46. A Basic Overview of Threads and Thread Handlers

The next chapter will be the first in a series of chapters intended to introduce the use of Android Services to perform application tasks in the background. It is impossible, however, to understand the steps involved in implementing services without first gaining a basic understanding of the concept of threading in Android applications. Threads and thread handlers are, therefore, the topic of this chapter.

46.1 An Overview of Threads

Threads are the cornerstone of any multitasking operating system and can be thought of as mini-processes running within a main process, the purpose of which is to enable at least the appearance of parallel execution paths within applications.

46.2 The Application Main Thread

When an Android application is first started, the runtime system creates a single thread in which all application components will run by default. This thread is generally referred to as the *main thread*. The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface. Any additional components that are started within the application will, by default, also run on the main thread.

Any component within an application that performs a time consuming task using the main thread will cause the entire application to appear to lock up until the task is completed. This will typically result in the operating system displaying an "Application is not responding" warning to the user. Clearly, this is far from the desired behavior for any application. This can be avoided simply by launching the task to be performed in a separate thread, allowing the main thread to continue unhindered with other tasks.

46.3 Thread Handlers

Clearly, one of the key rules of Android development is to never perform time-consuming operations on the main thread of an application. The second, equally important, rule is that the code within a separate thread must never, under any circumstances, directly update any aspect of the user interface. Any changes to the user interface must always be performed from within the main thread. The reason for this is that the Android UI toolkit is not *thread-safe*. Attempts to work with non-thread-safe code from within multiple threads will typically result in intermittent problems and unpredictable application behavior.

In the event that the code executing in a thread needs to interact with the user interface, it must do so by *synchronizing* with the main UI thread. This is achieved by creating a *handler* within the main thread, which, in turn, receives messages from another thread and updates the user interface accordingly.

46.4 A Basic Threading Example

The remainder of this chapter will work through some simple examples intended to provide a basic

introduction to threads. The first step will be to highlight the importance of performing time-consuming tasks in a separate thread from the main thread. Begin, therefore, by creating a new project in Android Studio, entering *ThreadExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *ThreadExampleActivity*, using the default for the layout resource files.

Click *Finish* to create the new project.

Load the *activity_thread_example.xml* file for the project into the Layout Editor tool. Select the default TextView component and change the ID for the view to *myTextView* in the Properties tool window.

With autoconnect mode disabled, add a Button view to the user interface, positioned directly beneath the existing TextView object as illustrated in Figure 46-1. Once the button has been added, click on the Infer Constraints button in the toolbar to add the missing constraints.

Change the text to “Press Me” and extract the string to a resource named *press_me*. With the button view still selected in the layout change the layout_width property to *wrap_content* in the Properties tool window, locate the *onClick* property and enter *buttonClick* as the method name.

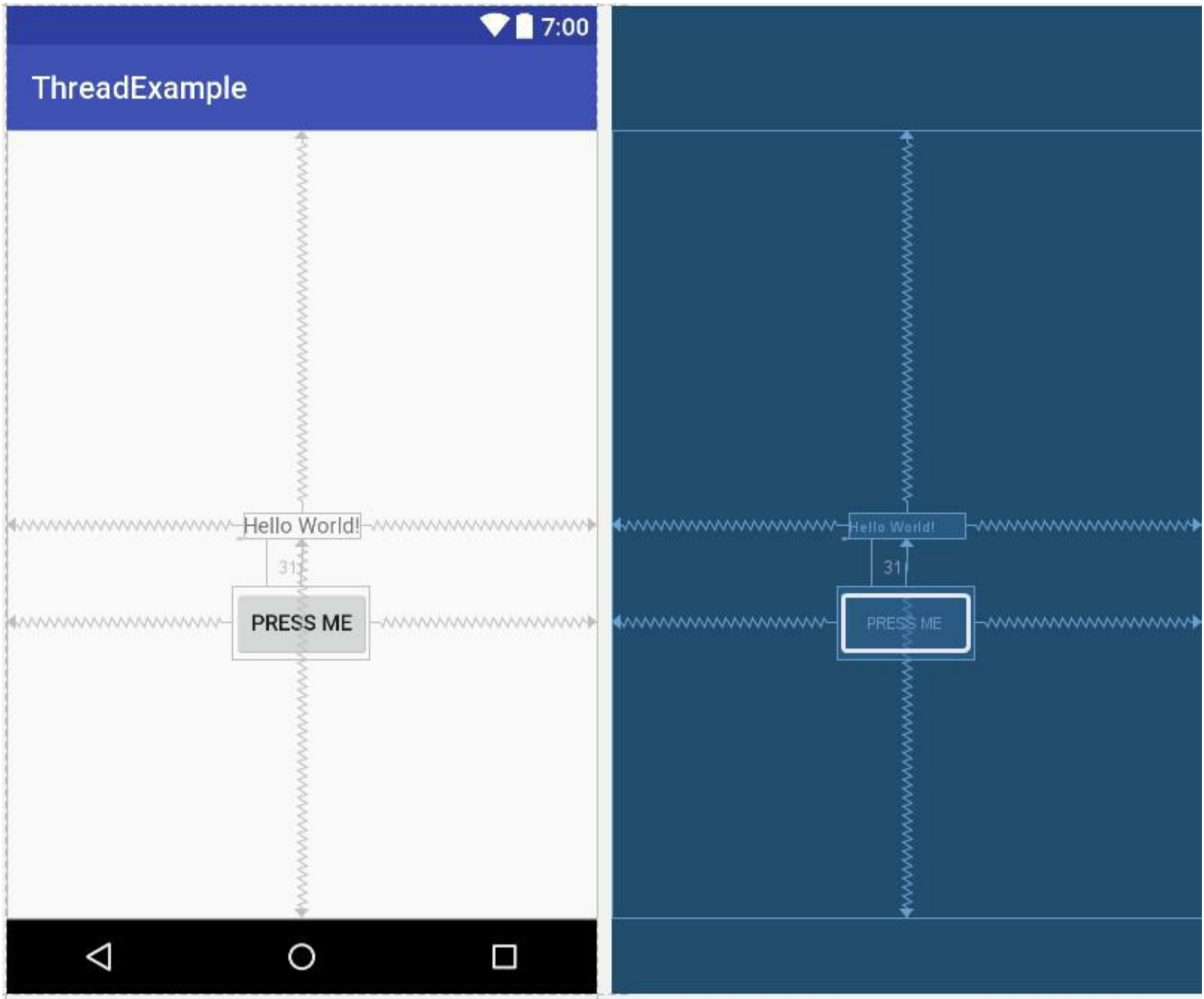

Figure 46-1

Next, load the *ThreadExampleActivity.java* file into an editing panel and add code to implement the *buttonClick()* method which will be called when the Button view is touched by the user. Since the goal here is to demonstrate the problem of performing lengthy tasks on the main thread, the code will simply pause for 20 seconds before displaying different text on the TextView object:

```
package com.ebookfrenzy.threadexample;

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

public class ThreadExampleActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_thread_example);
```

```
    }

    public void buttonClick(View view)
    {
        long endTime = System.currentTimeMillis() + 20*1000;

        while (System.currentTimeMillis() < endTime) {
            synchronized (this) {
                try {
                    wait(endTime - System.currentTimeMillis());
                } catch (Exception e) {
                }
            }
        }
        TextView myTextView =
                (TextView)findViewById(R.id.myTextView);
        myTextView.setText("Button Pressed");
    }
}
```

With the code changes complete, run the application on either a physical device or an emulator. Once the application is running, touch the Button, at which point the application will appear to freeze. It will, for example, not be possible to touch the button a second time and in some situations the operating system will, as demonstrated in Figure 46-2, report the application as being unresponsive:

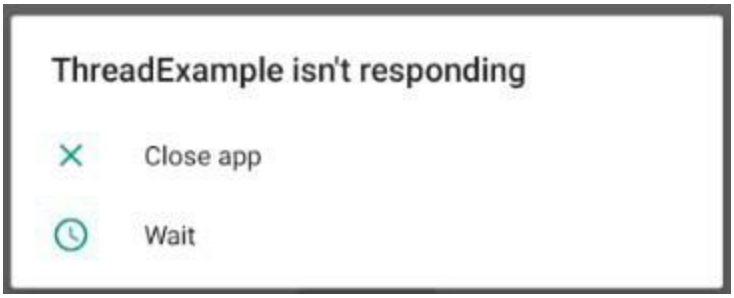

Figure 46-2

Clearly, anything that is going to take time to complete within the *buttonClick()* method needs to be performed within a separate thread.

46.5 Creating a New Thread

In order to create a new thread, the code to be executed in that thread needs to be placed within the *Run()* method of a *Runnable* instance. A new Thread object then needs to be created, passing through a reference to the Runnable instance to the constructor. Finally, the *start()* method of the thread object needs to be called to start the thread running. To perform the task within the *buttonClick()* method, therefore, the following changes need to be made:

```
public void buttonClick(View view)
{
        Runnable runnable = new Runnable() {
                public void run() {

                        long endTime = System.currentTimeMillis()
                                        + 20*1000;

                        while (System.currentTimeMillis() < endTime) {
                                synchronized (this) {
```

```
                                try {
                                        wait(endTime -
                                           System.currentTimeMillis());
                                } catch (Exception e) {}
                            }
                        }
                    }
            };
            Thread myThread = new Thread(runnable);
            myThread.start();
    }
```

When the application is now run, touching the button causes the delay to be performed in a new thread leaving the main thread to continue handling the user interface, including responding to additional button presses. In fact, each time the button is touched, a new thread will be created, allowing the task to be performed multiple times concurrently.

A close inspection of the updated code for the *buttonClick()* method will reveal that the code to update the TextView has been removed. As previously stated, updating a user interface element from within a thread other than the main thread violates a key rule of Android development. In order to update the user interface, therefore, it will be necessary to implement a Handler for the thread.

46.6 Implementing a Thread Handler

Thread handlers are implemented in the main thread of an application and are primarily used to make updates to the user interface in response to messages sent by other threads running within the application's process.

Handlers are subclassed from the Android *Handler* class and can be used either by specifying a Runnable to be executed when required by the thread, or by overriding the *handleMessage()* callback method within the Handler subclass which will be called when messages are sent to the handler by a thread.

For the purposes of this example, a handler will be implemented to update the user interface from within the previously created thread. Load the *ThreadExampleActivity.java* file into the Android Studio editor and modify the code to add a Handler instance to the activity:

```
package com.ebookfrenzy.threadexample;

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

public class ThreadExampleActivity extends AppCompatActivity {

    Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            TextView myTextView =
                    (TextView)findViewById(R.id.myTextView);
            myTextView.setText("Button Pressed");
        }
```

```
    };

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

The above code changes have declared a handler and implemented within that handler the *handleMessage()* callback which will be called when the thread sends the handler a message. In this instance, the code simply displays a string on the TextView object in the user interface.

All that now remains is to modify the thread created in the *buttonClick()* method to send a message to the handler when the delay has completed:

```
public void buttonClick(View view)
{

    Runnable runnable = new Runnable() {
            public void run() {

                long endTime = System.currentTimeMillis() +
                                                    20*1000;

                while (System.currentTimeMillis() < endTime) {
                    synchronized (this) {
                      try {
                            wait(endTime -
                              System.currentTimeMillis());
                      } catch (Exception e) {}
                }

            }
            handler.sendEmptyMessage(0);
        }
    };

    Thread myThread = new Thread(runnable);
    myThread.start();

}
```

Note that the only change that has been made is to make a call to the *sendEmptyMessage()* method of the handler. Since the handler does not currently do anything with the content of any messages it receives it is sent an empty message object.

Compile and run the application and, once executing, touch the button. After a 20 second delay, the new text will appear in the TextView object in the user interface.

46.7 Passing a Message to the Handler

While the previous example triggered a call to the *handleMessage()* handler callback, it did not take

advantage of the message object to send data to the handler. In this phase of the tutorial, the example will be further modified to pass data between the thread and the handler. First, the updated thread in the *buttonClick()* method will obtain the date and time from the system in string format and store that information in a Bundle object. A call will then be made to the *obtainMessage()* method of the handler object to get a message object from the message pool. Finally, the bundle will be added to the message object before being sent via a call to the *sendMessage()* method of the handler object:

```
package com.ebookfrenzy.threadexample;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.os.Handler;
import android.os.Message;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class ThreadExampleActivity extends AppCompatActivity {
.
.
.
    public void buttonClick(View view)
    {
        Runnable runnable = new Runnable() {
            public void run() {
                Message msg = handler.obtainMessage();
                Bundle bundle = new Bundle();
                SimpleDateFormat dateformat =
                        new SimpleDateFormat("HH:mm:ss MM/dd/yyyy",
                                Locale.US);
                String dateString =
                        dateformat.format(new Date());
                bundle.putString("myKey", dateString);
                msg.setData(bundle);
                handler.sendMessage(msg);
            }
        };
        Thread myThread = new Thread(runnable);
        myThread.start();

    }
}
```

Next, update the *handleMessage()* method of the handler to extract the date and time string from the bundle object in the message and display it on the TextView object:

```
Handler handler = new Handler() {
      @Override
      public void handleMessage(Message msg) {
              Bundle bundle = msg.getData();
              String string = bundle.getString("myKey");
              TextView myTextView =
                      (TextView)findViewById(R.id.myTextView);
              myTextView.setText(string);
```

```
                }
    };
```

Finally, compile and run the application and test that touching the button now causes the current date and time to appear on the TextView object.

46.8 **Summary**

The goal of this chapter was to provide an overview of threading within Android applications. When an application is first launched in a process, the runtime system creates a *main thread* in which all subsequently launched application components run by default. The primary role of the main thread is to handle the user interface, so any time consuming tasks performed in that thread will give the appearance that the application has locked up. It is essential, therefore, that tasks likely to take time to complete be started in a separate thread.

Because the Android user interface toolkit is not thread-safe, changes to the user interface should not be made in any thread other than the main thread. User interface changes can be implemented by creating a handler in the main thread to which messages may be sent from within other, non-main threads.