyncTask UI contains a Button that launches the AsyncTask, and a TextView that displays the status of the app.

1.1 Create the project and layout

1. Create a new project called SimpleAsyncTask using the **Empty Activity** template. Accept the defaults for all other options.
2. Open the `activity_main.xml` layout file. Click the **Text** tab.

*This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).
This PDF is a one-time snapshot. See developer.android.com/courses/fundamentals-training/toc-v2
for the latest updates.*

3. Add the `layout_margin` attribute to the top-level `ConstraintLayout`:

```
android:layout_margin="16dp"
```

4. Add or modify the following attributes of the "Hello World!" `TextView` to have these values. Extract the string into a resource.

Attribute	Value
<code>android:id</code>	<code>"@+id/textView1"</code>
<code>android:text</code>	<code>"I am ready to start work!"</code>
<code>android:textSize</code>	<code>"24sp"</code>

5. Delete the `app:layout_constraintRight_toRightOf` and `app:layout_constraintTop_toTopOf` attributes.
6. Add a `Button` element just under the `TextView`, and give it these attributes. Extract the button text into a string resource.

Attribute	Value
<code>android:id</code>	<code>"@+id/button"</code>
<code>android:layout_width</code>	<code>"wrap_content"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:text</code>	<code>"Start Task"</code>
<code>android:layout_marginTop</code>	<code>"24dp"</code>
<code>android:onClick</code>	<code>"startTask"</code>
<code>app:layout_constraintStart_toStartOf</code>	<code>"parent"</code>
<code>app:layout_constraintTop_toBottomOf</code>	<code>"@+id/textView1"</code>

7. The `onClick` attribute for the button will be highlighted in yellow, because the `startTask()` method is not yet implemented in `MainActivity`. Place your cursor in the highlighted text, press **Alt + Enter** (**Option + Enter** on a Mac) and choose **Create 'startTask(View)' in 'MainActivity'** to create the method stub in `MainActivity`.

Solution code for `activity_main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ready_to_start"
        android:textSize="24sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        android:onClick="startTask"
        android:text="@string/start_task"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView1"/>

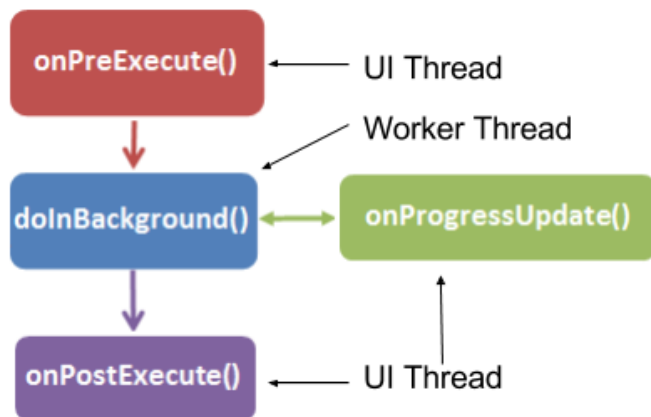
</android.support.constraint.ConstraintLayout>
```

Task 2: Create the AsyncTask subclass

[AsyncTask](#) is an abstract class, which means you must subclass it in order to use it. In this example the AsyncTask performs a very simple background task: it sleeps for a random amount of time. In a real app, the background task could perform all sorts of work, from querying a database, to connecting to the internet, to calculating the next Go move to beat the current Go champion.

An AsyncTask subclass has the following methods for performing work off of the main thread:

- [onPreExecute\(\)](#): This method runs on the UI thread, and is used for setting up your task (like showing a progress bar).
- [doInBackground\(\)](#): This is where you implement the code to execute the work that is to be performed on the separate thread.
- [onProgressUpdate\(\)](#): This is invoked on the UI thread and used for updating progress in the UI (such as filling up a progress bar)
- [onPostExecute\(\)](#): Again on the UI thread, this is used for updating the results to the UI once the AsyncTask has finished loading.



Note: A background or worker thread is any thread which is not the main or UI thread.

When you create an `AsyncTask` subclass, you may need to give it information about the work which it is to perform, whether and how to report its progress, and in what form to return the result.

When you create an `AsyncTask` subclass, you can configure it using these parameters:

- **Params:** The data type of the parameters sent to the task upon executing the `doInBackground()` override method.
- **Progress:** The data type of the progress units published using the `onProgressUpdated()` override method.
- **Result:** The data type of the result delivered by the `onPostExecute()` override method.

For example, an `AsyncTask` subclass called `MyAsyncTask` with the following class declaration might take the following parameters:

- A `String` as a parameter in `doInBackground()`, to use in a query, for example.
- An `Integer` for `onProgressUpdate()`, to represent the percentage of job complete
- A `Bitmap` for the result in `onPostExecute()`, indicating the query result.

```
public class MyAsyncTask
    extends AsyncTask <String, Integer, Bitmap>{}
```

In this task you will use an `AsyncTask` subclass to define work that will run in a different thread than the UI thread.

2.1 Subclass the `AsyncTask`

In this app, the `AsyncTask` subclass you create does not require a query parameter or publish its progress. You will only be using the `doInBackground()` and `onPostExecute()` methods.

1. Create a new Java class called `SimpleAsyncTask` that extends `AsyncTask` and takes three generic type parameters.

Use `Void` for the params, because this `AsyncTask` does not require any inputs. Use `Void` for the progress type, because the progress is not published. Use a `String` as the result type, because you will update the `TextView` with a string when the `AsyncTask` has completed execution.

```
public class SimpleAsyncTask extends AsyncTask <Void, Void, String>{}
```

Note: The class declaration will be underlined in red, because the required method `doInBackground()` has not yet been implemented.

2. At the top of the class, define a member variable `mTextView` of the type `WeakReference<TextView>`:

```
private WeakReference<TextView> mTextView;
```

3. Implement a constructor for `AsyncTask` that takes a `TextView` as a parameter and creates a new weak reference for that `TextView`:

```
SimpleAsyncTask(TextView tv) {  
    mTextView = new WeakReference<>(tv);  
}
```

The `AsyncTask` needs to update the `TextView` in the `Activity` once it has completed sleeping (in the `onPostExecute()` method). The constructor for the class will therefore need a reference to the `TextView` to be updated.

What is the weak reference (the [WeakReference](#) class) for? If you pass a `TextView` into the `AsyncTask` constructor and then store it in a member variable, that reference to the `TextView` means the `Activity` cannot ever be garbage collected and thus leaks memory, even if the `Activity` is destroyed and recreated as in a device configuration change. This is called creating a *leaky context*, and Android Studio will warn you if you try it.

The weak reference prevents the memory leak by allowing the object held by that reference to be garbage collected if necessary.

2.2 Implement doInBackground()

The `doInBackground()` method is required for your `AsyncTask` subclass.

1. Place your cursor on the highlighted class declaration, press **Alt + Enter** (**Option + Enter** on a Mac) and select **Implement methods**. Choose `doInBackground()` and click **OK**. The following method template is added to your class:

```
@Override
protected String doInBackground(Void... voids) {
    return null;
}
```

2. Add code to generate a random integer between 0 and 10. This is the number of milliseconds the task will pause. This is not a lot of time to pause, so multiply that number by 200 to extend that time.

```
Random r = new Random();
int n = r.nextInt(11);

int s = n * 200;
```

3. Add a try/catch block and put the thread to sleep.

```
try {
    Thread.sleep(s);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

4. Replace the existing `return` statement to return the `String` "Awake at last after sleeping for xx milliseconds", where xx is the number of milliseconds the app slept.

```
return "Awake at last after sleeping for " + s + " milliseconds!";
```

The complete `doInBackground()` method looks like this:

```
@Override
protected String doInBackground(Void... voids) {

    // Generate a random number between 0 and 10
    Random r = new Random();
    int n = r.nextInt(11);

    // Make the task take long enough that we have
    // time to rotate the phone while it is running
    int s = n * 200;

    // Sleep for the random amount of time
    try {
        Thread.sleep(s);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // Return a String result
    return "Awake at last after sleeping for " + s + " milliseconds!";
}
```

2.3 Implement `onPostExecute()`

When the `doInBackground()` method completes, the return value is automatically passed to the `onPostExecute()` callback.

1. Implement `onPostExecute()` to take a `String` argument and display that string in the `TextView`:

```
protected void onPostExecute(String result) {  
    mTextView.get().setText(result);  
}
```

The `String` parameter to this method is what you defined in the third parameter of your `AsyncTask` class definition, and what your `doInBackground()` method returns.

Because `mTextView` is a weak reference, you have to deference it with the `get()` method to get the underlying `TextView` object, and to call `setText()` on it.

Note: You can update the UI in `onPostExecute()` because that method is run on the main thread. You cannot update the `TextView` with the new string in the `doInBackground()` method, because that method is executed on a separate thread.

Task 3: Implement the final steps

3.1 Implement the method that starts the `AsyncTask`

Your app now has an `AsyncTask` class that performs work in the background (or it would if it didn't call `sleep()` as the simulated work). You can now implement the `onClick` method for the "Start Task" button to trigger the background task.

1. In the `MainActivity.java` file, add a member variable to store the `TextView`.

```
private TextView mTextView;
```

2. In the `onCreate()` method, initialize `mTextView` to the `TextView` in the layout.

```
mTextView = findViewById(R.id.textView1);
```

3. In the `startTask()` method, Update the `TextView` to show the text "Napping...". Extract that message into a string resource.

```
mTextView.setText(R.string.napping);
```

4. Create an instance of `SimpleAsyncTask`, passing the `TextView` `mTextView` to the constructor. Call `execute()` on that `SimpleAsyncTask` instance.

```
new SimpleAsyncTask(mTextView).execute();
```

Note: The `execute()` method is where you pass comma-separated parameters that are then passed into `doInBackground()` by the system. Because this `AsyncTask` has no parameters, you leave it blank.

Solution code for `MainActivity`:

```
package com.example.android.simpleasynctask;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

/**
 * The SimpleAsyncTask app contains a button that launches an AsyncTask
 */
```

```
* which sleeps in the asynchronous thread for a random amount of time.
*/
public class MainActivity extends AppCompatActivity {

    // The TextView where we will show results
    private TextView mTextView;

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

        mTextView = findViewById(R.id.textView1);
    }

    public void startTask(View view) {
        // Put a message in the text view
        mTextView.setText(R.string.napping);

        // Start the AsyncTask.
        new SimpleAsyncTask(mTextView).execute();
    }
}
```

3.2 Implement onSaveInstanceState()

1. Run the app and click the **Start Task** button. How long does the app nap?
2. Click the **Start Task** button again, and while the app is napping, rotate the device. If the background task completes before you can rotate the phone, try again.

Note: You'll notice that when the device is rotated, the TextView resets to its initial content, and the AsyncTask does not seem able to update the TextView.

There are several things going on here:

*This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).
This PDF is a one-time snapshot. See developer.android.com/courses/fundamentals-training/toc-v2
for the latest updates.*

- When you rotate the device, the system restarts the app, calling `onDestroy()` and then `onCreate()`. The `AsyncTask` will continue running even if the activity is destroyed, but it will lose the ability to report back to the activity's UI. It will never be able to update the `TextView` that was passed to it, because that particular `TextView` has also been destroyed.
- Once the activity is destroyed the `AsyncTask` will continue running to completion in the background, consuming system resources. Eventually, the system will run out of resources, and the `AsyncTask` will fail.
- Even without the `AsyncTask`, the rotation of the device resets all of the UI elements to their default state, which for the `TextView` is the default string that you set in the layout file.

For these reasons, an `AsyncTask` is not well suited to tasks which may be interrupted by the destruction of the Activity. In use cases where this is critical you can use a different type of background class called an `AsyncTaskLoader` which you will learn about in a later practical.

In order to prevent the `TextView` from resetting to the initial string, you need to save its state. You will now implement `onSaveInstanceState()` to preserve the content of your `TextView` when the activity is destroyed in response to a configuration change such as device rotation.

Note: Not all uses of `AsyncTask` require you to handle the state of the views on rotation. This app uses a `TextView` to display the results of the app, so preserving the state is useful. In other cases, such as uploading a file, you may not need any persistent information in the UI, so retaining the state is not critical.

3. At the top of the class, add a constant for the key for the current text in the state bundle:

```
private static final String TEXT_STATE = "currentText";
```

4. Override the `onSaveInstanceState()` method in **MainActivity** to preserve the text inside the `TextView` when the activity is destroyed:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Save the state of the TextView
```

```
        outState.putString(TEXT_STATE,
            mTextView.getText().toString());
    }
}
```

5. In `onCreate()`, retrieve the value of the `TextView` from the state bundle when the activity is restored.

```
// Restore TextView if there is a savedInstanceState
if(savedInstanceState!=null){
    mTextView.setText(savedInstanceState.getString(TEXT_STATE));
}
```

Solution code for MainActivity:

```
package android.example.com.simpleasynctask;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.TextView;

/**
 * The SimpleAsyncTask app contains a button that launches an AsyncTask
 * which sleeps in the asynchronous thread for a random amount of time.
 */
public class MainActivity extends AppCompatActivity {

    //Key for saving the state of the TextView
    private static final String TEXT_STATE = "currentText";

    // The TextView where we will show results
    private TextView mTextView = null;

    /**
```

```
* Initializes the activity.
* @param savedInstanceState The current state data
*/
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // Initialize mTextView
    mTextView = (TextView) findViewById(R.id.textview1);

    // Restore TextView if there is a savedInstanceState
    if(savedInstanceState!=null){
        mTextView.setText(savedInstanceState.getString(TEXT_STATE));
    }
}

/**
 * Handles the onClick for the "Start Task" button. Launches the
 * AsyncTask
 * which performs work off of the UI thread.
 *
 * @param view The view (Button) that was clicked.
 */
public void startTask (View view) {
    // Put a message in the text view
    mTextView.setText(R.string.napping);

    // Start the AsyncTask.
    // The AsyncTask has a callback that will update the text view.
    new SimpleAsyncTask(mTextView).execute();
}

/**
 * Saves the contents of the TextView to restore on configuration
 * change.
 * @param outState The bundle in which the state of the activity is
 * saved when it is spontaneously destroyed.
 */
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Save the state of the TextView
```