Poul Klausen

JAVA 16

Mobile phones and Android

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 16: MOBIL PHONES AND ANDROID

SOFTWARE DEVELOPMENT

Java 16: Mobil phones and Android: Software Development
1st edition

ISBN 978-87-403-2330-6
Peer review by Ove Thomsen, EA Dania

CONTENTS

	Foreword	**7**
1	**Introduction**	**9**
	Exercise 1: Calculations	22
2	**Widgets**	**24**
2.1	A Zipcode app	27
	Exercise 2: People	35
3	**Layout**	**37**
3.1	RelativeLayout	37
3.2	LinearLayout	47
3.3	TableLayout	51
3.4	GridLayout	51
3.5	A ConstraintLayout	57
3.6	Containers	60
	Exercise 3: People1	73

4	**User interaction**	**74**
4.1	A context menu	76
4.2	A main menu	82
4.3	Define a menu in XML	87
	Exercice 4: ImageApp	88
4.4	About activities	90
4.5	Multiple activities	94
	Problem 1: LoanApp	96
5	**Files**	**99**
5.1	Reading resources	99
	Exercise 5: Zipcodes	103
5.2	Ordinary files	103
5.3	Internal storage	104
	Exercise 6: NameBook	117
	Problem 2: PuzzleApp	119
6	**SQLite**	**121**
6.1	Creates a Database	124
6.2	SQL SELECT	128
6.3	SQL INSERT, UPDATE and DELETE	132
6.4	The other activities	137
	Exercise 7: PhoneBook	137
7	**Threads**	**141**
8	**Services**	**149**
8.1	The class Intent	153
8.2	ServiceApp1	154
8.3	ServiceApp2	158
8.4	ServiceApp3	161
8.5	ServiceApp4	164
8.6	ServiceApp5	169
8.7	Ring the phone	172
8.8	A local service – ServiceApp	176
8.9	Services and SQLite	184

9 A final example 196
9.1 Project start and MainActivity 196
9.2 The data access layer 198
9.3 Notes 199
9.4 Anniversaries 201
9.5 Appointments 202
9.6 The other tools 202
9.7 A project review 206

Appendix A, installing of Android Studio 207

FOREWORD

The current book is the sixteenth in this series of books on software development in Java, and it is at the same time the first of two books that focus on developing applications for Android, and especially for mobile phone applications. The books requires basic knowledge of Java and system development in general, and the goal is that the reader, after reading this book, is able to write simple applications to an Android phone. Important topics are the most common widgets, layouts and basically about activities, but also use of files and databases are processed. Mobile phone programming is in many ways different from programming to a usual computer, for example because the phone is another machine than a traditional computer, and the phone and, for that matter, other Android devices like tablets have other features that are not available on a PC. However, these features are dealt with in the following book.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer

technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

1. etBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
3. GlassFish as a web server and application server (from the book Java 11 onwards)
4. Android Studio, that is a tool to development of apps to mobil phones

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not "how to write" or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

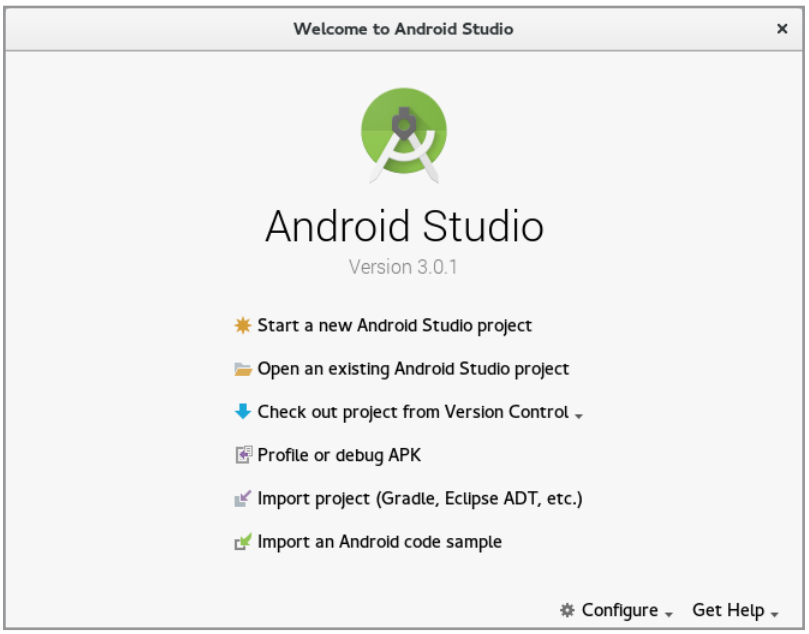
1 INTRODUCTION

In short, this book is about programming mobile phones, and although it is in principle not so different from writing programs to a regular PC, there are also significant differences. Firstly, the programs are written in Java so that side of the case should be in place, but what makes the programmming different are differences in the hardware.

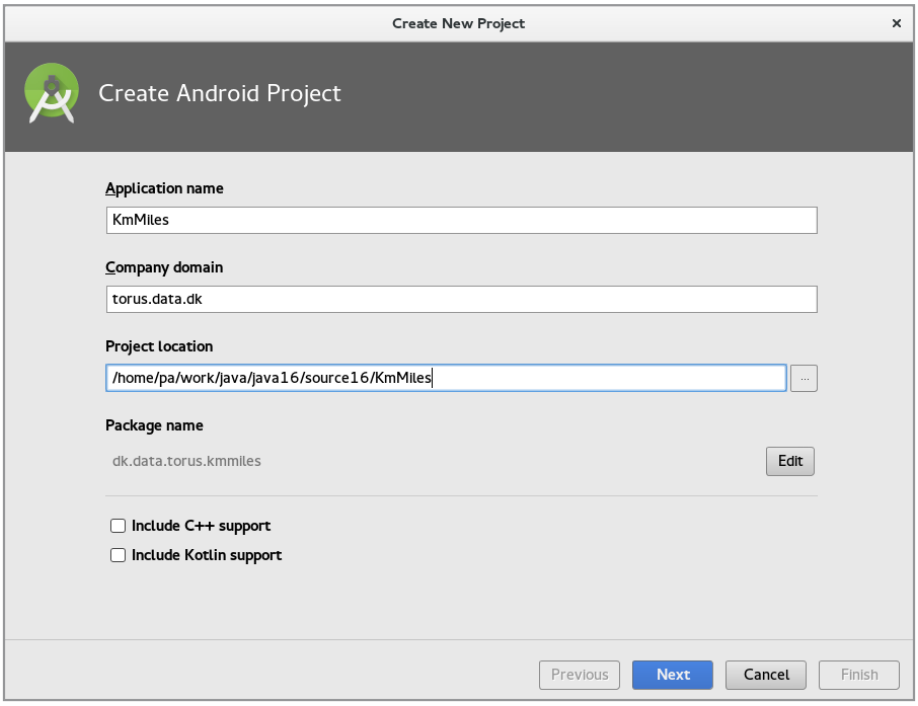
Android is used as an operating system on a variety of devices such as mobile phones, tablets, smart TV, and so on, and although the starting point for this book is a smart phone, they are also widely different with different screen sizes and other options. Furthermore, the development of the mobile phone market is so fast that writing programs for the latest mobile phones you can not be sure they are supported by only a few phones. Thus, there are a whole host of new factors to consider when writing applications for mobile phones.

As a development tool, for the rest of this book, I will use *Android Studio*, an IDE from Google. Appendix A explains how to download and install the product and what it takes to set up an *Android* application development environment. In the following, I would like to assume that *Android Studio* is installed and in the rest of this section, I will show you how to write a simple application that can convert kilometers to miles and the other way, convert miles to kilometers, and it is so a little more than *Hello World*. I want to focus on what you should do, but not so much about how it all works, which I want to postpone for later.

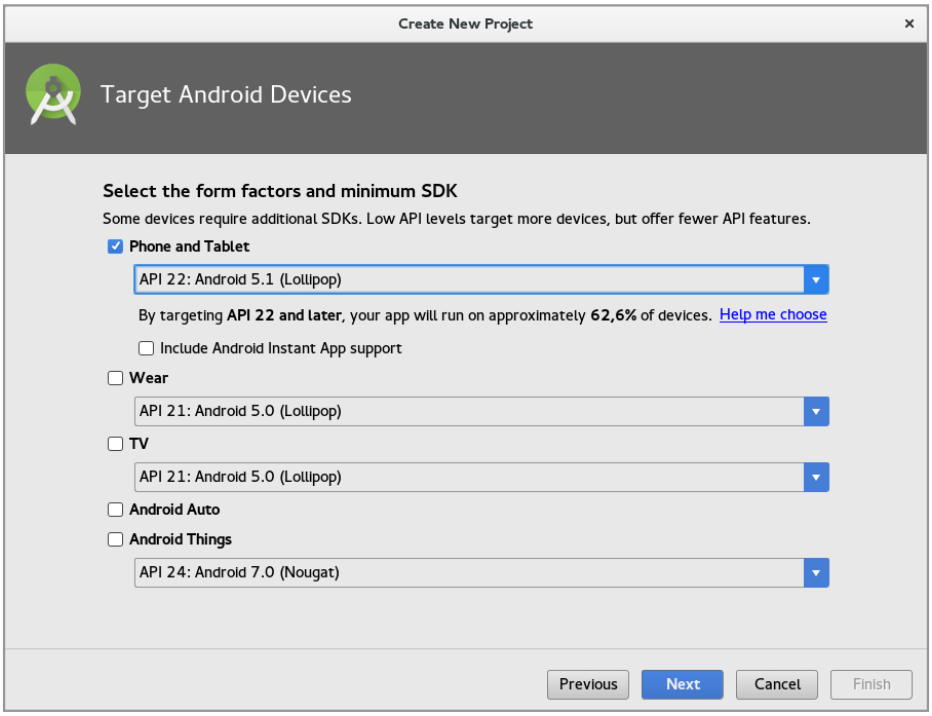
I start by opening Android Studio. Is it the first time you opens the program you get the following window:



Here I click on the top link *Start a new Android Studio project*, and the result is the following window where I have to enter the project name (here *KmMiles*), and where in the file system that the project should be created:



I have also entered a *Company domain* name, which is usually a web address, and the name is included in the project's package name. When you click Next, you get a window:

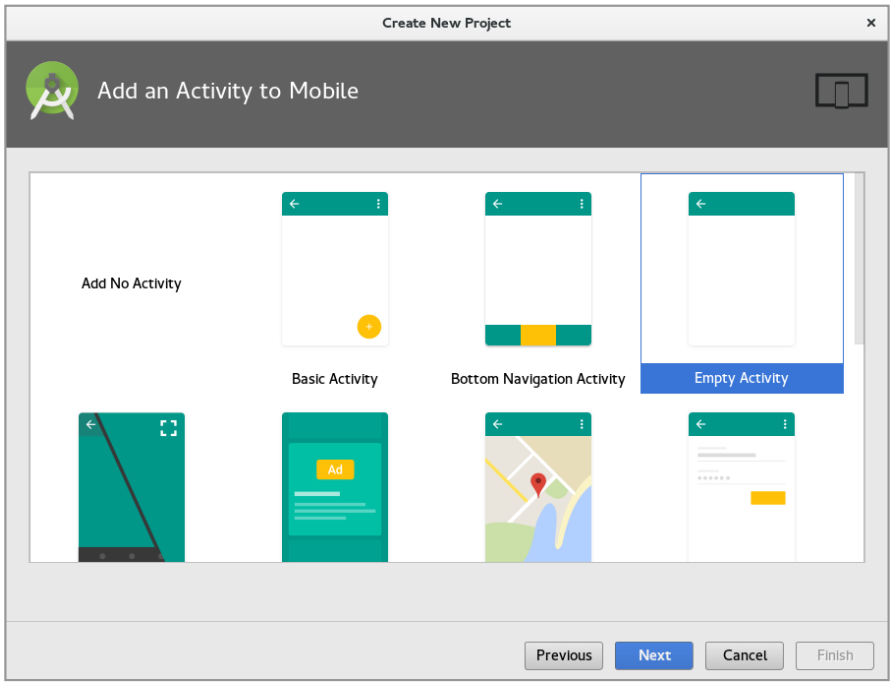


where to choose what it is for an application that you want to create and what version of the operating system you want to support. I have chosen that the application should be

Phone and Tablet

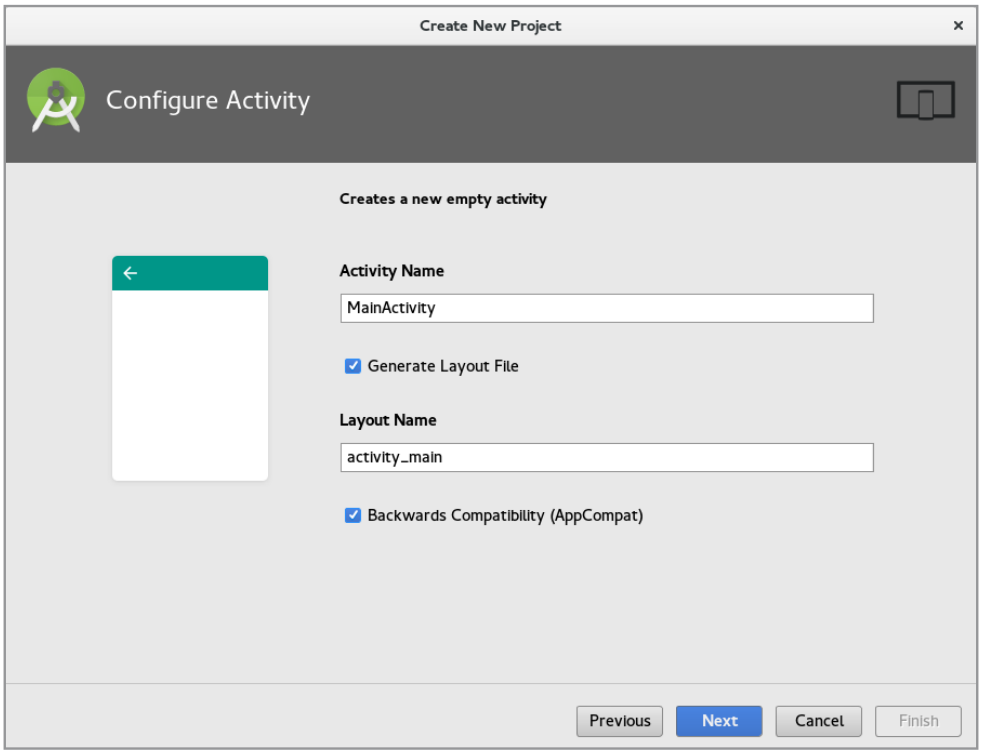
and that it should support API 22 for Android 5.1. You should note that Android Studio writes that I can expect my application to run on 62.6 percent of all devices. Instead, if I had chosen the latest version, I would like to know that my application could run on less than 1 percent of all devices.

In the next window, you must select the *activity* for the phone that corresponds to the overall layout of the screen and what elements should be on it:

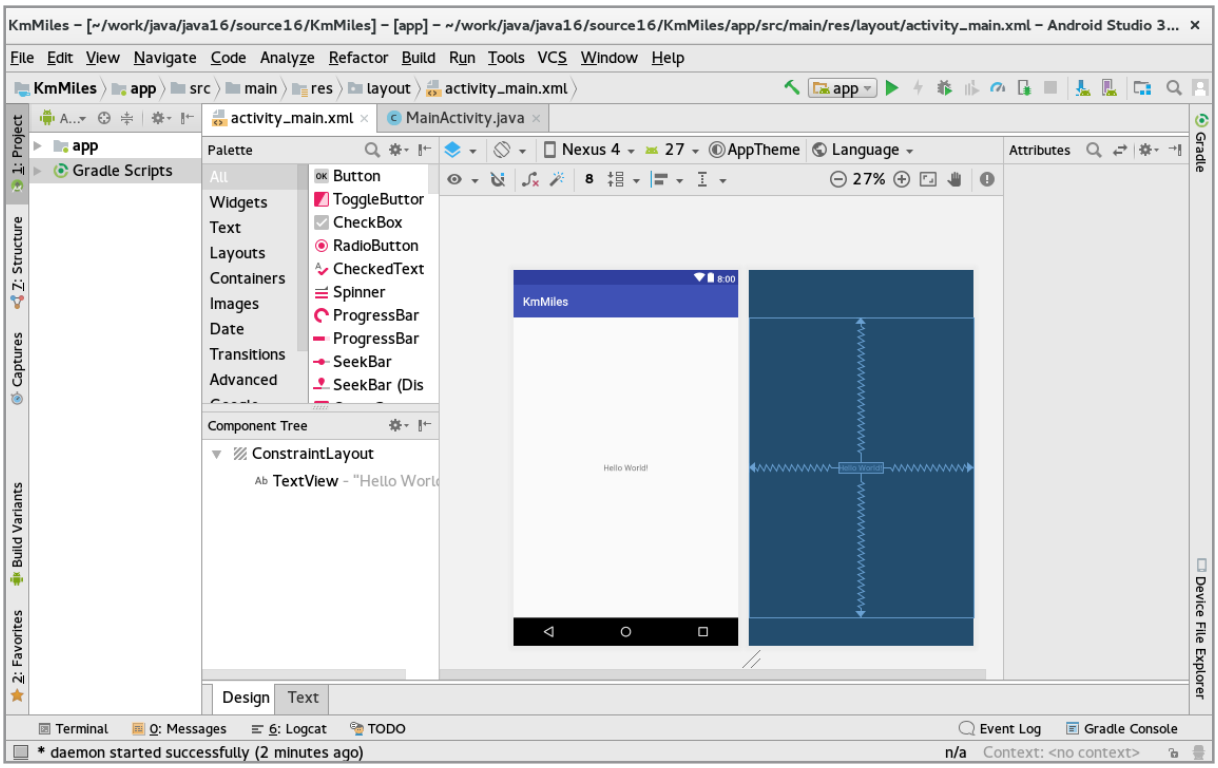


In this case, I have chosen *Empty Activity*, which is default. In the next window again, select the name for that activity, which becomes a class name, and you must also choose the name for the layout. There is no special reasons for choosing other names than what Android Studio has done as default, so I keep the names. When you click *Next*, you get a window where Android Studio builds the project and you then have to click *Finish*.

Is it the first time you run the program, Android Studio ask you to download more files, what you should do, and after Android Studio opening the window with the program's code, you may also need to click on a few download links, but then the project has been created and ready and you can start programming.


Create New Project
Configure Activity
Creates a new empty activity
Activity Name
MainActivity
Generate Layout File
Layout Name
activity_main
Backwards Compatibility (AppCompat)
Previous
Next
Cancel
Finish

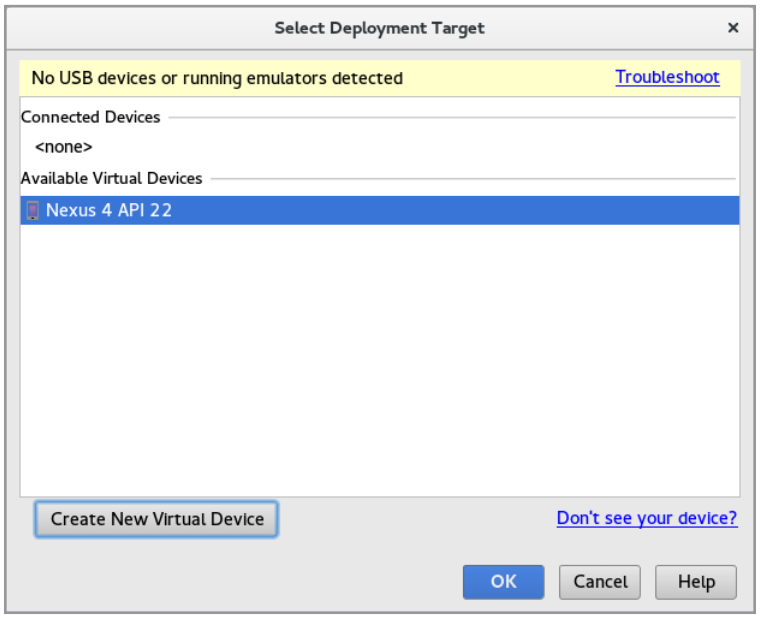
Below is a screenshot of Android Studio after the project has been created:



On the left side you have an overview of the project's files, which I do not want to mention further in this place. Otherwise, there are two tabs, where the first shows the application layout, while the other shows a Java class that corresponds to a main class in a conventional Java application.

The layout is called *activity_main* and is an XML document that defines the user interface. You can work with the document both in design mode where you can place components using the mouse, or you can work directly in the XML code. You can switch between the two modes by clicking on the bottom tabs (*Design* and *Text*).

The project is actually a complete application that you can build and test. To test the program, click the green arrow after *app* in the toolbar. When you do, you get the following window:

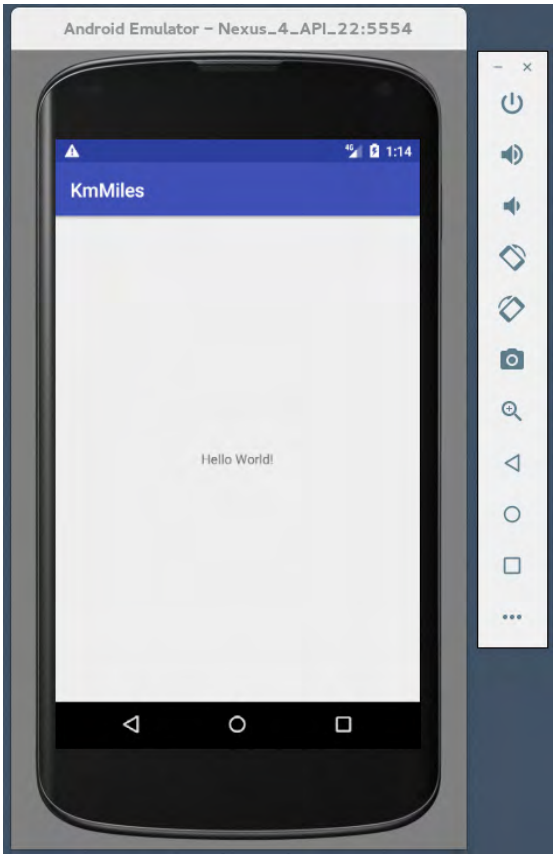


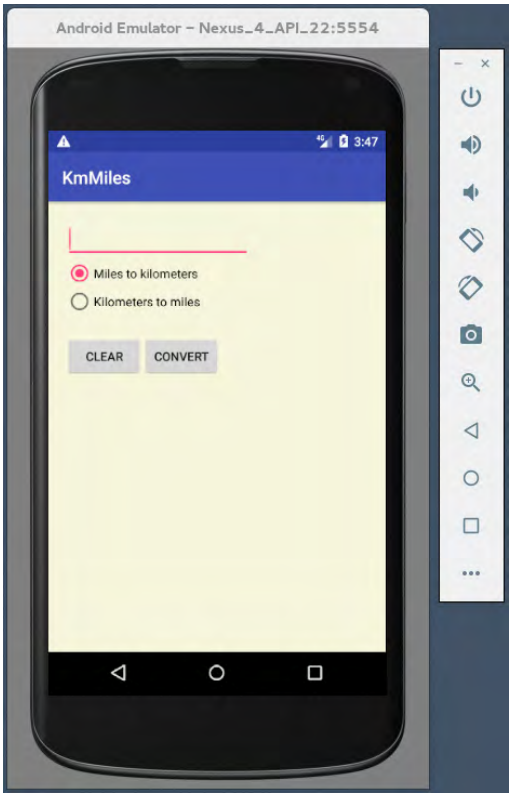
where to choose which device to test the program on. The first time the window is empty. There is no device connected, but you can create a virtual device by clicking the bottom button. Here you will first get an overview of which virtual devices are available. When I created the project, I chose that the application has to support API 22, and so I must choose a virtual device that does. I have chosen Nexus 4, and in the next window you will need to download the current device. It requires some downloads, but when that happens, you have a virtual device as shown in the above window.

When I then click OK, Android Studio will open the emulator (see below). It takes a while, because you have to download the emulator itself, and it takes a long time to start, but with a little hassle, you get a window that simulates a phone and with the current application active. For the time being, the application is trivial as it does nothing but write a text in the window. I will now expand the program from what Android Studio has auto generated to an application that opens an emulator as shown below, that is, an application where you can enter a number and then click on the *Convert* button, and depending on which radio button are checked, the number entered is interpreted as miles or kilometers and the number is converted to a value in the opposite unit. That's a classic conversion program. It is of course a very simple program that consists of

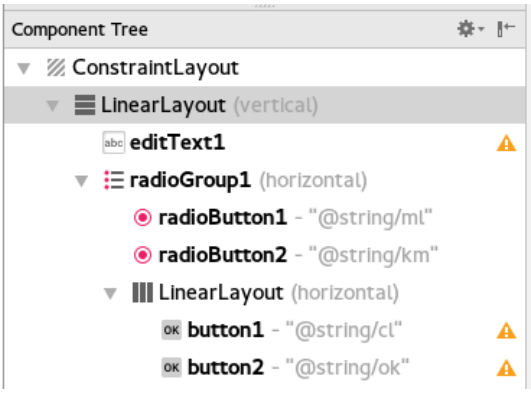
- a text field
- two radio buttons
- and two push buttons

but it is nevertheless a program that performs a data processing and not just shows a text.





To write the program I will start in the design window and drag components from the toolbox to the design window's *Component Tree*:



The design originally had a component that showed the text *Hello World*. I've deleted this component, and then I added

- a *LinearLayout*, there is some kind of layout manager
- a *EditText* (a *Plain Text*), there is an input field
- a *RadioGroup*, which is a container for components
- two *RadioButton* components
- a *LinearLayout*
- two *Button* components

You should note that the design panel shows how the individual components are nested in a hierarchy. Also note that if you select a component with the mouse, you get a panel on the right side of the window, where you can specify properties for that component.

If you examine the project files, see the *app* | *res* | *values*, there is a file called *strings.xml*. It is an XML document that defines strings and other resources that the application can apply and only one is defined, which is the program's name. I have expanded the file with 5 new definitions, the first 4 being texts for radio buttons and other buttons, while the last is for the background color of the window:

```
<resources>
 <string name="app_name">KmMiles</string>
 <string name="km">Kilometers to miles</string>
 <string name="ml">Miles to kilometers</string>
 <string name="ok">Convert</string>
 <string name="cl">Clear</string>
 <color name="bgColor">#F5F5DC</color>
</resources>
```

I will then display the finished XML for the user interface, which is essentially autogenerated by the designer window, but where I made some adjustments:

```
<?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:background="@color/bgColor"
  tools:context="dk.data.torus.kmmiles.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_marginTop="20dp"
    android:layout_marginStart="20dp">
    <EditText
      android:id="@+id/editText1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:ems="10"
      android:inputType="numberSigned|numberDecimal"/>
    <RadioGroup
      android:id="@+id/radioGroup1"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
```

```
<RadioButton
   android:id="@+id/radioButton1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:checked="true"
   android:text="@string/ml" />
<RadioButton
   android:id="@+id/radioButton2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="@string/km" />
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="horizontal">
   <Button
       android:id="@+id/button1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="20dp"
       android:text="@string/cl"
       android:onClick="onClick"/>
```

```
        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="20dp"
            android:text="@string/ok"
            android:onClick="onClick"/>
      </LinearLayout>
    </RadioGroup>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

When you see the code, you can not of course know that it should be written in this way, but you can easily understand the code and the meaning of the individual lines. You should especially note how to create an identifier for a component (for a *widget*), such as

```
android:id="@+id/editText1"
```

You should also note how to refer to resources (defined in *strings.xml*), and as an example:

```
android:text="@string/km"
```

Finally, note that an event handler has been defined for the two buttons. Basically, the document defines what components the window should have and how it should be laid out, and in addition, for each widget, a number of attributes are defined with values for corresponding properties for the objects that the finished program must consists of.

To perform the conversion, I have added a class named *Converter* to the directory (or package in the project pane) *app* | *java* | *dk.data.torus.kmmiles*:

```
package dk.data.torus.kmmiles;public class Converter
{
 private static final double factor = 1.609344;
 public static double toKm(double miles)
 {
  return miles * factor;
 }

 public static double toMl(double kilometers)
 {
  return kilometers / factor;
 }
}
```

The class is obviously trivial and is included solely to show that an Android application can contain Java classes in the same way as other Java applications and that it is just a usual Java class.

Back there is the main class *MainActivity*:

```
package dk.data.torus.kmmiles;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity
{
 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
 }

 public void onClick(View view)
 {
  EditText text = (EditText) findViewById(R.id.editText1);
  if (view.getId() == R.id.button1)
  {
   text.setText("");
  }
  else if (view.getId() == R.id.button2)
  {
   RadioButton ml = (RadioButton) findViewById(R.id.radioButton1);
   RadioButton km = (RadioButton) findViewById(R.id.radioButton2);
   if (text.getText().length() == 0)
   {
    Toast.makeText(this, "Please enter a valid number",
              Toast.LENGTH_LONG).show();
    return;
   }
   double value = Double.parseDouble(text.getText().toString());
   if (ml.isChecked()) text.setText(String.valueOf(Converter.toKm(value)));
   else text.setText(String.valueOf(Converter.toMl(value)));
  }
 }
}
```

As you can see, this is again a regular Java class. Here is *onCreate()* created by Android Studio, while the event handler should be written. There is not much to explain about the code, but you should notice how to refer to the individual components (widgets).

Then, in principle, the program is complete and can be translated and opened in the emulator, but in practice, more needs to be done before the program can be packed and placed for download on an app store. I will postpone that for later, but here a few words about what is needed. First, the program must be tested with different screen sizes, and that is, with different emulators. Next, the program must be tested on a physical design, which is actually quite simple. You can connect your device (phone) via a USB cable. For Android Studio to see the device, it must unlock the phone as a *Developer Device*. You can find out how on

```
https://developer.android.com/studio/run/index.
html?utm_source=android-studio
```

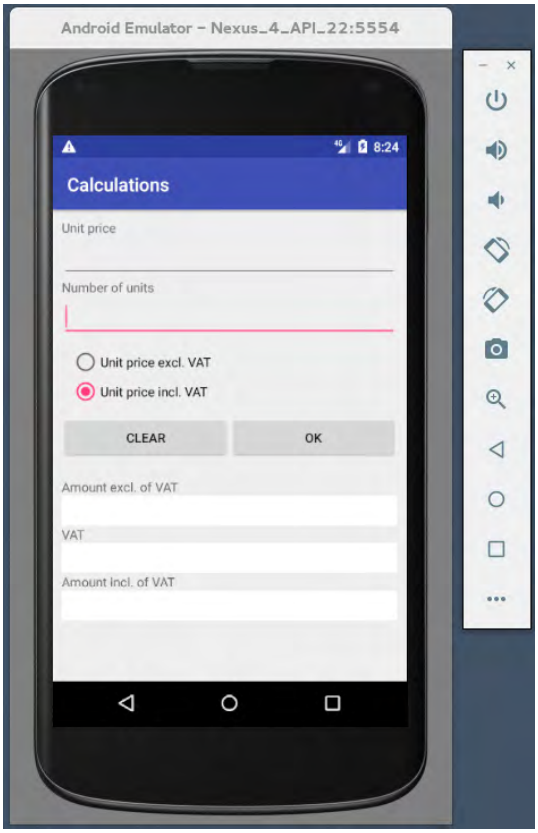
and then you can choose the physical device when you run the program, in the same way as for the emulator, after which the program is installed and running on the phone.

EXERCISE 1: CALCULATIONS

You must write a program using the same procedure as shown above. When you opens the program in the emulator, you should get a window as shown below that contains the following widgets:

- a *LinearLayout*
- a *TextView*
- an *EditText*
- a *ButtonGroup* with two *RadioButton* widgets
- a *LinearLayout* with two *Button* widgets
- a *TextView*
- an *EditText*
- a *TextView*
- an *EditText*
- a *TextView*
- an *EditText*

In the top entry field you must be able to enter a unit price, while in the next one must be able to enter the number of units. When clicking on the *OK* button, the application must calculate the total price without VAT (depending on which radio button is pressed), the VAT and total price with VAT, and then the three lower entry fields must be updated.



If you press the *Clear* button, all input fields must be blanked.

When you solve the exercise, you should place texts in the file *strings.xml* in the same way as in the introductory example.

The three bottom entry fields should in principle be read only. Try to find out how to do that.

2 WIDGETS

If you consider the introductory example, it contains 5 controls:

- a entry field – a *EditText* control
- two radiobuttons – *RadioButton* controls
- two buttons – *Button* controls

In Android, these are called controls for *widgets*, but the above are the same as you know from Swing or JavaFX. There are many such widgets, and some of the first to learn to write Android applications is of course learning which and how they are used. I do not want to systematically review the various components in this book, but illustrate some of them through examples, but basically they are used the same way you know it from controls in Java. Things are of course called something else, and there are other properties that you can set, but the principles are the same. Widgets are arranged in an extensive class hierarchy, where the overall basic class is called *View*. In this context, it is good to know the web address

```
https://developer.android.com/reference/classes.html
```

which is the basic reference for Android's many features.

As with Swing and JavaFX, the biggest challenges are getting a component placed in the window, where you now exactly want them to be, but also that the window looks nice at different sizes of the screen, but that is the subject of the next chapter.

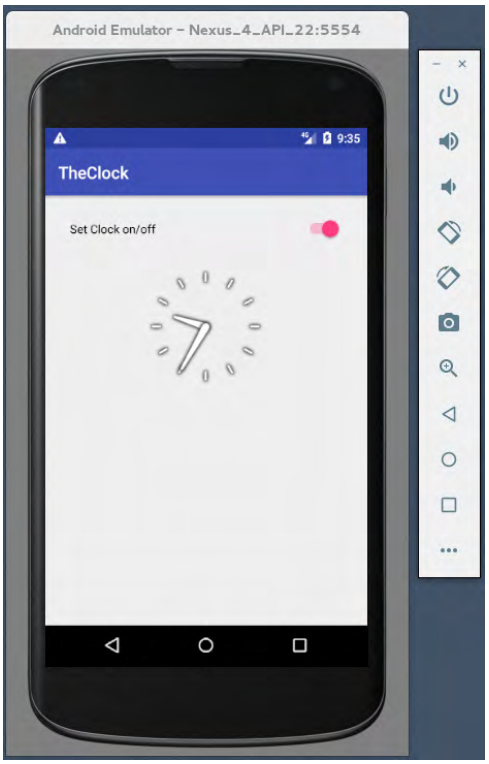
In general, Android has the same widgets that you know as controls in Swing / JavaFX, but maybe they are called something else. However, there are also new ones that are not found in Java, and as an example, I want to show an application with two widgets

- a *Switch*, that is an alternative to a *CheckBox*
- an *AnalogClock*, har shows a clock (the widget is now deprecated)

The project is called *TheClock*, and if you opens the application in the emulator, the result is as shown below. The application shows an analogue clock that counts in minutes. Over the clock there is a *Switch* (you know the widget from your phone) that shows a text and is a contact that you can turn on and off. If you run the application, you can hide the clock by clicking the *Switch* control, and you can display it again by clicking it on.

The project has been created in exactly the same way as the example in the introductory chapter, where I have followed the same wizard, and the only difference is that I have entered another project name. Then I deleted the *TextView* widget that Android Studio has added

and instead added a *LinearLayout* and again a *Switch* widget. *AnalogClock* does not exist in the toolbox (as it is deprecated), so I have to add it by entering the XML code.



The code is very simple and the most important is *main_activity.xml*, which defines the user interface:

```
<?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"
  tools:context="dk.data.torus.theclock.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <Switch
      android:id="@+id/switch1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:padding="30dp"
      android:checked="true"
      android:text="Set Clock on/off"
      android:onClick="onClick"/>
```

```
    <AnalogClock android:id="@+id/analog"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_centerHorizontal="true"
      />
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Again, it's easy enough to understand the code when you see it and remember that Android Studio has created the most. The only thing that is difficult to understand is the width and height attributes, but it is explained in the next chapter. Note that the *Switch* object has an event handler that must be written in the Java code and the task is changed the clock's visibility depending on the *Switch* contact's state:

```
package dk.data.torus.theclock;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
```

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

 public void onClick(View view)
 {
  Switch sw = (Switch) findViewById(R.id.switch1);
  AnalogClock cl = (AnalogClock) findViewById(R.id.analog);
  if (sw.isChecked()) cl.setVisibility(View.VISIBLE);
  else cl.setVisibility(View.INVISIBLE);
 }
}
```

2.1 A ZIPCODE APP

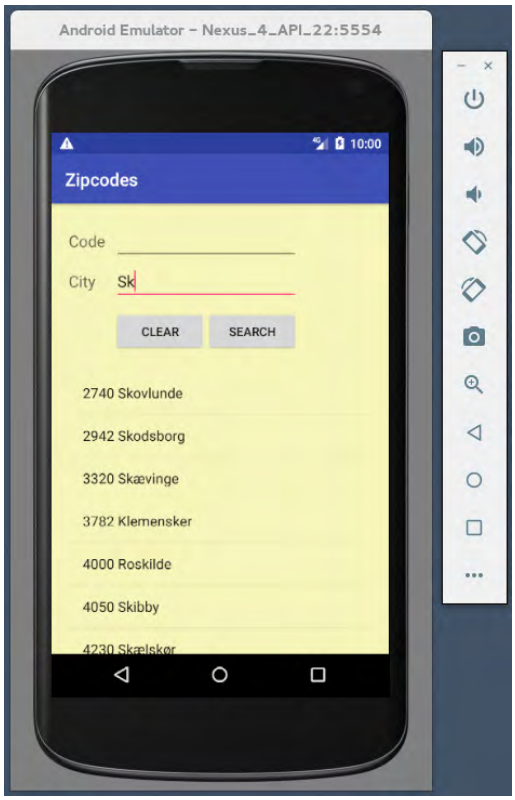
I want to exit this chapter about widgets with yet another simple example of an app. It must be an application where you can search for Danish postal codes by entering a search text for the postal code and the city name, respectively – an application that I have used as an example until several times. Compared with the above example, the application is somewhat larger and thus better reflects how a typical app for a mobile phone looks. If you perform the application in the emulator, you get the following result where *Sk* is searched for city name. The result is all the zip codes where the city name contains *Sk*. Instead if searching for a number, you get the zip codes where the number starts with the search text.

This time the code fills up a little more and I want to start with the resource file *strings.xml*:

```
<resources>
 <string name="app_name">Zipcodes</string>
 <string name="codelabel">Code</string>
 <string name="citylabel">City</string>
 <string name="clrlabel">Clear</string>
 <string name="seeklabel">Search</string>
 <color name="bgColor">#F5F5BA</color>
</resources>
```

which defines texts for the user interface as well as a background color. It is not a requirement that this is defined as resources, but it is recommended that you easily can change the

program without having to go around the code, and it also opens up to language versions of the application.



The user interface was built in the designer window by retrieving components from the toolbox, and then I have edited the XML code. It's filling up a bit:

```
<?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:background="@color/bgColor"
  tools:context="dk.data.torus.zipcodes.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="20dp"
    android:layout_marginLeft="20dp"
    android:layout_marginRight="20dp"
    android:orientation="vertical">
    <TableLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content">
```

```
<TableRow
   android:layout_width="match_parent"
   android:layout_height="match_parent" >
   <TextView
       android:id="@+id/textView"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginRight="10dp"
       android:textSize="18dp"
       android:text="@string/codelabel" />
   <EditText
       android:id="@+id/editText1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:ems="4"
       android:inputType="number" />
</TableRow>
<TableRow
   android:layout_width="match_parent"
   android:layout_height="match_parent" >
   <TextView
       android:id="@+id/textView2"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
```

```
            android:layout_marginRight="10dp"
            android:textSize="18dp"
            android:text="@string/citylabel" />
        <EditText
            android:id="@+id/editText2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:ems="10"
            android:inputType="textPersonName"/>
      </TableRow>
      <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="60dp"
            android:orientation="horizontal">
            <Button
                android:id="@+id/button1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:layout_marginTop="10dp"
                android:text="@string/clrlabel"
                android:onClick="onClick"/>
            <Button
                android:id="@+id/button2"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:layout_marginTop="10dp"
                android:text="@string/seeklabel"
                android:onClick="onClick"/>
        </LinearLayout>
      </TableRow>
    </TableLayout>
    <ListView
      android:id="@+id/listView1"
      android:layout_marginTop="20dp"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
    </ListView>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Again, the code is easy enough to understand. Basic the layout is a *LinearLayout* with two elements laid out vertically:

- a *TableLayout*
- a *ListView*

You can think of a *TableLayout* as a HTML table, where elements are organized in rows and columns. In this case, the component is used to create a form with the input fields and buttons. There are three rows where the first has a label (a *TextView* widget) and an input field (a *EditText* widget). Note which attributes are defined, and especially for the latter, that you can only enter integers. The next row is in principle identical, while the latter is a bit different. Each row has two cells, and the first cell in the last row has an empty label. The second cell has a *LinearLayout* with two buttons (*Button* widgets). Note that both buttons have an event handler, which must be defined in the program code – the class *MainActivity*. *ListView* is a widget that corresponds to a list box, and is temporarily empty as it should be initialized in the code.

As can be seen from the above, the XML section for an activity can quickly become extensive, but if you write applications for mobile phones, it is recommended (due to the screen size) that a window may only contain few widgets, and if there is a need for many, the application should in instead use more activities, what I will later show examples of.

Back there is the Java code, where this time in addition to *MainActivity* are two other classes, which are usual model classes. The idea is to show that an Android application consists of classes in the same way as other applications, and again, in principle, that it is not so much different to write mobile apps than other PC applications.

First, I want the following simple model class to be added to the project:

```
package dk.data.torus.zipcodes;

public class Zipcode
{
 private String code;
 private String city;

 public Zipcode(String code, String city)
 {
  this.code = code;
  this.city = city;
 }
```

```
 public String getCode()
 {
  return code;
 }

 public String getCity()
 {
  return city;
 }

 public String toString()
 {
  return getCode() + " " + getCity();
 }
}
```

which represents a zip code. Then I have added a class *Zipcodes* where the zip codes are defined as constants, and where the class creates a custom list with *Zipcode* objects:

```
package dk.data.torus.zipcodes;

import java.util.*;
```

```
public class Zipcodes
{
 private List<Zipcode> list = new ArrayList();

 public Zipcodes()
 {
  for (int i = 0; i < codes.length; ++i)
   list.add(new Zipcode(codes[i][0], codes[i][1]));
 }

 public List<Zipcode> search(String code, String city)
 {
  city = city.toLowerCase();
  List<Zipcode> lines = new ArrayList();
  for (Zipcode z : list)
   if (z.getCode().startsWith(code) && z.getCity().
   toLowerCase().contains(city))
    lines.add(z);
  return lines;
 }
 private static String[][] codes =
 {
  { "0800", "Høje Taastrup" },
  { "0900", "København C" },
  …
 }
}
```

Here is the important the method *search()* that is used to search for zip codes. Here you should especially note that it returns a *List<Zipcode>*, and thus a list of the *Zipcode* objects that match the search criteria.

Then there is the *MainActivity* class:

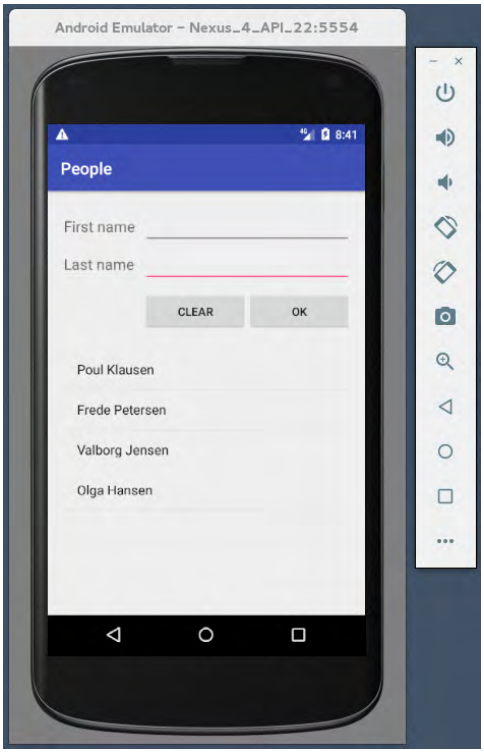
```
package dk.data.torus.zipcodes;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.ListView;
import java.util.*;
import android.widget.ArrayAdapter;

public class MainActivity extends AppCompatActivity
{
 private Zipcodes zipcodes = new Zipcodes();
```

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

 public void onClick(View view)
 {
  EditText text1 = (EditText) findViewById(R.id.editText1);
  EditText text2 = (EditText) findViewById(R.id.editText2);
  if (view.getId() == R.id.button1)
  {
   text1.setText("");
   text2.setText("");
  }
  else if (view.getId() == R.id.button2)
  {
   List<Zipcode> list = zipcodes.search(text1.getText().toString().trim(),
                            text2.getText().toString().trim());
   ArrayAdapter<Zipcode> adapter =
    new ArrayAdapter<Zipcode>(this, android.R.layout.
    simple_list_item_1, list);
   ListView lv = (ListView) findViewById(R.id.listView1);
   lv.setAdapter(adapter);
  }
 }
}
```

The class has an instance of the class *Zipcodes* and thus has the zip codes available. As for the event handler, only the code for the *Search* button contains something new. It you performs a search the result must be used to update the *ListView* component, which occurs with an *ArrayAdapter* for *Zipcode* objects. Here the parameters are not immediately obvious. The first is a reference to the current object and thus the object that consists of the *ListView*. The other parameter looks mysterious, but denotes a constant, which is defined as part of Android and indicates that it is an adapter for a *ListView*. Finally, the last parameter is the objects for which the *ListView* component is initialized. The adapter in question is subsequently used to initialize the component.

EXERCISE 2: PEOPLE

You must write a program similar to the example above which opens the window below. At the top there are two entry fields and at the bottom there is a *ListView*. In the two top fields you must be able to enter the name of a person and when you click *OK*, the name must be inserted in the list view. In the example below, 4 names are entered.



You can solve the exercise as described below:

1. Create a new project in Android Studio, which you can call *People*.
2. Create all texts in *string.xml*.
3. Write the design, that is, add all widgets by dragging them from the toolbox and dropping them into the *Component Tree*, and then adjust the XML code, but without adding event handlers.
4. Test the program in the emulator and repeat until you think it all looks nice.
5. Add a simple model class *Person* representing a person by a first name and a last name.
6. Add a *List<Person>* to your *MainActivity*.
7. Add an adapter between your list and your *ListView*. Note that the adapter creates a form of data binding between your list and the component.
8. Write the event handler to the *Clear* button.
9. Write the event handler to the *OK* button, which (if both a last name and first name are entered) must create a *Person* object and add it to your list.

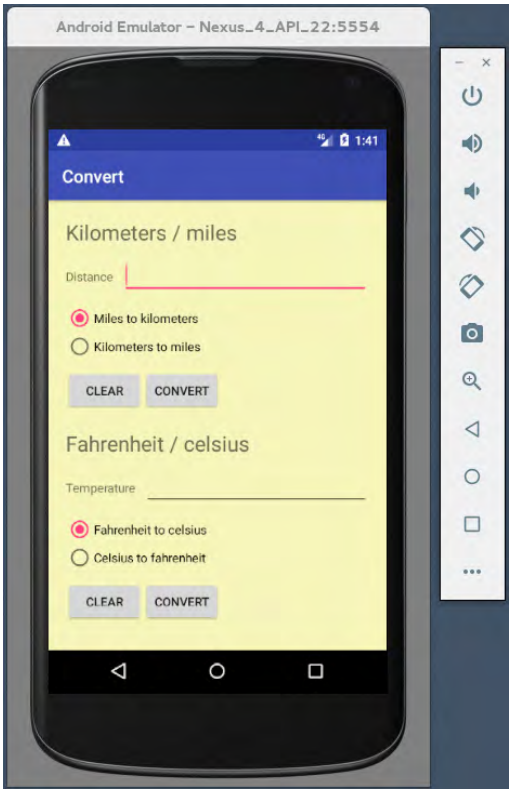
Then the program should be completed.

3 LAYOUT

In this chapter I will look a little more on layouts, which have of course already been part of the previous examples, but it is about how to place components in the window, thus matching what you know from Swing as layout managers. With this background (and with knowledge of panes in JavaFX) it is relatively simple to address how layouts works, what the previous examples also shows. However, there are some syntax, and the following is not a reference, but I will instead show through examples what is possible.

3.1 RELATIVELAYOUT

As example I will show an improved version of the program from chapter 1, which is enhanced by conversion between fahrenheit and celsius:

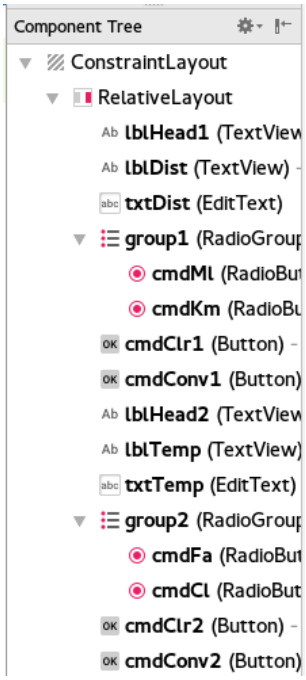


In addition to that, the main difference is that another the layout manager that has been used, which is called a *RelativeLayout*. The starting point is a new project, which I have called *Convert*. After the project has been created, Android Studio has created the following design for the user interface (that is *main_activity.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"
  tools:context="dk.data.torus.convert.MainActivity">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

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

The design contains a layout manager, called *ConstraintLayout*, and as I return to later, but you may notice that two attributes are defined *android:layout_width* and *android:layout_height*, both of which have the value *match_parent*. This means that the layout should apply all

the space as the parent – and this will say the screen – makes available, and it applies to both width and height. In this layout, Android Studio has inserted a *TextView* widget that I want to delete. I have then added a *RelativeLayout* to the layout window, and from the toolbox, I have draged the current widgets to the layout so that almost all widgets are now added to the window in the correct hierarchy (where I have changed the names of the individual widgets):



The result is that all widgets are in the top left corner of the window, and Android Studio has generated the following 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"
  tools:context="dk.data.torus.convert.MainActivity">
  <RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
      android:id="@+id/textView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="TextView" />
```

```
<TextView
   android:id="@+id/textView1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="TextView" />
<EditText
   android:id="@+id/editText"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:ems="10"
   android:inputType="textPersonName"
   android:text="Name" />
<RadioGroup
   android:layout_width="wrap_content"
   android:layout_height="wrap_content">
   <RadioButton
      android:id="@+id/radioButton4"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="RadioButton" />
   <RadioButton
      android:id="@+id/radioButton3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="RadioButton" />
</RadioGroup>
<Button
   android:id="@+id/button5"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Button" />
<Button
   android:id="@+id/button6"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Button" />
<TextView
   android:id="@+id/textView2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="TextView" />
<TextView
   android:id="@+id/textView3"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="TextView" />
```

```
<EditText
   android:id="@+id/editText2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:ems="10"
   android:inputType="textPersonName"
   android:text="Name" />
<RadioGroup
   android:layout_width="wrap_content"
   android:layout_height="wrap_content">
   <RadioButton
      android:id="@+id/radioButton6"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="RadioButton" />
   <RadioButton
      android:id="@+id/radioButton5"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="RadioButton" />
</RadioGroup>
```

```
    <Button
      android:id="@+id/button7"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Button" />
    <Button
      android:id="@+id/button8"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Button" />
  </RelativeLayout>
</android.support.constraint.ConstraintLayout>
```

Here you should especially note what values Android Studio has assigned for *android:layout_width* and *android:layout_height* respectively. For the *RelativeLayout*, it is *match_parent*, which means it should apply all the space that the parent component makes available, and here is a *ConstraintLayout* that fills the entire window. For the other components, *wrap_content* is used everywhere, which means that the component size is determined by the content, and if you take a *TextView* as an example, it is the component's text.

After I, as described above, have uncritically filled the widgets into the window, it's my job to make them sit nicely, and this is where the layout manager enters the image, as here is a *RelativeLayout*. As the name says, it sets the individual widgets relative to each other by specifying attributes, and so I have a process back where I adjust the XML code by adding attributes until it's all the way I want it. It's also as part of this process that I name the individual widgets so they have slightly more pronounced names than what Android Studio has assigned. Below is the completed 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:background="#F5F5BA"
  tools:context="dk.data.torus.convert.MainActivity">
  <RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="20dp">
    <TextView
      android:id="@+id/lblHead1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
```

```
   android:layout_marginBottom="10dp"
   android:text="Kilometers / miles"
   android:textSize="24dp"/>
<TextView
   android:id="@+id/lblDist"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/lblHead1"
   android:layout_alignBaseline="@id/txtDist"
   android:text="Distance" />
<EditText
   android:id="@+id/txtDist"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:layout_toEndOf="@id/lblDist"
   android:layout_below="@id/lblHead1"
   android:layout_marginLeft="10dp"
   android:inputType="numberSigned|numberDecimal"/>
<RadioGroup
   android:id="@+id/group1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/txtDist"
   android:layout_marginTop="10dp">
   <RadioButton
      android:id="@+id/cmdMl"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:checked="true"
      android:text="Miles to kilometers" />
   <RadioButton
      android:id="@+id/cmdKm"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="Kilometers to miles" />
</RadioGroup>
<Button
   android:id="@+id/cmdClr1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/group1"
   android:layout_marginTop="10dp"
   android:text="Clear"
   android:onClick="distClear"/>
<Button
   android:id="@+id/cmdConv1"
   android:layout_width="wrap_content"
```

```
   android:layout_height="wrap_content"
   android:layout_below="@id/group1"
   android:layout_toEndOf="@id/cmdClr1"
   android:layout_marginTop="10dp"
   android:text="Convert"
   android:onClick="distConvert"/>
<TextView
   android:id="@+id/lblHead2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/cmdClr1"
   android:layout_marginTop="20dp"
   android:layout_marginBottom="10dp"
   android:text="Fahrenheit / celsius"
   android:textSize="24dp"/>
<TextView
   android:id="@+id/lblTemp"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/lblHead2"
   android:layout_marginTop="20dp"
   android:layout_alignBaseline="@id/txtTemp"
   android:text="Temperature" />
```

```
    <EditText
      android:id="@+id/txtTemp"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_below="@id/lblHead2"
      android:layout_toEndOf="@id/lblTemp"
      android:layout_marginLeft="10dp"
      android:inputType="numberSigned" />
    <RadioGroup
      android:id="@+id/group2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_below="@id/txtTemp"
      android:layout_marginTop="10dp">
      <RadioButton
         android:id="@+id/cmdFa"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_weight="1"
         android:checked="true"
         android:text="Fahrenheit to celsius" />
      <RadioButton
         android:id="@+id/cmdCl"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_weight="1"
         android:text="Celsius to fahrenheit" />
    </RadioGroup>
    <Button
      android:id="@+id/cmdClr2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_below="@id/group2"
      android:layout_marginTop="10dp"
      android:text="Clear"
      android:onClick="tempClear"/>
    <Button
      android:id="@+id/cmdConv2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_below="@id/group2"
      android:layout_toEndOf="@id/cmdClr2"
      android:layout_marginTop="10dp"
      android:text="Convert"
      android:onClick="tempConvert"/>
  </RelativeLayout>
</android.support.constraint.ConstraintLayout>
```

In the following I will briefly mention what has happened. In the *ConstraintLayout*, I have defined a background color, and in the *RelativeLayout* I have defined a margin of 20. It does not matter where the individual widgets are placed, but are overall settings that are even independent of the current layout manager.

For the first *TextView* widget, I changed the name to *lblHead1*, and then I have defined a margin of 10 below the component. I have also changed the text (the text to be displayed on the screen) and I have defined a font size.

The next *TextView* widget I have called the *lblDist*, and then I have with *android:layout_below* defined that the component should be under *lblHead1*. This is where the relative layout comes in. Then, with *android:layout_alignBaseLine*, I have defined the text to be displayed on the same baseline as for the following component, which is the input field, and the goal is to get the two widgets aligned vertically relative to each other. Here you should especially note that in XML is allowed to refer to a widget (*txtDist*) that is not yet been defined. It is the next widget that is defined and in addition to the name being changed to *txtDist*, note that *android:layout_width* has been changed to *fill_parent* which means that the component width should fill the rest of the space as the parent component (here a *RelativeLayout*) has set aside, that is, it needs to use the rest of the window width. In addition, three attributes are defined where with *android:layout_toEndOf*, the component should be located to the right of the *TextField* component and with *android:layout_below* that it should sit under the header text and finally with *android:layout_marginLeft* that there is some space between the input field and the text in front.

The next widget is a *RadioGroup*. There is not much to explain here, but I have given it a name and defined that it should be below the input field and that there must be a margin of 10 to the input field. As for the two *RadioButton* widgets, there is also not much to notice except that I have changed the names again and indicate that the first one must be checked. Also note the attribute *android:layout_weight*. It is actually not necessary but inserted by Android Studio and means that the entire height is to be used, but in other contexts it is important.

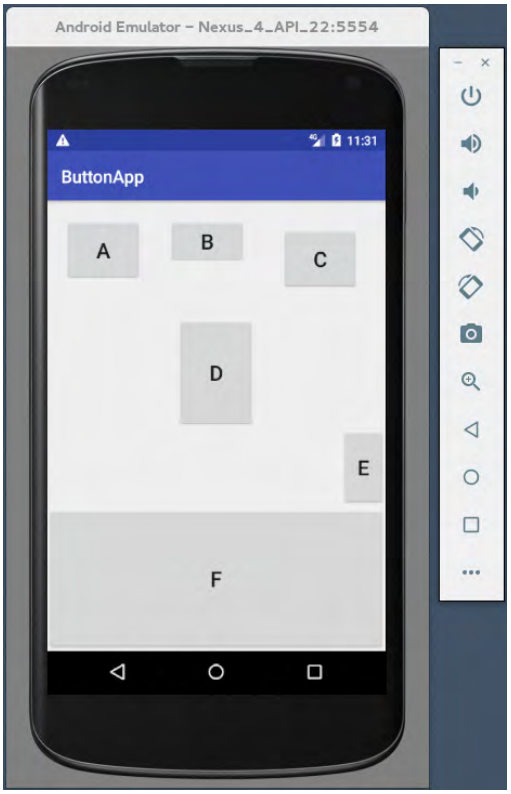
Next, follows the two (top) buttons, and compared to what has already been explained, there is nothing new, but you should note that for both buttons are defined event handlers.

The rest of the XML code is, in principle, just a repeat of the above, and the result of all is that the individual widgets are laid out nicely adjusted to each other. The procedure may sound a little bit comprehensive, and it is, but with a little practice it is actually a quite safe and also relatively effective way to design a complex window. The actual window is actually relative to mobile phones complex and should rarely be more complex.

You should note that this time I have not defined texts as resources, but directly in the XML code. There are no reasons, only for making it simple to write the code. Back there is the Java code, and here the class *Converter* is expanded with two new static conversion methods. The class *MainActivity* now has four event handlers for each of the four buttons. They are all simple and I do not want to display the code here. Then the application is complete and can be translated and tested in the emulator and can even be installed on a phone as described in the introductory chapter.

3.2 LINEARLAYOUT

In the first examples of this book, I have used a *LinearLayout*, which is a layout manager that composes components in a row or a column. It can instantly be compared to a *VBox* and a *HBox* in JavaFX, and like JavaFX (and Swing) layout managers can be nested and you can solve many layout challenges using a *LinearLayout*. The example shown below is called *ButtonApp* and puts 6 buttons on the screen using two nested *LinearLayout* managers:



The application does nothing but show the 6 buttons and they have no function so the only code that is to show is *main_activity.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"
  tools:context="dk.data.torus.buttonapp.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal">
      <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:padding="20dp"
        android:layout_weight="1"
```

```
            android:textSize="24dp"
            android:text="A" />
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="10dp"
            android:padding="10dp"
            android:layout_weight="1"
            android:textSize="24dp"
            android:text="B" />
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="30dp"
            android:padding="20dp"
            android:layout_weight="1"
            android:textSize="24dp"
            android:text="C" />
    </LinearLayout>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="24dp"
        android:layout_weight="2"
        android:layout_gravity="center"
        android:text="D" />
    <Button
        android:layout_width="50dip"
        android:layout_height="wrap_content"
        android:textSize="24dp"
        android:layout_weight="1"
        android:layout_gravity="right"
        android:text="E" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="24dp"
        android:layout_weight="3"
        android:text="F" />
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

The layout starts with a *LinearLayout* and it has an attribute *android:orientation* that tells that the components should to be vertically positioned. The first element is a new *LinearLayout* and hence a nested layout, and its orientation is instead *horizontal*. Here you should note that it sets *android:layout_height* to *wrap_content* so it does not take over the entire window.

The inner *LinearLayout* contains three buttons with the texts A, B and C respectively, and the result is that they are laid out horizontally. For each button, a margin is defined that indicates how much space there should be outside of the button. A *padding* is also defined, and it specifies how the distance should be from the text to the edges of the component (if it is a component with a frame). For all three buttons, *android:layout_weight* are 1, which indicates that they have to share the space equally and thus get the same width.

The outer *LinearLayout* has, in addition to the *LinearLayout* element above, three buttons, which are then vertically laid out below the layout with the buttons A, B and C. The three lower buttons D, E and F have the weights 2, 1 and 3. This means that they share the vertical space in the ratios 2/6, 1/6 and 3/6. With respect to the width of the components, note that for D, the width is determined of the content, while for E an absolute width is given. For F, it is the width of the parent and thus the width of the window. You should also note the attribute *android:layout_gravity*, which indicates alignment.

Previously, *LinearLayout* was a very used layout manager, and it still makes sense in situations where a very simple layout is needed.

3.3 TABLELAYOUT

I used this layout manager in the previous chapter in the connection with the zip code application, and I do not want to show another examples, but in fact there is a lot more to explain. The best to do is to think of it as an HTML table, and it supports concepts like colspan and placement of widgets in specific columns, but basically it's a layout manager that's easy to use.

3.4 GRIDLAYOUT

There is also a layout manager called *GridLayout* and, in principle, it corresponds to a *GridPane* in JavaFX. Basically, it is a layout manager that divides the screen into a number of cells, and you can then load components into the manager, and if nothing else, it will put all widgets in a large row, but you can specify a number of columns that so indicates how many columns there should be. You can also define the number of rows and you can also define a span for the individual cells. The *IntCalc* program is a simple calculator with the following layout:

The calculator works alone on integers, thus performing, for example, integer division. The input field is read-only, and you can only enter numbers by clicking the buttons. Expressions must be entered in postfix form, thus entering the operator after the arguments. For example, if you want to calculate

23 + 19

then you must enter 23 and click on Enter, which means that 23 will be saved on a stack. Then enter 19, and then click on the plus button and the program calculates the sum of the content of the display and the number popped from the stack, after which the display is updated with the result. The commands in the bottom row may not be quite obvious, but ! determines the factorial of the content of the display, ^ is power raising while _ changing the sign of the content of the display. Finally, C is command that deletes the content of the display, but it simultaneously deletes the content of the stack, although it should probably be two commands.

The program is made simple and consists solely of *activity_main.xml* and *MainActivity.java*. The XML section fills a lot, and I have only shown a smaller part below, as the rest alone refers to more buttons, which, of course, are, in principle, identical:

```
<?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"
  tools:context="dk.data.torus.intcalc.MainActivity">
  <GridLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:columnCount="5">
    <EditText
      android:id="@+id/txtDisplay"
      android:layout_width="0dp"
      android:layout_height="60dp"
      android:layout_columnWeight="1"
      android:layout_gravity="fill_horizontal"
      android:textSize="24dp"
      android:layout_columnSpan="5" />
    <Button
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_columnWeight="1"
      android:gravity="center"
      android:layout_gravity="fill_horizontal"
      android:textSize="24dp"
      android:onClick="modClicked"
      android:text="%" />
    <Button
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_columnWeight="1"
```

```
        android:gravity="center"
        android:layout_gravity="fill_horizontal"
        android:textSize="24dp"
        android:onClick="divClicked"
        android:text="/" />
    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_columnWeight="1"
        android:gravity="center"
        android:layout_gravity="fill_horizontal"
        android:textSize="24dp"
        android:onClick="mulClicked"
        android:text="*" />

    ...

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

First, note that a *GridLayout* is defined with 5 columns. Here you can also specify the number of rows, but it is not necessary as the rows are determined from the XML code. The first

widget is an *EditText* and you should note that its width is set to 0. This is because the width is determined by a weight attribute and a graverty attribute, which are the attributes that a *GridLayout* uses to determine how much components should fill. Also note that a *columnSpan* of 5 is defined for this component as the machine's display should fill the full width of the window. Finally, note that the component has an ID as it should be referred to in the Java code, but it is also the only one of the layout's widgets that has an ID.

All the remaining (20) widgets are *Button* components, and in fact there is not much to note beyond which attributes are defined for the size. Since weight anywhere is 1, it means that they all get the same width as *gravirty* is set to *fill_horizontal*.

The Java code also fills as there are 20 event handlers, and the following is only a part of the code:

```
package dk.data.torus.intcalc;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
import java.util.*;

public class MainActivity extends AppCompatActivity
{
 private Stack<Long> stack = new Stack();
 private boolean newValue = false;

 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  EditText text = (EditText) findViewById(R.id.txtDisplay);
  text.setKeyListener(null);
  text.setEnabled(false);
 }

 public void oneClicked(View view)
 {
  insert(view, "1");
 }

 …
```

```
public void addClicked(View view)
{
 calc2(view, '+');
}

…

private void insert(View view, String ch)
{
 EditText text = (EditText) findViewById(R.id.txtDisplay);
 try
 {
  if (newValue)
  {
   long value = Long.parseLong(text.getText().toString());
   stack.push(value);
   text.setText(ch);
   newValue = false;
  }
  else text.setText(text.getText() + ch);
 }
 catch (Exception ex)
 {
  text.setText("Error");
 }
}

…

private void calc2(View view, char opr)
{
 EditText text = (EditText) findViewById(R.id.txtDisplay);
 try
 {
  long value2 = Long.parseLong(text.getText().toString());
  long value1 = stack.pop();
  switch (opr)
  {
   case '+': value1 += value2; break;
   case '-': value1 -= value2; break;
   case '*': value1 *= value2; break;
   case '/': value1 /= value2; break;
   case '%': value1 %= value2; break;
   case '^': value1 = (long)Math.pow(value1, value2);
  }
  text.setText("" + value1);
  newValue = true;
 }
```

```
    catch (Exception ex)
    {
     text.setText("Error");
    }
  }

  …

}
```

The class has two instance variables, where the first one is the stack, while the other is used to control whether the content of the display should be placed on the stack when clicking on a digit. The constructor is expanded with two statements used to set the display read-only so that you can not directly enter numbers. Note that this means that no virtual keyboard is displayed.

If you click on a button for a digit, the method *insert()* is performed. It tests the variable *newValue*, and if it is true, it means entering a new number and the method tries to parse the content of the display into an integer and put it on the stack. Then the digit that is clicked on is inserted as the value of the display. If *newValue* is false, nothing else happens than the digit that is clicked is added to the content of the display.

Clicking on an operator with two arguments, the event handler calls the method *clac2()*. It parses the content of the display and pops the stack and execute the actual calculation, after which the display is updated. Note that *newValue* is set to *true* as the result must be added to the stack next time a button is clicked. Instead, click on an operator with only one argument (and it will say for the current machine ! and _) the method *calc1()* is called that changes the value of the display.

3.5 A CONSTRAINTLAYOUT

As the last layout manager, I would like to mention a *ConstraintLayout*, which is the layout manager that Android Studio defines as default for an activity. In principle, it works a bit like a *RelativeLayout*, but it is more flexible and the goal is that you can design a complex layout solely using the mouse and in the Android Studio's layout editor. That is, you can drag widgets to the window from the toolbox and drag it into the wanted places with the mouse. You can thus create the finished design without having to write XML yourself.

To define the position of a view component, it must have at least one horizontal and at least one vertical *constraint*, each constraint being an attachment or an alignment to another view or parent layout. Each constraint thus defines a view position along either a horizontal or vertical axis, but often there will be more constraints for each axis.

If you drop a view in the layout editor, it has no constraints, but sits where it is dropped. There are probably some attributes, and an example could be:

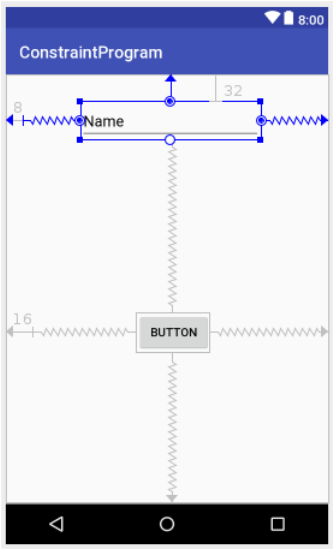
```
<Button
  android:id="@+id/button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Button"
  tools:layout_editor_absoluteX="148dp"
  tools:layout_editor_absoluteY="167dp" />
```

where the last two attributes tells you where the component is in the layout editor, but if you run the program, you will find that the button is located in the upper left corner, and the two attributes therefore do not matter to the finished program.

Below is the XML code after I have also added an *EditText* widget and then I have drawn both views to those positions where I think they should be:

```
<Button
  android:id="@+id/button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Button"
  tools:layout_editor_absoluteX="148dp"
  tools:layout_editor_absoluteY="168dp" />
<EditText
  android:id="@+id/editText"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:ems="10"
  android:inputType="textPersonName"
  android:text="Name"
  tools:layout_editor_absoluteX="85dp"
  tools:layout_editor_absoluteY="83dp" />
```

If you now select a view (and as esample the *EditText* component), you will see four small circles. If you point to one of them, you can drag the component against another component or layout, that this component should be placed in relation to. Below it is the layout manager itself. Next, you can adjust the distance under *Properties* to the right of the window, and in the toolbar you can specify alignment.



After I've done it for both widgets, the XML code is:

```
<Button
  android:id="@+id/button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_marginStart="16dp"
```

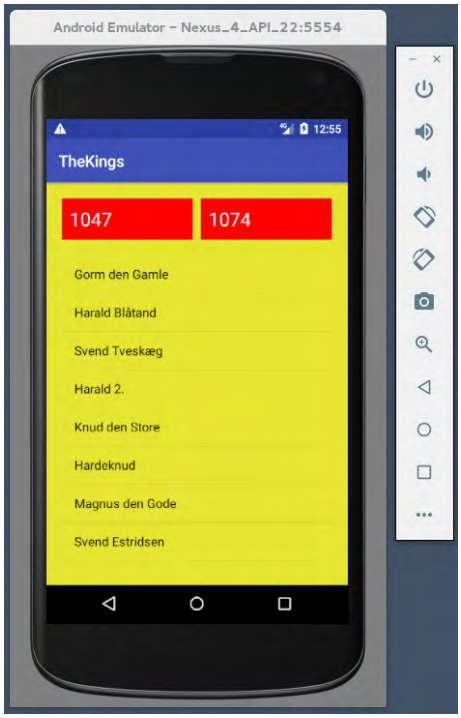
```
  android:layout_marginTop="24dp"
  android:text="Button"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintHorizontal_bias="0.5"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toBottomOf="@+id/editText" />
<EditText
  android:id="@+id/editText"
  android:layout_width="wrap_content"
  android:layout_height="46dp"
  android:layout_marginStart="8dp"
  android:layout_marginTop="32dp"
  android:ems="10"
  android:inputType="textPersonName"
  android:text="Name"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintHorizontal_bias="0.5"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent" />
```

and Android Studio has converted all editor constraints into actual constraints, after which the program can be translated and run, and the components are located corresponding to the above XML.

Of course, it requires a little learning how to use the layout editor, but if you do not write Android applications every day, it's worth the effort, because you're free to keep in mind the XML syntax, which is not everywhere, easy to understand. With the help of a *ConstraintLayout*, you can design a full-featured app without meeting the XML code. On the other hand, if you often write mobile apps, it can actually pay to work directly in XML. By learning how to use the different layout managers and how to determine the position and alignment of widgets, you get a very stable and safe layout, and you get fast very efficiently to works on complex activities. Here you should be aware that you can always switch to design mode to see how it all looks.

3.6 CONTAINERS

Android also offers multiple components that can display a family of objects. There are widgets called containers, and although layout managers are not directly containers, they have something to do with layout. In fact, I have already shown an example in the application *Zipcodes*, which used a *ListView*, which is a container that can display a number of objects. If the container contains more objects than there is room for on the screen, you can scroll the content, and you can also associate an event handler with the container that fires each time you select an object. In principle, the different containers are used in the same way and use a data source with the current objects, the container itself, such as a *ListView* and then an adapter that has the task to update the container. I will start with a program that shows an overview of *Danish kings*:



where there above are clicked on *Svend Estridsen* with the result that the government period appears in the top two fields. The design of the user interface is quite simple and contains in addition to two *LinearLayout* three widgets:

```
<?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:background="#E5E532"
  tools:context="dk.data.torus.thekings.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_margin="20dp">
    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal">
      <EditText
        android:id="@+id/txtFrom"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
```

```
            android:textSize="24dp"
            android:layout_marginRight="5dp"
            android:padding="10dp"
            android:textColor="#FFFFFF"
            android:background="#FF0000"/>
        <EditText
            android:id="@+id/txtTo"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:textSize="24dp"
            android:layout_marginLeft="5dp"
            android:padding="10dp"
            android:textColor="#FFFFFF"
            android:background="#FF0000"/>
    </LinearLayout>
    <ListView
        android:id="@+id/lstKings"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="20dp"/>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

There are two *EditText* widgets, and here is nothing new to explain, but note the names and which attributes that are defined. The latter is a *ListView*, where there is nothing to notice besides the name, but it is a container and the widget to be initialized with data, and to which an event handler is to be attached. It happens in the Java code.

First, the data resource and the starting point is the following model class:

```
package dk.data.torus.thekings;

public class King
{
 private String name;
 private int from;
 private int to;

 public King(String name, int from, int to)
 {
  this.name = name;
  this.from = from;
  this.to = to;
 }

 public String getName()
 {
  return name;
 }

 public int getFrom()
 {
  return from;
 }

 public int getTo()
 {
  return to;
 }

 public String toString()
 {
  return name;
 }
}
```

It is also a very simple class that does not contain anything other than what the current program needs, and here you should especially note the method *toString()* which returns

the value that appears for each object in the *ListView* container. As the second part of the data resource, the class *Kings* makes the current *King* objects available:

```
package dk.data.torus.thekings;

import java.util.*;

public class Kings
{
 private List<King> list = new ArrayList();

 public Kings()
 {
  for (String[] arr : data)
   list.add(new King(arr[0], Integer.parseInt(arr[1]),
    Integer.parseInt(arr[2])));
 }

 public List<King> getKings()
 {
  return list;
 }

 private static final String[][] data = {
         { "Gorm den Gamle", "0", "958" },
         { "Harald Blåtand", "958", "987" },
         …
 };
}
```

where I have only shown few data. The class is of course pseudo as all data are hard-coded and it is clear that this data should come from another source such as a file, database or equivalent, but since I have not yet shown how, it must wait for later , but it does not change the principle.

An adapter is actually a basic concept in Android, and the goal is to disconnect the data source from the presentation, so that for the programmer it happens in the same way, regardless of which data source it is and regardless of which list widget is used to display the data. In this case, the code of the class is in the class *MainActivity*:

```
package dk.data.torus.thekings;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
```

```
import android.widget.*;
import android.widget.AdapterView.*;
import java.util.*;

public class MainActivity extends AppCompatActivity
{
 private List<King> kings = (new Kings()).getKings();
 private EditText from;
 private EditText to;

 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  from = (EditText) findViewById(R.id.txtFrom);
  to = (EditText) findViewById(R.id.txtTo);
  disable(from);
  disable(to);
  ListView view = (ListView) findViewById(R.id.lstKings);
  view.setAdapter(new ArrayAdapter<King>(this,
   android.R.layout.simple_list_item_1, kings));
```

```
  view.setOnItemClickListener(new OnItemClickListener()
  {
   @Override
   public void onItemClick(AdapterView<?> parent, View view, final int position,
    long id)
   {
    update(position);
   }
  });
 }

 public void update(int position)
 {
  int a = kings.get(position).getFrom();
  int b = kings.get(position).getTo();
  from.setText(a == 0 ? "" : "" + a);
  to.setText(b == 9999 ? "" : "" + b);
 }

 private void disable(EditText view)
 {
  view.setKeyListener(null);
  view.setEnabled(false);
 }
}
```

Initially, a list of objects of the type *King* is created, which is then the program's data source. In addition, variables are defined for the two *EditText* components, and they are both initialized in *onCreate()*. In addition, for both objects, the method *disable()* is called, which disables both input fields so that you can not enter text. Next, a reference to the *ListViev* component is defined and initialized with an adapter:

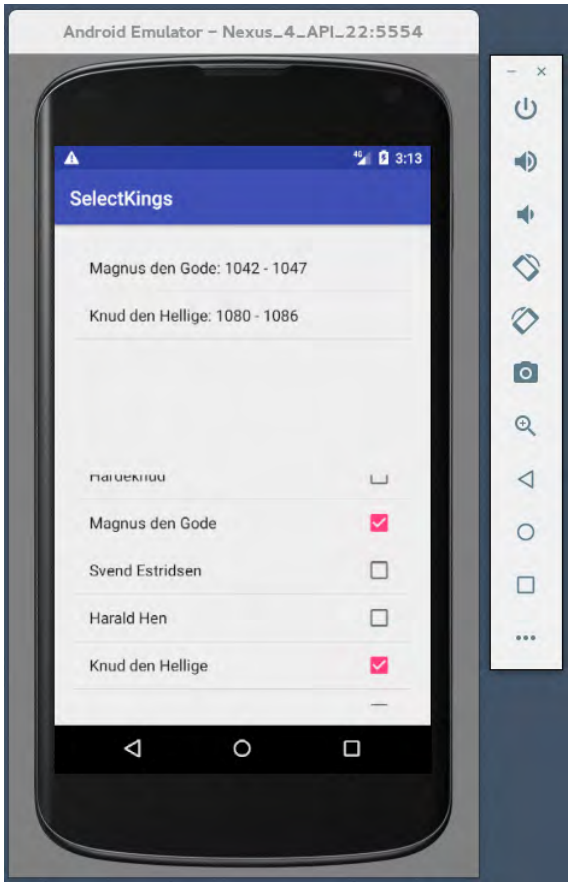
```
new ArrayAdapter<King>(this, android.R.layout.simple_list_item_1, kings)
```

The type is *ArrayAdapter* which is an adapter used for a data source that is an array or a list and thus a data source where the individual objects are indentified by an index. In addition, a parameter (for the generic type) tells that the adapter represents *King* objects. The constructor has three parameters, where the first is the object (here the current object) to which the container has to initialize is a member, while the last parameter is the data resource. Then there is the middle parameter that tells how the objects should be presented by the container, and here it is a constant whose meaning is not easy to understand, but it tells that an object should be presented as a string and thus that it is the value of the object's *toString()* method to de displayed.

As the last, an event handler is attached to the *ListView* component, and it happens with the method *setOnItemClickListener()*, where the parameter is an *OnItemClickListener*, which is an interface that defines the event handler. It has four parameters, where the most important is *position*, which indicates the index of the object that is clicked (tabbed with the finger). In this case, it performs the method *update()*, which updates the two top input fields with the king's government period.

I want to show a variation of the above application, which I have called *SelectKings*. The program opens the following phone, which again shows the Danish kings, but this time there are two *ListView* components, where the top shows the kings selected in the bottom. At the bottom you can this time select objects using checkboxes and you can select multiple and select more objects. In the example, there are selected two, and if you uncheck the checkboxes, the names disappear from the top *ListView.*

Again, the program consists of an XML file for defintion of the user interface, as well as the class *MainActivity* (and the model classes *King* and *Kings*), and the program primarily shows how to dynamically update the user interface using an adapter.



The layout in the XML file is this time the following:

```
<LinearLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:layout_margin="20dp">
  <ListView
    android:id="@+id/lstKings1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:layout_marginBottom="20dp"/>
  <ListView
    android:id="@+id/lstKings2"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:choiceMode="multipleChoice"
    android:drawSelectorOnTop="false" />
</LinearLayout>
```

There is not much to note, but you should notice how to indicate in the bottom that select multiple should be possible. In addition, note how in the design with *layout_weight* to define that the two components should share the space equally. The most important thing happens in the Java part, and here the classes *King* and *Kings* are unchanged while *MainActivity* is changed:

```
package dk.data.torus.selectkings;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
import android.widget.AdapterView.*;
import java.util.*;

public class MainActivity extends AppCompatActivity
{
 private List<King> kings = (new Kings()).getKings();
 private List<String> lines = new ArrayList();
 private ListView view;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  view = (ListView) findViewById(R.id.lstKings1);
  ListView view2 = (ListView) findViewById(R.id.lstKings2);
  view2.setAdapter(new ArrayAdapter<King>(this,
   android.R.layout.simple_list_item_multiple_choice, kings));
  view2.setOnItemClickListener(new OnItemClickListener()
  {
   @Override
   public void onItemClick(AdapterView<?> parent, View view, final int position,
    long id)
   {
    update(kings.get(position));
   }
  });
 }

 private void update(King king)
 {
  String line = toString(king);
  if (!lines.remove(line)) lines.add(line);
  view.setAdapter(new ArrayAdapter<String>(MainActivity.this,
   android.R.layout.simple_list_item_1, lines));
 }
```

```
 private String toString(King king)
 {
  if (king.getFrom() != 0 && king.getTo() != 9999)
   return String.format("%s: %d - %d", king.
   getName(), king.getFrom(), king.getTo());
  if (king.getFrom() != 0)
   return String.format("%s: %d -", king.getName(), king.getFrom());
  if (king.getTo() != 9999)
   return String.format("%s: - %d", king.getName(), king.getTo());
  return king.getName();
 }
}
```

Same as in the previous example, a *List* of *King* objects is created that is used as data source for the bottom *ListView*. In addition, another list is created for strings that is used as data source for the top *ListView*, and a reference to the top *ListView* is also defined. The reason is that you do not have to find the reference each time you clicks on the bottom *ListView*. This reference is initialized in *onCreate()*. Here, the bottom *ListView* is also initialized, in principle similar to the previous example, but this time another parameter for the second parameter to the adapter is used:

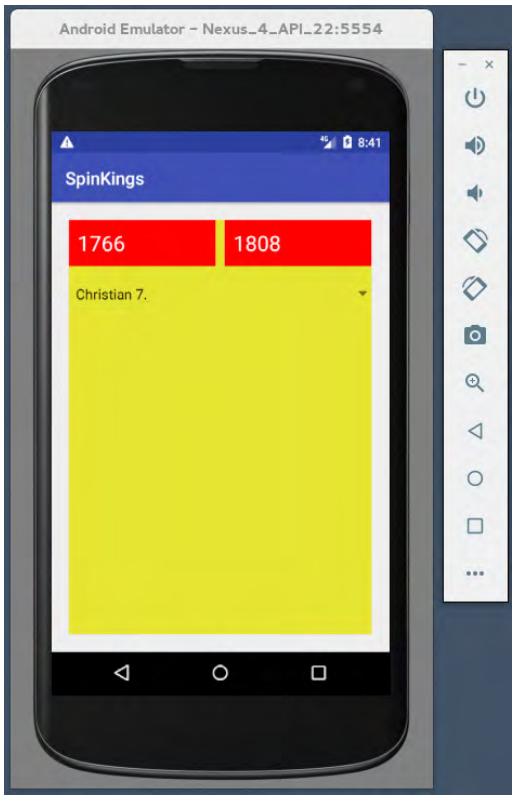
```
android.R.layout.simple_list_item_multiple_choice
```

It says that the *King* objects this time not only should be displayed as strings but with a subsequent checkbox and that it should be possible to select multiple. Instead, if you had only been able to select single, you would instead have written:

```
android.R.layout.simple_list_item_single_choice
```

The bottom *ListView* has in the same way as in the previous example attached an event handler, but it performs something different now. The method *update()* converts the current *King* object into a string, and if this string is already in data source *lines*, it is deleted and otherwise added. An adapter is then used to update the top *ListView*.

As the last example of kings, the program *SpinKings* opens the following window:



The program is almost identical to the program *TheKings*, and there is only one difference that instead of a *ListView*, a *Spinner* is used, which corresponds to a *ComboBox*, as you know it from Swing or JavaFX. The code is essentially the same as well, and for the XML part, there is only one difference in which the *ListView* element has been replaced with a *Spinner* element:

```
<Spinner
  android:id="@+id/lstKings"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:drawSelectorOnTop="true"
  android:layout_marginTop="20dp"/>
```

Regarding the Java code, the two model classes *King* and *Kings* are unchanged, but the class *MainActivity* is changed, and it is only the method *onCreate()*:

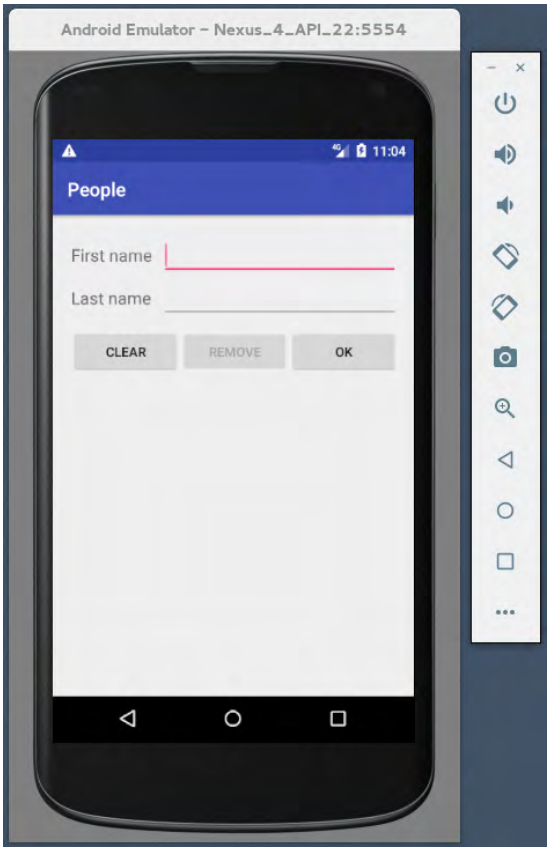
```
protected void onCreate(Bundle savedInstanceState)
{
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 from = (EditText) findViewById(R.id.txtFrom);
 to = (EditText) findViewById(R.id.txtTo);
 disable(from);
 disable(to);
 Spinner view = (Spinner) findViewById(R.id.lstKings);
 view.setAdapter(new ArrayAdapter<King>(this,
  android.R.layout.simple_spinner_item, kings));
 view.setOnItemSelectedListener(new OnItemSelectedListener()
 {
  @Override
  public void onItemSelected(AdapterView<?> parent,
   View view, int position,long id)
  {
   update(position);
  }
  @Override
  public void onNothingSelected(AdapterView<?> parent)
  {
   from.setText("");
   to.setText("");
  }
 });
}
```

Here is the most important of course the event handler, that this time has another listener, which is an interface that defines two methods.

EXERCISE 3: PEOPLE1

Start by creating a copy of the project from exercise 2. You can call the copy for *People1*. You must add a few changes. The user interface must be almost the same. The only difference is that a new button must be added (see below). Once you have entered one or more persons and they have been inserted into the *ListView* container, you must be able to click on a person, and the name of the person must then be inserted into the two top input fields so the names can be edited. When you click OK, no new person should be added, but the list should be updated with the changed name. If you have clicked on a name and it is inserted in the input fields, you must also be able to delete the name by clicking the new button.

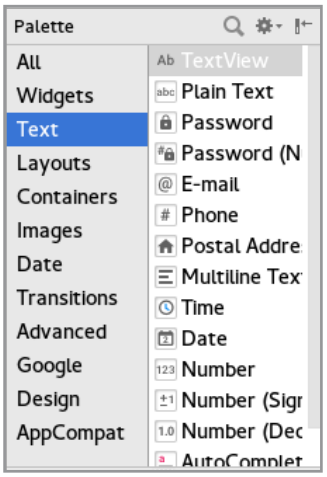
Note that the new button from the start is disabled, and it must be enabled only if a name has been clicked in the list.



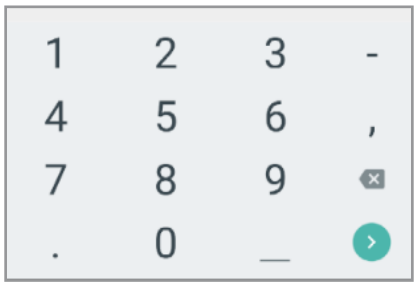
4 USER INTERACTION

In this chapter I will look a little more on the basic user interaction, including how to use menus and dialogs. However, I want to start with the virtual keyboard.

An Android device may well have a hardware keyboard, but most devices, including smartphones, have a virtual keyboard that pops up when you click in an input field. An input field is an *EditText* widget, but if you look at the design editor under the *Text* palette, you will see that there are many options, and each option tells you how to display the content, but primarily which virtual keyboard to use:

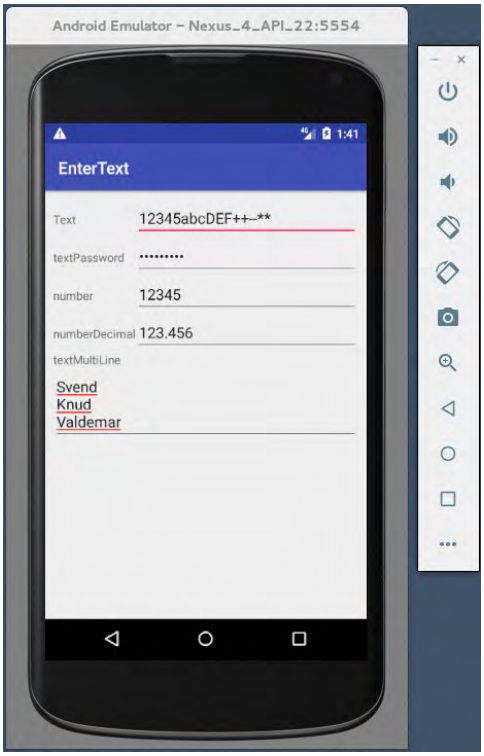


Each line will insert an *EditText* view, and the different options indicate the value of the attribute *InputType*. As you can see, there are many options, and in particular, you should note that there are attributes to typical tasks like entering text, email addresses, numbers, and so on. It is important to note that the attribute *InputType* primarily tells how the virtual keyboard should look and thus which keys there are. For example, if you have chosen *Number*, you will get a keyboard that only has keys to digits, sign on and thus a very limited keyboard:



Note that you can not switch to a keyboard with letters. The program *EnterText* can be used to experiment and show the meaning of the virtual keyboards. The program opens a window that contains 5 *TextView* widgets and 5 *EditText* widgets. I do not want to show *activity_main.xml* here because the only thing to note is the value of the attribute *InputText*, but you should note the following:

- The top one is blank, which means you can enter anything.
- The next has the value *textPassword*, which means that what you enter appears as dots.
- The third has the value *number* and means that you can enter integers.
- The fourth has the value *numberDecimal* so you can enter decimal numbers.
- The latter has the value *textMultiline* and means that you can enter more lines, and then as what you know as a *TextArea*.



As mentioned above, I will not show the XML code as it fills a lot without adding so much new. Basically, it is a *TableLayout* with *TextView* and *EditText* widgets, but the layout manager is placed in a *ScrollView*:

```
<ScrollView
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="10dp">
    …
  </TableLayout>
</ScrollView>
```

This means that the user can scroll the content vertically. If you want to scroll horizontally, use a *HorizontalScrollView* instead.

4.1 A CONTEXT MENU

Menus are frequently used in mobile phone apps, and in this section I will show you how to attach a context menu to a control. As example, I will use the program *TheKings*, as I have shown above, where I want to attach a context menu for the single kings in the *ListView*

component. I have started with a copy of the project, which I have called *AboutKings*. The user interface is exactly the same, so the changes concern only the Java section. Therefore, I do not want to show the XML code for the user interface again.

The class *Kings* contains a static array, where data for Danish kings are defined as constants. For each king there are three values, but the array is expanded with a fourth value, which is a brief description of the current king. The information is obtained from

```
http://kongehuset.dk/monarkiet-i-danmark/kongerakken
```

but regarding the program, the information is not critical (it's just a text) and the text is not translated into English. Similar to this extension, the class *King* is also expanded with an additional attribute that returns that description, and the class is expanded by a method *getKing()*, which returns a string that is the king's name and government period.

Then there's the class *MainActivity*, which is expanded a lot:

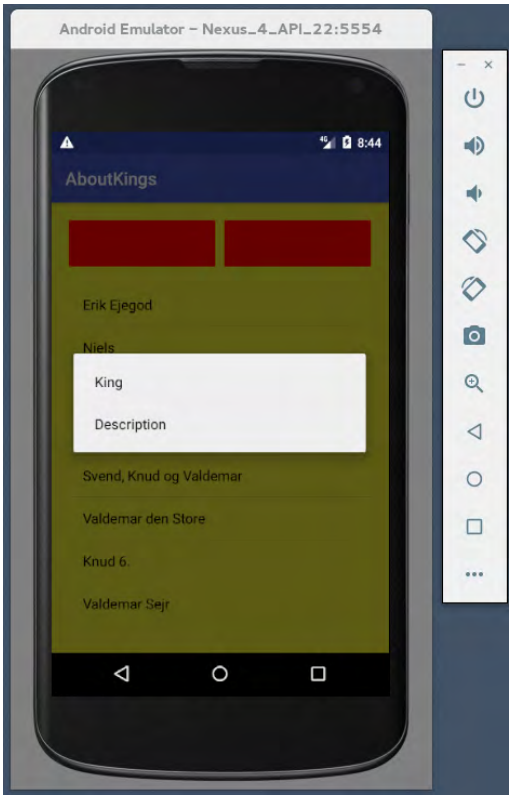
```
public class MainActivity extends AppCompatActivity
{
 public static final int MENU_NAME = Menu.FIRST + 1;
 public static final int MENU_TEXT = Menu.FIRST + 2 ;
 private List<King> kings = (new Kings()).getKings();
 private EditText from;
 private EditText to;

 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  from = (EditText) findViewById(R.id.txtFrom);
  to = (EditText) findViewById(R.id.txtTo);
  disable(from);
  disable(to);
  ArrayAdapter<King> adapter =
   new ArrayAdapter<King>(this, android.R.layout.
    simple_list_item_1, kings);
  ListView view = (ListView) findViewById(R.id.lstKings);
  view.setAdapter(adapter);
  view.setOnItemClickListener(new OnItemClickListener()
  {
   @Override
   public void onItemClick(AdapterView<?> parent, View view,
    final int position, long id) {
    update(position);
   }
  });
  registerForContextMenu(view);
 }
 @Override
 public void onCreateContextMenu(ContextMenu menu, View v,
  ContextMenu.ContextMenuInfo menuInfo)
 {
  menu.add(Menu.NONE, MENU_NAME, Menu.NONE, "King");
  menu.add(Menu.NONE, MENU_TEXT, Menu.NONE, "Description");
 }

 @Override
 public boolean onContextItemSelected(MenuItem item)
 {
  AdapterView.AdapterContextMenuInfo menuInfo =
   (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
  King king = kings.get(menuInfo.position);
```

```
  switch (item.getItemId()) {
   case MENU_NAME:
    Toast.makeText(this, king.getKing(), Toast.LENGTH_LONG).show();
    return (true);
   case MENU_TEXT:
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle(king.getName());
    builder.setMessage(king.getText());
    builder.setPositiveButton("Close", new
      DialogInterface.OnClickListener() {
   public void onClick(DialogInterface dialog, int which) {
   }
    });
    builder.show();
    return (true);
  }
  return (super.onContextItemSelected(item));
 }

 public void update(int position)
 {
  int a = kings.get(position).getFrom();
  int b = kings.get(position).getTo();
  from.setText(a == 0 ? "" : "" + a);
  to.setText(b == 9999 ? "" : "" + b);
 }

 private void disable(EditText view)
 {
  view.setKeyListener(null);
  view.setEnabled(false);
 }
}
```

First, two constants are defined that define two menu items in a context menu. Actually, they must just be unique integers, but the documentation recommends doing it as shown in this example, as Android creates other menu items in other contexts, and they must have a unique identifier within the same activity. The last statement in *onCreate()* says that a context menu must be registered for the program's *ListView* component. If there were other widgets that should have a context menu, they should be registered in the same way.

Then there is the method *onCreateContextMenu()* that creates the menu. In this case, it is simple with two menu items, but of course there could be more. Finally, there is the event handler, which in this case is performed every time a menu item is selected in the context menu. The parameter is the menu item that is selected. First of all, which line is clicked

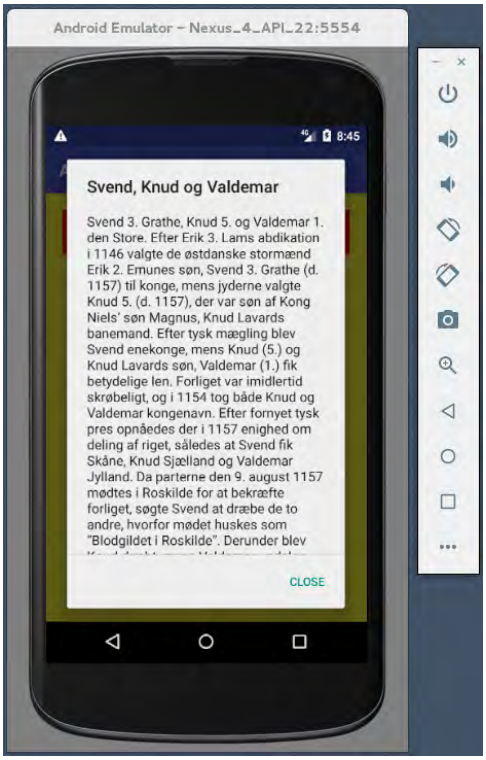
and the corresponding *King* object is determined. If it is the first menu item, nothing happens except a simple Toast with a text that is the king's name and government period. There it is a message that is shown over the window and automatically disappears after a short period of time determined by the devic. All you can control is that a parameter can indicate whether it takes a long time (few seconds = 3.5) or less time (= 2 seconds). Below is the window in which the context menu is activated:



How to activate a context menu depends on the Android version (there are no mouse that you can right-click), but in the newer Android versions it happens when you hold your finger on the component until the menu pops up.

The second menu item opens an alert, which is a simple message box. You create an alert with a sophisticated builder, which, as a parameter, that indicates the activity that should be the owner. The builder then defines what should be in the title line and the message itself. You must then define which buttons there should be. There may be two buttons that you defines with the methods *setPositiveButton()* and *setNegativeButton()*, and in this case there is only one. When you click on a button, the dialog closes no matter what button it is, but an event handler is assigned for the button. In this case it is empty, but in other situations an action could be performed when clicking the button.

As an example, below is shown how an alert looks after clicking on a name. What the text tells is not important, but the text has been chosen so much that the entire text can not be displayed, but you automatically get a scrollbar.



4.2 A MAIN MENU

I will then show how to define a main menu, which is called on an *option* menu in Android. As example, I would like to use a modified version of the application *SelectKings*, which I have called *MoreKings*. If you open the program, you get a window as shown below, where there is a *ListView* that shows the kings with a checkbox, but in the title bar, called the *ActionBar* window, there are now three vertical dots showing that there is a menu. The menu has two menu items, where one menu item deletes all the kings with a checkmark in the checkboxes, while the other menu item performs a reset and thus reinsertes all the kings. The program still has the same context menu as in the previous example.

Due to the latter, the classes *King* and *Kings* are replaced by the corresponding classes from the previous example. *activity_main.xml* is almost identical to the corresponding file from *SelectKings* with the one difference that the one *ListView* component is deleted, so the XML part is now quite simple. The new things to note therefore only concerns *MainActivity.java*:

```
package dk.data.torus.selectkings;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
```

```
import android.util.*;
import android.app.AlertDialog;
import android.content.*;
import java.util.*;

public class MainActivity extends AppCompatActivity
{
 public static final int MENU_NAME = Menu.FIRST + 1;
 public static final int MENU_TEXT = Menu.FIRST + 2 ;
 public static final int MENU_REMOVE = Menu.FIRST + 3;
 public static final int MENU_RESET = Menu.FIRST + 4;
 private List<King> kings;
 private ListView view;
 private ArrayAdapter<King> adapter;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  view = (ListView) findViewById(R.id.lstKings2);
  reset();
  registerForContextMenu(view);
 }
```

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenu.ContextMenuInfo menuInfo)
{
 menu.add(Menu.NONE, MENU_NAME, Menu.NONE, "King");
 menu.add(Menu.NONE, MENU_TEXT, Menu.NONE, "Description");
}

@Override
public boolean onContextItemSelected(MenuItem item)
{
 AdapterView.AdapterContextMenuInfo menuInfo =
  (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
 King king = kings.get(menuInfo.position);
 switch (item.getItemId()) {
  case MENU_NAME:
   Toast.makeText(this, king.getKing(), Toast.LENGTH_LONG).show();
   return (true);
  case MENU_TEXT:
   AlertDialog.Builder builder = new AlertDialog.Builder(this);
   builder.setTitle(king.getName());
   builder.setMessage(king.getText());
   builder.setPositiveButton("Close", new
     DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int which) {
      }
   });
   builder.show();
   return (true);
 }
 return (super.onContextItemSelected(item));
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
 menu.add(Menu.NONE, MENU_RESET, Menu.NONE, "Reset");
 menu.add(Menu.NONE, MENU_REMOVE, Menu.NONE, "Remove");
 return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
 switch (item.getItemId()) {
  case MENU_RESET:
   reset();
   return(true);
  case MENU_REMOVE:
   remove();
```

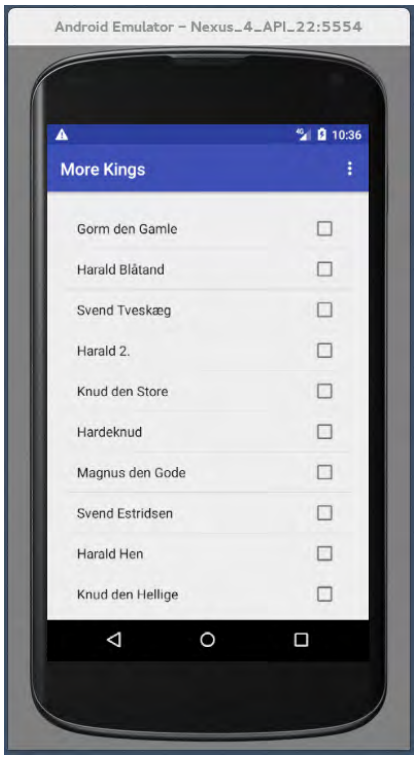
```
      return(true);
    }
    return(super.onOptionsItemSelected(item));
  }

  private void reset()
  {
    kings = (new Kings()).getKings();
    view.setAdapter(adapter = new ArrayAdapter<King>(this,
     android.R.layout.simple_list_item_multiple_choice, kings));
  }

  private void remove() {
    SparseBooleanArray checkedItems = view.getCheckedItemPositions();
    if (checkedItems != null) {
      for (int i = checkedItems.size() - 1; i >= 0; --i) {
       if (checkedItems.valueAt(i)) {
         King king = kings.get(checkedItems.keyAt(i));
         adapter.remove(king);
       }
      }
      for (int i = 0; i < view.getCount(); ++i) view.setItemChecked(i,false);
    }
  }
}
```

When you see the code, there is not so much mystery in it, although it has been changed a lot compared to *SelectKings*. First, note that the four constants that will identify the menu items. In addition, there are three instance variables that refer to the program's data, the *ListView* component, as well as an adapter. The *ListView* component is initialized in the method *reset()*, called from *onCreate()*. The reason is that the method must also be performed when selecting *Reset* in the menu. The two following methods are the methods from the previous example and are used to create the context menu and to the event handler.

The method *onCreateOptionsMenu()* creates the main menu. There is not much to explain except for the parameters and the value *Menu.NONE*. As the name says, of course, it means that you do not have a value for a parameter. In this case, it is a question that menu items can be divided into groups, which is not used here.

The method *onOptionsItemSelected()* is the associated event handler, which is also simple, but it calls the method *remove()* to delete all items selected in the *ListView* component. Here you should especially note how to determine the indexes (positions) of the items that are selected and that the result is a *SparseBooleanArray*, which you can think of as an arrays of indexes on items that are selected. You must also note the last statement that is required to remove checkmarks in the *ListView* component.

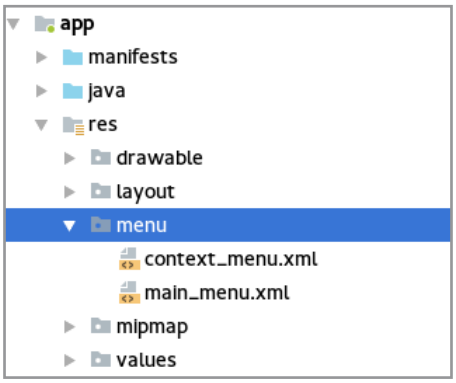
The result, along with the previous program, is that it is simple enough to associate menus with an app.



4.3 DEFINE A MENU IN XML

In the above examples, the menus are defined in Java, and there is nothing wrong with that, but you can also do it as a resource in XML, and in fact, it is recommended that you do so, partly to separate the code and the definition of user interface, and partly to facilitate language versions of an app. In addition, it is very simple.

The project *LastKings* is a copy of the above project, but during the directory *res* I have created a new subdirectory with the name *menu*, that includes two xml documents:



The content of the first document are:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/king_menu"
    android:title="Respond" />
  <item android:id="@+id/text_menu"
    android:title="Description" />
</menu>
```

and the other is in principle identical:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/remove_menu"
    android:title="Remove" />
  <item android:id="@+id/reset_menu"
    android:title="Reset" />
</menu>
```

In the Java code, the methods *onCreateContextMenu()* and *onCreateOptionsMenu()* should be changed

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenu.ContextMenuInfo menuInfo)
```

```
{
 new MenuInflater(this).inflate(R.menu.context_menu, menu);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
 new MenuInflater(this).inflate(R.menu.main_menu, menu);
 return(super.onCreateOptionsMenu(menu));
}
```

and the 4 constants for the menu items can be deleted. The event handlers must also be changed so that they do not use the constants but instead the identifiers from the XML documents. As an example, below is shown one of the two event handlers:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
 switch (item.getItemId()) {
  case R.id.reset_menu:
   reset();
   return(true);
  case R.id.remove_menu:
   remove();
   return(true);
 }
 return(super.onOptionsItemSelected(item));
}
```

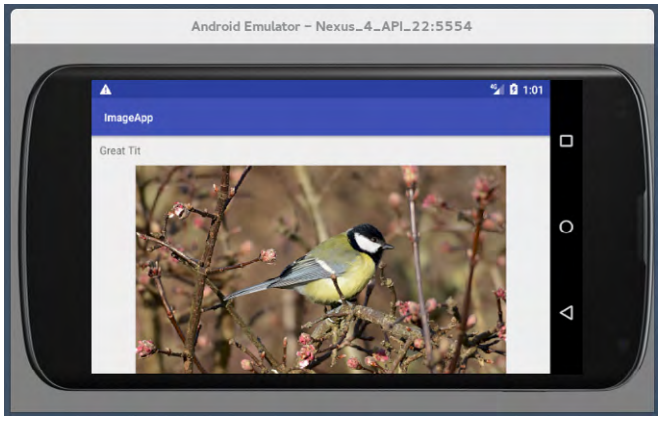
and then everything works again.

EXERCICE 4: IMAGEAPP

Create a new project that you can call *ImageApp*. The book's source code has a folder called images containing four images. Copy these images to your project:

```
ImageApp/app/src/main/res/drawable
```

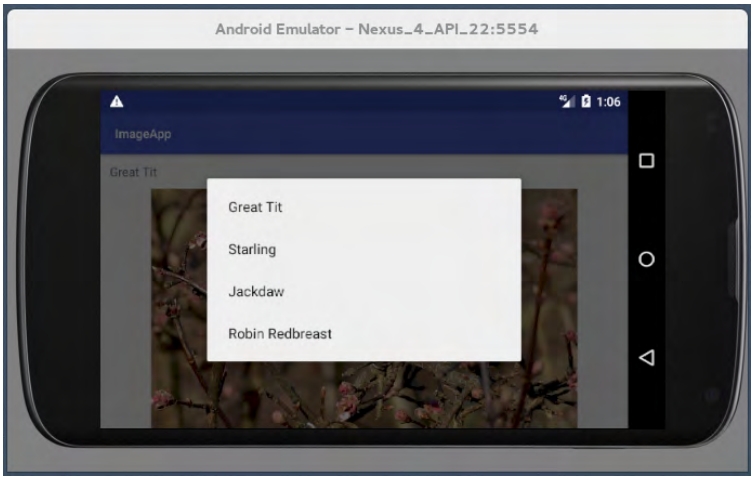
The images are *img1.jpg*, *img2.jpg*, *img3.jpg* and *img4.jpg*, and after copying the images, they are added as resources to your project. The project must (in the emulator) open the following window, which has two components in the form of a *TextView* and an *ImageView*:



In the file *strings.xml*, you must define 4 strings, which are the names of the birds that the images shows:

```
<resources>
  <string name="app_name">ImageApp</string>
  <string name="lbl_img1">Great Tit</string>
  <string name="lbl_img2">Starling</string>
  <string name="lbl_img3">Jackdaw</string>
  <string name="lbl_img4">Robin Redbreast</string>
</resources>
```

To the *TextView* component, a context menu has to be defined with four menu items (se below). The menu must be defined in XML, and if you select a menu item, the program must display the corresponding image and update the *TextView* component with the correct name.



Once you have tested the program, you can optionally replace the pictures with your own pictures and maybe have 6 images instead of 4.

4.4 ABOUT ACTIVITIES

I want to conclude this chapter by looking a little closer to the class *Activity*, and as shown in the previous examples, each mobile app has an Activity object. One might think of an *Activity* as an object that represents the program's window and implements the logic that the application uses, and a program may therefore have more activities. An activity can be in four modes

1. *Launched*, when an activity comes in to existence, usually as the result of a user performing some action to starts the application.
2. *Running* at which the users actually sees your activity, after all of the various setup steps are performed.
3. *Killed*, that is a state in which the Android OS has determined to terminate the appplication.
4. *Shut down* at which all persistent state information is deleted.

Changes are made on the basis of callback methods, where *onCreate()* is an example, but there are some others, and it's all methods that you can override. In the following, I will mention the main callback methods.

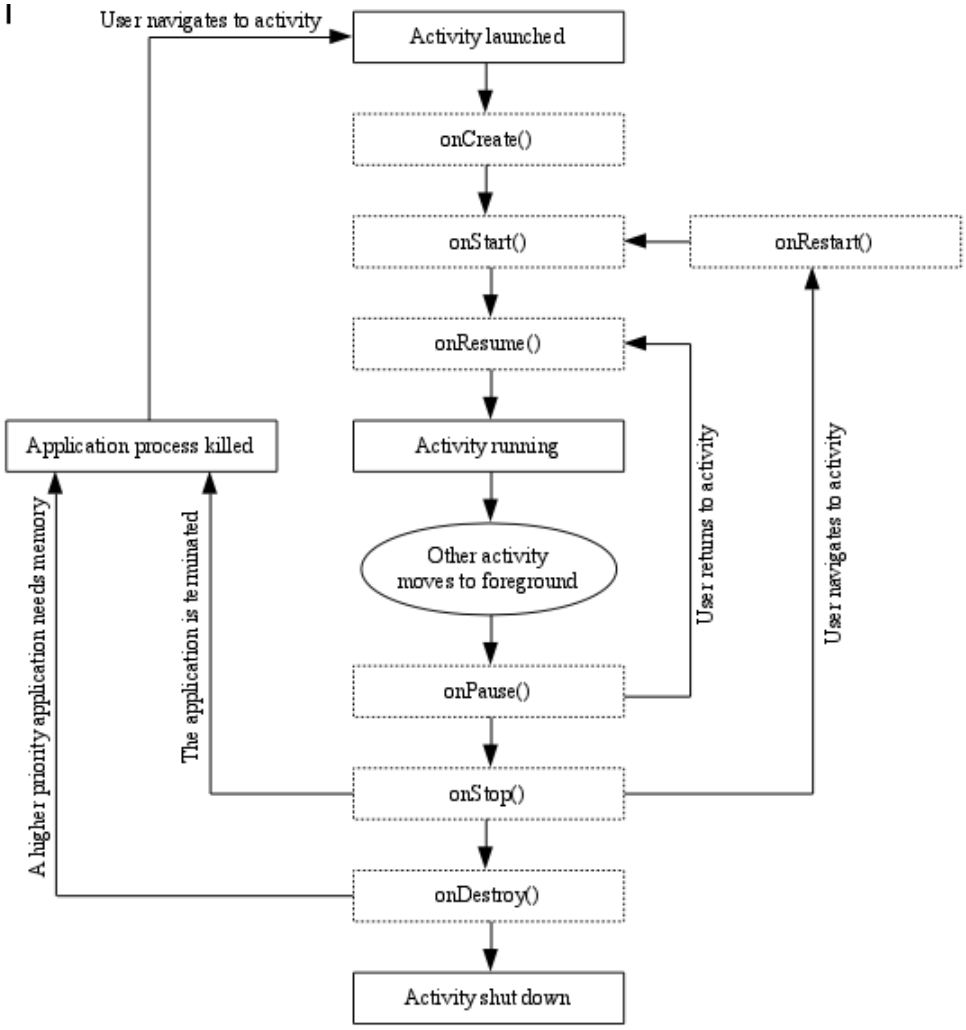
onCreate() is executed when an activity starts its life cycle, typically because a user has requested the activity or the activity is restarted due to changes in the environment, for example due to a rotation of the devic. It is for restart that *onCreate()* has a parameter of the type *Bundle*, that can contain information about the state of the activity. Basically, *onCreate()* is used for:

1. Load the layouts that the activity uses so the components in the user interface can be constructed in the method *onStart()*.
2. Initialize the activitiy's instance variables.

There are some tasks that relate to *onCreate()* and primarily concerns restart of the activity, including keeping track of existing global resources from the last time the activity was active. For example, if you programmatically change the layout depending on an user interaction of the application.

onStart(), that is executed after *onCreate()* has created the required objects from the layout definitions and performed other initialization work, it is *onStart()* which is tasked with presenting the current UI elements for the user. There is seldom the need to override this method. And if you do, it is important to call *super.onStart()*.

onResume() is executed immediately before the user interface becomes visible and the method performs almost the same as *onStart()*, but is performed in conjunction with another state switch as you can se in the following state diagram:



onPause() is executed after an application is switched to the background state after Android has suspended the application. Viewed from the programmer, the method can be used to ensure data is saved when the application is suspended.

onStop() sends the activity in background, and you can think of the method as the opposite of *onStart()* and the activity is no longer visible to the user, while the view and properties still exist and are transferred to *onResume()* and *onStart()* if they are performed. Note that *onStop()* is not necessarily performed, if the Android decides to terminate the application, and it is therefore rare to override this method.

onRestart() is executed at a state shift from stopped to start, then allowing to some extend to decide how to restart the activity.

onDestroy() is performed when an activity is closed, but note that it concerns an activity and not the entire application. The method can only be used for a cleanup that relates to the current activity.

The above state diagram should show where and when the different callback methods are performed in connection with state shift. In the above examples I have only used *onCreate()*, but as the examples use more of the phone's facilities, more of the above callback methods are also interesting.

There are several methods that relates to change of state for an activity and can be overwritten, and as example I would like to mention two:

1. *onRestoreInstanceState()* that is executed when a user leaves an activity by clicking the back button or performing other logic that is part of the application and the method can be used in situations where there is a need to save user data.
2. *onSaveInstanceState()* which is responsible for storing an activity's state information, primarily associated with *onDestroy()*.

4.5 MULTIPLE ACTIVITIES

As the last in this chapter, I will show an example of an app that has two activities. The application is trivial and should only show how to add more activities to an application. You must note that an application can have all the activities that are needed.

I have started with a project called *MoreActivities*, and after I have created the project, there is one activity called *MainActivity*, where the layout is called *activity_main.xml*. Next, I have under project files right-click on *app* and selected

New | *Activity* | *Empty Activity*

and in the next window I have for the names selected *SecondActivity* and *activity_second.xml*. As a result, my project now contains two activities.

For the first (that is *MainActivity*) I have added an event handler:

```
package dk.data.torus.moreactivities;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;
import android.view.*;

public class MainActivity extends AppCompatActivity
{

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

 public void openActivity(View view)
 {
  Intent intent = new Intent(this, SecondActivity.class);
  startActivity(intent);
 }
}
```

There is not much to explain besides taking note that the handler opens another activity, and the result is that the phone's screen changes content and shows the other activity. The XML code and thus the layout are as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/activity_main"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_margin="20dp"
  tools:context="dk.data.torus.moreactivities.MainActivity">
  <TextView
    android:id="@+id/lbl_main"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:text="Go to the second activity" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Open Activity"
    android:layout_below="@id/lbl_main"
    android:layout_marginTop="50dp"
    android:layout_centerHorizontal="true"
    android:onClick="openActivity" />
</RelativeLayout>
```

which is an extremely simple layout with a text and a button.

For the other activity, almost the same thing must happen. The XML code is in principle identical. Just another text appears and the button has another event handler:

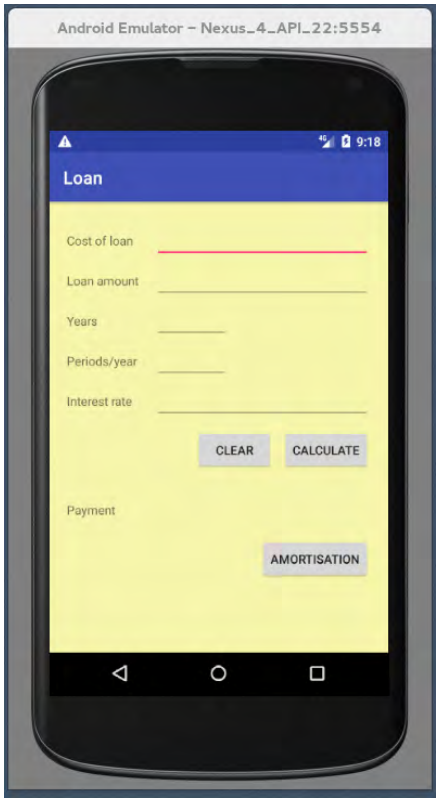
```
public class SecondActivity extends AppCompatActivity
{
 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_second);
 }
```

```
 public void closeActivity(View view)
 {
  finish();
 }
}
```

That's all, and then the application can be translated and tested, and the result is an application with two activities.

PROBLEM 1: LOANAPP

In the book Java 3 (and also elsewhere) I have shown an application that implements a loan calculator where the user can calculate the payment on a loan. This time the loan every time is an annuity loan and the program must open the following window:

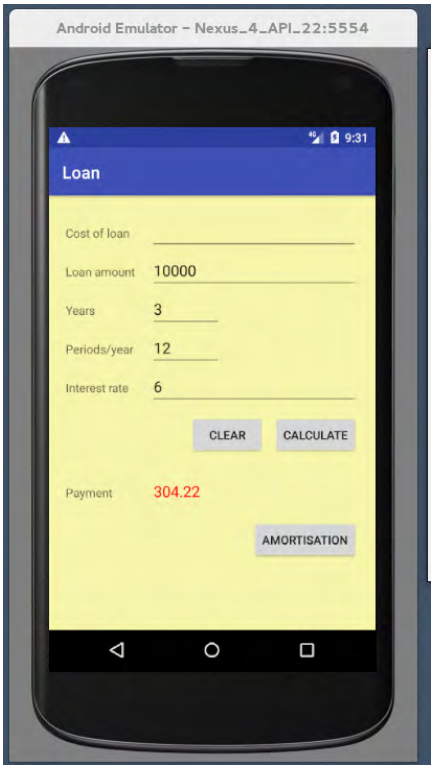


which contains only labels, entry fields and buttons. In the two top fields one must be able to enter the borrowing costs and the loan principal respectively, while you in the next two fields you must be able to enter the number of years for the repayment of the loan and the number of terms per year. In the last field you must be able to enter the interest rate, which is entered as the interest rate in percent per year. When you click the *CALCULATE* button, the program must calculate the term payment and show it in the field after the text *Payment*, which is a read-only *EditText*.

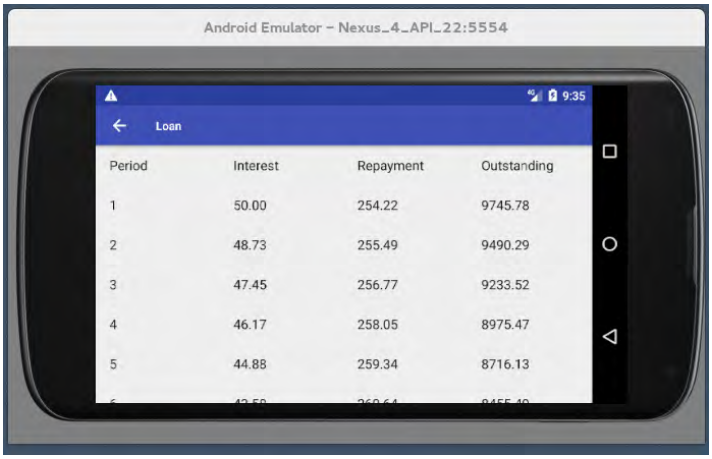
Before calculating (when clicked on the *CALCULATE* button), the program must validate data:

1. The cost must not be negative
2. The loan must be positive
3. The number of years must be an integer between 1 and 60 (both inclusive)
4. The number of terms per year must be an integer between 1 and 12 (both inclusive)
5. The interest rate must be positive and less than 50

In the case of an error, the program must display an error message in the form of a simple Toast. Below is shown the window after a loan calculation has been made:



If you click on the *AMORTIZATION* button, the program must open another activity that shows an amortization plan:



Here you should especially note that the window's Action Bar has a back button. It's part of the task to find out how to add this button – it's quite simple and you can try to search the Internet. The actual table is shown with a *GridView*. It's also part of the task to find out how to use a *GridView*, but it's almost the same as a *TextView*.

5 FILES

In view of the above examples, the subjects shown can now be a good move in developing mobile phone apps, but the phone provides a range of other services beyond what is applicable to ordinary computers, where I can mention SMS and phone, GPS, camera and more. These are facilities that I will first treat in the following book, but it is also possible to save data in files and databases, which is necessary for many apps, and in this and the following chapter I will focus on how to do that. This chapter will look at files, including how to manipulate the file system just like on a regular PC, but I want to start somewhere else, namely files that you only can read. If there are files that only have interest in the current application, the files can be saved as resource files within the application. Examples could be text, images, audio and more.

5.1 READING RESOURCES

If you create a project in Andoid Studio, under project files there will be two directories *java* and *res*, where the latter contains resource files. It are files that are packed with the

program code in the *apk* file. Here you will find, among other things, a subdirectory *layout* that contains *activity_main.xml* that defines the application's user interface and at the same location will be other xml documents for layout of other activities. You have also met the file *strings.xml* under the subdirectory *values*. Making data available to the program in this way has several advantages, such as:

- It is easy to define data, that is always available for the program.
- It ensures a separation of code and data.
- It allows to place resources in a library so that they can be used by multiple applications.
- Simple data can be made available in a standard format such as XML.

There are also disadvantages, among other things, to mention:

- Data is fundamentally read-only, although in principle it is possible to modify the content of a resource.
- It can be problematic/cumbersome to ensure data is up to date.

To show an example of how to do, I will return to the example with the Danish kings, which represent data that rarely changes. It is therefore data that can be conveniently placed as a resource, but also because it is data that does not fill up much. The data in question is stored in an XML document named *regents.xml* and has the following content:

```
<regents>
 <regent name="Gorm den Gamle" to="958"/>
 <regent name="Harald Blåtand" from="958" to="987"/>
 <regent name="Svend Tveskæg" from="987" to="1014"/>
 <regent name="Harald 2." from="1014" to="1018"/>
 …
 <regent name="Frederik 9." from="1947" to="1972"/>
 <regent name="Margrethe 2." from="1972"/>
</regents>
```

I have then created a project, which I have called *DanishRegents*. The layout defines only a single component:

```
<ListView
  android:id="@+id/lstReg"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

and the only thing to do is to fill this *ListView* with the names of the Danish kings. In order to make data available, I have created a subdirectory named *raw* and copied file *regents.xml* to this subdirectory. Then the document is a resource for the application. I have then added a model class *Regent*, which can represent a regent, and apart from the name, the class is identical to the class *King* that I used previously. Back there is only the class *MainActivity*:

```
package dk.data.torus.danishregents;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
import java.util.*;
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import android.widget.AdapterView.*;

public class MainActivity extends AppCompatActivity
{
 List<Regent> regents = new ArrayList();

 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  try
  {
   InputStream stream = getResources().openRawResource(R.raw.regents);
   DocumentBuilder builder =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
   Document doc = builder.parse(stream, null);
   NodeList nodes = doc.getElementsByTagName("regent");
   for (int i = 0; i < nodes.getLength(); ++i)
   {
    Element elem = (Element)nodes.item(i);
    String attr = elem.getAttribute("from");
    int from = attr == null || attr.length() == 0 ?
     Integer.MIN_VALUE : Integer.parseInt(attr);
    attr = elem.getAttribute("to");
    int to = attr == null || attr.length() == 0 ?
     Integer.MAX_VALUE : Integer.parseInt(attr);
    regents.add(new Regent(elem.getAttribute("name"), from, to));
   }
   stream.close();
  }
```

```
  catch (Exception e) {
  }
  ListView view = (ListView)findViewById(R.id.lstReg);
  view.setAdapter(new ArrayAdapter<Regent>(this,
   android.R.layout.simple_list_item_1, regents));
  view.setOnItemClickListener(new OnItemClickListener()
  {
   @Override
   public void onItemClick(AdapterView<?> parent, View view,
    final int position, long id)
   {
    Regent regent = regents.get(position);
    Toast.makeText(MainActivity.this, regent.getRegent(),
     Toast.LENGTH_LONG).show();
   }
  });
 }
}
```

First, note that there are a number of new import statement, among other things because the program should read and parse a XML document. The class has a single instance variable, which is a *List* for *Regent* objects, and thus to the objects that the *ListView* component should display. Otherwise, the class only has an *onCreate()*. It starts by loading the resource.

Here you should note how to refer to a program's resources using the method *getResources()* and how to load a raw file using the method *openRawResource()* and that the result is a regular *InputStream*. With this stream available, a *DocumentBuilder* can read the document and parse it as XML, after which you can create the current *Regent* objects and initialize the list. After the data is loaded and the list is initialized, an adapter is added to the *ListView* component. Additionally, an event handler has been added so that the program shows a *Toast* if you click on a name in the *ListView* component.

EXERCISE 5: ZIPCODES

In section 2.1 I shows an application *Zipcodes*, where you can search for Danish zip codes. Start by creating a copy of the project *Zipcodes*. Under the directory *res* you must create a subdirectory named *raw*. The book's source code contains a file *zipcodes.txt*, which contains the Danish zip codes as a comma-separated file. Copy this file to the new directory *raw*. In the class *Zipcodes*, all zip codes are defined in a static array. You must now delete this array and change the class so that the Zipcode objects are created by instead reading the content of your resource *zipcodes.txt*. You can do the following:

1. Delete the static array
2. Add a parameter of the type *InputStream* to the class' constructor
3. The constructor must then initialize the list of zip codes by reading the content of the stream represented by the parameter

A single change must be added to the class *MainActivity*. In *onCreate()* you must create a reference to your resource and send it as a parameter to the constructor in the class *Zipcodes*. Then everything should work again.

5.2 ORDINARY FILES

You can also use common files, as you know it from PC, and not enough, it goes a long way in the same way as described in the book Java 5 about IO. The main difference is that files can be stored in two places, either internally where they are stored together with the program in internal storage or external where they can be stored on an external SD card. I only want to show the first option because the ability to save files remotely to a SD card is a security problem with Android, and that's why Google has tightened the requirements for external files significantly and they have probably not found the final solution yet. It's actually possible to open up a device so that you can easily create files on a SD card, but it is not recommended, so I'm looking only at the internal files, but with respect to the code, the difference is not big.

When files are created internally, they are stored in the phone's memory which in this context is called internal storage. It has its pros and cons where I can mention:

- Internal storage is something that always are there and is part of any Android device, but it's limited in size and significantly less than what, an SD card makes available and there's a not insignificant risk that if applications often saves files in internal storage, so that it will be filled up.
- Basically, files stored in internal storage are associated with an application and can only be used immediately by this application. Should the file be used by other applications, special efforts are required, and in fact, it is not at all the intention to share information between applications using internal files.
- Files stored in internal storage are perceived as part of the application they are associated with, and if you delete the application, you also deletes its internal files.

Internal files are thus primarily suitable for storing smaller amounts of data that must be available for a program the next time it starts up, but there is also a need for that in many examples, so the conclusion is not to avoid using internal files, but just think about what they are used for.

As mentioned, internal files are associated with an application, and the application generally has unlimited access to a file and can thus both read and write the file – the application simply owns the file.

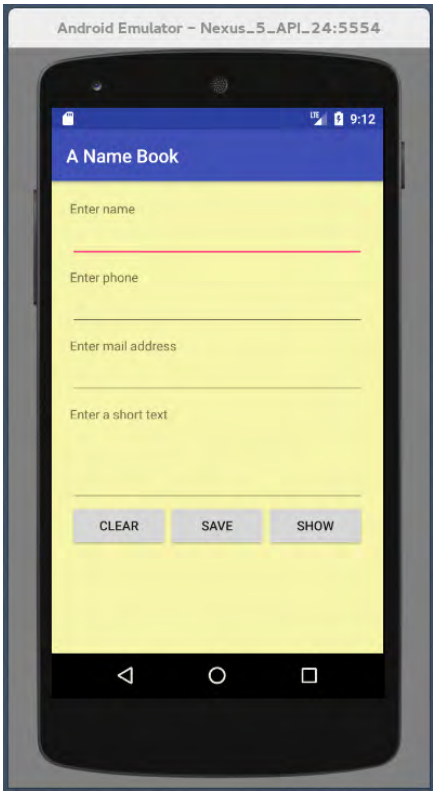
When you open a file, you can specify the following modes:

- MODE_PRIVATE, that is default and only the application that creates the file is allowed to access the file
- MODE_APPEND

In the following, I will look at an example that shows how to use a file.

5.3 INTERNAL STORAGE

The program is a simple contact program, where you can enter a name, a phone number, an email address and a text for a person:

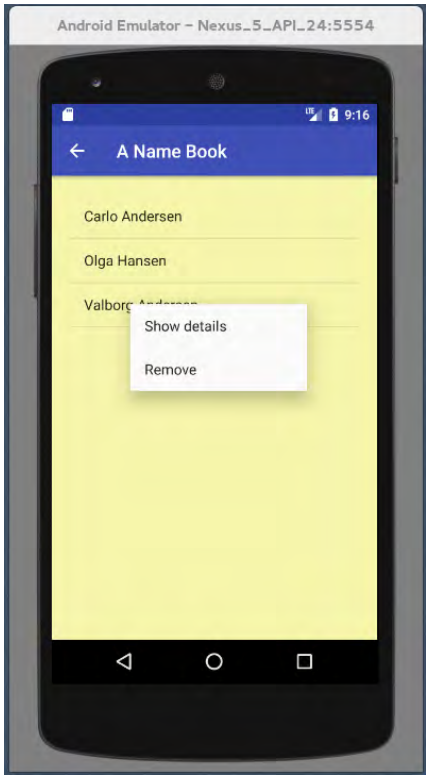

Android Emulator - Nexus_5_API_24:5554
9:12
A Name Book
Enter name
Enter phone
Enter mail address
Enter a short text
CLEAR
SAVE
SHOW

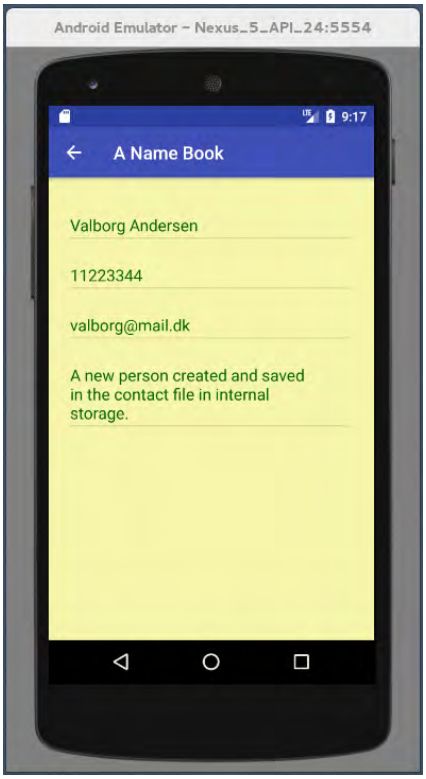
Once you have entered a contact (the name must be entered) you can save the contact by saving it to a file in internal storage. If you click the *SHOW* button, you will see the window below that shows an overview of the contacts that were entered. If you hold your finger on a contact, you will get a context menu with two functions, where you can delete a contact (and then delete in the file) and choose to see the details for the contact (see the window below). It is therefore an application with three activities.

I do not want to look at the XML code for the three layouts, as here is nothing new, so the following mainly concerns the Java code. Here are two things that you should notice:

1. how to write to a file, which is the primary purpose of the example
2. how to transfer a parameter from one activity to another

The program uses the following model class where I have not shown get and set methods:






```
public class Person implements Serializable
{
 private String name;
 private String phone;
 private String mail;
 private String text;

 public Person(String name, String phone, String mail, String text)
 {
  this.name = name;
  this.phone = phone;
  this.mail = mail;
  this.text = text;
 }

 public Person(String line) throws Exception
 {
  String[] elems = line.split("\f");
  if (elems.length != 4) throw new Exception("Illegal text for person");
  name = elems[0];
  phone = elems[1].trim();
  mail = elems[2].trim();
  text = elems[3].trim();
 }

 …
```

```
 public String getPerson()
 {
  StringBuilder builder = new StringBuilder(name);
  builder.append("\f");
  builder.append(phone.length() == 0 ? "\t" : phone);
  builder.append("\f");
  builder.append(mail.length() == 0 ? "\t" : mail);
  builder.append("\f");
  builder.append(text.length() == 0 ? "\t" : text);
  return builder.toString();
 }

 @Override
 public String toString()
 {
  return name;
 }
 public boolean equals(Object obj)
 {
  …
 }
}
```

The class represents a person with the four desired properties. It is the idea that contacts should be saved as text lines in a file, and it is therefore necessary to separate the fields so you can read in the file again. A seperation mark must be used, which is just a character that does not appear in the text fields, and here I use a form feed (page shift). The method *getPerson()* thus returns a string with the person's fields separated by form feeds. You should note that if a field is empty (what the fields may be if it is not the name), a tab will be saved. That's what you can call a "work around", and the reason is the *split()* method in the *String* class ignores blank fields.

The class has two constructors, and here you should first notice the last one that has a *String* as a parameter and assumes that the *String* consists of 4 fields separated by form feeds. The fields are determined by separating strings into fields using the *split()* method, and note that for the last three fields a *trim()* is performed, and the purpose is to remove any tab inserted by the above method.

Finally, you must note that the class is defined *Serializable*. The reason is that it should be possible to transfer a *Person* object as parameter from one activity to another.

This class is a major part of the work of saving person objects in a file. Reading and writing to a file takes place in the class *Persons*, which represents all contacts:

```
public class Persons implements Serializable
{
 private static final String filename = "papersons";
 private List<Person> persons = new ArrayList();
 private transient Context context;

 public Persons(Context context) throws Exception
 {
  this.context = context;
  FileInputStream stream = null;
  try {
   stream = context.openFileInput(filename);
  }
  catch (Exception ex)
  {
   return;
  }
  BufferedReader reader = new BufferedReader(new
   InputStreamReader(stream));
  for (String line = reader.readLine(); line
   != null; line = reader.readLine())
   try
   {
```

```
    persons.add(new Person(line));
   }
   catch (Exception ex)
   {
   }
  reader.close();
 }

 public Person getPerson(int n)
 {
  return persons.get(n);
 }

 public List<Person> getPersons()
 {
  return persons;
 }

 public void save(Person person) throws Exception
 {
  FileOutputStream stream =
   context.openFileOutput(filename, Context.MODE_APPEND);
  BufferedWriter writer = new BufferedWriter(new
   OutputStreamWriter(stream));
  writer.write(person.getPerson());
  writer.newLine();
  writer.close();
  persons.add(person);
 }

 public void remove(Context context, Person person) throws Exception
 {
  if (persons.remove(person))
  {
   FileOutputStream stream =
    context.openFileOutput(filename, Context.MODE_PRIVATE);
   BufferedWriter writer = new BufferedWriter(new
    OutputStreamWriter(stream));
   for (Person p : persons)
   {
    writer.write(p.getPerson());
    writer.newLine();
   }
   writer.close();
  }
  else throw new Exception("Person could not be removed");
 }
}
```

As you study the code, you should first notice that all of the basics are the same as you know it from IO in Java, and that even the same classes are used. The class is defined *Serializable* as it should be transferred as parameter to an activity. The class has three variables, where the first is the filename, the other an *ArrayList* to *Person* objects, and the latter is a *Context* that represents an activity. It is used when creating a file as it is associated with an activity. Note that it is defined *transient*. The reason is that it should not be serialized when a *Person* object is transferred to another activity.

The class' constructor loads the file's lines and parses them to *Person* objects. Basically, it is similar to the same classes that you generally use to read a text file in Java. There is only one place where there is something new and that's where the file opens:

```
stream = context.openFileInput(filename)
```

This happens with the method *openFileInput()*, which is a method in the *context* object that opens a file in internal storage and returns an *InputFileStream* object. Then it all goes as with other text files in Java.

When saving an entered contact, you call the method *save()* with a *Person* object as parameter. Here's not much new, and the only is how to open the file:

```
context.openFileOutput(filename, Context.MODE_APPEND)
```

which opens an output file in internal storage and again it is with a method in the *context* object. The method returns a *FileOutputStream* object. The file opens in append mode to add to the file and if the file does not exist, it will be automatically created.

Finally, there is the method *remove()* to delete a *Person* in the file. The file is this time opened as

```
context.openFileOutput(filename, Context.MODE_PRIVATE)
```

which means that an existing content is overwritten. The method works by writing the entire content of the list *persons* back into the file (but not the object that is deleted). This is necessary because it is a sequential file, and something similar would apply if you could change the information for a contact.

Then there is the class *MainActivity*:

```
public class MainActivity extends AppCompatActivity {
 private Persons persons;
 private EditText txtName;
 private EditText txtPhone;
 private EditText txtMail;
 private EditText txtText;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  loadPersons();
  txtName = (EditText)findViewById(R.id.txtName);
  txtPhone = (EditText)findViewById(R.id.txtPhone);
  txtMail = (EditText)findViewById(R.id.txtMail);
  txtText = (EditText)findViewById(R.id.txtText);
 }

 public void onClear(View view)
 {
  clear();
 }
```

```
public void onSave(View view)
{
 String name = txtName.getText().toString().trim();
 String phone = txtPhone.getText().toString().trim();
 String mail = txtMail.getText().toString().trim();
 String text = txtText.getText().toString().trim();
 if (name.length() > 0)
 {
  try
  {
   persons.save(new Person(name, phone, mail, text));
   Toast.makeText(this, getResources().getString(R.string.msg_save),
     Toast.LENGTH_LONG).show();
   clear();
  }
  catch (Exception ex)
  {
   Toast.makeText(this, getResources().getString(R.string.err_save),
     Toast.LENGTH_LONG).show();
  }
 }
}

public void onShow(View view)
{
 Intent intent = new Intent(this, ShowActivity.class);
 intent.putExtra("persons", persons);
 startActivity(intent);
}

private void clear()
{
 txtName.setText("");
 txtPhone.setText("");
 txtMail.setText("");
 txtText.setText("");
 txtName.requestFocus();
}

private void loadPersons()
{
 try
 {
  persons = new Persons(this);
 }
 catch (Exception ex)
 {
  Intent home = new Intent(Intent.ACTION_MAIN);
```

```
    home.addCategory(Intent.CATEGORY_HOME);
    home.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    startActivity(home);
   }
  }
}
```

In *onCreate()* not much else happens than the method instantiates variables to the 4 input fields, but the method calls *loadPersons()*, which initializes the variable *persons* by creating a new *Persons* object, thus reading all contacts in the file. Note the parameter, which is the *Context* object. The constructor in the class *Persons* can raise an exception if the file for one reason or another can not be read correctly. If so, there is nothing more to do than stop the application, and the method can be perceived as a standard way to stop an app.

In addition to the method *onCreate()*, the class primarily has three event handlers for the three buttons. Here, *onSave()* is the most comprehensive, as it must create a *Person* object corresponding to what the user has entered. Otherwise, the method does not do much else than to call the method *save()* in the class *Persons*, and finally the method shows a *Toast* that tells whether the contact was saved or not.

The event handler *onShow()* must open another activity, and it happens the same way as shown earlier with one difference:

```
intent.putExtra("persons", persons);
```

It indicates that the object *persons* must be sent as a parameter and identified by the key *persons*. The class *Intent* has many overrides of *puExtra()*, including overrides for all the simple types and strings, but there is also an override for a serializable object, which is why the classes *Person* and *Persons* are defined *Serializable*.

Then there is the class *ShowActivity*, which is the window that shows an overview of all contacts:

```
public class ShowActivity extends AppCompatActivity {
 private Persons persons;
 private ListView view;
 private ArrayAdapter<Person> adapter;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_show);
  Intent intent = getIntent();
  persons = (Persons)intent.getSerializableExtra("persons");
  view = (ListView) findViewById(R.id.lstView);
  reset();
  getSupportActionBar().setDisplayHomeAsUpEnabled(true);
  registerForContextMenu(view);
 }

 @Override
 public boolean onSupportNavigateUp(){
  finish();
  return true;
 }

 @Override
 public void onCreateContextMenu(ContextMenu menu, View v,
  ContextMenu.ContextMenuInfo menuInfo)
 {
  new MenuInflater(this).inflate(R.menu.context_menu, menu);
 }

 @Override
 public boolean onContextItemSelected(MenuItem item) {
```

```
  AdapterView.AdapterContextMenuInfo menuInfo =
   (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
  Person person = persons.getPerson(menuInfo.position);
  switch (item.getItemId()) {
   case R.id.details_menu:
    Intent intent = new Intent(this, DetailActivity.class);
    intent.putExtra("person", person);
    startActivity(intent);
    return(true);
   case R.id.remove_menu:
    try
    {
     persons.remove(this, person);
     reset();
     Toast.makeText(this, getResources().getString(R.string.msg_rem),
      Toast.LENGTH_LONG).show();
    }
    catch (Exception ex)
    {
     Toast.makeText(this, getResources().getString(R.string.err_rem),
      Toast.LENGTH_LONG).show();
    }
    return(true);
  }
  return (super.onContextItemSelected(item));
 }

 private void reset()
 {
  view.setAdapter(adapter = new ArrayAdapter<Person>(this,
   android.R.layout.simple_list_item_1, persons.getPersons()));
 }
}
```

The code fills a part, but it is primarily the event hander for the window's context menu that fills. Note how the method *onCreate()* determines the parameter transferred from *MainActivity*:

```
Intent intent = getIntent();
persons = (Persons)intent.getSerializableExtra("persons");
```

where it is important to use the same key and the correct type of cast. Also note that *onCreate()* defines that the *Action Bar* for the window should have a back button:

```
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
```

and corresponding override the method *onSupportNavigationUp()*. The context menu is defined as a resource, and the class must implement *onConextItemSelected()*. The method fills a lot, but it does not contain anything new. You should note that this time you transfer a *Person* object to the activity *DetailActivity*. I do not want to show the code for *DetailActivity*, as there is nothing new.

EXERCISE 6: NAMEBOOK

You shoul expand the *NameBook* example, such that you can also change the information about a contact. Start with a copy of the project *NameBook*. You must add a new activity to the project that can be used to edit a contact. Note that this activity should have a layout similar to *MainActivity*, except that you can not edit the name and there should only be a single button. The new activity should be opened by expanding the context menu with a new menu item in *ShowActivity*. In addition, the class *Persons* must be expanded with a new method so you can update the changes. It should be done in the same way as deletion, where all activates are saved.

There is a single challenge, namely when to save the changes. The easiest thing is to do that in *ShowActivity*, but it requires that the new activity can return a value when you click *Save* and the window is closed. You should solve the problem that way, especially because it may be useful in other contexts. To solve the problem, follow these steps:

In *ShowActivity* you must add a constant (the value is not important):

```
private static final int PERSON_UPDATED = 1;
```

You must then add the following method:

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
 if (requestCode == PERSON_UPDATED)
 {
  if (resultCode == RESULT_OK) {
   // here you can save the changes to the file
   Person person = (Person)data.getSerializableExtra("person");
   …
   }
  }
 }
}
```

The method *onActivityResult()* is now a method that can be called from your new activity when it ends and wishes to send a value back. It requires the new activity to be opened as:

```
startActivityForResult(intent, PERSON_UPDATED);
```

In your new activity, you can then write the event handler for the *Save* button as follows:

```
public void onSave(View view)
{
 person.setPhone(txtPhone.getText().toString().trim());
 person.setMail(txtMail.getText().toString().trim());
 person.setText(txtText.getText().toString().trim());
 Intent data = new Intent();
 data.putExtra("person", person);
 setResult(RESULT_OK, data);
 finish();
}
```

PROBLEM 2: PUZZLEAPP

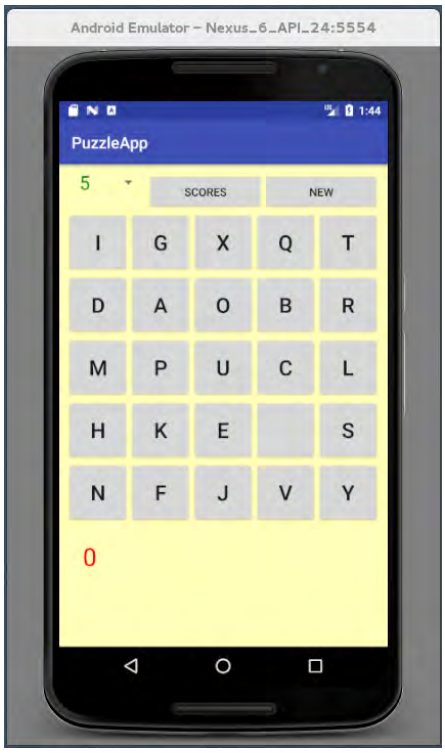
As the final example in the book Java 4, I have shown a program that simulates a simple game, which consists of a square of 5 × 5 pieces arranged in random order. One of the pieces is empty, and you can move a piece by moving a neighbor to the empty piece. In the case below (the square on the left) you can then move the pieces H, E, L and J (you can not move diagonally). The game is solved when the pieces are arranged as shown in the example to the right.

K	G	V	M	R
I	S	X	U	F
C	O	A	E	T
Q	Y	H		L
N	D	P	J	B

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	

The square is basically a 5 × 5 square, but it must in the same way as in Java 4 be possible to configure the program so that you can also play with a 3 × 3 square, a 7 × 7 square and a 9 × 9 square. Also the program must have a high score list where to save the five best results. A result consists of the number of movements, used to solve the square, and it is therefore to solve the square by moving as few pieces as possible.

The program should work in the same way as in Java 4, but must be written to a mobile phone this time and the user interface could be something like the following:

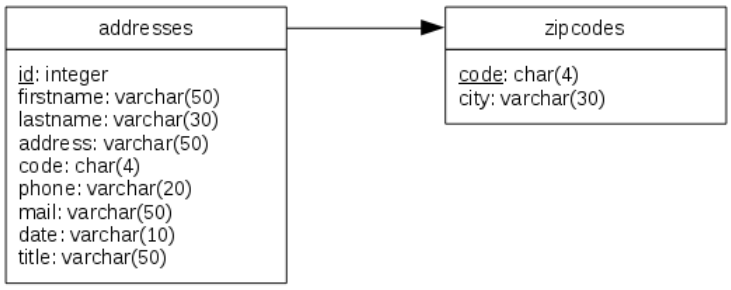


You can reuse much of the code from Java 4 – if not directly, then using smaller adjustments. Of course, the two challenges are the user interface, which needs to be written in a different way, and to save the high score list that must be saved in local file.

6 SQLITE

You can also write apps for mobile phones that use databases, because Android is born with a very simple database called *SQLite*. The database is also widely used in many other places than on Android devices. It's a simple database (what exactly is also the goal) and it has far from all the facilitators that other database products (like *MySQL*, *Oracle*, etc.) make available, but it's a complete SQL database where you can create local databases and performs the usual database operations – almost as you know it from *MySQL*. In this chapter, I will show you how to use *SQLite* through an example.

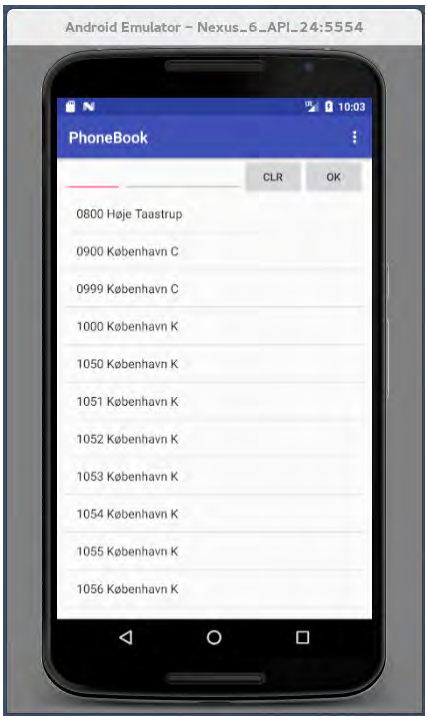
As example, I will write an app called *PhoneBook* and it corresponds to the *Contact* app with which the phone is born, but it is not intended as an alternative to that application, but only to show the use of a database, and it should also be a example of a slightly larger app and thus a slightly more realistic app than what the previous examples have shown. The application's data must be stored in a database with two tables:

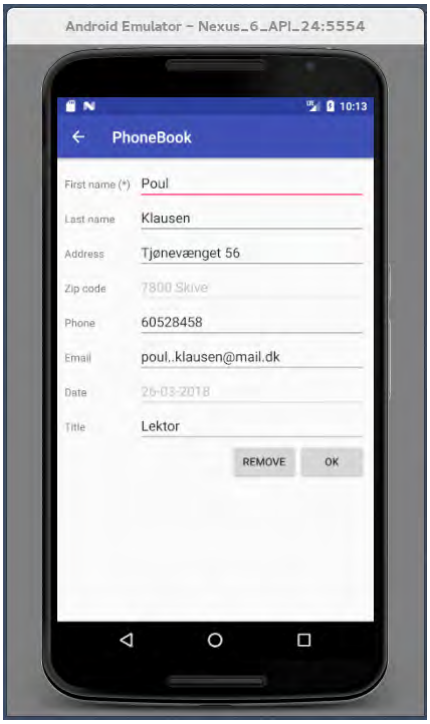


where one table is a table with Danish zip codes, while the other contains contact information about persons. The column *code* in table *addresses* is a foreign key to the table *zipcodes*. When you open the application, you get a window that shows an overview of zip codes (see below). At the top there are a two input fields and two buttons that act as a filter for zip codes. In general, contacts are organized by zip codes and in order to find a contact, hold your finger on a zip code, after which you get a context menu, where you can either view the contacts organized under that zip code or add a new contact. If you choose to view existing contacts, there are three options:

1. There are no contacts, and you just get a simple Toast that tells you.
2. There is a single contact and the program opens a window that displays the informations (see below). In this window you can also edit the contact and delete it.
3. You will get a window that shows all contacts organized under the current zip code. I have not shown the window here, but it is nothing but a *ListView* with the names of those contacts. If you here click on a name, you will come to the window with all the contact informations.

As the last concerning the user interface, *MainActivity* has an option menu with a single menu item. Clicking on it opens an activity where you can enter search criteria (first name, last name, address and title), and the program will find the contacts that match the search criteria. The result has the same three options as described above.





It was the program's functionality, and then how the program is written. There are no less than 4 activities, but none of them contain anything new, so I do not want to display the XML code for the layout, and the following only concerns the Java code and here primarily how to

1. creates a database
2. performs a SQL INSERT
3. performs a SQL UPDATE
4. performs a SQL DELETE
5. performs a SQL SELECT

and the following will be centered on these 5 topics.

The program must be initialized with the Danish zip codes, and to make it easy, I have added a resource:

raw/zipcodes.txt

which is a comma-separated file with the Danish zip codes.

6.1 CREATES A DATABASE

You can create a database in two ways. You can do it extern on the developer machine, and then copy the database to the application as a resource, but it can easily lead to difficulties if the database has to be updated later. For that reason, there is a helper class that makes it easy to create the database in the code and ensures that the database is automatically created the first time the application is used, but also ensures that the database is updated if you change it later – assuming that you have added the required code. In the present case, I have added the following class:

```
package dk.data.torus.phonebook;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import java.io.*;

public class DbHelper extends SQLiteOpenHelper
{
 public static final java.util.Random rand = new java.util.Random();
 public static final String ZTABLE_NAME="zipcodes";
 public static final int ZCOLNO_CODE = 0;
 public static final int ZCOLNO_CITY = 1;
 public static final String[] ZTABLE_COLUMNS = new String[]{ "code", "city" };
 public static final String ATABLE_NAME="addresses";
 public static final int ACOLNO_ID = 0;
 public static final int ACOLNO_FIRRSTNAME = 1;
 public static final int ACOLNO_LASTNAME = 2;
 public static final int ACOLNO_ADDRESS = 3;
 public static final int ACOLNO_CODE = 4;
 public static final int ACOLNO_PHONE = 5;
 public static final int ACOLNO_MAIL = 6;
 public static final int ACOLNO_DATE = 7;
 public static final int ACOLNO_TITLE = 8;
 public static final String[] ATABLE_COLUMNS =
  new String[]{ "id", "firstname", "lastname", "address", "code", "phone",
   "mail", "date", "title" };
 private static final String DBFILENAME="phonebook.db";
 private static final int DBVERSION = 1;
 private static final String ZINITIAL_SCHEMA =
  "create table zipcodes (code char(4) primary
   key, city varchar(30) not null)";
 private static final String AINITIAL_SCHEMA =
  "create table addresses (" +
  "id integer primary key autoincrement," +
  "firstname varchar(50) not null," +
  "lastname varchar(30)," +
```

```
  "address varchar(50)," +
  "code char(4) not null," +
  "phone varchar(20)," +
  "mail varchar(50)," +
  "date varchar(10)," +
  "title varchar(50),"+
  "foreign key (code) references zipcodes (code))";
 private Context context;

 public DbHelper(Context context) {
  super(context, DBFILENAME, null, DBVERSION);
// super(context, DBFILENAME, null, rand.nextInt(20) + 1);
  this.context = context;
 }

 @Override
 public void onCreate(SQLiteDatabase db) {
  db.execSQL(ZINITIAL_SCHEMA);
  db.execSQL(AINITIAL_SCHEMA);
  db.execSQL(insertZipcodes());
 }

 @Override
 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
//    db.execSQL("DROP TABLE IF EXISTS zipcodes");
//    db.execSQL("DROP TABLE IF EXISTS addresses");
//    db.execSQL(ZINITIAL_SCHEMA);
//    db.execSQL(AINITIAL_SCHEMA);
//    db.execSQL(insertZipcodes());
 }

 @Override
 public void onDowngrade(SQLiteDatabase db,
int oldVersion, int newVersion) {
//    db.execSQL("DROP TABLE IF EXISTS zipcodes");
//    db.execSQL("DROP TABLE IF EXISTS addresses");
//    db.execSQL(ZINITIAL_SCHEMA);
//    db.execSQL(AINITIAL_SCHEMA);
//    db.execSQL(insertZipcodes());
 }

 private String insertZipcodes()
 {
  InputStream stream = context.getResources().
   openRawResource(R.raw.zipcodes);
  StringBuilder builder =
   new StringBuilder("insert into zipcodes (code, city) values ");
  try (BufferedReader reader = new BufferedReader(new
```

```
   InputStreamReader(stream)))
  {
   addRow(builder, reader.readLine());
   for (String line = reader.readLine(); line
    != null; line = reader.readLine())
   {
    if (line.length() > 0)
    {
     builder.append(",");
     addRow(builder, line);
    }
   }
  }
  catch (Exception e) {
  }
  return builder.toString();
}

private void addRow(StringBuilder builder, String line)
{
 String[] elems = line.split(",");
 builder.append("('");
 builder.append(elems[0]);
```

```
    builder.append("','");
    builder.append(elems[1]);
    builder.append("')");
   }
  }
```

Initially, a random generator has been defined and it should not be part of a the finished application, but I will explain the stetement in a while. Otherwise, the class defines a large number of constants that defines the database. These constants are not absolutely necessary, but they can make the following code easier to read and understand, and they can make the maintenance of the code significantly easier. It is therefore advisable to introduce these constants, and in this case there are two sets of constants for each of the two database tables. When you see the constants, it's easy enough to understand the meaning. Note that there is a constant for the database name itself, and note that there is a constant for the version number. Also note that there are constants for the two SQL expressions that create the database tables.

The class inherits *SQLiteOpenHelper*, and the constructor in the base class checks, if there is a database with that name. If not, it will perform *onCreate()*. If the database is found, the version number is tested and it is larger than the database version number the method *onUpgrade()* is performed. Is the version number smaller than the database version number the method *onDowngrade()* is performed. A *SQLiteOpenHelper* class should therefore override these three methods – however, in many cases you do not override the last one (the version number will always be higher).

In this case, *onCreate()* creates the two tables. Note that it is the constructor in the base class that creates the database, while *onCreate()* creates the tables. *onCreate()* calls the method *insertZipcodes()*. It uses a *StringBuilder* to dynamically create a *SQL INSERT* using the resource with the zip codes, and the method returns the result as a string. *onCreate()* has a parameter of the type *SQLiteDatabase* and it has methods, such as header *execSQL()*, which performs a SQL statement. The method is used to create the two tables and can generally be used to execute a SQL statement represented as a string and thus also the result of *insertZipcodes()*.

The program also overrides the other two methods from the base class: *onUpgrade()* and *onDowngrade()*. Both are trivial as all the code has been commented out, but the meaning is that you have to insert the code to be used if the version number is increased (until you define the final version). When you develop a program, you will typically be interested in creating the database each time you run the program. This can be enforced by increasing the version number, and then adding the code to *onUpgrade()*, which deletes the existing tables and re-creates them. That is the goal of my random generator that every time the program runs, it gets a random version number (see the constructor) and the result is that either *onUpgarde()* or *onDownload()* is performed. You may, of course, remember to remove this action before applying the application, otherwise you delete the database every time the application is running.

6.2 SQL SELECT

To the project has been added two model classes *Zipcode* and *Person*, which represent a zip code and a contact, respectively. Both classes are simple and consist only of get and set methods, but you should note that both classes are defined *Serializable*, so that objects of the classes can be transferred as parameters to other activities.

There is also a class called *Zipcodes*, which represents all zip codes:

```
package dk.data.torus.phonebook;

import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import java.util.ArrayList;
import java.util.List;

public class Zipcodes implements java.io.Serializable
{
 private List<Zipcode> zipcodes = new ArrayList();

 public Zipcodes(SQLiteDatabase db)
 {
  try {
   Cursor cursor = db.query(DbHelper.ZTABLE_NAME, DbHelper.ZTABLE_COLUMNS,
    null, null, null, null, null);
   for (cursor.moveToFirst(); !cursor.isAfterLast(); cursor.moveToNext())
    String code = cursor.getString(DbHelper.ZCOLNO_CODE);
    String city = cursor.getString(DbHelper.ZCOLNO_CITY);
    zipcodes.add(new Zipcode(code, city));
   }
   cursor.close();
  }
  catch (Exception ex)
  {
   zipcodes.clear();
  }
 }

 public List<Zipcode> getZipcodes()
 {
  return zipcodes;
 }
}
```

The class has a single instance variable, which is used for the *Zipcode* objects. The list is initialized in the constructor, whose parameter is an open database. The constructor starts

by creating a *Cursor*, which is an object that represents a SQL SELECT and can be used to traverse the result. In this case, a SELECT statement is performed using the method *query()*, which is quite a complex method

```
query(String table,
      String columns[],
      String selection,
      String selectionArgs,
      String groupBy,
      String having,
      String orderBy,
      String limit)
```

The method has a parameter for each of the elements that can be included in a SELECT statement, and above, I have only set values for the first two parameters, since the statement should retrieve all zip codes in the database. You should note how the table name and column names are defined using the constants in the class *DbHelper*. After the statement has been completed, you can traverse the rows using the *Cursor* object, an object that basically works in the same way as a *ResultSet*.

There is also a class called *Persons* that, in principle, works in the same way, but where the constructor instead determines a list of *Person* objects:

```
package dk.data.torus.phonebook;

import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import java.util.ArrayList;
import java.util.List;

public class Persons implements java.io.Serializable
{
 private List<Person> persons = new ArrayList();

 public Persons(SQLiteDatabase db, Zipcode zipcode)
 {
  try {
   String[] params = new String[]{zipcode.getCode()};
   Cursor cursor =
    db.rawQuery("select * from addresses where code = ?", params);
   for (cursor.moveToFirst(); !cursor.isAfterLast(); cursor.moveToNext()) {
    int id = cursor.getInt(DbHelper.ACOLNO_ID);
    String fname = cursor.getString(DbHelper.ACOLNO_FIRRSTNAME);
    String lname = cursor.getString(DbHelper.ACOLNO_LASTNAME);
    String addr = cursor.getString(DbHelper.ACOLNO_ADDRESS);
    String phone = cursor.getString(DbHelper.ACOLNO_PHONE);
    String mail = cursor.getString(DbHelper.ACOLNO_MAIL);
    String date = cursor.getString(DbHelper.ACOLNO_DATE);
    String title = cursor.getString(DbHelper.ACOLNO_TITLE);
    persons.add(new Person(id, fname, lname, addr, zipcode, phone,
     mail, date, title));
   }
   cursor.close();
  }
  catch (Exception ex)
  {
   persons.clear();
  }
 }

 public Persons(SQLiteDatabase db, String firstname, String lastname,
  String address, String persontitle)
 {
  try {
   Cursor cursor = db.query(DbHelper.ATABLE_NAME, DbHelper.ATABLE_COLUMNS,
    "firstname like ? and lastname like ? and
     address like ? and title like ?",
```

```
    new String[] { firstname + "%", lastname + "%", "%" + address + "%",
      "%" + persontitle + "%" }, null, null, null);
   for (cursor.moveToFirst(); !cursor.isAfterLast(); cursor.moveToNext())
   {
    int id = cursor.getInt(DbHelper.ACOLNO_ID);
    String fname = cursor.getString(DbHelper.ACOLNO_FIRRSTNAME);
    …
    persons.add(new Person(id, fname, lname, addr, getZipcode(db, code),
     phone, mail, date, title));
   }
   cursor.close();
  }
  catch (Exception ex)
  {
   persons.clear();
  }
 }
 public List<Person> getPersons()
 {
  return persons;
 }

 private Zipcode getZipcode(SQLiteDatabase db, String code)
 {
  Cursor cursor = db.query(DbHelper.ZTABLE_NAME, DbHelper.ZTABLE_COLUMNS,
   "code = ?", new String[] { code }, null, null, null);
  cursor.moveToFirst();
  return new Zipcode(cursor.getString(DbHelper.ZCOLNO_CODE),
   cursor.getString(DbHelper.ZCOLNO_CITY));
 }
}
```

Here is the most important thing the two constructors which based on search criteria determines a number of contacts in the database. The difference is how they define the SQL SELECT statement.

The first finds all contacts that have a specific zip code, but this time the *Cursor* object is created using a method *rawQuery()*. It has two parameters, the first being a regular SELECT statement written as a string. In the same way as you have seen before, the SQL statement may have parameters as indicated by ? and the last parameter for the method *rawQuery()* should be an array of values to be substituted for the ? placeholders. After the statement is completed, the *Cursor* uses the object in the same way as above, but this time, *Person* objects are created. Note how the individual columns are referenced with constants from the class *DbHelper*.

The other constructor has in addition to a reference to an open database four parameters, all of which are strings. The constructor must find all contacts where the first name starts with a certain value, where the last name starts with a certain value, where the address contains a certain value and the title contains a particular value. The *Cursor* object is created this time using the method *query()*, but the first four parameters are used to define a WHERE part. The actual WHERE part is a string (an expression), and the following parameter is an array of strings to the expression's placeholders. Otherwise, the two constructors seems to work identically, but note that in the last case it is necessary to determine the zip code by reading the database.

6.3 SQL INSERT, UPDATE AND DELETE

The application has an activity called *PersonActivity*, which is used to display contact details (see the window above), but also used to create new contacts, edit existing contacts, and delete contacts. The code is therefore similarly comprehensive:

```
package dk.data.torus.phonebook;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;
import android.widget.*;
import android.view.*;
import android.database.sqlite.SQLiteDatabase;
import android.content.ContentValues;
import java.util.*;

public class PersonActivity extends AppCompatActivity {
 private Zipcode zipcode = null;
 private Person person = null;
 private EditText txtZip;
 private EditText txtFname;
 private EditText txtLname;
 private EditText txtAddr;
 private EditText txtPhone;
 private EditText txtMail;
 private EditText txtDate;
 private EditText txtTitle;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_person);
  Intent intent = getIntent();
  txtFname = (EditText) findViewById(R.id.txtFname);
  txtLname = (EditText) findViewById(R.id.txtLname);
  txtAddr = (EditText) findViewById(R.id.txtAddr);
  txtPhone = (EditText) findViewById(R.id.txtPhone);
  txtMail = (EditText) findViewById(R.id.txtMail);
  txtDate = (EditText) findViewById(R.id.txtDate);
  txtTitle = (EditText) findViewById(R.id.txtTitle);
  txtZip = (EditText) findViewById(R.id.txtZip);
  Object obj = intent.getSerializableExtra("zipcode");
  if (obj != null) {
   zipcode = (Zipcode) obj;
   Calendar cal = Calendar.getInstance();
   txtDate.setText(String.format("%02d-%02d-
    %d", cal.get(Calendar.DAY_OF_MONTH),
     cal.get(Calendar.MONTH) + 1, cal.get(Calendar.YEAR)));
  }
  else
  {
   person = (Person)intent.getSerializableExtra("person");
   txtFname.setText(person.getFirstname());
```

```
   txtLname.setText(person.getLastname());
   txtAddr.setText(person.getAddress());
   txtPhone.setText(person.getPhone());
   txtMail.setText(person.getMail());
   txtDate.setText(person.getDate());
   txtTitle.setText(person.getTitle());
   zipcode = person.getZipcode();
  }
  getSupportActionBar().setDisplayHomeAsUpEnabled(true);
  txtZip.setText(zipcode.toString());
  disable(txtDate);
  disable(txtZip);
 }

 @Override
 public boolean onSupportNavigateUp(){
  finish();
  return true;
 }

 public void onOk(View view)
 {
  String fname = txtFname.getText().toString().trim();
  if (fname.length() > 0)
  {
   String lname = txtLname.getText().toString().trim();
   String addr = txtAddr.getText().toString().trim();
   String phone = txtPhone.getText().toString().trim();
   String mail = txtMail.getText().toString().trim();
   String date = txtDate.getText().toString().trim();
   String title = txtTitle.getText().toString().trim();
   DbHelper dbHelper = new DbHelper(this);
   SQLiteDatabase db = dbHelper.getWritableDatabase();
   ContentValues values = new ContentValues(8);
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_FIRRSTNAME], fname);
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_LASTNAME], lname);
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_ADDRESS], addr);
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.
    ACOLNO_CODE], zipcode.getCode());
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_PHONE], phone);
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_MAIL], mail);
   values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_TITLE], title);
   if (person == null)
   {
    values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_DATE], date);
    db.insert(DbHelper.ATABLE_NAME, null, values);
   }
   else
   {
```

```
    Calendar cal = Calendar.getInstance();
    values.put(DbHelper.ATABLE_COLUMNS[DbHelper.ACOLNO_DATE],
      String.format("%02d-%02d-%d", cal.get(Calendar.DAY_OF_MONTH),
      cal.get(Calendar.MONTH) + 1, cal.get(Calendar.YEAR)));
    String[] args = { "" + person.getId() };
    db.update(DbHelper.ATABLE_NAME, values, "id = ?", args);
   }
   db.close();
   onSupportNavigateUp();
  }
}

public void onRem(View view)
{
 if (person != null)
 {
  DbHelper dbHelper = new DbHelper(this);
  SQLiteDatabase db = dbHelper.getWritableDatabase();
  String[] args = { "" + person.getId() };
  db.delete(DbHelper.ATABLE_NAME, "id = ?", args);
  db.close();
  onSupportNavigateUp();
 }
}
```

```
 private void disable(EditText view)
 {
  view.setKeyListener(null);
  view.setEnabled(false);
 }
}
```

The class defines several variables, but for the sake of the code, the most important are the first two, which define a *Zipcode* and a *Person* object, respectively. The other variables are components in the user interface and are initialized in *onCreate()*. When the relevant activity is opened, a parameter is transferred, either a *Zipcode* object (if you want to create a new contact) or a *Person* object (if you want to view or edit a contact). *onCreate()* starts by referencing a *Zipcode* object, and if this reference is not null, it is a *Zipcode* object that has been transferred and the variable *zipcode* is initialized (while the variable *person* is still *null*). Next, the field *txtDate* is set to *date*, as this field must contain the date of last change of the contact. If *obj* is zero, it is because the transferred parameter is a *Person* object, and the variable *person* is then initialized with this object. It is the object to be displayed in the user interface, and all fields are initialized with the object's values. Finally, note that the *txtDate* and *txtZip* fields are read-only as these fields can not be changed.

The class defines two event handlers *onOk()* and *onRem()*, where the first one is used when the *OK* button is clicked while the last one is used when clicking on the button *REMOVE*. *onOk()* tests whether a first name has been entered (it is assumed that a contact must have a first name). If so, the handler determines the other values entered in the user interface, and then create an object that represents the database. Here you must note the syntax as well as the database is opend *writeable*. Next, a *ContentValue* object is created containing the values to be written to the database, whether it is because a new row must be created or it is an existing row to be updated. Note again the syntax and note how to refer to the individual columns using the constants defined in the class *DbHelper*. Next, the variable *person* is tested, and if it is *null*, it is an INSERT and otherwise it should be an UPDATE. Basically, it happens the same way, in the one case, you use the method *insert()* method, while in the second case, you use the method *update()*, except that in the latter case you should specify a WHERE part.

Finally, there is the last event handler used to delete a contact and thus perform a SQL DELETE. It's basically done just in the same way as above, but where you must use the method *delete()*.

6.4 THE OTHER ACTIVITIES

The aplicationen has three additional activities. *PersonsActivity* displays a list of contacts found in the database, and it's a simple activity with a *ListVew*, and clicking on a contact *PersonActivity* is shown. The Java code contains nothing regarding databases. The activity *SearchActivity* is also simple and is used to enter search criteria, and the code is executed a SQL SELECT, but using the class *Persons*. Finally, there is *MainActivity*, which binds it all together and fills a lot. I do not want to display the code for these three activities, as they do not contain anything new regarding databases, and by the way, nothing new.

EXERCISE 7: PHONEBOOK

In this exercise you will need to make some changes to the above program *PhoneBook*, and the goal is to add some small improvements, but primarily to work a bit more with database programming and *SQLite*. Start by creating a copy of the project.

As the first change, expand the table *addresses* with an additional column, which you can call *text* and whose type should be *VARCHAR(1000)*. The idea is that it should be possible to associate a short description with a contact. You must start changing the *DbHelper* class:

1. delete the variable *rand* at the start of the class
2. delete the line with the comment in the constructor
3. delete the method *onDowngrade()*
4. add an *int* constant with the name *ACOLNO_TEXT* and the value 9 to the class
5. add a *String "text"* to the array *ATABLE_COLUMNS*
6. update the SQL expression (the *String AINITIAL_SCEMA*) such that it creates a column with the name *text* and the type *varchar(1000)*.

Next, you must change the method *onUpgrade()* to the following:

```
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
 if (newVersion > oldVersion) {
  try {
   db.execSQL("ALTER TABLE addresses ADD COLUMN
    text VARCHAR(1000) DEFAULT ''");
  }
  catch (Exception ex)
  {
  }
 }
}
```

Test the program. The result should now be that the database table *addresses* have a new column, but it still contains the old contacts.

You must then expand the program such that you can also enter a text, which means that the window for entering and editing contacts must be expanded with an input field:

You can do the following:

1. Expand the class *Person* with an additional *String* property that you can call *text*. It is also necessary to expand the two constructors with an extra parameter for the new property.
2. The class *Persons* must be changed so it now takes into consideration that the database table has a new column and the class *Person* a new property.
3. The *activity_person.xml* layout should be expanded with the new field (and a label), which should be a multiline *EditText*.
4. *PersonActivity* must be updated so that the value of the new field is stored in the database.

Test application to make sure the text is saved. Remember to test both creating new contacts and editing existing contacts.

As the program is written right now, all texts are hard-coded, ie texts for labels, texts for buttons, toast messages and menu items. Replace these texts with constants in *string.xml*, while defining other texts for the 4 activity title lines, so you can see where in the application you are working.

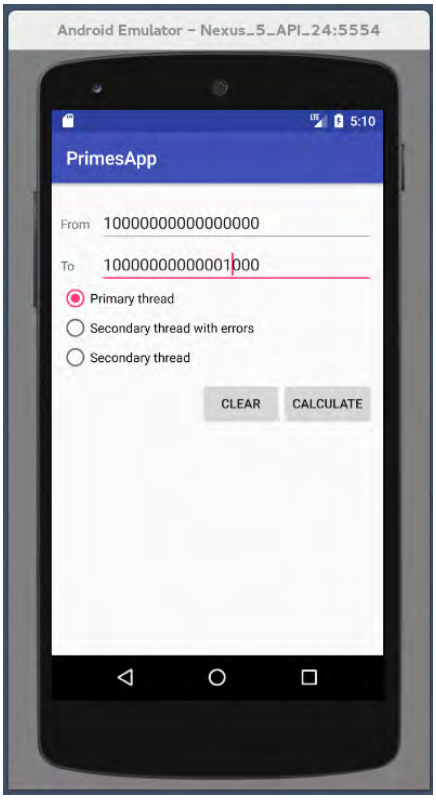
Currently, you can not change the zip code. It's probably a little too restrictive. Solve this problem when the field of the zip code always must contain a legal zip code.

The application has another problem that you may not discover before installing the application on a physical device. The problem is that the activity *PersonActivity* has so many fields that they are covered by the virtual keyboard. You can solve this problem by placing the window layout in a *ScrollView*.

7 THREADS

Android is a multi-threaded operating system, so you can use threads almost the same way you know it from PC. Once an Android application is started, the runtime system will create a primary thread, and all the application's components will as defaulted be running in this thread. The primary thread's main task is to handle the user interface, including processing events, but the thread can of course perform any other operation, whichever is the default. Like what you know from PC, the primary thread can start a secondary thread, and there may be many good reasons for which are the same as you know from earlier, but there are also the same challenges that if a secondary thread directly updates the user interface, so the program will stop with an exception. If you use threads, you need a little more, and the purpose of this chapter is to show what is needed.

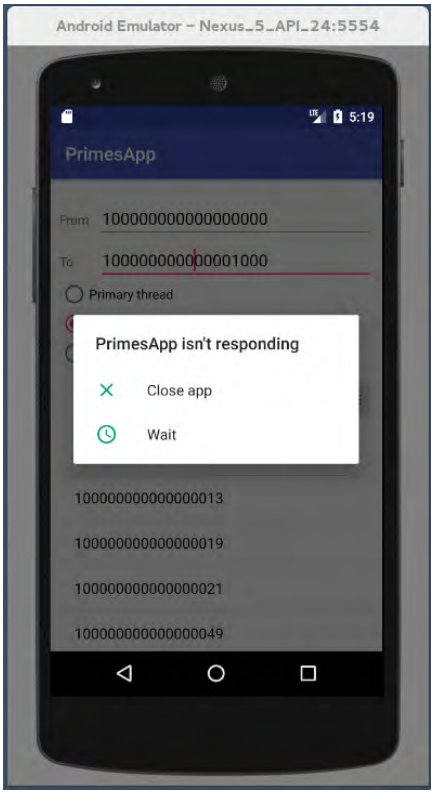
I will use an example that I have used earlier, which is an application that opens the following window:



In the two input fields, you can enter two positive integers, and click on *CALCULATE*, the program will determine all prime numbers between the two entered numbers and insert them into a *ListView* under the buttons. The idea to use the prime numbers is that for large numbers it takes a long time to determine whether a number is prime and if you

have entered as large numbers as above, the result is that the program intermittently adds a prime to the *ListView* component. To limit the memory consumption, the two entered numbers must only differ by 1000, but you can of course change if you want to.

Under the input fields there are three radio buttons. If you perform the calculation and the top radio button is pressed, the calculations are performed in the primary thread and the meaning is that you will find that the application is not responding and if the numbers are sufficiently large, you will receive a warning as shown below where you can interrupt the program:



If the program is allowed to perform the calculations, then after all the prime numbers are found, it will definitely update the *ListView* component with all the numbers found.

If, on the other hand, you perform the program while the middle radio button is pressed, the calculations are performed in second thread, which updates the *ListView* component whenever a prime is found. Since nothing special has been done to update the component, the result is that the program stops with an exception. On the other hand, if you performs the program while the lower radio button is pressed, runs everything as it should. The program updates the *ListView* component and the program is alive, but this time without causing problems, as it happens with a *Handler* object.

The user interface consists of 10 components that are placed using a *RelativeLayout*, and the design does not contain anything new so I do not want to display the XML code here, so where there is something new is in the class *MainActivity*. The full code is as follows:

```
package dk.data.torus.primesapp;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
import android.os.*;
import android.content.*;
import java.util.*;

public class MainActivity extends AppCompatActivity
{
 private EditText txtFrom;
 private EditText txtTo;
 private RadioButton cmdPrim;
 private RadioButton cmdSec1;
 private RadioButton cmdSec2;
 private ListView lstPrimes;
```

```
private ArrayAdapter<Long> adapter;
private List<Long> primes = new ArrayList();
private Handler messageHandler;
private Handler primeHandler;

@Override
protected void onCreate(Bundle savedInstanceState)
{
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 txtFrom = (EditText) findViewById(R.id.txtFrom);
 txtTo = (EditText) findViewById(R.id.txtTo);
 cmdPrim = (RadioButton) findViewById(R.id.cmdPrim);
 cmdSec1 = (RadioButton) findViewById(R.id.cmdSec1);
 cmdSec2 = (RadioButton) findViewById(R.id.cmdSec2);
 lstPrimes = (ListView) findViewById(R.id.lstPrimes);
 lstPrimes.setAdapter(adapter = new ArrayAdapter<Long>(this,
  android.R.layout.simple_list_item_1, primes));
 messageHandler = new MessageHandler(this);
 primeHandler = new PrimeHandler(this, adapter);
}

public void onClear(View view)
{
 adapter.clear();
}

public void onCalculate(View view)
{
 try {
  long from = Long.parseLong(txtFrom.getText().toString());
  long to = Long.parseLong(txtTo.getText().toString());
  if (from > 0 && to > from && to - from <= 1000)
  {
   if (cmdPrim.isChecked()) calc1(from, to);
   else if (cmdSec1.isChecked()) calc2(from, to);
   else calc3(from, to);
   return;
  }
 }
 catch (Exception ex)
 {
 }
 Toast.makeText(this, "Illegal values", Toast.LENGTH_LONG).show();
}

private void calc1(long a, long b)
{
 for (long t = a; t <= b; ++t) if (isPrime(t))
 {
```

```
   adapter.add(t);
  }
 }

 private void calc2(long from, long to)
 {
  final long a = from;
  final long b = to;
  Runnable runnable = new Runnable() {
   public void run() {
    for (long t = a; t <= b; ++t) if (isPrime(t))
    {
     adapter.add(t);
    }
   }
  };
  (new Thread(runnable)).start();
 }

 private void calc3(long from, long to)
 {
  final long a = from;
  final long b = to;
  Runnable runnable = new Runnable() {
   public void run() {
    for (long t = a; t <= b; ++t) if (isPrime(t))
    {
     Message msg = primeHandler.obtainMessage();
     Bundle bundle = new Bundle();
     bundle.putLong("prime", t);
     msg.setData(bundle);
     primeHandler.sendMessage(msg);
    }
    messageHandler.sendEmptyMessage(0);
   }
  };
  (new Thread(runnable)).start();
 }

 private boolean isPrime(long n)
 {
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
   if (n % t == 0) return false;
  return true;
 }
}
```

The class ends with an method *isPrime()* that tests whether a number is a prime and is a well-known method that I have used many times. The class defines no less than 10 instance variables. The first are references to components in the user interface and the names tells what they are use for. Then there is a *List<Long>* for the content of the *ListView* component and an associated adapter, and finally there are two variables of the type *Handler* that I returns to in a little while. In *onCreate()*, there is also nothing new, except for the last two statements that create objects for the two *Handler* variables (explained below).

The program has two event handlers for the buttons, the first one is trivial and does nothing but delete the content of the *ListView* component. The other is the event handler for the button *CALCULATE*, and even if the code fills, the handler is simple enough. It must determine the values of the two numbers entered in the user interface and test if they are legal. If this is not the case, a simple Toast will appear with an error message. Are the values legal the event handler calls one of the methods

1. *calc1()*
2. *calc2()*
3. *calc3()*

depending on which radio button is pressed. Is it the top radio button the method *calc1()* is called with the two numbers as parameters and the method consists of a loop that updated the *ListView* component with the primes determined within the current range:

```
for (long t = a; t <= b; ++t) if (isPrime(t))
{
 adapter.add(t);
}
```

Since *calc1()* is a common method, it is executed by the primary thread, which in principle is fine enough, but the method can take a long time to perform (for large input parameters) and during that time the program does not respond and you may receive a warning as shown above.

If instead *calc2()* is executed, the same happens, but this time in a secondary thread, in which the thread performs the above loop. Once a prime has been found, the method *add()* of the adapter is called, which due to the data binding will update the user interface, and as it happens from another thread, the program will fail and stop with an exception.

This problem is solved in *calc3()* which updates the user interface using a *Handler* object, which is an object performed in the primary thread. At the end of the file *MainActivity.java* is defined two classes:

```
class MessageHandler extends Handler
{
 private Context context;

 public MessageHandler(Context context)
 {
  this.context = context;
 }

 @Override
 public void handleMessage(Message msg) {
  Toast.makeText(context, "Primes generated", Toast.LENGTH_LONG).show();
 }
}

class PrimeHandler extends Handler
{
 private Context context;
 private ArrayAdapter<Long> adapter;
 public PrimeHandler(Context context, ArrayAdapter<Long> adapter)
 {
```

```
  this.context = context;
  this.adapter = adapter;
 }

 @Override
 public void handleMessage(Message msg) {
  Bundle bundle = msg.getData();
  Long prime = bundle.getLong("prime");
  adapter.add(prime);
 }
}
```

Both classes override the method *handleMessage()*, which is a callback method performed by the primary thread, and the difference between the two classes is, among other things, which parameters are transmitted. In the first class, *handleMessage()* is trivial as it simply performs a *Toast*, but in the second class, a parameter must be transferred. It happens with a *Bundle* object, and in this case it must be a *Long*, which is the prime to be added to the *ListView* component. Note, in particular, that the parameter has the type *Message*, and also note the syntax for referring to the transferred value.

With these two *Handler* types available, the program can create instance variables of the types in question, what happens in *onCreate()*.

Then there is the method *calc3()*, which is in principle identical to *calc2()* and performs the task of determining the prime numbers in a secondary thread, but the method *run()* is changed a bit:

```
public void run() {
 for (long t = a; t <= b; ++t) if (isPrime(t))
 {
  Message msg = primeHandler.obtainMessage();
  Bundle bundle = new Bundle();
  bundle.putLong("prime", t);
  msg.setData(bundle);
  primeHandler.sendMessage(msg);
 }
 messageHandler.sendEmptyMessage(0);
}
```

In the *for* loop, the object *primeHandler* is used to create a *Message* object. In addition, a *Bundle* object is initialized with the current prime and this *Bundle* object is linked to the *Message* object. Hereafter the *primeHandler* object's *sendMessage()* method is called. When the loop is completed, the second handler object is used to send a message, but here the method *sendEmptyMessage()* is called.

8 SERVICES

A service is a component that is performed in the background without direct interaction with the user, and a service has no user interface. Services are used to perform special operations and often operations that take a long time. For example, it could be downloading a file from the Internet, testing for new data, and more. A service is performed with higher priority than activities, and there is therefore less chance that a service will be interrupted by Android if there is insufficient resources available. However, it is possible to provide a service with the same priority as foreground activities, and if so, it is necessary to have a visible indication that tells that the service is active, something that is used by video and music players.

There are two main types of services:

1. A service is be *started* when an application (an activity) starts it by calling the method *startService()*. Once started, a service can run in the background indefinitely, even if the application that started the service is destroyed. Usually, a started service performs a single operation (as download or upload a file over the network) and does not return a result to the caller. When the operation is done, the service should stop itself.

2. A service is *bounded* when an application component binds to it by calling the method *bindService()*. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and also across processes. A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

By default, a service is performed in the same process as an application's primary thread, but if it is a service that is started by an application, that service will usually perform its work in a secondary thread, and the service will then be terminated with the end of the secondary thread.

Android performs a number of predefined services that any application may apply if it is allowed. Use of these services usually occurs with a special *Manager* class using a method *getSystemService()*, and the *Context* class defines more constants for these services. An application may, in addition to these system services, define and apply new services that behave in the same way as system services. *Custom* services can generally be started from other Android components, that is, activities, broadcast receivers and other services.

The entire lifetime of a service happens between the call of a method *onCreate()* and the call of the method *onDestroy()* where *onCreate()* is used to all initial setup, while the release of resources is done in *onDestroy()*. The methods are called for all services, whether they're created by *startService()* or *bindService()*. A service begins its work with a call to either *onStartCommand()* or *onBind()*. Each method is handed an *Intent* object passed to either *startService()* or *bindService()*, respectively. If the service is *started*, it ends its work when *onStartCommand()* returns, and if the service is *bound*, it ends when *onUnbind()* returns.

A service must be defined in *AndroidManifest.xml* and written as a class that inherits the class *Service* or one of its subclasses. As an example, an Android component (and so far, to say an activity) can start a service with the method *startService()*, and as alternative it can use *bindService()* that allows for direct communication with the service.

A *started* service starts its life with the method *startService()* and if the service is not already running, a service object is created in the service's *onCreate()* method. Once running, a service can notify the user of events using *Toast* notifications or *Status Bar* notifications, but the service can not directly call the user.

The method *onStartCommand()* returns an *int* that indicates how services should be restarted if it is terminated by Android and there are the following options:

1. *Service.START_STICKY*, which means that the service is restarted, but such that the *Intent* object is null and used if the service maintains its own state and does not depend on the *Intent* object.
2. *Service.START_NOT_STICKY*, which indicates that the service will not be automatically restarted.
3. *Service.START_REDELIVER_INTENT*, which means the same as the first option, but with the difference that the original *Intent* object is sent to *onStartCommand()*.

One can end a service with the method *stopService()* and a service can end itself with *stopSelf()*.

To create a bound service, you must implement the method *onBind()* that returns an *IBinder* that defines the interface for communication with the service. Other applications can then call *bindService()* to retrieve the interface and begin calling methods on the service. The service is only alive as long there are applications that is bound to it, and when there are no components bound to the service, the system destroys it, and you do not need to stop a bound service in the way you must when the service is started with *onStartCommand()*. There are two ways you can define the *IBinder* interface:

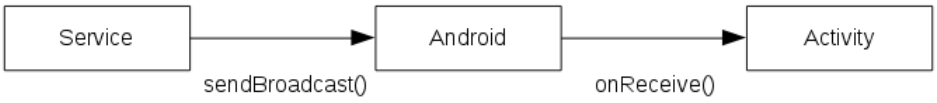
1. Extending the *Binder* class. If your service is private to your own application and runs in the same process as the application, which is often the case, you should create your interface by extending the *Binder* class and returning an instance of it from *onBind()*. Then the client receives the *Binder* object and can use it to directly access *public* methods available in either the *Binder* implementation or the *Service*. When the service is merely a background worker for the application it is the way to create the *Binder*, and the only reason not to create the interface this way is because the service has to be used by other applications or across separate processes.
2. Using a *Messenger.* If you your interface must work across different processes, you can create an interface for the service with a *Messenger*. In this manner, the service defines a *Handler* that responds to different types of *Message* objects. This *Handler* is the basis for a *Messenger* that can share an *IBinder* with the client, allowing the client to send commands to the service using *Message* objects. Also the client can define its own *Messenger* so the service can send messages back. This is the simplest way to perform interprocess communication, because the *Messenger* queues all requests into a single thread so that you don't have to design your service to be thread-safe.

Application components can bind to a service by calling *bindService(),* and Android then calls the service's *onBind()* method, which returns an *IBinder* object for interacting with

the service. It happens asynchronous and *bindService()* returns immediately and does not return the *IBinder* object. To receive the object, the client must create an instance of *ServiceConnection* and pass it to *bindService()*. *ServiceConnection* is an interface and defines a method *onServiceConnected()* that the system calls to get the *IBinder* object. The interface also defines a method *onServiceDisconnected()* that Android calls when the connection to the service is lost because the service has crashed or has been killed. The method is not called when the application unbinds. To disconnect from a service, you must call the method *unbindService()*. When the application is destroyed, it will automatical unbind from the service, but you should always unbind when you're done so that the service can shutdown while it is not longer being used.

A service can also be a foreground service, which is a service associated to something that the user is aware of and thus not a candidate for the system to kill when low on memory. A foreground service must provide a notification in the status bar, which is a notification that cannot be dismissed unless the service is either stopped or removed from the foreground. To start a service in the foreground you use the method *startForeground()*, that takes two parameters, where the first is an *int* that uniquely identifies the notification and the notification for the status bar.

In the simple case, there is no need for any communication between activity and service. The service receives an *Intent* object from the activity component, and the service then performs its work, but in other contexts there is a need for direct communication between an activity and a service, and here you can use a receiver. An activity can detect a broadcast receiver for an event and the service can send notifications for this event and typically responds that the service will signal the activity object that it has completed its work:



A broadcast receiver is a class which extends *BroadcastReceiver* and which is registered as a receiver in an application other in *AndroidManifest.xml* or in code. The receiver will be able to receive intents generated with the method *Context.sendBroadcast()*. The class *BroadcastReceiver* defines the method *onReceive()* that the application can use to get the received object.

It was briefly a little about the theory, and in the following I will show what exactly to be written to use services, at least the start of simple services and the use of local services, but the subject is actually relatively complex, and I will return to it in the next book. Among other things, *BroadcastReceiver* objects are not featured in this book.

First, however, I would like to attach a few comments to the class *Intent*.

8.1 THE CLASS INTENT

The class *Intent* has not alone to do with services, and I have already used the class many times without mentioning what it is for a class. As the class plays a crucial role in services, I will briefly mention it here.

The class primarily applies to activities, services and broadcast receivers. These components are created using messages called intents, which are a form of late binding of components, so it always happens in the same way. An *Intent* object is a data structure that contains an abstract description of an operation to be performed or a description of something that has happened and has to be advertised. There are several mechanisms to send intents to different types of components:

1. An *Intent* object can be passed to *Context.startActivity()* or *Activity.startActivityForResult()* to launch an activity or get an existing activity to do something new. It can also be passed to *Activity.setResult()* to return information from the activity that has been performed with *startActivityForResult()*.

2. An *Intent* object can be passed to *Context.startService()* to initiate a service or deliver new instructions to a running service. Similarly, an intent can be passed to *Context.bindService()* to establish a connection between the calling activity and a service. It can also be used to initiate the service if it's not already running.
3. *Intent* objects can be passed to broadcast methods (see the next book) such as *Context.sendBroadcast()*, *Context.sendOrderedBroadcast()*, or *Context.sendStickyBroadcast()* and are delivered to all interested broadcast receivers.

Intents are really useful with services to exchange information and notification between the calling object and the service. You are encouraged to examine the documentation for the *Intent* class.

8.2 SERVICEAPP1

A service is written as a class inheriting the class *Service* or *IntentService*, where the last is derived from the first. The goal of an *IntentService* is to simplify the start of a service as much as possible. The service carries out requests asynchronously in their own thread, and it can take as long as necessary and does not block the application. The individual request is added to a queue and executed one at a time. An *IntentService* is a background service that is started by an activity, and then there is no direct communication between the service and the activity. A typical application is the start of an action that takes a long time, such as downloading a file.

The example *ServiceApp1* opens the following window:


Android Emulator - Nexus_6_API_24:5554
12:46
ServiceApp1
CLICK
START SERVICE

If you click on the top button, the window gets a random color and clicking on the bottom button starts a service that is an IntentService. The service is written as the following class:

```
import android.app.IntentService;

import android.content.Intent;
import android.net.Uri;
import android.media.RingtoneManager;
import android.media.Ringtone;

public class TheService extends IntentService
{
 public TheService() {
  super("TheService");
 }

 @Override
 protected void onHandleIntent(Intent intent)
 {
  long n = intent.getLongExtra("todo", 0);
  for (long i = 0; i < n; ++i) Math.sin(Math.sqrt(i));
  Uri notification =
   RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
  Ringtone r =
   RingtoneManager.getRingtone(getApplicationContext(), notification);
  r.play();
 }
}
```

The class should only have a default constructor who performs the base class' constructor with the service name as parameter and furthermore it must override the method *onHandleIntent()*, which is the method that performs the desired action. In this case, the method performs a long (meaningless) action, after which it activates a ringtone. With regard to the action, the goal is to show how you with an *Intent* can transfer values to a service and that the service is doing something for a while, before you because of the sound observes that the service has performed its work.

When an application uses a service, it must be defined in the program's manifest

```
<application
  …
  </activity>
  <service android:name=".TheService" />
</application>
```

A detail that is important to remember.

With regard to the program (*MainActivity*), the goal of the top button is to show that the application is not blocked while the service performs its work – and also a little to show how to change the background color of the window. When you try out the program, note that you can start multiple services and how they are queued. The code for *MainActivity* is as follows:

```
package serviceapp1.dk.data.torus.serviceapp1;

import android.graphics.Color;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.content.*;
import java.util.Random;

public class MainActivity extends AppCompatActivity {
 private static Random rand = new Random();

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

 public void onClick(View view)
 {
  getWindow().getDecorView().setBackgroundColor(getColor());
 }

 public void onStart(View view)
 {
  Intent intent = new Intent(this, TheService.class);
  intent.putExtra("todo", 100000000L);
  startService(intent);
 }

 private int getColor()
 {
  return Color.rgb(rand.nextInt(256), rand.nextInt(256), rand.nextInt(256));
 }
}
```

Here the most important is the event handler *onStart()*, that starts a service. First, an *Intent* object is created for the service and the service is started with the method *startService()*.

8.3 SERVICEAPP2

Often, an *IntentService* does not provide enough options, for example, if you want to stop a running service, and you will then have to write a service that directly inherits the class *Service*. The program *ServiceApp2* shows how. The program's window is almost identical to the above, just there is a third button:

which is used to stop a running service. The service is this time written as:

```
package serviceapp2.dk.data.torus.serviceapp2;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.widget.Toast;

public class TheService extends Service {

 @Override
 public void onCreate() {
  Toast.makeText(this, "Service created", Toast.LENGTH_LONG).show();
 }

 @Override
 public IBinder onBind(Intent arg0) {
  return null;
 }

 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
  long timeout = System.currentTimeMillis() + 30000;
```

```
  while (System.currentTimeMillis() < timeout) {
   synchronized (this) {
    Toast.makeText(this, "Service " + startId + " is working",
     Toast.LENGTH_LONG).show();
    try {
     wait(5000);
    }
    catch (Exception e) {
    }
   }
  }
  return Service.START_STICKY;
 }

 @Override
 public void onDestroy() {
  Toast.makeText(this, "Service destroyed", Toast.LENGTH_LONG).show();
 }
}
```

A background service (which does not have a user interface) can communicate with the environment in two ways, and the one with a simple *Toast*, which is just a simple message that appears briefly on the screen. The class inherits this time the class *Service* and a service has an *onCreate()* method where you can initialize the service. In this case, nothing happens other than the service shows a *Toast* that says it was created. There is also a method called *onBind()* and is inserted by Android Studio, but it is only used for a bounded service. Then there is the last method *onDestroy()* that is performed when the service is stopped, and in this case it also shows just a simple *Toast*.

Finally, there is *onStartCommand()*, which is the method that is performed when the service is started, thus when the starting activity performs *startService()*. In this case, a loop is running for approximate 30 seconds. For each iteration, a *Toast* is performed, after which service is waiting 5 seconds.

The class *MainActivity* is almost identical to the previous example, just the event handler for the *Start* button is a bit simpler (no parameter must be transferred), and an event handler is added to the *Stop* button:

```
public void onStart(View view)
{
 Intent intent = new Intent(this, TheService.class);
 startService(intent);
}
```

```
public void onStop(View view)
{
 Intent intent = new Intent(this, TheService.class);
 stopService(intent);
}
```

If you try the program and click the start button, you will find that the other buttons do not seem to work and you will not see *Toasts* either – at least not before the service has finished its work, after which the background changes color (if you have clicked the button). If you have not clicked the *Stop* button and do it now, you will see a *Toast*, telling that the service is stopped. The reason is that the service and the program itself are executed in the same thread, and as the service temporarily suspends the thread, the program will appear to be unresponsive – and you may also get a message from Android, that it is the case.

8.4 SERVICEAPP3

This example is exactly the same as the previous example, just showing the solution to the above problem. That is, the layout and *MainActivity* (and the manifest) are completely unchanged. The solution is, of course, that the service must do its work in a second thread,

which in principle is quite simple, but there are still a few things to be aware of. The code is as follows:

```
package serviceapp3.dk.data.torus.serviceapp3;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.widget.Toast;
import android.os.*;

public class TheService extends Service {
 private boolean running = false;

 @Override
 public void onCreate() {
  Toast.makeText(this, "Service created", Toast.LENGTH_LONG).show();
 }

 @Override
 public IBinder onBind(Intent arg0) {
  return null;
 }

 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
  if (running) Toast.makeText(this, "Service already running",
   Toast.LENGTH_LONG).show();
  else {
   final int currentId = startId;
   Runnable r = new Runnable() {
    public void run() {
     long timeout = System.currentTimeMillis() + 30000;
     running = true;
     while (running && System.currentTimeMillis() < timeout) {
      synchronized (this) {
       sendMessage(currentId);
       try {
        wait(5000);
       } catch (Exception e) {
       }
      }
     }
     stopSelf();
    }
   };
```

```
    Thread t = new Thread(r);
    t.start();
   }
   return Service.START_STICKY;
  }

  @Override
  public void onDestroy()
  {
   running = false;
   Toast.makeText(this, "Service destroyed", Toast.LENGTH_LONG).show();
  }

  private void sendMessage(int id)
  {
   final int currentId = id;
   Handler handler = new Handler(TheService.this.getMainLooper());
   handler.post(new Runnable() {
    @Override
    public void run() {
     Toast.makeText(getApplicationContext(),
       "Service " + currentId + " is working", Toast.LENGTH_LONG).show();
    }
   });
  }
}
```

Note that at the start of the class, an instance variable is defined. It must be used to ensure that the service can only start one thread. It is not necessary, but it is included to show how to stop a thread started by a service. Note, in particular, that the variable is set to *false* in the method *onDestroy()*, and it is executed either because the thread stops or because *onDestroy()* is called from an activity. The method *onStartCommand()* starts testing this variable, and if it is *true*, nothing else happens than the method performs a *Toast*. Otherwise, the method creates a thread that performs the same as in the previous example, in which in an iteration it performs a *Toast* followed by a *wait()*. However, there is a small problem. The message is now sent from a secondary thread, and although it is only a *Toast*, the thread must still update the user interface. It requires a *Handler* and is performed in the method *sendMessage()*. A *Handler* is linked to a message loop, and since a thread does not have a message loop, the method *getMainLooper()* is used. The *Handler* object will then use thet method *post()* to insert a *Runnable* object that performs the desired *Toast*.

8.5 SERVICEAPP4

In this section I will show an example of local service and thus a service that the application can bind to and communicate with. That it is a local service means that it is a service performed in the same process as the application and that it is automatically terminated when the application close. The program opens the following window with a *TextView* and 3 *Button* components:

After the program has started, the text shows *Disconnected*, and clicking on the two top buttons nothing happens because the program is not yet connected to the service. If you click on the bottom button, the text changes to *Connected*, as the program is now bounded to the service, and when you click on the two top buttons, the text displays the current date and time, respectively. These values are determined by the program (the buttons event handlers) by calling a service method, and it is the service that reads the clock.

The user interface is trivial and should not be displayed here, and the manifest has been changed in the same way as in the previous examples in this chapter. The code for the service is as follows:

```
package dk.data.torus.serviceapp4;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;
import java.util.Calendar;

public class TheService extends Service {
 private final IBinder binder = new TheBinder();
```

```
 @Override
 public IBinder onBind(Intent intent) {
  return binder;
 }

 public String getCurrentDate() {
  Calendar cal = Calendar.getInstance();
  return String.format("%04d-%02d-%02d", cal.get(Calendar.YEAR),
   cal.get(Calendar.MONTH) + 1, cal.get(Calendar.DAY_OF_MONTH));
 }

 public String getCurrentTime() {
  Calendar cal = Calendar.getInstance();
  return String.format("%02d:%02d:%02d:%03d", cal.get(Calendar.HOUR_OF_DAY),
   cal.get(Calendar.MINUTE), cal.get(Calendar.SECOND),
   cal.get(Calendar.MILLISECOND));
 }

 public class TheBinder extends Binder {
  TheService getService() {
   return TheService.this;
  }
 }
}
```

First note that the class inherits the class *Service*, but this time I have not overridden the methods *onStartCommand()* and *onDestroy()*. Instead, the method *onBind()* is implemented, so it returns a *Binder* object. This object is created at the beginning of the class as an object of an inner class. It is a class inheriting the class *Binder* (which implements an interface *IBinder*), and it implements a single method *getService()*, which returns a reference to the current service. In addition, the class *TheService* implements the two service methods, both of which are simple.

Then there is the class *MainActivity*, which is responsible for starting the service in question by binding to it:

```
package dk.data.torus.serviceapp4;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
import android.os.*;
import android.content.*;
```

```
public class MainActivity extends AppCompatActivity
 implements ServiceConnection {
 private TheService theService;
 private boolean isBound = false;
 private TextView txtResult;

 @Override
 public void onServiceConnected(ComponentName className, IBinder service) {
  TheService.TheBinder binder = (TheService.TheBinder) service;
  theService = binder.getService();
  isBound = true;
  txtResult.setText("Connected");
 }

 @Override
 public void onServiceDisconnected(ComponentName className) {
  isBound = false;
  theService = null;
 }

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
```

```
  setContentView(R.layout.activity_main);
  txtResult = findViewById(R.id.txtResult);
 }

 public void onDate(View view)
 {
  if (isBound) txtResult.setText(theService.getCurrentDate());
 }

 public void onTime(View view)
 {
  if (isBound) txtResult.setText(theService.getCurrentTime());
 }

 public void onConnect(View view)
 {
  if (!isBound) {
   Intent intent = new Intent(this, TheService.class);
   bindService(intent, this, Context.BIND_AUTO_CREATE);
  }
 }
}
```

Here you should first notice that the class implements the interface *ServiceConnection*. This interface defines two methods, the first being called *onServiceConnected()* and is executed after the program has connected to the service. The other is called *onServiceDisconnected()* and executed if the program has been disconnected from the service. In this case, the program binds to the service in the event handler *onConnect()*, and if the program is not already bound, an *Intent* object is created for the current class (program) and service class. Then, the program binds using the method *bindService()*, and note the last parameter telling that the service should be created if it is not already. After *bindService()* is performed, Android calls the method *onServiceConnected()*, and its last parameter has the type *IBinder* and is a reference to the service and is the return value of the service's *onBind()* method. This value is used to reference the service's *getService()* method, so the program has a reference to service's methods, which can then be used in the two event handlers for the top buttons.

You should note that a local service such as the above can be perceived as part of the sole program and the current service is not typical as both service methods are trivial. Typically, a program needs to have performed a work that takes time, and the program then starts a service that performs the work in a secondary thread.

8.6 SERVICEAPP5

The next example shows a remote service, and it's just a simple example, but more in the next book. The application opens a window with two buttons. Clicking on the top button, the background color of the window is changed and clicking on the bottom button sends a message (the current date and time) to a remote service, which after 5 seconds shows a *Toast* with the message. The mening with the 5-second delay and the color change button are that you should find that the service does not block the application as it is performed in another process, than the application. The service is quite simple:

```
package dk.data.torus.serviceapp5;

import android.app.Service;
import android.content.Intent;
import android.os.*;
import android.widget.Toast;

public class TheService extends Service {
 private final Messenger theMessenger = new Messenger(new TheHandler());

 class TheHandler extends Handler {
  @Override
  public void handleMessage(Message msg) {
   Bundle data = msg.getData();
   synchronized (this) {
    try {
     wait(5000);
    }
    catch (Exception e) {
    }
    Toast.makeText(getApplicationContext(), data.getString("message"),
     Toast.LENGTH_LONG).show();
   }
  }
 }

 @Override
 public IBinder onBind(Intent intent) {
  return theMessenger.getBinder();
 }
}
```

The class inherits *Service* as the other examples. It defines an inner class that inherits *Handler*, and it overrides the method *handleMessage()*. The method's parameter has the type *Message* and represents the message that a client sends. It is wrapped in a *Bundle* object.

The method starts blocking the current thread for 5 seconds, after which it displays a *Toast* message with the message sent.

TheService class has a single instance variable, which is a *Messenger* object that is created with an object of the above inner class (it is a *Handler*) as a parameter. A *Messenger* object has a *Binder*, and the method *onBind()* returns this *Binder*.

In order for the service to be created in a process other than the program, enter it in the manifest:

```
<service android:name=".TheService"
  android:process=":app5_process" >
</service>
```

Finally, there is the class *MainActivity*:

```
package dk.data.torus.serviceapp5;

import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.os.*;
```

```
import android.content.*;
import android.graphics.Color;
import java.util.Calendar;
import java.util.Random;

public class MainActivity extends AppCompatActivity {
 private static final Random rand = new Random();
 private Messenger theService = null;
 private boolean isBound;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  bindService(new Intent(this, TheService.class), theConnection,
   Context.BIND_AUTO_CREATE);
 }

 private ServiceConnection theConnection = new ServiceConnection() {
  public void onServiceConnected(ComponentName className, IBinder service) {
   theService = new Messenger(service);
   isBound = true;
  }
  public void onServiceDisconnected(ComponentName className) {
   theService = null;
   isBound = false;
  }
 };

 public void sendMessage(View view)
 {
  if (!isBound) return;
  Message msg = Message.obtain();
  Bundle bundle = new Bundle();
  bundle.putString("message", getMessage());
  msg.setData(bundle);
  try {
   theService.send(msg);
  }
  catch (RemoteException e) {
   e.printStackTrace();
  }
 }

 public void onClick(View view)
 {
  getWindow().getDecorView().setBackgroundColor(getColor());
 }
```

```
 private int getColor()
 {
  return Color.rgb(rand.nextInt(256), rand.nextInt(256), rand.nextInt(256));
 }

 private String getMessage() {
  // returns the mesage
 }
}
```

When you see the code, it is simple and as in the previous example, the method should use a *ServiceConnection* object. Instead of inheriting the interface *ServiceConnection*, the class creates an anonymous object. There is no particular reason for that in addition to showing that it is possible. The binding to the service is done in *onCreate()* and occurs in the same way as in the previous example. Otherwise, you should primarily note the method *sendMessage()* that is event handler the button *Send* and the method that sends a message to the service.

8.7 RING THE PHONE

The following example opens a simple window as shown below. If you click on the top button, the phone rings and continue until you click the bottom button. Note that if you click on the top button and then close the program, the phone will continue to ring. This is because the program starts a service, and since it is not a local service, it will still be active after the application is closed.

After the phone has ringed for 10 seconds, the phone will open the window below (allowing you to stop ringing), regardless of whether the application is closed or not.

I do not want to show the layout for the user interface, and *MainActivity* is also simple:

```
public class MainActivity extends AppCompatActivity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  ((Button) findViewById(R.id.cmdStart)).setOnClickListener(
   new View.OnClickListener() {
    public void onClick(View view) {
     startService(new Intent(MainActivity.this, SimpleService.class));
    }
  });
```

```
  ((Button) findViewById(R.id.cmdStop)).setOnClickListener(
   new View.OnClickListener() {
    public void onClick(View view) {
     stopService(new Intent(MainActivity.this, SimpleService.class));
    }
  });
 }
}
```

There is only an *onCreate()* and there is not much about services. An event handler to a button is an object of a class that implements the interface *View.OnClickListener*. This interface defines a single method called *onClick()* and has a *View* object as parameter. The method associates event handlers with the two buttons by creating objects of anonymous classes. The event handler for the button *cmdStart* has a single statement, which calls the method *startService()*, where the parameter is an *Intent* object for the current class and also the class *RingService*, which is the service class. With regard to the method *startService()*, it is a method in the *Context* class as an *Activity* inheritance, and the method starts a *RingService* service – if it is not already running. The event handler for the second button is implemented accordingly, but instead calls the method *stopService()*.

The services itselves are written as follows:

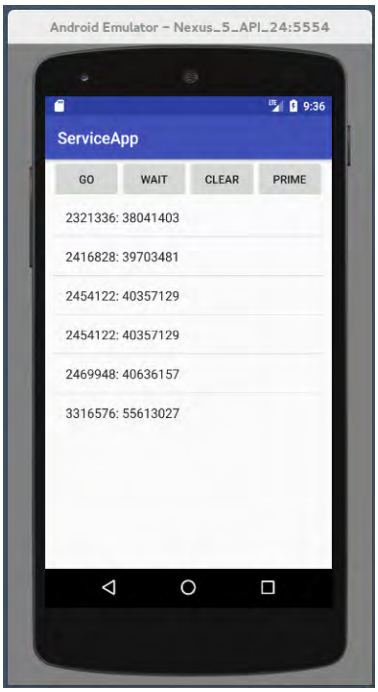
```
public class RingService extends Service {
 private MediaPlayer player = null;

 @Nullable
 @Override
 public IBinder onBind(Intent intent) {
  return null;
 }
```

```
 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
  if (player == null) {
   Runnable r = new Runnable() {
    public void run() {
     synchronized (this) {
      try {
       wait(10000);
      } catch (Exception e) {
      }
     }
     Intent intent = new Intent();
     intent.setAction(Intent.ACTION_MAIN);
     intent.addCategory(Intent.CATEGORY_LAUNCHER);
     intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
     ComponentName cn = new ComponentName(RingService.this,
      dk.data.torus.ringthephone.MainActivity.class);
     intent.setComponent(cn);
     startActivity(intent);
    }
   };
   player = MediaPlayer.create(RingService.this,
    Settings.System.DEFAULT_RINGTONE_URI);
   player.setLooping(true);
   player.start();
   Thread t = new Thread(r);
   t.start();
  }
  return START_STICKY;
 }

 @Override
 public void onDestroy() {
  super.onDestroy();
  if (player != null) player.stop();
 }
}
```

The class has a variable of the type *MediaPlayer* that is used to ring the phone, but otherwise, the class overrides two methods from the class *Service*. The first is called from the event handler for the first button when it calls *startService()* and it creates a *MediaPlayer* object (if it is not already created) and says it should continue to ring. Then, the method starts a thread that waits for 10 seconds, and then opens the window *MainActivity,* even if the application is closed. The syntax is not simple and by no means obvious and there is not much more than to take it into account. The second method is called equivalent from the event handler to the bottom button when it performs the method *stopService()*.

8.8 A LOCAL SERVICE – SERVICEAPP

I will then show an application that uses a local service. The application opens the following window:



The program uses a service that always finds the next prime number. In the beginning, it's fast, but if the program has run for a long time, it starts to take time to determine the next prime. The window has 4 buttons and a *ListView*. If you click on the button *PRIME*, the prime for which the service is reached is added to the *ListView*, partly as the number's index in the sequence of primes and partly as the actual prime number. Clicking the button *CLEAR* will delete the content of the *ListView*. If you click on the button *WAIT*, the service stops its work, but it can be restarted (meaning it continues) by clicking *GO*. Above, after the program has been running for a while, there are clicked

- PRIME
- PRIME
- WAIT
- PRIME
- PRIME
- GO
- PRIME
- and after a while PRIME

The layout is simple and contains only 5 components:

```
<?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"
  tools:context="dk.data.torus.serviceapp.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:orientation="vertical">
    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:orientation="horizontal" >
      <Button
        android:id="@+id/cmdRestart"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:onClick="onGo"
        android:text="Go" />
```

```
        <Button
           android:id="@+id/cmdPause"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:layout_weight="1"
           android:onClick="onWait"
           android:text="Wait" />
        <Button
           android:id="@+id/cmdClear"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:layout_weight="1"
           android:onClick="onClear"
           android:text="Clear" />
        <Button
           android:id="@+id/cmdCalc"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:layout_weight="1"
           android:onClick="onService"
           android:text="Prime" />
     </LinearLayout>
     <ListView
        android:id="@+id/lstView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
     </ListView>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

and the only thing to note is the names that I have given the buttons and the *ListView*, as well as the names of the event handlers that the buttons calls. The code for *MainActivity* is as follows:

```
package dk.data.torus.serviceapp;

import android.support.v7.app.AppCompatActivity;
import android.content.ComponentName;
import android.content.*;
import android.os.*;
import android.view.*;
import android.widget.*;
import java.util.*;

public class MainActivity extends AppCompatActivity
 implements ServiceConnection {
```

```
private ArrayAdapter<String> adapter;
private List<String> primesList = new ArrayList();
private LocalService service;

@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 lstView = (ListView)findViewById(R.id.lstView);
 adapter = new ArrayAdapter<String>(this,
  android.R.layout.simple_list_item_1, android.R.id.text1, primesList);
 lstView.setAdapter(adapter);
}

@Override
protected void onResume() {
 super.onResume();
 Intent intent= new Intent(this, LocalService.class);
 bindService(intent, this, Context.BIND_AUTO_CREATE);
}

@Override
protected void onPause() {
```

```
  super.onPause();
  unbindService(this);
 }

 public void onGo(View view) {
  service.start();
 }

 public void onWait(View view) {
  service.pause();
 }

 public void onService(View view) {
  adapter.add(String.format("%d: %d", service.
    getIndex(), service.getPrime()));
  Toast.makeText(this, "List is updated", Toast.LENGTH_SHORT).show();
 }

 public void onClear(View view) {
  adapter.clear();
 }

 @Override
 public void onServiceConnected(ComponentName name, IBinder binder)
 {
  if (service == null)
    service = ((LocalService.PrimesBinder) binder).getService();
  Toast.makeText(MainActivity.this, "Service connected",
    Toast.LENGTH_SHORT).show();
 }

 @Override
 public void onServiceDisconnected(ComponentName name)
 {
  service = null;
  Toast.makeText(MainActivity.this, "Service disconnected",
    Toast.LENGTH_SHORT).show();
 }
}
```

The first thing to note is that the class implements an interface *ServiceConnection*. This interface defines two methods to be overridden. The first is called *onServiceConnected()*, and it is executed when the application starts and binds to the service. It initializes a variable *service*, which then refers to the service in question. In addition, a simple *Toast* is performed, so you can see that the program is bound to the service. The second method is called *onServiceDisconnected()* and is executed if the application disconnects from the service, typically because Android can terminate the service.

In addition to the variable *service*, there is nothing to note about the class' variables, and the same applies to *onCreate()*, where there is nothing new. On the other hand, the class overrides both the method *onResume()* and *onPause()*, and these methods bind or unbinds the application to the service, thus resulting in the above two methods being performed.

Finally, there are the 4 event handlers for the buttons, which are generally trivial, but three of them use methods at the service to pause, restart, and read its value.

The service is also a class and is written as follows:

```
package dk.data.torus.serviceapp;

import android.app.Service;
import android.content.Intent;
import android.os.*;

public class LocalService extends Service {
 private final IBinder binder = new PrimesBinder();
 private long prime = 2;
 private int index = 0;
 private boolean running = false;

 @Override
 public IBinder onBind(Intent intent)
 {
  start();
  return binder;
 }

 @Override
 public void onRebind(Intent intent) {
  start();
 }

 @Override
 public boolean onUnbind(Intent intent) {
  running = false;
  return true;
 }

 public int getIndex()
 {
  return index;
 }
```

```
public long getPrime()
{
 return prime;
}

public void start()
{
 if (!running) {
  running = true;
  Thread th = new Thread(new Generator());
  th.start();
 }
}

public void pause()
{
 running = false;
}

public class PrimesBinder extends Binder {
 LocalService getService() {
  return LocalService.this;
 }
}
```

```
 private class Generator implements Runnable
 {
  public void run()
  {
   long t = index == 0 ? 3 : prime + 2;
   for ( ; running ; t += 2)
    if (isPrime(t))
    {
     ++index;
     prime = t;
    }
  }

  private boolean isPrime(long n)
  {
   if (n == 2 || n == 3 || n == 5 || n == 7) return true;
   if (n < 11 || n % 2 == 0) return false;
   for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
    if (n % t == 0) return false;
   return true;
  }
 }
}
```

First, note that the class inherits the class *Service*, which is what makes it to a service. Then note the class' variables, where the first has the type *IBinder*, which is an interface that defines how components can communicate with the service. The variable refers to a *PrimesBinder* object, which is a public inner class inheriting the class *Binder*, and the class does nothing but implement a method that returns a reference to the current service. The following variables are for the current prime number, while the last indicates whether the service is paused.

The class overrides three methods from the class *Service*, which is performed when a component binds or rebinds the service, and a method if a component unbinds. Here is the last trivial, in which it just sets *running* to false, while the others call the method *start()*. It is the method that starts the work of the service by starting a thread that runs until the variable *running* becomes *false*. When the thread terminates (*running* becomes false) the thread stops and the variables *index* and *prime* are not updated, but their value are retained. This means that if the method *start()* is executed again, it will start a new thread, which will then continue from where the previous one stopped.

The thread itself is implemented by an inner class *Generator*, but here is nothing new and the method *isPrime()*, is as you have seen it many times.

Back there is only one thing, that the manifest must be updated:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.serviceapp">
  <application
    …
    </activity>
    <service
      android:name=".LocalService"
      android:label="A local service" >
    </service>
  </application>
</manifest>
```

8.9 SERVICES AND SQLITE

As the last example of a service, I will show a program that starts a service that updates a database every 20 seconds. The example also includes another program (a client) that can read the content of the database and the goal is to show

1. how an application and a service can write to a database
2. how another application can read the same database

However, I will start with a database adapter that is not necessary for the purpose of this task, but a very used pattern. In fact, it is nothing but a class that encapsulates the class *SQLiteOpenHelper* and the database operations. In this case, the database must have only one table that has three columns:

```
create table posts (
 id integer primary key autoincrement,
 time varchar(19) not null,
 post varchar(100) not null)
```

where the column *time* contains a clock time of the form *YYYY-MM-DD HH:MM:SS*, while the last column is just a text. The adapter can then be written as follows:

```
package dk.data.torus.launchposts;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
```

```
public class DbAdapter {
 public static final String TABLE_NAME="posts";
 public static final int COLNO_ID = 0;
 public static final int COLNO_TIME = 1;
 public static final int COLNO_POST = 2;
 public static final String[] TABLE_COLUMNS =
  new String[]{ "id", "time", "post" };
 private static final String DBFILENAME="posts.db";
 private static final int DBVERSION = 1;
 private static final String INITIAL_SCHEMA =
  "create table posts (id integer primary key autoincrement, " +
  "time varchar(19) not null, " +
  "post varchar(100) not null)";
 private final Context context;
 private DatabaseHelper dbHelper;
 private SQLiteDatabase db;

 public DbAdapter(Context context) {
  this.context = context;
  dbHelper = new DatabaseHelper(context);
 }
```

```
public DbAdapter open() {
 db = dbHelper.getWritableDatabase();
 return this;
}

public void close() {
 dbHelper.close();
}

public long insert(String time, String post) {
 ContentValues values = new ContentValues();
 values.put(TABLE_COLUMNS[COLNO_TIME], time);
 values.put(TABLE_COLUMNS[COLNO_POST], post);
 return db.insert(TABLE_NAME, null, values);
}

public boolean update(String time, String post) {
 ContentValues values = new ContentValues();
 values.put(TABLE_COLUMNS[COLNO_TIME], time);
 values.put(TABLE_COLUMNS[COLNO_POST], post);
 return db.update(TABLE_NAME, values, "id = 1", null) != 0;
}

public boolean delete(int id) {
 return db.delete(TABLE_NAME, "id = ?", new String[] {"" + id}) != 0;
}

public void delete() {
 db.delete(TABLE_NAME, null, null);
}

public Cursor getRows() {
 Cursor cursor =
  db.query(TABLE_NAME, TABLE_COLUMNS, null, null, null, null, null);
 return cursor;
}

public Cursor getRow(int id) {
 Cursor cursor = db.query(TABLE_NAME, TABLE_COLUMNS, "id = ?",
  new String[] {"" + id}, null, null, null, null);
 return cursor;
}

public Cursor getRows(String date) {
 Cursor cursor = db.query(TABLE_NAME, TABLE_COLUMNS, "time like ?",
  new String[] { date + "%"}, null, null, null, null);
 return cursor;
}
```

```
 private static class DatabaseHelper extends SQLiteOpenHelper
 {
  DatabaseHelper(Context context) {
   super(context, DBFILENAME, null, DBVERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
   db.execSQL(INITIAL_SCHEMA);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
   db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
   onCreate(db);
  }

  @Override
  public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
   db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
   onCreate(db);
  }
 }
}
```

In fact, the class is easy enough to understand and contains, among other things, what I have previously shown as a *DbHelper* class. In addition, there are the database operations and which will depend on the specific database and how it will be used. In this case, there are more than the applications uses, just to give an impression of which operations you typically want. The most important reason for the class is that it makes it easy to use the same database from multiple programs.

The project that creates the database and the service is called *LaunchPosts*, and it will thus be the application to which that database is a belongs. The project uses two classes, which I have used in previous examples and it are the class *King*, which represents a Danish king and the class *Kings*, which is a list of *King* objects and has a method that returns a *List<King>*. The service is as follows:

```
package dk.data.torus.launchposts;

import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
```

```
import android.os.IBinder;
import android.support.annotation.Nullable;
import java.util.*;

public class PostService extends Service {
 private static final Random rand = new Random();
 private Kings kings = new Kings();
 private Timer timer;
 private TimerTask timerTask;

 public PostService() {
 }

 public PostService(Context applicationContext) {
  super();
 }

 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
  super.onStartCommand(intent, flags, startId);
  startTimer();
  return START_STICKY;
 }
```

```
@Override
public void onDestroy() {
 super.onDestroy();
 stoptimertask();
}

public void startTimer() {
 timer = new Timer();
 initializeTimerTask();
 timer.schedule(timerTask, 20000, 20000);
}

public void initializeTimerTask() {
 timerTask = new TimerTask() {
  public void run() {
   writePost(getPost());
  }
 };
}

public void stoptimertask() {
 if (timer != null) {
  timer.cancel();
  timer = null;
 }
}

@Nullable
@Override
public IBinder onBind(Intent intent) {
 return null;
}

private void writePost(String post)
{
 Calendar cal = Calendar.getInstance();
 String time = String.format("%04d-%02d-%02d %02d:%02d:%02d",
  cal.get(Calendar.YEAR), cal.get(Calendar.MONTH) + 1,
  cal.get(Calendar.DAY_OF_MONTH), cal.get(Calendar.HOUR_OF_DAY),
  cal.get(Calendar.MINUTE), cal.get(Calendar.SECOND));
 Context context = getApplicationContext();
 DbAdapter db = new DbAdapter(getApplicationContext());
 db.open();
 Cursor cursor = db.getRows();
 cursor.moveToFirst();
 if (cursor.isAfterLast()) db.insert(time, post); else db.update(time, post);
 cursor.close();
 db.close();
}
```

```
 private String getPost()
 {
  King king = kings.getKings().get(rand.nextInt(kings.getKings().size()));
  if (king.getFrom() > 0 && king.getTo() < 9999)
   return String.format("Kind regards: %s, %d - %d", king.getName(),
    king.getFrom(), king.getTo());
  if (king.getFrom() > 0) return String.format("Kind regards: %s, %d -",
   king.getName(), king.getFrom());
  if (king.getTo() < 9999) return String.format("Kind regards: %s, - %d",
   king.getName(), king.getTo());
  return "Kind regards: " + king.getName();
 }
}
```

Please note that it is a normal service, and since *onBind()* is trivial, it is not a service that clients can connect to, but a service that starts and runs until it's done, and in this case it's until *onDestroy()* is performed. *onStartCommand()* is simple, but it starts a timer ticking every 20 seconds, and every time it ticks, it creates with the last method a greeting from a Danish king, which is sent to the method *writePost()*. This method uses the database adapter to update the database with the current greeting. Here you should note that the method updates the same row every time (the database table has only 1 row) and the reason is that the database should not grow beyond all limits. The goal is only to show that a service can write to a database.

Running the *LaunchPosts* application opens the following window:

The window has 6 components. The top two are *TextView* components and are updated if you click the buttons. If you click on the top button, the program reads the database and the fields are updated with the record that is read. The next tests where the service is running and updates the top *TextView* component. The Button *Start* starts the service, while the last stops the service. The user interface is of course simple, but the class *MainActivity* contains some code:

```
package dk.data.torus.launchposts;

import android.app.ActivityManager;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
 private TextView txtDate;
 private TextView txtPost;
 private PostService service;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 txtDate = (TextView) findViewById(R.id.txtDate);
 txtPost = (TextView) findViewById(R.id.txtPost);
 service = new PostService(this);
}

public void onCheck(View view) {
 checkState(service.getClass());
}

public void onStart(View view) {
 if (!isRunning(service.getClass()))
 {
  startService(new Intent(this, service.getClass()));
  if (isRunning(service.getClass())) txtDate.setText("Post service started");
  else txtDate.setText("Service could not be started");
 }
 else txtDate.setText("Service is already started");
}

public void onStop(View view) {
 stopService(new Intent(MainActivity.this, PostService.class));
 if (isRunning(service.getClass()))
  txtDate.setText("Service could not be stopped");
 else txtDate.setText("Post service stopped");
}

public void onRead(View view)
{
 try {
  DbAdapter db = new DbAdapter(getApplicationContext());
  db.open();
  Cursor cursor = db.getRows();
  cursor.moveToFirst();
  if (!cursor.isAfterLast())
  {
   txtDate.setText(cursor.getString(DbAdapter.COLNO_TIME));
   txtPost.setText(cursor.getString(DbAdapter.COLNO_POST));
  }
  else txtDate.setText("Database is empty");
  cursor.close();
  db.close();
 }
```

```
    txtDate.setText(ex.getMessage());
   }
  }

  private boolean isRunning(Class<?> serviceClass) {
   ActivityManager manager =
    (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
   for (ActivityManager.RunningServiceInfo service :
    manager.getRunningServices(Integer.MAX_VALUE)) {
    if (serviceClass.getName().equals(service.service.getClassName())) {
     return true;
    }
   }
   return false;
  }

  private void checkState(Class<?> serviceClass)
  {
   ActivityManager manager =
    (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
   for (ActivityManager.RunningServiceInfo service :
    manager.getRunningServices(Integer.MAX_VALUE)) {
    if (serviceClass.getName().equals(service.service.getClassName())) {
     txtDate.setText("Post service: Running");
     return;
    }
   }
   txtDate.setText("Post service: Not started");
  }
}
```

In particular, note the two last methods. Here's the first the test where the servise identified by a *Class* is running. There is not much to explain in addition to noting that, but shortly, you use the method *getSystemService()* to determine which services are running and the result is an *ActivityManager* object. This object has a collection of running services, and you can traverse this collection to determine if that service is running. The last method works in the same way, but it updates the *TextView* component. The two methods are used from the event handlers, and here you should especially note how to indicates the *Class* object.

The event handlers *onStart()* and *onStop()* in principle do not contain anything new besides using the above methods, and the same applies to *onCheck()*. Finally, there is the event handler *onRead()*, which reads in the database, and it happens with the adapter. Note that the adapter's get methods returns a *Cursor* object, and you could make a nicer wrapper by returning a *King* object or a list of *King* objects instead.

If you look at the class *MainActivity* (and also the class *PostService*), there is nothing about that the database should be able to be used from another program. Nor should much happen, but only that the other program should use the same database adapter and then add an addition to the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  android:sharedUserId="dk.data.torus.app"
  package="dk.data.torus.launchposts">
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
```

```
    <service
      android:name="dk.data.torus.launchposts.PostService"
      android:enabled="true" >
    </service>
  </application>
</manifest>
```

Then to the second program, called *ReadPosts*. Firstly, the manifest of this program must be updated in the same way as shown above, so it has the same *sharedUserId*. The program's user interface is almost identical to the above, just missing the three bottom buttons. Then a copy of the class *DbAdapter* has been added, and finally there are *MainActivity*:

```
public class MainActivity extends AppCompatActivity {
 private TextView txtDate;
 private TextView txtPost;
 private Context sharedContext = null;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  txtDate = (TextView)findViewById(R.id.txtDate);
  txtPost = (TextView)findViewById(R.id.txtPost);
  try {
   sharedContext = this.createPackageContext("dk.data.torus.launchposts",
    Context.CONTEXT_INCLUDE_CODE);
  } catch (Exception e) {
   sharedContext = this;
  }
 }

 public void onRead(View view)
 {
  …
 }
}
```

Here the event handler *onRead()* is the same and all you need to note is that the same *Context* should be used for the database and how to determine this with *context.getResources()*.

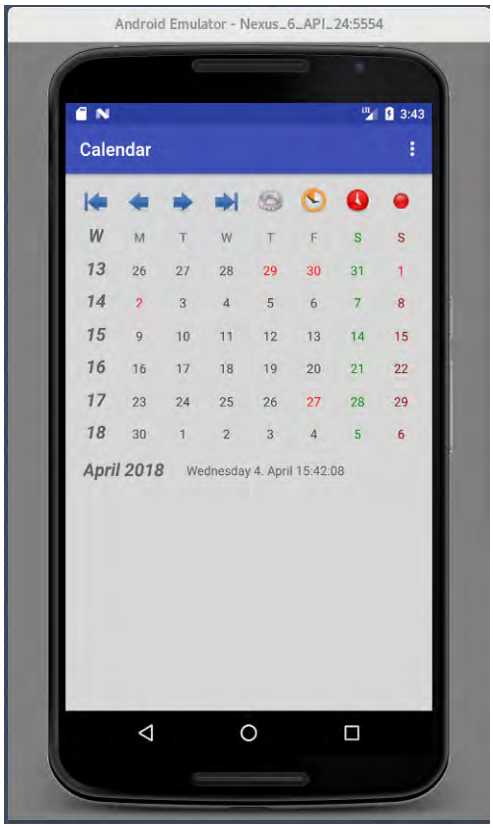
After that, both programs can be installed on the phone and it all should work. If you install the programs on a physical device, be sure to stop the service after you have tested the programs as it will continue and update the database every 20 seconds.

9 A FINAL EXAMPLE

As the final example of this book, I will write the same program, which was the final example of the book Java 8, and thus a calendar program where you can navigate the calendar and enter notes and appointments. The program should basically have the same features as in Java 8, and it is a goal to reuse as much of the design and code as possible. One can thus think of the task as a project, where you have to migrate a PC application to a mobile phone.

The purpose of the example is to show a slightly larger program and partly to apply many of the elements for developing mobile phone apps mentioned in this book. Regarding the requirements of the program, please refer to the Java 8 book, and the following will only focus on how I developed the program, what I have reused from the previous version, and what has been implemented differently, including primarily in relation to Android and mobile phones. The program is developed through a series of small iterations or sprints, and the following is a brief presentation of these sprints and including the most important decisions.

9.1 PROJECT START AND MAINACTIVITY



The overall design of the window is shown here, where the icons (buttons) in the toolbar are used for:

- to navigate the calendar a year back
- to navigate the calendar a month back
- to navigate the calendar a month onwards
- to navigate the calendar a year onwards
- to navigate the calendar for a particular month a certain year
- set a stopwatch
- set a count down
- set an alarm

The calendar's other functions are located in the main menu of the program and as a context menu for the calendar itself. The function export to XML is not included in this mobil phone version. Initially, only the first four buttons in the toolbar are implemented, so you can navigate the calendar one month back or forth and one year back or forth.

From the previous version of the program I have copied the two classes *Date* and *MainModel*. Both these classes are placed in a sub package called *models*. The class *Date* is completely

unchanged from the fact that the package name has been changed. In principle, this also applies to the class *MainModel*, but something here has no meaning yet and is therefore commented out.

The user interface is defined entirely in XML. It's extensive (fills a lot), but the user interface is simple to design using copy/paste, and in the code is defined references to individual components, and the result is a user interface that is simple and stable to update. As a result, the class *MainActivity*until this place is simple to write.

When reviewing the code, you should first notice how to create an *ImageButton* in XML, which is a button with an image. The individual images are added as resources to *res. drawable*. In the Java code, you should notice how to refer to the individual widgets using their name and not their ID. You should also note how the event handlers are linked to the four buttons in the code.

9.2 THE DATA ACCESS LAYER

The program should be able to maintain data regarding notes, appointments and anniversaries. In the original PC application, I used object serialization, but in the current version I would instead use a SQLite database. The goal of this iteration is to clarify the model to persist data. The result is as follows:

1. The classes *Note*, *Appointment* and *Anniversery* are copied from the Java 8 project. These model classes have not been changed.
2. The interface *Repository* is copied and it is also unchanged.
3. The class *Time* is copied. It is expanded with an additional constructor that parses a string on the form *HH:MM* to a *Time* object.
4. Similarly, the class *Date* is expanded with a constructor that parses a string on the form *YYYY-MM-DD* to a *Date* object.

The reason for the last two changes is that *SQLite* does not have types for *Date* and *Time* columns.

A package *dal* has been added to the project, and here a I have created a *DbHelper* class for three tables *notes*, *appointments* and *anniversaries*.

The class *IORpository* is copied from the original project. The class has the same interface (*Repository*), but otherwise the class has been completely modified so it now persists data in a SQLite database. The class has not been tested in addition to it has an error-free

translation, and the test is associated with the implementation of the individual calendar functions. One must therefore expect that there will be changes to the class during the following three sprints.

9.3 NOTES

This sprint is about implementing maintenance of notes and in principle includes 3 features

1. To create and maintain a note (CRUD)
2. To show an overview of all notes
3. To delete all notes within an interval of two dates

The two lower functions are selected from the program's options menu, while the first one is selected either from the context menu or from the list of notes. The result is that three new activities have been added to the project, but there are also significant changes to the class *MainActivity*.

Since all database operations are implemented in the previous iteration, it should be in place and there are no changes to these operations. However, the class *IORepository* and thus the interface *Repository* are expanded with a new method used to search notes by title.

The first of the above activities is called *NoteActivity* and is used for CRUD operations. It opens from the program's context menu, or from the activity *NotesActivity*. It is a simple activity and can be called with two parameters, which can be a *Date* object and is used if the activity is opened from the context menu, and it may be a *Note* object and used if opened from *NotesActivity*. In the latter case, you will see all the note's data, you can edit the note (not the date that specifies when the note is created) and you can delete it.

The other activity is called *NotesActivity* and shows an overview of all notes. If it has no parameter, it shows all notes, but you can search for a title that works like a filter. If the activity is called without parameter, it is from the program's options menu. If a note has been created for a specific day in the calendar, the context menu is expanded with an additional menu item called *Show notes*, and selecting this menu item opens *NotesActivity* again, but this time with the date as parameter, and the window only shows notes for the current day.

The last activity is called *RemoveActivity* and is very simple. It's only a matter of being able to enter two dates and then call a database operation that deletes all notes within that range. The function is performed from the program's options menu. Since it basic is the same

function that will be used for appointments and anniversaries, the activity has a parameter indicating what the function is to be used for.

Then there is *MainActivity* where there are changes. There are of course changes to open the above activities, but the biggest change concerns the context menu. First, each date field should be able to open its own context menu, which corresponds to that

```
registerForContextMenu()
```

must be performed for all *TextView* widgets that represent a date. In order to read the date, one should also be able to determine which *TextView* one has held his finger on, and you can not do that immediately. One solution is that when you touch a *View* object, an *onTouch()* method is performed, which is an event handler defined in the interface *OnTouchListener*. If you associate this event with all *TextView* components for dates, you can initialize an instance variable with the reference of the last viewed *View*, and when the event handler for the context menu is executed, you can determine which *View* relates to the event. It requires that you also know its coordinates (row and column), but it can be done by attaching a *tag* object to the component. It sounds a bit complex and can be called for a work around.

Another problem to be solved is that the context menu should be dynamic. In addition to the three fixed menu items, there must be an additional menu item for each of the three, if a note, appointment or anniversary has been created for that date. Finally, there must be an additional menu item if the date is a public holiday. For these reasons, the menu is now defined in the code (and the file *context_menu.xml* is removed).

Finally, it should be possible in the user interface to see (with a red color) and for a given date if there are extensions to the context menu. It requires a change in model (the three static methods previously commented out should be activated and changed slightly).

9.4 ANNIVERSARIES

In the next sprint, I will implement maintenance of anniversaries. In principle, they must be done in the same way as maintenance of notes, but it is simpler as most challenges are solved, and since the function can be implemented to a large extent using copy/paste. Basically, two activities must be added:

1. *AnniversaryActivitiy* to CRUD functions
2. *AnniversariesActivity* to overview of anniversaries

Both activities can be implemented by copying the corresponding activities for notes. In the same way as for the maintenance of notes, it has been necessary to add an additional search method to *IORepository* regarding anniversaries.

Finally, there is removing anniversaries, but it is already implemented in *RemoveActivity*. However, it has been necessary to change the method

```
public boolean deleteAnniversaries(SQLiteDatabase db, Date from, Date to)
```

in the class *IORepository*.

As part of this iteration, *MainActivity* has also been updated, but it's only a matter of updating the menus event handlers.

At the end of the iteration, the following changes have been made:

1. I have deleted the interface *Repository*.
2. I have changed the name of *IORepository* to the *Repository* and at the same time changed the class to a singleton.

The reason for this changes is primarily that it provides a slightly simpler code, which does not always require instances of the class *IORepository*. Another reason is that the *Repository* class contains several methods that are not used, but they are there because of the interface and as part of the conversion of the class *IORepository* to the current project. I want to remove these methods as part of the next sprint.

9.5 APPOINTMENTS

This sprint is essentially identical to the previous one, but it is less about copy/paste. The CRUD window is more complex as it also has to show an overview of other appointments for the same day, and also the program must validate that an appointment does not consolidate with an already created appointment. The code for that is taken from the PC version of the program.

After this sprint, all functions relating to the database are completed and all menu item functions are in place (except the clock menu item). Finally, the class *Repository* is updated where the unused methods are removed.

9.6 THE OTHER TOOLS

The task of this iteration is to implement the last functions, that is, the four tools in the toolbar and to turn the clock on and off.

The tool: Select a month

It's a simple feature. Choosing this tool in the toolbar opens an activity *GotoActivity*, where you can enter a year and select a month. If you click *OK*, you will return to *MainActity*, and the calendar will show the selected month. In addition to adding the *GotoActivity* class, a class *Month* has been added that is used to return the selected month.

The tool: Stopwatch

The stopwatch appears under the calendar as a button and a read-only *EditText* component, which displays the clock. By default, these two components are invisible. Clicking on the button in the toolbar displays the components and the clock starts. If you click on the button

again, the components disappear, and the button in the toolbar is thus an *on/off* button. If the stopwatch is visible and running, and you click the button below the calendar, it will stop the clock and you can start it again by clicking the button.

To implement the stopwatch, the following types are added to the project's model:

- *Stopwatch*
- *StopEvent*
- *StopListener*

These types are taken from the PC version and are largely unchanged. *Stopwatch* is in principle an enclosure of a timer ticking every second and every time the timer is ticking, a notification is sent as a *StopEvent* to a *StopListener*. It is an interface that *MainActivity* implements.

The tool: Count Down

The counter is displayed below the stopwatch as a button and a read-only *EditText* component, which displays the clock. By default, these two components are invisible. If you click on the button in the toolbar, you switch to an activity *WatchActivity*, where you have to set

the clock (the clock counts down to 0). Clicking on *OK* displays the components and the clock starts. If you click on the button again, nothing happens (it's not an *on/off* button), but clicking on the button below the calendar stops the counter and the two components become invisible again.

When starting a counter, the program simultaneously starts a service that runs to just before the counter is 0. When it occurs, the phone rings for 10 seconds and the *MainActivity* window opens, which it is not already. The service class is called *CounterService*.

In order to implement the counter, the following types are also added to the project model:

- *Watch, som er basisklasse til Counter*
- *TimeEvent*
- *TimeListener*
- *Counter*

These types are taken from the PC version and are largely unchanged. *Counter* is in principle an enclosure of a timer ticking every second and every time the timer is ticking, a notification is sent as a *TimeEvent* to a *TimeListener*. It is an interface that *MainActivity* implements. Notifications are also sent when the counter starts and stops.

The tool: Alarm

This tool is in principle identical to the above counter and appears in the window under the counter. The difference is that an alarm selects an alarm clock (with the activity *WatchActivity*) and the alarm occurs when the clock reaches this time. The following types have been added:

- *Alarm*
- *AlarmService*

The tool: Clock

It's a simple feature. In the menu you can choose to turn a watch on and off. The function consists only of starting a timer that ticks every second and the following types have been added to the model:

- *Clock*
- *ClockEvent*
- *ClockListener*

The AppointmentService

In fact, another feature still lacks, that must warning the user if it is time for an appointment. The function is implemented as a service that starts when the program is started (if it is not already running). The service tests for appointments every second minute:

1. If an appointment occurs within 5 minutes, the service sends a Toast.
2. If an appointment occurs within 15 minutes, the service sends a Toast.
3. If an appointment occurs within 1 hour, the service sends a Toast.

If the time for an appointment has passed, the following occurs:

1. The service in question is deleted in the database.
2. The phone rings for 2 seconds.
3. The service opens the Calendar's *MainActivity* (if it is not already).
4. The service will insert a notification in the window's action bar.

Appointments have a property *Save*, which indicates that an appointment should not to be deleted (a feature inherited from the PC version of the program). This feature does not really make sense, and as it complicates the current service, the option is removed. The database has not been changed, and the only thing that has happened is that in the acitivity *AppointmentActivity*, a *CheckBox* has been removed.

For the purpose of notifications from the service, the program's options menu is expanded with a function so that you can delete all notifications in the action bar.

The service starts when the application starts, if it is not already running.

Then there is the implementation of the service itself, which is not quite simple, and which uses to send notifications to the client, something that I have not mentioned. The service starts a thread that performs the work, and the code for this thread (at least the algorithm) is retrieved from the PC program. What complicates the service is that the thread must communicate with the program either in the form of Toast messages or notifications to the status bar. It is therefore necessary to create *Handler* objects for the communication. Finally, there is the code that sends notifications, which is quite complex and difficult to figure out. If you not only want to take it as it is, studying the online documentation best describes the meaning of it all.

9.7 A PROJECT REVIEW

The last iteration includes review of the code, further testing, and rectification of deficiencies and inconveniences.

Regarding the latter, two things have changed:

When maintaining notes, appointments and anniversaries, the user interface must be updated to see that a change has occurred for a date in the current month. Predictable, it applies only to changes from the options menu. The change consists primarily of switching activity using the method *startActivityForResult()* instead of *startActivity()*.

With regard to the service *AppointmentService*, it may not be desirable for this service to run all the time, at least not if it is a demo program. The option menu is therefore expanded with two new menu items for starting and stopping this service. You must therefore manually start the service, but then it should also run until the device is booted. One might consider instead starting the sevice at the start of the program (as it is now) and retaining the menu items and thus having the opportunity to stop the service.

Otherwise, only minor color adjustments, a new icon, and changes have been introduced so that you use the correct virtual keyboard.

The largest part of this iteration has been used on the test, and some errors are corrected.

If you look at the final result, there should be things that should be different, and for example, some of the application's activities should be replaced by dialogs, but here I lack the foundation so it has to wait for later. The biggest problem, however, is that the program does not work at all – at least not as desired. As long as the phone is switched on, it all works well in principle, but if the phone has gone to sleep, the three services can not wake the application, and so the idea goes a little to the ground. Thus there is another pendant to the next book.

APPENDIX A, INSTALLING OF ANDROID STUDIO

The following is a brief introduction to how to set up a development environment for Android. There are several options, and you can add a plugin to NetBeans, thus using your well-known development tool. However, most people use either *Eclipse* or *Android Studio*, and as I apply the latter, it is the product as the following reviews. An *Android* developer tool requires two things:

1. A common IDE, where you can enter and edit the code.
2. An emulator, which is a program running a virtual Android machine on your PC, simulating a mobile phone, for example, to test the programs you write before they are installed on a physical device such as a phone.

In fact, the integration between the two products is the challenge, as it seems from the developer must be easy and effective without big problems. That's why there are not so many who uses NetBeans as a platform for software development for Android.

To write Android applications you need two things:

1. *Android SDK* that best can be characterized as a number of libraries that contains all the components that an Android application may need.
2. *Android Tools*, which are a number of tools used in the development of apps, including the emulator and more.

When Android came on the market, the most widely used development environment was *Eclipse* combined with the above two products, and it has since been a stable and efficient development platform. When I mention Eclipse here, it's also because the product is worth acquainting for completely other reasons, as it is generally an effective tool for developing a Java application, and it's in every way an alternative to NetBeans, and many prefer Eclipse rather than NetBeans. There may be a long discussions about – and will be – which of the two products is the best and in fact it is probably the product that for the user is the best known and has the most experience with. In these books I have used NetBeans everywhere, but it is good to know that there is something called Eclipse and you can easily end up in a development organization using this tool, especially if the tasks relate to the development of apps for mobile phones or other mobile devices.

The two products *Android SDK* and *Android Tools* come from Google, as is the company behind Android's development. Together with these products, Google has also developed an IDE for the development of Android applications. It has been quite a long time underway, but today it has reached a stage where it is quite user-friendly and efficient tool. That's the reason why I'll use this tool everywhere, a tool which Google calls *Android Studio*.

In principle, it is quite simple to install Android Studio, as it is all packed into a single package that can be downloaded and installed instantly, and it is quite easy under Windows, whereas Linux requires a little more. In the following, I will outline what I have done to set up a development environment on a Fedora machine, and it's actually not easy to write a precise guide, partly because updates and changes are constantly being added and partly because the commands depend a little of the current Linux distribution. Fortunately, there are many guides online, and the best thing to do is follow these instructions, if it is manuals of recent date. The following are therefore primarily references to good guidelines.

In order for you to develop Android applications in practice, you need to have installed virtualization software, supported by a hardware accelerator, as the emulator otherwise runs intolerably slow. You can see how at the following addresses (for Fedora and Ubuntu, respectively):

```
https://fedoraproject.org/wiki/Getting_started_with_virtualization
```

```
https://software.intel.com/en-us/blogs/2012/03/12/how-to-
start-intel-hardware-assisted-virtualization-hypervisor-
on-linux-to-speed-up-intel-android-x86-emulator
```

The emulator itself is 32 bit software, and if you have a 64 bit machine (whatever you have), you should also install a special library called *ia32-libs*. For Fedora you can see how at the following address:

```
https://ask.fedoraproject.org/en/question/9556/how-do-
i-install-32bit-libraries-on-a-64-bit-fedora/
```

and for Ubuntu, the following document explains how, and also how to download and install *Android Studio*:

```
http://www.vogella.com/tutorials/Android/article.html
```

The document is also a brief introduction to what Android is, and also shows how to write the first application.

With the above referrals (or alternatives), setting up your development environment should run without the big challenges, but keep in mind that Android is a dynamic world that is constantly changing.