Mobile and Ubiquitous Computing

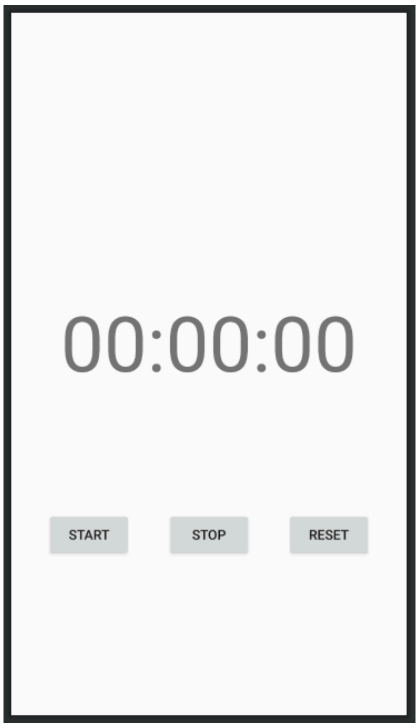
Labsheet 6 – Bundles and Threads

In this labsheet we're going to be going into a little more detail on how activities are managed and we're going to have a little thing about threading and how we can do some work off the main UI thread. This week is a little more hands-off; I'll give you some instructions, but I am hoping that you will need less step-by-step guidance now we're halfway through the labsheets.

Activities, threads, bundles

In this labsheet we're going to develop a stopclock application. Create a new project with an Empty Activity. It's going to consist of a single activity and a single layout. Within the layout, change the `textSize` of the `TextView` that's already there to 80sp, using either the text or design view. Set the default text of this `TextView` to **00:00:00**.

Below this, you need three buttons: one to **start** the stop-clock, one to **stop** the stop-clock, and one to **reset** the timer value to zero. Lay these buttons out however you'd like. Here's how I have done it:



Activity Overview

Each of the three buttons will control the stop-clock in a given manner, through their respective `onClick` handlers (`timerStart`, `timerStop`, `timerReset`). We'll update the timer itself through use of a method that we'll define called `runTimer()`. This should run every second to check whether the stop-clock is running, increment the number of seconds, and display this in the text view. The state of the stopclock will be recorded in two private variables, one to record the `seconds` that have passed since the stop-clock started, and one called `running` to record whether the stop-clock is running or not.

When the user clicks the Start button, `running` should be set to `true` so the stop-clock will start appropriately. When the user clicks on the Stop button, `running` should be set to `false`. If we reset the stop-clock by clicking the Reset button, should `running` be `true` or `false`? Also when we click reset, what value should seconds be set to?

Code all of this into your `MainActivity.java` class file. Remember, all of the `onClick` methods that you create must take a `View` object as an argument, and return nothing.

The trickier method here now is the `runTimer()` method. In this, we need to access the TextView (`timer`), and format the contents of `seconds` into hours, minutes and seconds, and then display the result. We need this code to keep looping so that it increments the seconds variable and updates the text view every second. We need to do this in such a way that the main Android thread isn't blocked; this would make the application completely unresponsive while it was waiting. We can do this by using a background thread. However, with Android, *only* the main thread can update the UI (so our thread could do the counting of each second, but not update the displayed time). To overcome this, we need to us a `Handler` object.

Handlers

The `Handler` class in Android allows you to schedule code that should be run at some point in the future. You can also use it to post code that needs to run on a different thread. For us, the `Handler` can schedule the stop-clock code to run every second.

To use a handler, we have to wrap the code that we want to run – the seconds to hour, minutes and seconds conversion – in a `Runnable` object, in its `run()` method, and then use the `Handler`'s `post()` and `postDelayed()` methods to specify when you want the code to run. The `post()` method posts code that needs to be run as soon as possible. It usually takes a single argument, an object of type `Runnable`, which contains a job you want to run, put in the Runnable object's `run()` method.

The `postDelayed()` method is very similar to the `post()` method, except that you refer to code you wish to run at a certain point in the future. It takes two parameters, a `Runnable` and a `long`. The long is the number of milliseconds you wish to delay the code by. You'll need to start this method with statement to find your `TextView` and another to create your `Handler` instance. They need to be tagged `final` otherwise Android will refuse to allow cross-thread changes.

```java
final TextView timer = (TextView) findViewById(R.id.timer);
final Handler thread = new Handler();
```

Next, we want some code to run. Type `thread.post(new Runnab`… and you should find that Android Studio offers an autocomplete for what you're doing. Accept the autocomplete suggestion and it will create a new `run()` method for you.

You're going to need something that converts the seconds that you're counting into a stopclock-style display. This requires using the modulus and some lateral thinking. It's not the focus of this lab and I don't want you to get bogged down in it, so I'm happy just to give you the code to do it:

```
int hours = seconds/360;
int minutes = (seconds%3600)/60;
int secs = seconds%60;
String time = String.format("%d:%02d:%02d",hours, minutes, seconds);
timer.setText(time);
```

You need to add some code in the method that will increment the number of seconds each time this runs depending on whether the stopclock is running or not. Once you've done that you need one more line of code that is going to reuse our handler to re-run our current handler (`this`) 1000-ms from now:

```
thread.postDelayed(this,1000);
```

Once you've put all of this together you should have a functioning stopclock! If not, double check your onClick handlers for your buttons and make sure you have something incrementing the number of seconds.

Bundles and state management

Try rotating the stopclock app you've just created as it's running. What do you notice? It seems to reset the screen to 0:00:00 whenever you rotate it, doesn't it? When the screen rotates, Android sees this, and quite aggressively destroys the activity, plus any variables currently being used by the `runTimer()` method. After rotating, the `onCreate()` method is rerun, and `runTimer()` gets called once again, essentially recreating the stopclock activity. We need to save the current state of the activity so that it survives this destruction/recreation. How do we do this?

To save the current state of the activity we need to implement the `onSaveInstanceState()` method in our activity class (i.e., MainActivity). This always gets called before the activity gets destroyed, which means you get an opportunity to save any values you want to retain before they disappear forever.

`onSaveInstanceState()` takes one parameter, a `Bundle`. This allows you to gather together different types of data into a single object. What data should we save?

```
public void onSaveInstanceState(Bundle savedInstanceState)
```

Of course, the running state and the number of seconds that have passed.

```
    savedInstanceState.putInt(“seconds”, seconds);
    savedInstanceState.putBoolean(“running”, running);
```

Now, we also need to retrieve these values when the app is created. Conveniently, the `onCreate()` method is passed a `Bundle` parameter by Android! This is part of the boilerplate code that turns up in all the applications you've created so far but that you might not have looked at yet. See if you can get those values back from this using appropriate getter methods (instead of 'put' think 'get'). **Tip:** check first that the Bundle object is not null. If it is null, nothing will be loaded!

Extension

If you've managed to get through all this, you'll notice that our stopclock is not terribly precise. The state is updated a maximum of once per second, which means it doesn't always seem to immediately respond and it can feel a bit slow stopping. See if you can refactor this code to get it working more expeditiously.

Tests

This lab sheet wouldn't be complete unless I gave you some tests to to check your code against. First of all we just want to check that you have the right views and that they are visible in the activity. This is a simple test that you should be familiar with, but note you might need to have a go at refactoring your View ids to get these to pass/compile for you:

```
@Rule
public ActivityScenarioRule act = new
ActivityScenarioRule<>(MainActivity.class);

@Test
public void checkViews(){
    onView(withId(R.id.start)).check(matches(isDisplayed()));
    onView(withId(R.id.stop)).check(matches(isDisplayed()));
    onView(withId(R.id.stop)).check(matches(isDisplayed()));
    onView(withId(R.id.timer)).check(matches(isDisplayed()));
}
```

Let's test some of the more complex functionality of the application now. The test below starts the clock, lets it run for a few seconds and clicks the stop button. It gives things a second the 'settle' (because our clock is not very accurate and could keep running after being stopped). Then, the current time shown is recorded, and we test to see if the time has actually moved on from the default (i.e., checks that it is *not* 00:00:00).

We then wait a couple of seconds and check that the time currently shown still matches the time we recorded after stopping – this tells us whether the stop button is actually stopping the values on the clock changing. We then start the clock again, and after five seconds check that the time has moved on again (i.e., that pressing start after stopping causes the clock to resume). We then hit reset and after a brief pause to let the UI catch up, we check that the clock has indeed been reset.

```
@Test
public void startStopResetTimer(){
    onView(withId(R.id.start)).perform(click());
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {}

    onView(withId(R.id.stop)).perform(click());
    try {
        //Give the clock time to settle
        Thread.sleep(1000);
    } catch (InterruptedException e) {}

    String timeNow = getText(withId(R.id.timer));
    onView(withId(R.id.timer)).check(matches(not(withText("00:00:00"))));

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {}

    onView(withId(R.id.timer)).check(matches(withText(timeNow)));
    onView(withId(R.id.start)).perform(click());

    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {}

    onView(withId(R.id.timer)).check(matches(not(withText(timeNow))));
    onView(withId(R.id.reset)).perform(click());

    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}

    onView(withId(R.id.timer)).check(matches(withText("00:00:00")));
}
```

Note that the `getText()` method is not built in but is some code I have borrowed:

```
// Thanks StackOverflow for this!
// https://stackoverflow.com/questions/23381459/how-to-get-text-from-textview-using-
espresso
String getText(final Matcher<View> matcher) {
    final String[] stringHolder = { null };
    onView(matcher).perform(new ViewAction() {
        @Override
        public Matcher<View> getConstraints() {
            return isAssignableFrom(TextView.class);
        }

        @Override
        public String getDescription() {
            return "getting text from a TextView";
        }

        @Override
        public void perform(UiController uiController, View view) {
            TextView tv = (TextView)view; //Save, because of check in
getConstraints()
            stringHolder[0] = tv.getText().toString();
        }
    });
    return stringHolder[0];
}
```

Funnily enough I spent ages debugging this test because it kept failing… turns out the original code was wrong and even though it started off as 00:00:00 it was instantly being set to 0:00:00 instead. This was causing the test to fail. Testing *does work*, and this is not a bug I'd have noticed otherwise!

Note that calling `Thread.sleep()` is generally *not* a desirable way to ask Android to wait for something to happen. It is 'hacky' at best in most place. There are idling resources and other techniques for waiting for things like animations to complete or network calls to return. The time for these kinds of actions to complete might vary significantly with the device and other conditions and so are likely to fail in a non-deterministic way. I don't want to introduce these concepts this week though, so we'll just stick to the 'bad' `Thread.sleep()` approach this week.