Poul Klausen

JAVA 17

More about Java and Android

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 17: MORE ABOUT JAVA AND ANDROID

SOFTWARE DEVELOPMENT

Java 17: More about Java and Android: Software Development
1st edition

ISBN 978-87-403-2422-8
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

The current book is the seventeenth in this series of books on software development in Java, and it is a continuation of the previous book on mobile phone applications development. The book requires a knowledge about Java and Android corresponding to what is dealt with in the book Java 16, and the goal is to introduce topics that were not available in this book. Among other things, the book will introduce how to use fragments that are important for developing larger applications and ensuring independence of the current device. Other important topics are to introduce how to use the facilities (such as GPS and Camera) that a telephone relative to a conventional PC makes available. Finally, the book will expand the substance regarding services and present solutions to some of the shortcomings mentioned at the end of the previous book. The goal is that the reader after reading the previous book and this book is able to in practice develop Android applications written in Java.

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. NetBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onward)
3. GlassFish as a web server and application server (from the book Java 11 onward)
4. Android Studio, that is a tool to development of apps to mobile 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 the previous book, I have dealt with how to write Java applications to an Android device and here primarily a mobile phone. Like so much else, it's a big area with many special issues, and compared to what has been said in the previous book, there's a lot left, and the goal of the following is to mention something of what's missing.

A mobile phone (and other Android devices such as a tablet) is a computer and is programmed in principle in the same way as any other PC, but there are two important differences:

1. It is a very small computer, both physically and in terms of capacity.
2. The device has other and more features than a traditional PC.

Slightly simplified, you can say that the previous book deals with what is special because of the first of the above differences, while this book deals with the last difference. For any computer program with a user interaction, the two key issues are:

1. How to program the application's user interface and for a mobile phone it is how to program an activity and navigates between the program's activities.
2. How to persist data, and for a mobile phone, it is done using a SQLite database that provides many of the same facilities as known from traditional database systems.

These two issues are addressed in the previous book, but compared to a conventional PC, a mobile phone provides other features as

- that you can make and receive phone calls
- that you send and receive SMS messages
- that using the GPS system you can determine the location
- that the device has a camera and you can take a picture
- that you can invoke the user's attention by vibrating the phone
- and certainly many others

That are features that are also available to the programmer. Compared to the finished program, it offers some new possibilities, but it also means learning how to use these features. Android makes it available in the form of classes, and for the most part, it's a matter of seeing what to write. This book provides typical examples of how the particular features can be used.

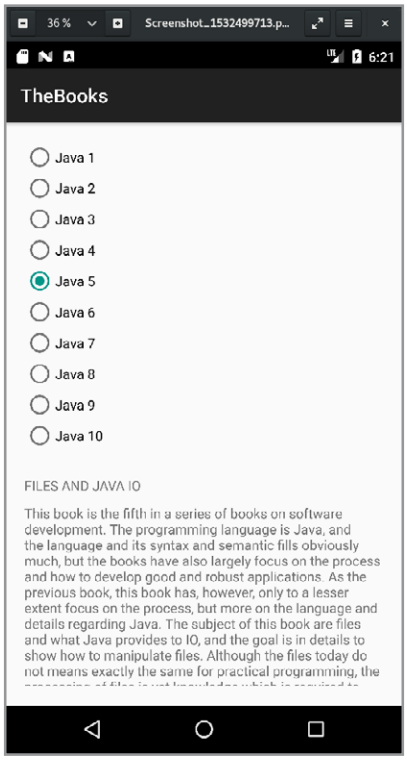
However, the book starts with fragments, which is the solution to develop applications that are flexible and customize the user interface to the screen of the current device. An activity should use the screen differently if the device is a tablet rather than a smaller phone. Finally, the current book has more about services that are extremely important in terms of developing applications for Android devices such as a phone.

2 FRAGMENTS

I want to start this book with fragments, which was actually something of what I lacked in the last example of the previous book. Just like an activity, a *fragment* is a layout, which is an XML document, that defines the layout and content of a window (part of a window = a fragment) and then a Java class. A window (an activity) can thus consist of several fragments, each fragment defining a part of the window. The use is that at run time you can replace a fragment with another fragment, and the window can thus get a different layout, and fragments can therefore often be used as an alternative to having more activities. An even more important the application, however, is a better use of the screen, which is particularly interested in larger screens like tablets of similar. Here the screen can accommodate more activities, and as this is not possible, you can instead view multiple fragments side by side. Finally, fragments are used for tabbed displays, list displays and better dialog boxes than what is shown in the previous book.

You should be aware that fragments are by no means an alternative to activities, but fragments can simplify design of an application's layout and facilitate easier reuse of a layout in multiple activities. An activity can consist of multiple fragments, but a particular fragment object can only be associated with one activity. You can also say that fragments group components of the user interface together with the associated logic.

I want to start with a simple example called *TheBooks* that is an application that does not use fragments. The program opens the following window:



The goal is to show how the program can be modified, so it uses fragments. It should be mentioned immediately that fragments are not the solution to everything, and if the application should only do what is the case in the application *TheBooks*, it makes no sense to use fragments, and in the current example, the goal is thus only to show what a fragment is , but first a little about how the application is made. The layout consists solely of *activity_main.xml*:

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

 <LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:layout_margin="20dp">

   <ScrollView
     android:layout_width="match_parent"
     android:layout_height="0dp"
     android:layout_weight="3" >
     <LinearLayout
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:orientation="vertical">
       <RadioGroup
         android:id="@+id/bookGroup"
         android:tag="1"
         android:layout_height="wrap_content"
         android:layout_width="wrap_content" >
         <RadioButton
           android:id="@+id/java1"
           android:tag="1"
           android:layout_height="wrap_content"
           android:layout_width="wrap_content"
           android:text="Java 1" />
         <RadioButton
           android:id="@+id/java2"
           android:tag="2"
           android:layout_height="wrap_content"
           android:layout_width="wrap_content"
           android:text="Java 2" />
```

```
            …
         </RadioGroup>
      </LinearLayout>
   </ScrollView>

   <ScrollView
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="2"
      android:layout_marginTop="20dp" >
      <LinearLayout
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:orientation="vertical">
         <TextView
            android:id="@+id/txtName"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
         <TextView
            android:id="@+id/txtDetails"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"/>
      </LinearLayout>
   </ScrollView>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

The layout is extensive as it defines 20 *RadioButton*'s (where I have only shown the first 2), but otherwise there is nothing new in the layout. It basically consists of two *ScrollView* elements in a vertically oriented *LinearLayout*, and the two *ScrollView* elements share the space in the ration 3:2. Note that each *RadioButton* element defines a *tag* attribute used to determine in the code which radio button is clicked.

With regard to the Java code, a class has been added that defines a book with two properties:

```
public class Book
{
 private String name;
 private String text;
 public Book(String name, String text)
 {
  this.name = name;
  this.text = text;
 }
 public String getName()
 {
```

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

and finally there is added a class that creates 20 *Book* objects:

```
public class Books
{
 private List<Book> list = new ArrayList();
 public Books()
 {
  for (String[] arr : data) list.add(new Book(arr[0], arr[1]));
 }
 public List<Book> getBooks()
 {
```

```
  return list;
 }
 private static final String[][] data = {
      { "BASIC SYNTAX AND SEMANTICS", "This book is the first … " },
      …
 };
}
```

Finally, there is the class *MainActivity*, where there is nothing to explain:

```
public class MainActivity extends Activity {
 private List<Book> books = (new Books()).getBooks();
 private TextView name;
 private TextView text;
 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  name = findViewById(R.id.txtName);
  text = findViewById(R.id.txtDetails);
  RadioGroup radios = findViewById(R.id.bookGroup);
  radios.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener()
  {
   @Override
   public void onCheckedChanged(RadioGroup group, int checkedId) {
    RadioButton radio = findViewById(checkedId);
    int n = Integer.parseInt(radio.getTag().toString()) - 1;
    name.setText(books.get(n).getName());
    text.setText(books.get(n).getText());
   }
  });
 }
}
```

So far, it's a simple application that does not add much new, but I want to divide the user interface into two fragments so each fragment contains one of the two *ScrollView* elements. I have started with a copy of *TheBooks* project and have called the copy for *TheBooks1*. As a first step, I've added a file *fragment_books.xml* to the *res* / *layout* directory:

```
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
```

```
    <RadioGroup
      android:id="@+id/bookGroup"
      android:tag="1"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content" >
      <RadioButton
        android:id="@+id/java1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:text="Java 1" />
      <RadioButton
        android:id="@+id/java2"
        android:tag="2"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:text="Java 2" />
      …
    </RadioGroup>
  </LinearLayout>
</ScrollView>
```

The content is nothing but a copy of the first *ScrollView* element from the *activity_main.xml* – with quite a few changes. It is necessary to add a *namespace* element to the *ScrollView* element, and then the value of the *width* attribute is changed from 0 to *wrap_content* just as the *weight* attribute is removed. The reason is that it should not be the fragment that determines how much it should fill, but instead the layout that uses the fragment.

Then, in the same way, I've added a layout for the last *ScrollView* element, which I've called *fragment_book.xml*:

```
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
      android:id="@+id/txtName"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
```

```
      android:id="@+id/txtDetails"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginTop="10dp"/>
  </LinearLayout>
</ScrollView>
```

A fragment consists (usually) in the same way as an activity of a layout in the form of a XML file and then a Java class. I have therefore added the following class for the first fragment:

```
package dk.data.torus.thebooks;

import android.os.Bundle;
import android.app.*;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class BooksFragment extends Fragment {
 public BooksFragment() {
 }
```

```
 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container,
  Bundle savedInstanceState) {
  return inflater.inflate(R.layout.fragment_books, container, false);
 }
}
```

Obviously, it's not easy to guess it should look like this, but initially you can notice that it is a class that inherits *Fragment*. When a fragment is created, Android will call multiple callback methods, where the most commonly to override is *onCreateView()*. This method returns the *View* hierarchy that represents the fragment and which Android associates with the current activity. The method *onCreateView()* has three parameters, the first being a reference to a *LayoutInflator*, an object that reads and uses a layout resource for the current activity. The other parameter is a reference to a *ViewGroup* object in the activity in which the fragment is to be linked. The last parameter is mentioned later. In this case, the method *onCreateView()* does not do much besides performing the method *inflate()* on the *LayoutInflater* object, where the parameter is the name of the resource as well as the container where the fragment should be inserted.

There is a corresponding class called *BookFragment*, where the code is almost identical to the above except from referring to another resource.

Then there is the layout *activity_main.xml*, which of course has to be changed. Here, most of the code is moved to the two fragments, and these code blocks must be replaced by fragment elements:

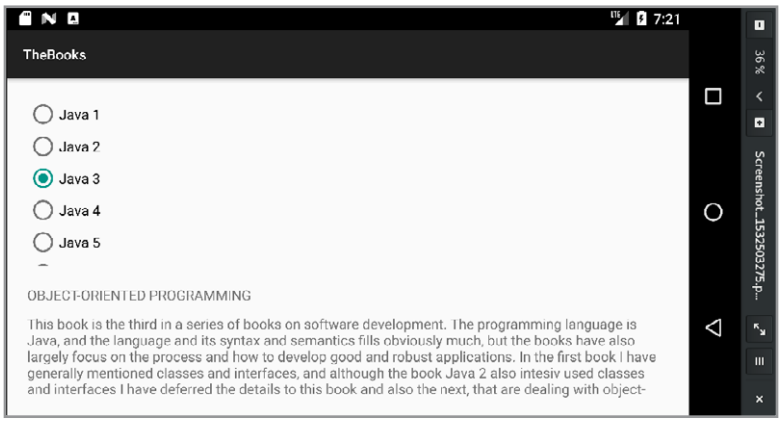
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="dk.data.torus.thebooks.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_margin="20dp">
    <fragment
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="3"
      android:name="dk.data.torus.thebooks.BooksFragment"
      android:id="@+id/fragmentBooks"/>
    <fragment
```

```
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="2"
        android:layout_marginTop="20dp"
        android:name="dk.data.torus.thebooks.BookFragment"
        android:id="@+id/fragmentBook"/>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

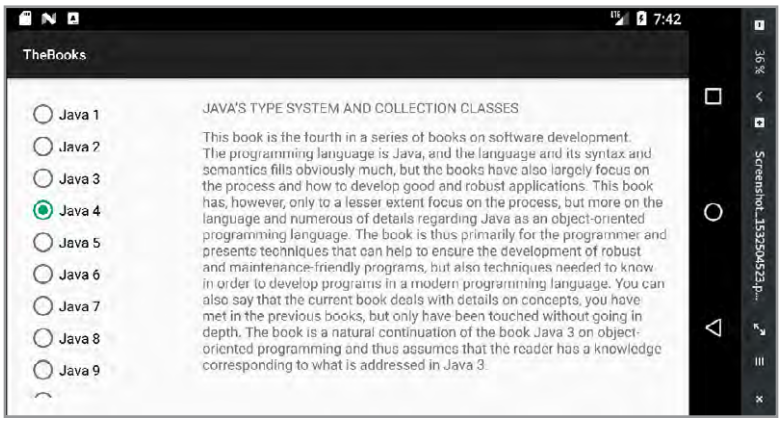
Each fragment element has an *id* attribute (which is not used in this example) and a *name* attribute that indicates what it is for a *Fragment* to be instantiated and associated. Then the new version of the program is complete and can be translated and run. The program works exactly the same as before, and the difference is that the user interface is now constructed using two fragments. As mentioned, in this case, there are no benefits, but you can notice that the layout is divided into smaller building blocks, which could then be used in other similar activities.

2.1 ROTATE THE PHONE

The above program shows two *ScrollView* components below each other, but rotating the phone so that it is held horizontally, you get a poor use of the screen as the part that contains the titles captures most of the screen:



Here it would be better with the following layout:



Of course, the solution is to define a different layout, so make sure it is used if the phone is held horizontally. Of course, this can be programmed, but since the layout is defined as a resource, it is a built in feature in Android that the system on runtime selects a resource depending on how the phone is held.

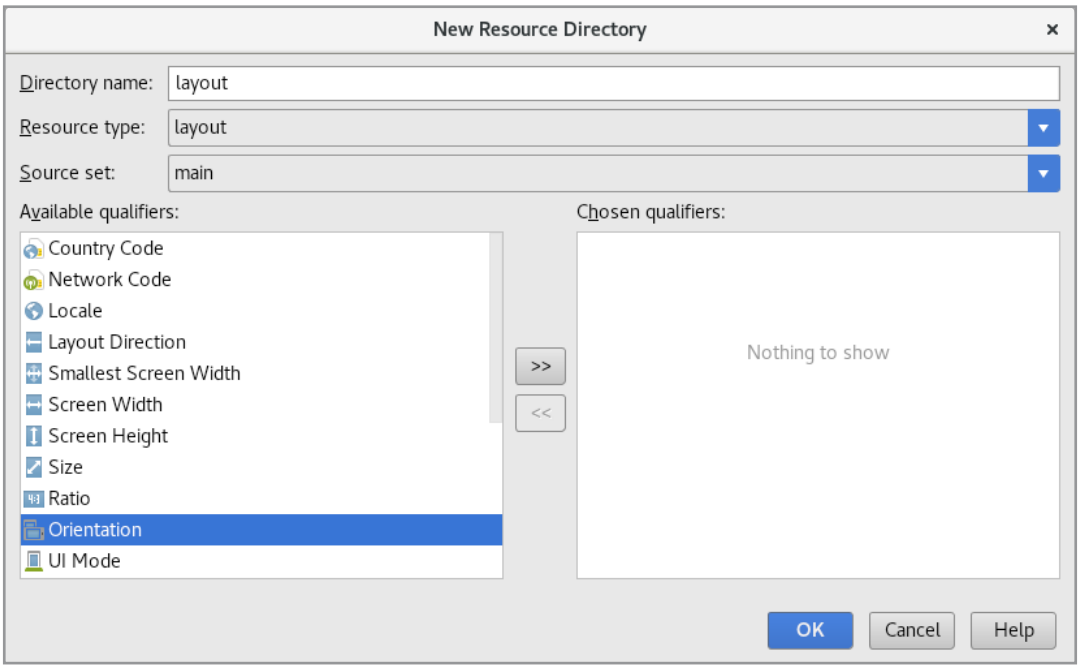
This is the first example where fragments can be useful. I'm going to write a **new** version of *activity_main.xml*, but it's a very simple layout resource, since all the work has been moved

into two other resources, and it's the same resources to be used regardless of whether it's the one or the second layout. I start with a copy of *TheBooks1* project, which I have called *TheBooks2*.

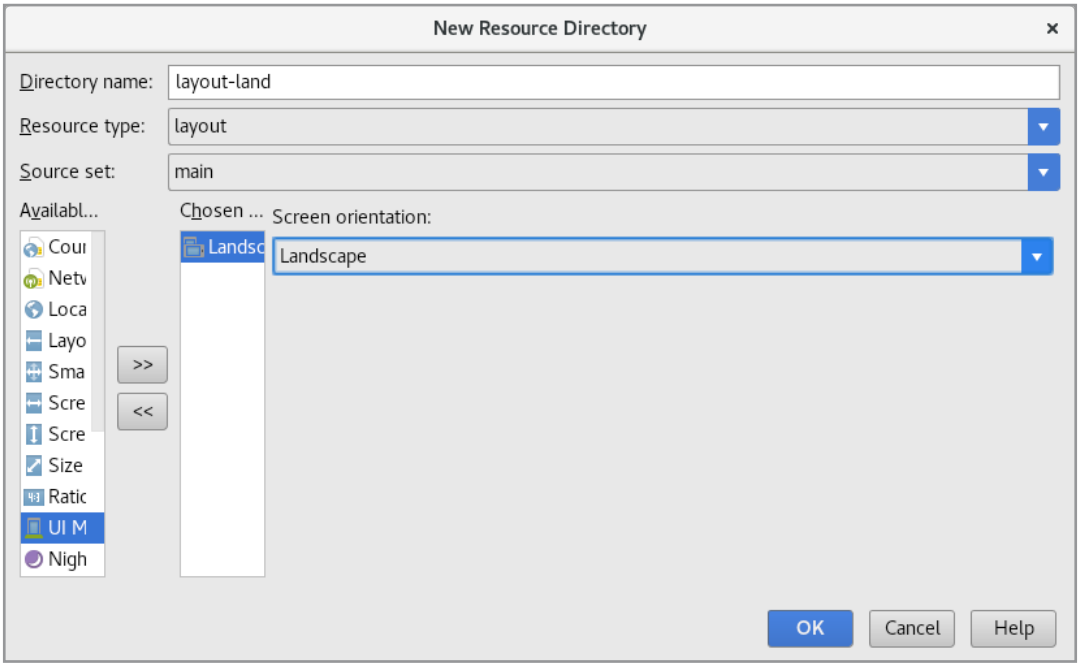
As a first step, you must add a new directory called *res / layout-land*. Doing this is easiest by right-clicking on the directory *res* and then selecting

```
New | Android resource directory
```

Then you get the following window:



where, as a *Resource type*, I have chosen *layout*. Next, under *Available qualifiers*, I have selected *Orientation*, and after I clicked on the button >> I get the following window:



Here I have under *Screen orientation* selected *Landscape*. When you click *OK*, then Android Studio creates a directory named *layout-land* under *res*.

You must then copy the file *activity_main.xml* to the new directory (if Android Studio does not display the new directory, you can do it from outside using *Files*). The project now has two versions of *activity_main.xml*, and here the new will be adjusted as it fits landscape mode. There is not much to do, and below is shown what is changed:

```
<?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.thebooks.MainActivity">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:layout_margin="20dp">
    <fragment
```

```
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:name="dk.data.torus.thebooks.BooksFragment"
        android:id="@+id/fragmentBooks"/>
    <fragment
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3"
        android:layout_marginLeft="20dp"
        android:name="dk.data.torus.thebooks.BookFragment"
        android:id="@+id/fragmentBook"/>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

After that, the program can be translated and tested and you will discover that if the phone is held vertically, the old design is used and held horizontal, the new design is used. In both cases, the same fragments are used and the runtime system takes care of (since there is now a *layout-land* directory) that it is the right design used for *MainActivity*.

2.2 THE SCREEN SIZE

Above I have shown how to get Android to apply a particular layout depending on the phone's orientation, where Android, based on a particular name convention, decides which resource to use. You can do the same with regard to the screen size. However, it is not quite simple as today there are many different screen sizes, among other things because you also have to take into account tablets.

I will start with a copy of the above project, which I have called *TheBooks3*. I will first implement the solution with regard to the rotation of the phone in a different way. I have changed the name of the version of *activity_main.xml* found under the directory *layout-land* to *activity_main_wide.xml*, and then I have copied the file to the directory *layout* and then deleted the directory *layout-land* again. The result is that the directory *layout* now contains four files:

As the next step, I have the same way as shown above (with the same wizard) added a resource directory:

```
New | Android resource directory
```

with the name *values-land*. For this directory, I have added the following XML file, which I have called *refs.xml*:

```
<resources>
  <item type="layout" name="activity_main">
    @layout/activity_main_wide
  </item>
</resources>
```

It is a resource in a *land* directory, and the result is that if the device is held horizontal, Android will see that the runtime system as layout for references to *activity_main.xml* it must use *activity_main_wide.xml* instead of *activity_main.xml*. If you try the application now, you will see that it all works again and that the phone uses a different layout depending on whether it is held vertical or horizontal. For the time being, there is no benefit in this solution compared with the previous example. The technique is called *layout aliasing*, and the advantage is that you can easily expand the solution to include devices with a different screen size.

Today, as mentioned, there are many screen sizes that vary in terms of physical size and resolution. Similarly, Android uses a measurement unit called *density independent pixel* and is called *dp*, and it is the unit of measurement that I have used throughout the previous book. 1 dp is defined as the physical size of 1 pixel on a 160 dpi device and is therefore independent of the screen's physical pixel size. For example, if you consider a 7 inch screen on a 160 dpi device, there are 1120 pixels along the diagonal. Is the screen format 16 × 9 it gives with a slightly rounding a resolution of 1000 × 600 dp.

Android defines screen sizes using *resource screen size qualifiers*, and here are three options that are defined using a resource directory:

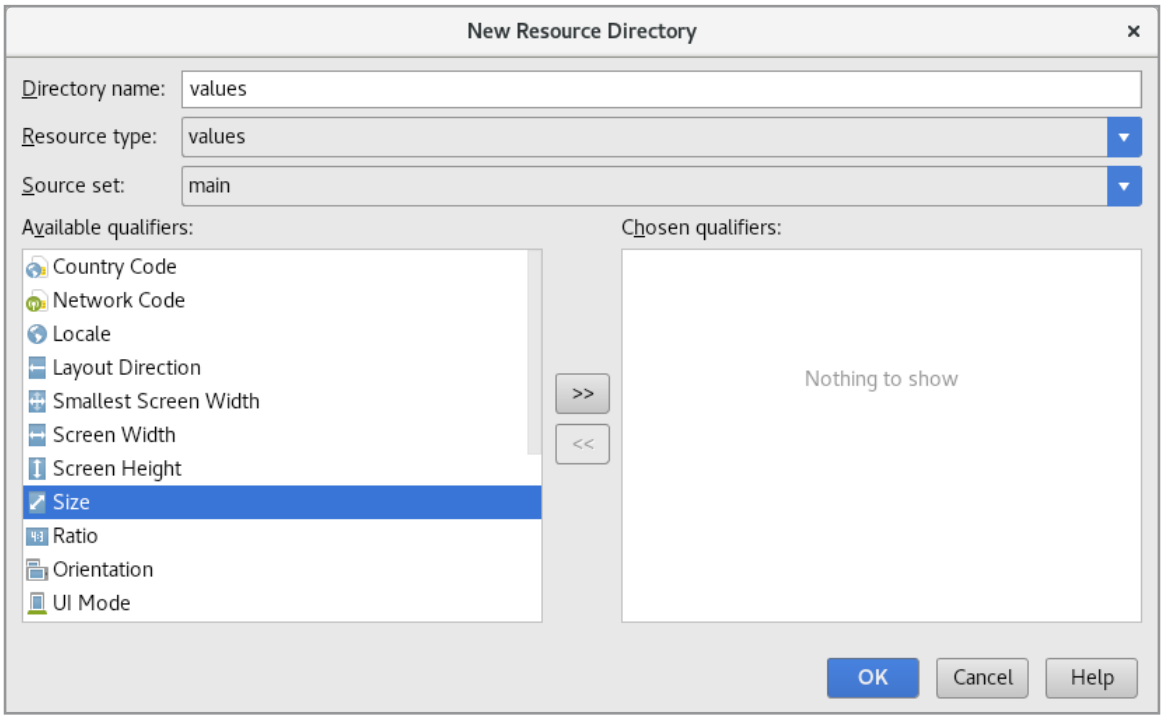
1. *Smallest width screen size qualifier*. It corresponds to the number of dp for the screen's narrowest point independent of the device orientation. The name of a resource directory is *sw*, followed by the desired screen size. For example, a layout resource directory for devices with a smallest width of at least 600 dp is called *layout-sw600dp*.
2. *Available width screen size qualifier*. It corresponds to the number of dp measured left to right for the device's current orientation. If you change the device orientation you also changes the available width. The name of a resource directory is *w*, followed the width. For example, a layout resource directory for devices with available width of at least 600 dp is called *layout-w600dp*.
3. *Available height screen size qualifier*. It corresponds to and behaves identically to the available width screen size qualifier, except that *h* is used instead of *w* and measured dp from top to bottom for the device's current orientation. A layout resource directory for a device with an available height for at least 600 dp is called *layout-h600dp*.

Instead of specifying resource directories using resource screen size qualifiers, you use an older way of splitting devices into four groups: *small*, *normal*, *large*, or *xlarge*, and a resource directory name may, for example, be *layout-large*.

In the present case, I have added a resource directory *values-large* in the same way as above to select

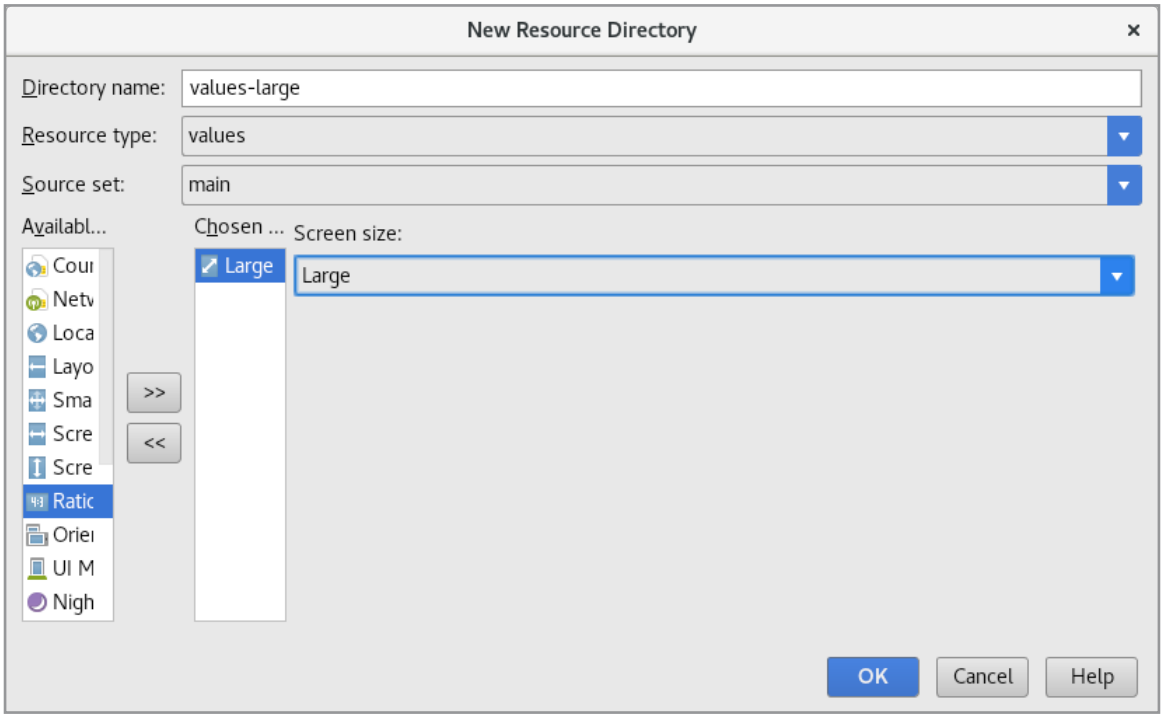
```
New | Android resource directory
```

In the following window I have selected values for *Resource Type* and selected *Size*, and when I click on the >> button I will select *large*. When you click OK then Android Studio creates the appropriate resource directory.


New Resource Directory
Directory name:
values
Resource type:
values
Source set:
main
Available qualifiers:
Country Code
Network Code
Locale
Layout Direction
Smallest Screen Width
Screen Width
Screen Height
Size
Ratio
Orientation
UI Mode
>>
<<
Chosen qualifiers:
Nothing to show
OK
Cancel
Help


It's only an opportunity if you act on it
IKEA.SE/STUDENT
IKEA
everyone deserves good design
save water. shower together
© Inter IKEA Systems B.V. 2009

Similarly, I have created another resource directory, this time called *values-v600dp*. The difference is that in the above screen, I will select the screen width name, and in the below window I will enter the name:



After creating these two directories, the file *refs.xml* from the *values-land* directory must be copied to both of the new directories.

After that, the program is complete and you will see, if executing the program on a phone if is running it in the same way as the previous example, but opening the application on a tablet, it will use *activity_main_wide.xml* each time regardless of whether the tablet is held vertically or horizontally. It has a screen that is always big enough to be used in landscape layout.

2.3 COMMUNICATION BETWEEN FRAGMENTS

The above program solve using fragments a number of challenges regarding Android apps, and the program works as desired. There are two fragments and the two fragments must communicate with each other so that if in one fragment is clicked on a radio button, the second fragment must be updated. In the program, this communication is solved by *MainActivity*, which receives a notification when a radio button is clicked and an event handler will then determine which radio button is clicked to find the data that the other fragment is to be updated with and then update the fragment's *TextView* components.

In principle, there is nothing wrong with it, but it means that the program's activity and fragments are closely linked, and it makes it more difficult to reuse the individual fragments in other activities.

In this section, I want to show a version of the program that implements a looser coupling of the fragments. The starting point is a copy of the previous version of the program, which I have called *TheBooks4*, and when you study the code, you will not immediately see the benefits, because the code fills more and is more complex, but for larger applications with multiple activities that use the same fragments the work are done well, as the result is a program that is easier to maintain. In the following, I will show what has changed, and the changes only concerns the Java code.

When objects in Java has to communicate and you want to use a loose coupling, the usual pattern is to define an interface. Similarly, I have added the following interface to the project:

```
public interface SelectBookListener {
 void onSelectedBookChanged(int id);
}
```

The interface defines a single method and the meaning is that the class *BooksFragment* must send a notification to a *SelectBookListener* if a radio button is clicked. The parameter of the method must be the index of the radio button that is clicked on. The class *BooksFragment* must be changed and the result could be:

```
package dk.data.torus.thebooks;

import android.os.Bundle;
import android.app.*;
import android.view.*;
import android.widget.*;

public class BooksFragment extends Fragment
 implements RadioGroup.OnCheckedChangeListener
{
 public BooksFragment() {
 }

 @Override
 public void onCheckedChanged(RadioGroup radioGroup, int id)
 {
  Activity activity = getActivity();
```

```
  if (activity instanceof SelectBookListener)
  {
   RadioButton radio = activity.findViewById(id);
   int n = Integer.parseInt(radio.getTag().toString()) - 1;
   ((SelectBookListener) activity).onSelectedBookChanged(n);
  }
 }

 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container,
  Bundle savedInstanceState) {
  View view = inflater.inflate(R.layout.fragment_books, container, false);
  RadioGroup group = view.findViewById(R.id.bookGroup);
  group.setOnCheckedChangeListener(this);
  return view;
 }
}
```

First, note that the class now implements the interface *RadioGroup.OnCheckeChangeListener*, and the class then must implement the method *onCheckedChanged()*. First, the method *getActivity()* is used to determine the activity to which this fragment is associated. If it is a *SelectBookListener* (what a *BooksFragment* object can not know – nor should it know),

the method determines which radio button is clicked (note that the code is copied from *MainActivity*). Then the type of the current activity is casted to a *SelectBookListener* object, after which the method *onSelectedBookChanged()* is called with the index of the book as a parameter. You should also note that *onCreateView()* has been changed so it now determines a reference to the layout's *RadioGroup* component and uses it to associate an event handler to the *RadioGroup* component.

The *BookFragment* class has also been changed:

```
package dk.data.torus.thebooks;

import android.os.Bundle;
import android.app.*;
import android.view.*;
import android.widget.TextView;
import java.util.List;

public class BookFragment extends Fragment {
 public static final List<Book> books = (new Books()).getBooks();
 private TextView name;
 private TextView text;

 public BookFragment() {
 }

 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container,
  Bundle savedInstanceState) {
  View view = inflater.inflate(R.layout.fragment_book, container, false);
  name = view.findViewById(R.id.txtName);
  text = view.findViewById(R.id.txtDetails);
  return view;
 }

 public void setBook(int id) {
  name.setText(books.get(id).getName());
  text.setText(books.get(id).getText());
 }
}
```

Note that the class now has three variables, all of which are copied from *MainActivity*. That is, the list of *Book* objects is part of this class, which is not natural, but to solve the reference problems, it is made *static*, so multiple instances of the class *BookFragment* do not create each their copy. A better option would be instead to place book information like

String resources. In other contexts, the information would be retrieved from a database, and the parameter to *setBook()* would then be a key. *setBook()* updates the two *TextView* components that create references to the components in *onCreateView()*.

Back there is *MainActivity*, which has now been changed to the following:

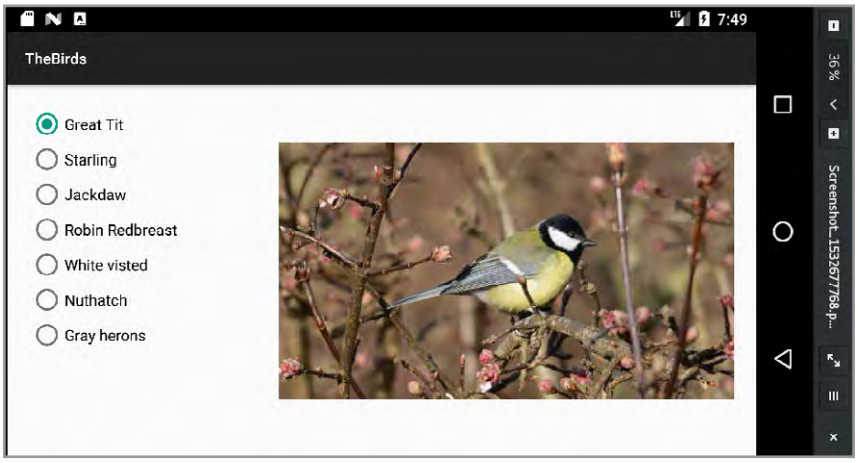
```
package dk.data.torus.thebooks;

import android.app.*;
import android.os.Bundle;

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

 @Override
 public void onSelectedBookChanged(int id) {
  FragmentManager fragmentManager = getFragmentManager();
  BookFragment bookFragment =
    (BookFragment) fragmentManager.findFragmentById(R.id.fragmentBook);
  if(bookFragment != null) bookFragment.setBook(id);
 }
}
```

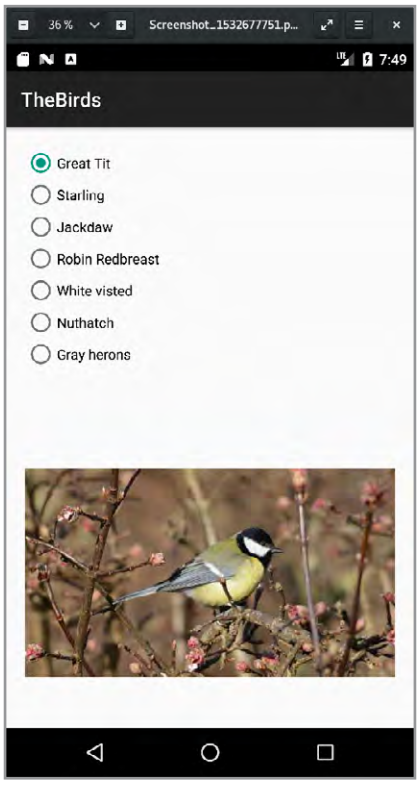
All variables are removed, and *onCreate()* is trivial (as Android Studio has created it). The class implements the interface *SelectBookListener* and therefore needs to implement the event handler *onSelectedBookChanged()*. It is called from *BooksFragment* when clicking on a radio button and the parameter is the index of the book that is clicked. When that happens, you should call the method *setBook()* in the class *BookFragment*. You can not immediately refer to a *Fragment* from an activity, but you can do it with a *FragmentManager* object, and any activity has a method that returns such an object. With this object available, you can determine a reference to the fragment and succeed, you can call *setBook()*.

After that, the changes have been implemented – after a relatively long way – but the result is a solution with weaker couplings between the classes.

EXERCISE 1: THEBIRDS

You must write an application that you can call *TheBirds*. The book's source directory contains a sub directory of 7 pictures. When the program opens, it must show an overview of these pictures as well as the first picture. When you click the name of a picture, it is that picture that will appear in the window. The user interface must consist of two fragments, one showing the list, while the other shows the picture. Additionally, the program must support the phone's rotation. You do not have to take into account different screen sizes.





2.4 TWO ACTIVITIES

The above application is a so-called *list / detail* application and shows two different layouts depending on how the phone is held or if the device is a tablet. It's an application that shows an overview of a number of objects (here that is book titles) and clicking on a book will show you the details. If the phone is held vertically, the list is displayed at the top, and if the phone is held horizontally or in the case of a large screen, the list is displayed on the left and the details on the right. Such applications are widely used, but for space reasons, it is often interesting when the phone is held vertically, that the entire screen is used to the list, and when you touch an object to view details they appear in another activity. In this section I want to show what is needed for the above application to work that way.

I start with a copy of *TheBooks4* project, which I have called *TheBooks5*. First, I've added a new empty activity to the project, which I've called *BookActivity*. The layout is as follows:

```
<?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.thebooks.BookActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_margin="20dp">
    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="12pt"
      android:text="&lt;&lt;"
      android:onClick="goBack"/>
    <fragment
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="2"
      android:layout_marginTop="20dp"
      android:name="dk.data.torus.thebooks.BookFragment"
      android:id="@+id/fragmentBook"/>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

In addition to a button, it is only a matter of the fragment element being copied from the layout *activity_main.xml*. This element must at the same time be removed from *activity_main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="dk.data.torus.thebooks.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_margin="20dp">
    <fragment
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="3"
      android:name="dk.data.torus.thebooks.BooksFragment"
      android:id="@+id/fragmentBooks"/>
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

The result is that *MainActivity* in the case where the phone is held vertically will have the entire screen available for the list. If your phone is held horizontal or is it a tablet, the program works as before and uses the layout *activity_main_wide.xml*.

Below is shown the class of the new activity:

```
public class BookActivity extends Activity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_book);
  Intent intent = getIntent();
  int id = intent.getIntExtra("bookIndex", -1);
  if (id != -1) {
   FragmentManager fm = getFragmentManager();
   BookFragment bookFragment =
    (BookFragment) fm.findFragmentById(R.id.fragmentBook);
   bookFragment.setBook(id);
  }
 }
```

```
 public void goBack(View view)
 {
  Intent intent = new Intent(this, MainActivity.class);
  startActivity(intent);
  finish();
 }
}
```

In *onCreate()* the program test if a parameter has been transferred (which is, if any, the index on a book), and if so, a *FragmentManager* is used to find a reference to the *Fragment* object and the method *setBook()* is called.

Back there is the class *MainActivity*, where the event handler *onSelectedBookChanged()* needs to be updated:

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

 @Override
 public void onSelectedBookChanged(int id) {
  FragmentManager fm = getFragmentManager();
  BookFragment bookFragment =
    (BookFragment) fm.findFragmentById(R.id.fragmentBook);
  if(bookFragment == null || !bookFragment.isVisible()) {
   Intent intent = new Intent(this, BookActivity.class);
   intent.putExtra("bookIndex", id);
   startActivity(intent);
  }
  else bookFragment.setBook(id);
 }
}
```

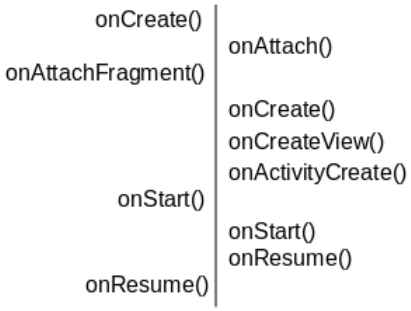
In the same way as previously the program attempts to determine a reference to a *BookFragment*. If this is not possible (or such a fragment is not visible), the new activity opens with the book's index as a parameter, and otherwise nothing else will happen than the fragment is updated.

When you try out the program, please note that if it is a tablet or you held the phone horizontally, so it works as before.

2.5 THE FRAGMENT LIFE CYCLE

In the book Java 16 (section 4.4) I have mentioned the activities life cycle and the callback methods performed in conjunction with a state shift. There is something similar for fragments, and since fragments are linked to activities, there is of course a close connection. Here it is worth noting that it is often the case, that an activity is created, closed down and re-created many times, and a good example of an event is if the phone is rotated.

Just as there is for an activity associated a number of callback methods, the same applies to fragments, and of course, there is a connection as to when these methods are performed. When a fragment is initialized, the following callback methods are performed, where the methods on the left relate to the activity, while the methods to the right relate to the fragment:



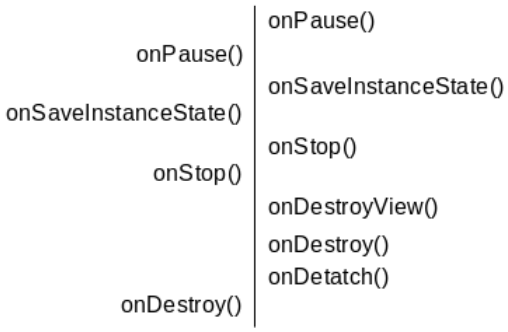
The most important method is *onCreateView()* as illustrated in the previous examples. When a fragment is associated with an activity, a method *onAttach()* is performed, and then the activity is known in the fragment and you can use the method *getActivity()*. Then the fragment sends a notification to the activity, after which *onCreate()* and *onCreateView()* are executed, where the layout is created in the last. The two methods are the most important and for many fragments they are the only callback methods that are overridden. The difference is that if the initialization has actions that takes time as connect to a data source, complex calculations and allocation of resources it should be performed in *onCreate()* and the rest and the primary the initialization of the layout should be done in *onCreateView()*. The reason is that it is often only the fragment's view hierarchy that is shut down and therefore it is not necessary to perform time-consuming initialization every time something happens to the phone.

The four methods

1. *onAttach()*
2. *onAttachFragment()*
3. *onCreate()*
4. *onCreateView()*

are always performed together, which is important if an activity is associated with multiple fragments. When this sequence of methods is performed for all fragments, the remaining callback methods can be performed individually. Here *onActivityCreated()* tells the fragments that the activity is created, while *onStart()* and *onResume()* basically have the same function for a fragment as the case is for an activity.

When a fragment is hidden or is shut down, something similar happens:



Basically, it is the same as for an activity, but after *onStop()*, for each fragment three methods are performed:

1. *onDestroyView()*
2. *onDestroy()*
3. *onDetach()*

and they are performed together. Then the fragment is no longer associated with an activity and you can no longer perform *getActivity()*.

There are several reasons for this separation of how to create and how to close a fragment and, among other things, be utilized that if a fragment is hidden, it is only *onDestroyView()* that is executed and once the fragment is visible again, it is only *onCreateView()* that is performed.

You should also be aware of the method *onSaveInstanceState()*, which has the same function as for an activity and is used to persist a fragment's state before it is closed.

2.6 A LISTFRAGMENT

If you look at the example that I have used in this chapter, the overview of books are listed with radio buttons. Here it would be more obvious to use a *ListView* in the fragment with the overview, and it would also be more flexible if the program should be expanded with new books. The subject of this section is to replace the radio buttons with a *ListView*. Immediately, it looks simple, but you can not just change the fragment_*books.xml* layout and replace all *RadioButton* components with a *ListView* component. Instead, it is necessary to change the *BooksFragment* class to a *ListFragment*, but it is also quite simple.

I starts with a copy of *TheBooks4* project and I have called the copy *TheBooks6*. However, I will start by modifying the data source a bit so that a *Book* object gets an additional field:

```
public class Book
{
 private String name;
 private String title;
 private String text;
 public Book(String name, String title, String text)
 {
  this.name = name;
  this.title = title;
  this.text = text;
 }
```

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

 public String getTitle()
 {
  return title;
 }

 public String getText()
 {
  return text;
 }

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

Of course, it also requires that the class *Books* is changed, as an additional parameter must now be added when creating a *Book* object. In addition, the class has been changed to a singleton:

```
public class Books
{
 private static Books instance = null;
 private List<Book> list = new ArrayList();

 private Books()
 {
  for (String[] arr : data) list.add(new Book(arr[0], arr[1], arr[2]));
 }

 public static Books getInstance()
 {
  if(instance == null){
   synchronized (Books.class) {
    if(instance == null) instance = new Books();
   }
  }
  return instance;
 }
```

```
 public List<Book> getBooks()
 {
  return list;
 }

 private static final String[][] data = {
      { "Java 1", "BASIC SYNTAX AND SEMANTICS", "This … " },
      { "Java 2", "PROGRAMS WITH A GRAPHICAL USER INTERFACE", "This … " },
      …
 };
}
```

With these changes, the class *BookFragment* can be updated as follows:

```
public class BookFragment extends Fragment
{
 private TextView name;
 private TextView text;

 public BookFragment() {
 }
```

```
 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container,
  Bundle savedInstanceState) {
  View view = inflater.inflate(R.layout.fragment_book, container, false);
  name = view.findViewById(R.id.txtName);
  text = view.findViewById(R.id.txtDetails);
  return view;
 }

 public void setBook(int id) {
  List<Book> books = Books.getInstance().getBooks();
  name.setText(books.get(id).getTitle());
  text.setText(books.get(id).getText());
 }
}
```

That is, the variable *book* has been removed, and the method *setBook()* is changed a bit.

In addition, the changes have nothing to do with the fact that I want to use a *ListView* instead of *RadioButton* components. These changes take place in the class *BooksFragment*. Instead of *Fragment*, the class must inherit *ListFragment*. It is a specialization of a *Fragment* that encapsulates a *ListView* component and thus a fragment with a specific purpose and layout. The fragment therefore does not use a XML file with the layout. The class *BooksFragment* must be changed to the following:

```
public class BooksFragment extends ListFragment {
 public BooksFragment() {
 }

 @Override
 public void onActivityCreated(Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  List<Book> books = Books.getInstance().getBooks();
  ArrayAdapter<Book> adapter = new ArrayAdapter<Book>(getActivity(),
   android.R.layout.simple_list_item_1, books);
  setListAdapter(adapter);
 }

 @Override
 public void onListItemClick(ListView l, View v, int position, long id) {
  Activity activity = getActivity();
  if (activity instanceof SelectBookListener)
  {
   ((SelectBookListener) activity).onSelectedBookChanged(position);
  }
 }
```

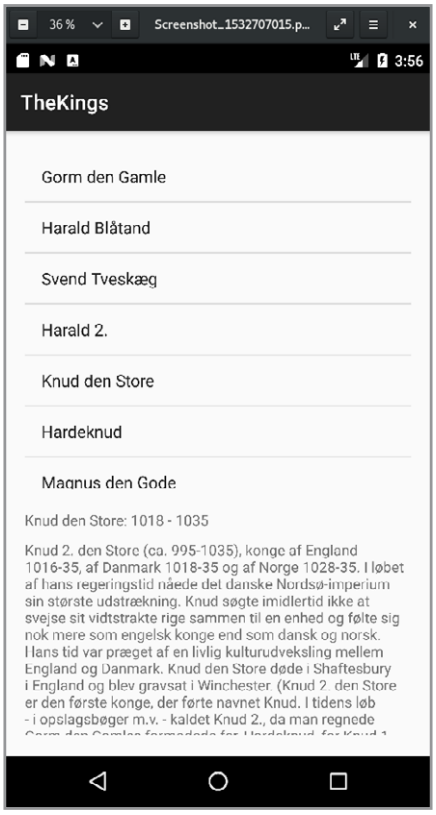
Note that the class now inherits *ListFragment* and that there is no *onCreateView()* method. Instead, the callback method *onActivityCreated()* is overriden, and it usually creates an adapter and initializes the fragment's data source. This happens with *setListAdapter()*. Here you should be aware that the class *ListFragment* has a method *getListView()*, which returns the encapsulated *ListView* object. You can therefore use the *ListView* class methods as usually, with one exception: You can not initialize the component's data with the *ListView* class's *setListAdapter()*.

To send a notification when selecting a name in the list, the event handler *onListItemClick()* is overridden to send a notification to a listener – if it is a *SelectBookListener*.

Then the program is complete and works now as before. The layout *fragment_books.xml* is no longer used and can be deleted.

EXERCISE 2: THEKINGS

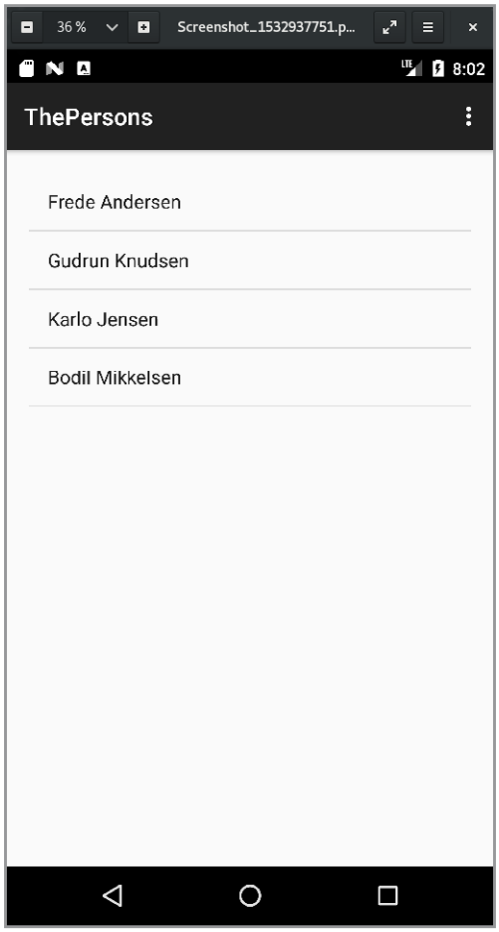
You must write an application similar to the above (*TheBooks6*), but instead, the application must show an overview of *Danish* kings. If you open the application, you must get the following windows where clicked on *Knud den Store*:



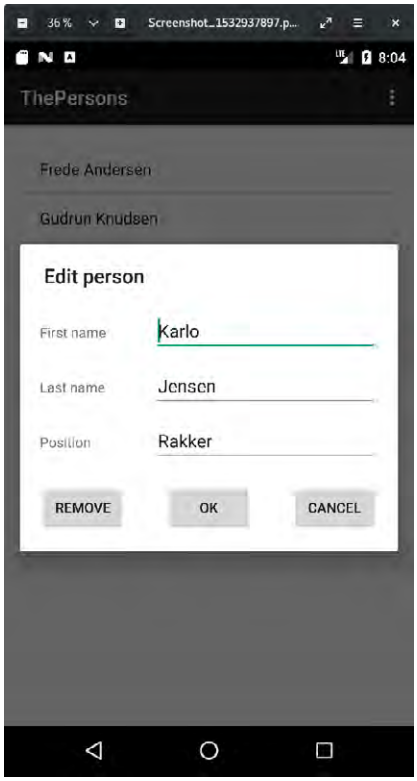
Start creating a new project that you can call *TheKings*. In the previous book there is a project called *AboutKings*, which contains two classes *King* and *Kings*. Copy these classes to your new project. The class *King* should not be changed, but you should change the class *Kings* so it's a singleton. Then the application is written as shown above.

2.7 A DIALOGFRAGMENT

A fragment can also be used to open a dialog box. In the previous book, I have discussed dialog boxes, and although the class *Dialog* makes much more available for dialog boxes, it is easier and better to wrap a *Dialog* object in a fragment and that is exactly what the class *DialogFragment* does. For example, it solves a problem by rotating the phone if a dialog box is open. The program *ThePersons* opens the following window:



which shows 4 people in a list. Clicking on a name opens a dialog box (see below), where you can edit information about the person and optionally delete the person. The program has a menu with a single menu item and clicking on it opens the same dialog box where you can create a person instead.



The example actually has some code. Not because it is difficult to use a *DialogFragment*, but because there are nevertheless some details that you need to know and here how to transfer data to the dialog and receive notifications. The program is born with 4 people just for the window to show something, but you should note that the changes you make (people that are created or modified) are not saved. Of course, it would be relatively simple to add this option.

First, a model class is defined for a person:

```
public class Person implements java.io.Serializable
{
 private static int lastKey = 0;
 private int key;
 private String firstname;
 private String lastname;
 private String position;
```

Here, there is not much to be aware of, but a person is identified by a key that is assigned continuously when *Person* objects are created, and that is the purpose of the static variable *lastKey*. Also note that the class is defined *Serializable* as an object of the class should be able to be transferred as a parameter to a dialog box. There is also a class that is a collection of *Person* objects:

```
public class Persons
{
 private static Persons instance = null;
 private List<Person> list = new ArrayList();

 private Persons()
 {
  list.add(new Person("Frede", "Andersen", "Skarpretter"));
  list.add(new Person("Gudrun", "Knudsen", "Heks"));
  list.add(new Person("Karlo", "Jensen", "Rakker"));
  list.add(new Person("Bodil", "Mikkelsen", "Sypige"));
 }

 public static Persons getInstance()
 {
  if(instance == null){
   synchronized (Persons.class) {
    if(instance == null) instance = new Persons();
   }
  }
  return instance;
 }
```

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

 public void update(Person person)
 {
  for (int i = 0; i < list.size(); ++i)
   if (list.get(i).getKey() == person.getKey())
   {
    list.set(i, person);
    return;
   }
  list.add(person);
 }

 public void remove(Person person)
 {
  list.remove(person);
 }
}
```

There is also not much to be aware of. The class is implemented as a singleton. The method *update()* is used to either change an existing person or add a new one.

The window (the program's activity) shows a list of *Person* objects, and it is nothing but a *ListFragment*, and the layout of the window is quite simple:

```
<LinearLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:layout_margin="20dp">
  <fragment
    android:layout_width="match_parent"
    android:layout_height="fill_parent"
    android:name="dk.data.torus.thepersons.PersonsFragment"
    android:id="@+id/fragmentPersons"/>
</LinearLayout>
```

where there is nothing else to note than the fragment element's *name* attribute refers to a *PersonsFragment*:

```
public class PersonsFragment extends ListFragment {
 public PersonsFragment() {
 }
```

```
 @Override
 public void onActivityCreated(Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  refresh();
 }

 @Override
 public void onListItemClick(ListView l, View v, int position, long id) {
  Activity activity = getActivity();
  if (activity instanceof SelectPersonListener)
  {
    ((SelectPersonListener) activity).personSelected(position);
  }
 }

 public void refresh()
 {
  ArrayAdapter<Person> adapter = new ArrayAdapter<Person>(getActivity(),
    android.R.layout.simple_list_item_1, Persons.getInstance().getPersons());
  setListAdapter(adapter);
 }
}
```

Here there is nothing new in relation to the previous section. Initialization of the *ListView* component has been moved in its own method so that it can be called from *MainActivity* when the list's data model is changed. An event handler to the list that determines a reference to the fragment's activity and if it has the type *SelectPersonListener* is called a method *personSelected()* which is a notification to the activity with which item (index for) that is clicked. *SelectPersonListener* is an interface that only defines the method *personSelected()*. If you run the program now (with the changes shown above) the window will open and display the list of the four people.

If you click (and that is, touching the finger) on a person, the program must open a dialog box with the person's data, and the same should be done if you select the *Add* menu in the main menu of the program. All fields must be blank at this time. A dialog box requires a layout that is customarily created as a resource, where in this case there must be three *TextView* components, three *EditText* components and three *Button* components. The layout fills a lot, and I do not want to display the XML code here, but it is a resource under the *layout* directory called *dialog_person.xml*, and it is nothing but the 9 components deployed using *LinearLayout* managers. In return, I will show the entire Java code:

```
public class PersonDialog extends DialogFragment {
 public static final int CMD_OK = 1;
 public static final int CMD_REMOVE = 2;
 private View dlgView;
 private Person person;
```

```
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setStyle(DialogFragment.STYLE_NORMAL, 0);
 person = (Person)getArguments().getSerializable("person");
}

public View onCreateView(LayoutInflater inflater, ViewGroup container,
 Bundle savedInstanceState)
{
 View view = inflater.inflate(R.layout.dialog_person, container, false);
 View cmdCancel = view.findViewById(R.id.cmdCancel);
 cmdCancel.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View view) {
   dismiss();
  }
 });
 View cmdOk = view.findViewById(R.id.cmdOk);
 cmdOk.setOnClickListener(new OkHandler());
 View cmdRemove = view.findViewById(R.id.cmdRemove);
 cmdRemove.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View view) {
```

```
    if (person.getLastname().length() > 0) {
     Activity parent = getActivity();
     if (parent instanceof OnDialogClickListener) {
      ((OnDialogClickListener)parent).onDialogClick(CMD_REMOVE);
      dismiss();
     }
    }
   }
  });
  ((EditText)view.findViewById(R.id.txtFname)).setText(person.getFirstname());
  ((EditText)view.findViewById(R.id.txtLname)).setText(person.getLastname());
  ((EditText)view.findViewById(R.id.txtJob)).setText(person.getPosition());
  Dialog dialog = getDialog();
  dialog.setTitle(person.getLastname().length() > 0 ?
   "Edit person" : "Add person");
  dialog.setCanceledOnTouchOutside(false);
  dlgView = view;
  return view;
 }

 public interface OnDialogClickListener {
  void onDialogClick(int buttonId);
 }

 class OkHandler implements View.OnClickListener {
  @Override
  public void onClick(View view) {
   String fname = ((EditText)dlgView.findViewById(R.id.txtFname)).
    getText().toString().trim();
   String lname = ((EditText)dlgView.findViewById(R.id.txtLname)).
    getText().toString().trim();
   String job = ((EditText)dlgView.findViewById(R.id.txtJob)).
    getText().toString().trim();
   if (fname.length() > 0 && lname.length() > 0 && job.length() > 0)
   {
    person.setFirstname(fname);
    person.setLastname(lname);
    person.setPosition(job);
    Activity parent = getActivity();
    if (parent instanceof OnDialogClickListener) {
     ((OnDialogClickListener)parent).onDialogClick(CMD_OK);
     dismiss();
    }
   }
  }
 }
}
```

First, note that the class inherits *DialogFragment*. This means that the class is a *Fragment* and that it is a fragment that is a wrapper of a *Dialog* object. A *DialogFragment* has a style that can be

1. STYLE_NORMAL (default)
2. STYLE_NO_TITLE
3. STYLE_NO_FRAME
4. STYLE_NO_INPUT

A dialog box consists of three parts, which are the layout area itself, a title and a frame. The first style constant indicates that all three parts must be displayed, the other that there should be no title, the third that there should be neither a title nor a frame, and finally the last that there should neither be a title nor a frame, and the dialog box should not receive input events. In this case, the style is STYLE_NORMAL, and you must define it in *onCreate()* with *setStyle()*. The last parameter to *setStyle()* is 0 and indicates that Android must decide which theme to use, and thus it is possible to set your own. When the dialog opens and *onCreate()* is executed, a *Person* object is transferred, and you should notice how to make a reference to this object with *getArguments()*.

Then there is *onCreateView()*, which this time is quite extensive. First of all, there must be associated event handlers with the three buttons. For the *Cancel* button, it's a simple handler, which should not do other things than to close the dialog. This happens with the method *dismiss()*. For the OK button, a handler of the type *OkHandler* is assigned, that is an inner class. Finally, there is the button *Remove* where the handler is created as an object of an anonymous class. The handler tests whether the person has a last name (and thus it is not a new person). If this is the case, the activity for the fragment is determined and if it has the type *OnDialogClickListener* (an interface defined as an internal interface in the class *PersonDialog*), a method *onDialogClick()* is called with the constant *CMD_REMOVE* as parameter. The goal is that the activity containing the fragment (and that is, displaying the dialog) will receive a notification that tells which button has been clicked.

After the three event handlers are attached to the buttons, the three *EditText* components are initialized with the values of the *Person* object that has been transferred. As the final step, a reference is made to the *Dialog* object that the *DialogFragment* is a wrapper for. It is used to define the text of the title. In general, a dialog box closes if you click somewhere outside the dialog box. If you do not want it, you can say that it should not be the case, which happens with the statement:

```
dialog.setCanceledOnTouchOutside(false);
```

Back there is the class *OkHandler*, but there are not so much to notice. The handler determines the values entered and if entered something in all three fields, the object is initialized and a notification is sent to the dialog box activity, which tells which button is clicked.

Then there is *MainActivity*, where you should notice that the class implements the two interfaces:

```
public class MainActivity extends Activity
 implements PersonDialog.OnDialogClickListener, SelectPersonListener
{
 public static final int MENU_ADD = Menu.FIRST + 1;
 private Person person;

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

 @Override
 public boolean onCreateOptionsMenu(Menu menu) {
  menu.add(Menu.NONE, MENU_ADD, Menu.NONE, "Add");
```

```
  return(super.onCreateOptionsMenu(menu));
 }

 @Override
 public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId() == MENU_ADD)
  {
   person = new Person("", "", "");
   openDialog();
  }
  return(super.onOptionsItemSelected(item));
 }

 public void personSelected(int pos)
 {
  person = Persons.getInstance().getPersons().get(pos);
  openDialog();
 }

 private void openDialog()
 {
  PersonDialog dlg = new PersonDialog();
  Bundle args = new Bundle();
  args.putSerializable("person", person);
  dlg.setArguments(args);
  dlg.show(getFragmentManager(), null);
 }

 public void onDialogClick(int id)
 {
  if (id == PersonDialog.CMD_OK)
  {
   Persons.getInstance().update(person);
   refreshList();
  }
  else if (id == PersonDialog.CMD_REMOVE)
  {
   Persons.getInstance().remove(person);
   refreshList();
  }
 }
 private void refreshList()
 {
  FragmentManager fm = getFragmentManager();
  PersonsFragment personsFragment =
   (PersonsFragment) fm.findFragmentById(R.id.fragmentPersons);
  personsFragment.refresh();
 }
}
```

The class creates a main menu with a simple menu item used to create a new person. If you select this menu item, a new *Person* object is created (with all three fields blank). Next, the method calls *openDialog()* that is the method that opens the dialog. Here you should first notice how to transfer the parameter and how to open the dialog with the method *show()* and where the parameter is the activity's *FragmentManager*. *personSelected()* works in the same way, instead of creating a new *Person* object, an existing object is determined from the transferred index, and it is the object that *openDialog()* transfers to the dialog.

Finally, note the method *onDialogClick()*, which is the method that receives notifications from the dialog. This happens in two instances where the *OK* button is clicked or where the *REMOVE* button is clicked.

2.8 FRAGMENT TRANSACTIONS

As the last example of fragments, I will show a program where, instead of opening a new activity, dynamically, I will replaces a fragment with another. In section 2.4, I showed a version of *TheBooks* application, where in the case of the phone being held vertically, the application uses the entire screen to the list and to view details, the program opens another activity. If the phone is held horizontally, the screen is divided into two fragments, showing the list and details, respectively. The example is called *TheBooks5*. I will now change the program so if the phone is held vertically and you click on a book in the list, then details will appear by dynamically replacing the fragment with the list by a another fragment that shows the details.

The starting point is a copy of the *TheBooks5* project, which I have called *TheBooks7*. Since the fragment now has to be assigned dynamic, and that is, in the Java code, the layout used if the phone is held vertically has been changed and that is, *actvity_main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="dk.data.torus.thebooks.MainActivity">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:id="@+id/layoutRoot"
    android:layout_margin="20dp">
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

So I have changed two things in which the *LinearLayot* element has an *id* (so it can be referenced from the code) and the fragment element has been removed, and the layout is reasonably trivial. Otherwise, the changes primarily concern the class *MainActivity*, but it is also necessary to change the class *BookFragment*, where two constants are added, just like *onCreateView()* is updated:

```
public class BookFragment extends Fragment {
 public static final String ID = "bookid";
 private static final int NO_ID = -1;
 public static final List<Book> books = (new Books()).getBooks();
 private TextView name;
 private TextView text;

 public BookFragment() {
 }
 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container,
  Bundle savedInstanceState) {
  View view = inflater.inflate(R.layout.fragment_book, container, false);
  name = view.findViewById(R.id.txtName);
  text = view.findViewById(R.id.txtDetails);
  Bundle args = getArguments();
```

```
  int id = args != null ? args.getInt(ID, NO_ID) : NO_ID;
  if (id != NO_ID) setBook(id);
  return view;
 }

 public void setBook(int id) {
  name.setText(books.get(id).getName());
  text.setText(books.get(id).getText());
 }
}
```

It is not entirely clear what these changes should benefit from, but the goal is that it should be possible to transfer a parameter to *onCreateView()* which must be the index of the book selected by the user from the list. The first constant is alone a name to identify the parameter while the other is the default value indicating that an argument has not been transmitted. A parameter or argument is encapsulated in a *Bundle* object, and this object is determined in *onCreateView()* with *getArguments()*, and if there is an argument, you try to determine the value as an *int*. Is it successful and it is not the default value, *setBook()* is called with the value as parameter.

On the other hand, the changes in *MainActivity* are more extensive:

```
public class MainActivity extends Activity implements SelectBookListener {
 private boolean isDynamic;

 @Override
 protected void onCreate(Bundle savedInstanceState)
 {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  FragmentManager fm = getFragmentManager();
  Fragment bookFragment = fm.findFragmentById(R.id.fragmentBook);
  isDynamic = bookFragment == null || !bookFragment.isInLayout();
  if (isDynamic) {
   FragmentTransaction ft = fm.beginTransaction();
   BooksFragment booksFragment = new BooksFragment();
   ft.add(R.id.layoutRoot, booksFragment, "bookList");
   ft.commit();
  }
 }

 @Override
 public void onSelectedBookChanged(int id) {
  FragmentManager fm = getFragmentManager();
  if (isDynamic)
  {
```

```
    FragmentTransaction ft = fm.beginTransaction();
    BookFragment bookFragment = new BookFragment();
    Bundle args = new Bundle();
    args.putInt(BookFragment.ID, id);
    bookFragment.setArguments(args);
    ft.replace(R.id.layoutRoot, bookFragment, "bookText");
    ft.addToBackStack(null);
    ft.commit();
   }
   else {
    BookFragment bookFragment =
      (BookFragment) fm.findFragmentById(R.id.fragmentBook);
    bookFragment.setBook(id);
   }
  }
  @Override
  public void onBackPressed() {
   if (getFragmentManager().getBackStackEntryCount() > 0 ){
    finish();
    startActivity(getIntent());
   } else {
    super.onBackPressed();
   }
  }
}
```

First, a variable *isDynamic* is introduced, which is used to control whether or not dynamic updating of fragments is used, and that is where the device is oriented vertically or horizontally. The variable is initialized in *onCreate()* using a *FragmentManager*, which tests whether the *BookFragment* fragment exists, and thus where the layout *activity_main.xml* or *activity_main_wide.xml* is used. If *isDynamic* is *true*, a fragment must be dynamically inserted. However, it requires that you use a *transaction*. A *FragmentManager* can start a transaction as a *FragmentTransaction*, for which you can add operations regarding fragments. In this case, only one operation is added, which is an *add()* operation that adds a *BooksFragment* to the activity. Here you should note that the *LinearLayout* element in the XML document now has an *id*. Also note the last parameter for the method *add()*, which simply means that the fragment gets a name, so it can be referred to later. The result is that when the *FragmentTransaction* object performs a *commit()*, then operations that are added to the transaction are performed as one single operation.

Also the event handler *onSelectedBookChanged()* has been changed. If the variable *isDynamic* is *true* (and the phone is held vertically) then the fragment with the list must be replaced with the fragment showing the details. Therefore, a transaction is created again. In addition, the creation of a *BookFragment* object and the *id* of the book that is selected is associated as

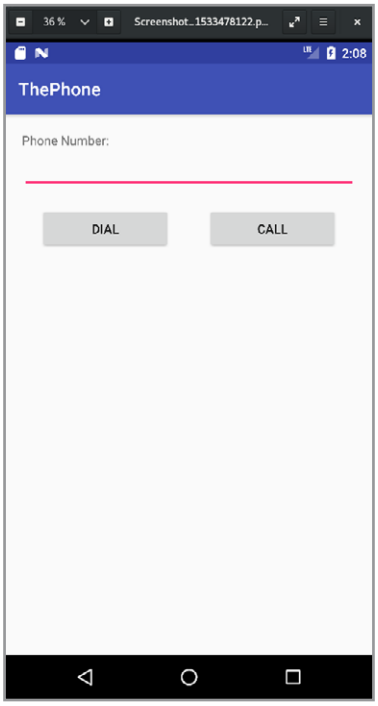
an argument with a *Bundle* object. Note that here you use the name defined as a constant in the *BookFragment* class. After the *BookFragment* object has been created and initialized with an argument, replace the fragment with the list, by adding a *replace()* to the transaction. Now, it may be so that the back button still works, so you must perform the operation *addToBackStack()* so that the current window (the current transaction) is stacked.

After that everything should work – at least in theory, but it does not apply to the back button if the phone is held vertically. Therefore, the last method only solves the problem to some extent. It may be a good job to search for a better solution.

After the program has been tested, I can finally delete *BookActivity.java* and *activity_book.xml* as these files are no longer used.

3 THE PHONE

When writing Android applications, it's a bit different from usual PC applications, simply because the devices often makes other facilities available, and if the device is a phone, you can make phone calls. In this chapter I will show how to do it in software and thus write applications where the user is able to make a phone call. From the programmer, there are two options, in which case it is the user of the phone that makes a call, while in the other case it is a question of writing a program that others can activate if they make a call to the current phone. Here is the last not quite simple. Firstly, it requires some service that receives notifications regarding calls, and there are some security issues in it, and the issues is not discussed here. The first problem, in which case, it is the user of an application that makes a call is quite simple, and the following shows in an example how. If you open the application *ThePhone*, you get the following window:



Here's an entry field, so you can enter a phone number, and there are two buttons that you can use to make calls to the number you have entered. If you click on the *DIAL* button, you get the usual window, like when you else make a phone call, and the number you entered has been inserted and you have as usual to press the *Call* button to make the phone call. If you click on *CALL*, the phone immediately executes a call without the user's intervention – if otherwise allowed. You must assign rights in *AndroidManifest.xml*, but with the latest phones it is not enough and the phone shows a simple dialog box where the user must accept that the phone is calling. In this case the manifest is updated as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.thephone">
  <uses-permission android:name="android.permission.CALL_PHONE"/>
  <uses-feature
    android:name="android.hardware.telephony"
    android:required="true"/>
  <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>
  </application>
</manifest>
```

The element *uses-permission* allows a program to make phone calls, without the user having to click the *Call* button. The element *uses-feature* is used to tell that the application can only be used (can only run) on a device that supports phone calls. Although most Android devices are still phones, the number of Android devices that are not grows, and on such devices it makes no sense to make phone calls. Note that instead, you can test where the device can make phone calls in software if the above procedures are too restrictive:

```
PackageManager device = getPackageManager();
if (device.hasSystemFeature(PackageManager.FEATURE_TELEPHONY) {
}
else {
};
```

I do not want to show the layout of the current application as it contains nothing new, but the Java code is as follows:

```
package dk.data.torus.thephone;

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

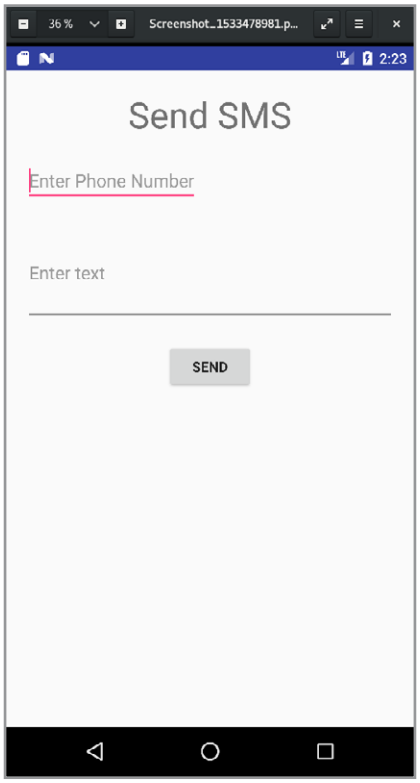
```
import android.telephony.*;
import android.net.*;
import android.Manifest;
import android.content.pm.PackageManager;

public class MainActivity extends AppCompatActivity {
 private static final int REQUEST_CODE = 100;

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

 public void onDial(View view) {
  TelephonyManager tm = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
  if (tm.getCallState() == TelephonyManager.CALL_STATE_IDLE) {
   String number = "tel:" +
    ((EditText) findViewById(R.id.txtNumber)).getText().toString();
   startActivity(new Intent(Intent.ACTION_DIAL, Uri.parse(number)));
  } else Toast.makeText(this, "The phone is in use", Toast.LENGTH_LONG).show();
 }
```

```
public void onCall(View view) {
 TelephonyManager tm = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
 if (tm.getCallState() == TelephonyManager.CALL_STATE_IDLE) {
  call();
 } else Toast.makeText(this, "The phone is in use", Toast.LENGTH_LONG).show();
}

public void call()
{
 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M &&
  checkSelfPermission(Manifest.permission.CALL_PHONE) !=
  PackageManager.PERMISSION_GRANTED) {
  requestPermissions(new String[] { Manifest.permission.CALL_PHONE },
   REQUEST_CODE);
 }
 else {
  String number =
   ((EditText) findViewById(R.id.txtNumber)).getText().toString();
  Intent intent = new Intent(Intent.ACTION_CALL);
  intent.setData(Uri.parse("tel:" + number));
  startActivity(intent);
  Toast.makeText(this, "Yoy have called " + number, Toast.LENGTH_LONG).show();
 }
}

@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
 int[] grantResults) {
 if (requestCode == REQUEST_CODE) {
  if (grantResults[0] == PackageManager.PERMISSION_GRANTED) call();
  else
   Toast.makeText(this, "You have no permission", Toast.LENGTH_LONG).show();
 }
}
}
```

The method *onDial()* is the event handler for the *DIAL* button and is quite simple. The first thing that happens is that the method determines a *TelephonyManager* object. It has a method *getCallState()* that is used to test if there is already a phone call in progress. It is not absolutely necessary, but recommended, otherwise you may interrupt an already ongoing conversation. If not, determine the phone number entered and note that it is preceded by the prefix *tel:* which tells that it is a phone number. Next, the call is executed with *startActivity()* and an *Intent* object. The first parameter to the *Intent* object indicates that a phone call must be made in the usual way, that is where the user get a dialog for entering the number while the other parameter are the number.

If you click on the *CALL* button, the event handler *onCall()* is performed, and it is similar to the above, but the call itself takes place in a method *call()*. Here you should note that the method *startActivity()* only can be performed if the permission *CALL_PHONE* is specified in *AndroidManifest.xml*. Otherwise, you simply get a translation error. The method starts by testing how old the phone is and if the manifest has the required permission. If it is a newer phone, it means that the phone opens a dialog box where the user must give permission for the phone to make a call. If you do so, the event handler will perform *onRequestPermissionsResult()*, which then performs *call()* again, but this time, so that's the *else* part of the *if* statement that is performed. The result is that a call is made.

3.1 SEND A SMS



Similar to the example above and how to make a phone call, I will in the following example show you how to send a SMS, and in principle it happens in the same way. The example is called *SendSMS*, and it opens the window above to enter the phone number and the message. I do not want to show the layout, but in the same way as for the use of the phone, the manifest must allow the application to send a SMS:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.sendsms">
  <uses-permission android:name="android.permission.SEND_SMS" />
  <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>
  </application>
</manifest>
```

Sending an SMS is quite simple and happens with the following method:

```
private void send()
{
 SmsManager sm = SmsManager.getDefault();
 sm.sendTextMessage(phone, null, message, null, null);
 Toast.makeText(getApplicationContext(), "Your SMS is sent",
  Toast.LENGTH_LONG).show();
}
```

and here the two variables *phone* and *message* must be initialized with the phone number and the message to be send, respectively. However, the conditions are a bit the same as for phone calls, although in *AndroidManifest.xml*, it has been stated that the application is allowed to send a SMS, it is not sure that the device allows it, but it opens a dialog box where the user explicitly accepts sending of the message. There must therefore be a little more code:

```
private void sendMessage() {
 phone = txtPhone.getText().toString().trim();
 message = txtText.getText().toString().trim();
 if (ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) !=
  PackageManager.PERMISSION_GRANTED) {
  if (!ActivityCompat.shouldShowRequestPermissionRationale(this,
   Manifest.permission.SEND_SMS)) ActivityCompat.requestPermissions(this,
    new String[]{ Manifest.permission.SEND_SMS }, REQUEST_SEND_SMS);
 }
 else send();
}

@Override
public void onRequestPermissionsResult(int
requestCode, String permissions[],
 int[] grantResults) {
 if (requestCode == REQUEST_SEND_SMS) {
  if (grantResults.length > 0 && grantResults[0] ==
   PackageManager.PERMISSION_GRANTED) send();
  else Toast.makeText(getApplicationContext(),
   "Error sending SMS, please try again", Toast.LENGTH_LONG).show();
 }
}
```

that is a reminiscent of what you saw in the previous example. The code is not so easy to review, so it is recommended to study the documentation, but briefly the following happens. The method

```
checkSelfPermission(this, Manifest.permission.SEND_SMS) !=
 PackageManager.PERMISSION_GRANTED)
```

tests whether the application in the manifest has specified the permission *SEND_SMS*, and if so the method

```
shouldShowRequestPermissionRationale()
```

is performed and a dialog box should appear to send a SMS. If that is the case, Android opens a dialog box, and when the user here clicks accept (or the opposite) the event handler is performed:

```
onRequestPermissionsResult()
```

4 USING THE CAMERA

In this chapter I will show you how you from a program can use the phone's camera. I will only show how to take a picture and save it to a file in external storage. In fact, it's quite simple, and when the code fills a lot, it's because you can not immediately save the image in a file. I want to show an example called *TheCamera*, which opens the following window where a picture has been taken:

When the picture is a little simple, it is because the picture is taken with the emulator. When the window opens, it only shows the button and clicking the button starts the phone's camera and you can take a picture saved in a file named *image.jpg*. If you take more pictures, the previous one is just overwritten. When you close the camera, the image is read from the file and displayed in an *ImageView* as shown above.

The layout of the window is simple and defines only two components embedded with a *LinearLayout*:

```
<?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.thecamera.MainActivity">
  <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_margin="10dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
      android:id="@+id/cmdCapture"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="20dp"
      android:text="Take a Picture" />
    <ImageView
      android:id="@+id/imageView"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:scaleType="centerInside" />
  </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Since the application should be able to write to external storage, you must define a permission in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  package="dk.data.torus.thecamera">
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  <application
    android:allowBackup="true"
    …
  </application>
</manifest>
```

Then there's the Java code, which is immediately harder to understand:

```
package dk.data.torus.thecamera;

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

```
import android.provider.*;
import android.app.*;
import android.graphics.*;
import android.os.*;
import android.net.*;
import android.support.v4.content.ContextCompat;
import android.support.v4.app.ActivityCompat;
import android.content.pm.*;
import java.io.*;

public class MainActivity extends AppCompatActivity
{
 private static final int REQUEST_IMAGE = 100;
 private static final int REQUEST_PERMISSION = 1;
 private Button cmdCapture;
 private ImageView imageView;
 private File file;

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

```
 StrictMode.VmPolicy.Builder builder = new StrictMode.VmPolicy.Builder();
 StrictMode.setVmPolicy(builder.build());
 cmdCapture = findViewById(R.id.cmdCapture);
 cmdCapture.setOnClickListener(new View.OnClickListener() {
  public void onClick(View view) {
   try
   {
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    intent.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(file));
    startActivityForResult(intent, REQUEST_IMAGE);
   }
   catch (Exception ex)
   {
    Toast.makeText(MainActivity.this, "The Device has no Camera",
     Toast.LENGTH_SHORT).show();
   }
  }
 });
 if (ContextCompat.checkSelfPermission(this,
  android.Manifest.permission.WRITE_EXTERNAL_STORAGE) !=
   PackageManager.PERMISSION_GRANTED)
 {
  if (ActivityCompat.shouldShowRequestPermissionRationale(this,
   android.Manifest.permission.WRITE_EXTERNAL_STORAGE)) {}
  else ActivityCompat.requestPermissions(this,
   new String[]{android.Manifest.permission.WRITE_EXTERNAL_STORAGE},
   REQUEST_PERMISSION);
 }
 imageView = findViewById(R.id.imageView);
 file = new File(Environment.getExternalStorageDirectory(), "image.jpg");
}

@Override
public void onRequestPermissionsResult(int code, String permissions[],
 int[] grant)
{
 if (code == REQUEST_PERMISSION)
 {
  if (grant.length > 0 && grant[0] == PackageManager.PERMISSION_GRANTED)
   Toast.makeText(MainActivity.this, "Permission Granted",
    Toast.LENGTH_SHORT).show();
  else Toast.makeText(MainActivity.this, "Permission Denied",
   Toast.LENGTH_SHORT).show();
 }
}
```

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
 if(requestCode == REQUEST_IMAGE && resultCode == Activity.RESULT_OK)
 {
  try
  {
   FileInputStream in = new FileInputStream(file);
   Bitmap userImage = BitmapFactory.decodeStream(in, null, null);
   imageView.setImageBitmap(userImage);
  }
  catch (Exception ex)
  {
  }
 }
}
```

and the hardest to figure out is *onCreate()*. The class first defines two constants that have no other meaning than to identify two actions (and the values are of no importance). Next, variables are defined for the two components defined in the layout, and finally a *File* variable is defined.

onCreate() starts with the two usual statements, and then there are the following statements:

```
StrictMode.VmPolicy.Builder builder = new StrictMode.VmPolicy.Builder();
StrictMode.setVmPolicy(builder.build());
```

They opens up to refer to a file outside of application's storage. In fact, it's a bit of a work around, and the Android documentation recommends doing it in a different (and slightly more complex) way. As the next step, the variable *cmdCapture* is initialized, and here is the most important of course the association of the event handler. The handler must open the device's camera (if there is a camera, why the statements are located in a try / catch block). This happens by creating an *Intent* object, as with the constant

```
MediaStore.ACTION_IMAGE_CAPTURE
```

refers to the phone's camera. To tell where the image is to be saved, the variable *file* is sent as a parameter, and finally the camera is opened with the method *startActivityForResult()*. This means that when the camera closes (the activity that displays the camera), so the method *onActivityResult()* is called and whose *requestCode* shows that the method is called because it is the camera that closes, and then the image is read from the file and displayed in the layout's *ImageView*.

After the variable *cmdCapture* is initialized in *onCreate()*, it is necessary to test whether the application has write access to external storage and if not assign this access. It happens the same way I've shown it earlier, and if it's a newer device, a dialog box opens, allowing the user to use external storage. Here is *onRequestPermissionsResult()* a method called as the result of the user's choice in the dialog, so you can see if you get the permission you want.

5 LOCATIONS

An important use of phones is to decide where you are and thus the geographical location and possibly display this location on a map. A phone is provided by the producer with programs that can do that, but it may also be of interest for the programmer, either to determine the location in their own apps and to display a map, where a given location is shown. Both parts are actually quite easy, at least if you use a special API provided by Google (*Google Play Services*). However, this API is not part of the Android SDK, so you need to download the API and refer to it in the current project. It's also simple enough, as long as you want to decide on your location, but if you also want to use Google Maps, you need a little more because you must be registered and have a key from their *Google Account*, and you can not freely quote its applications as there is license conditions for Google Maps. It should be added that other maps than Google's are available, but here are similar / other challenges. Therefore, I would just like to show you how to determine the location without using Google Play Services, and for developing your own applications it may also be what you typically need.

You can determine your location from three different location providers:

1. GPS_PROVIDER, which is the provider that delivers the most accurate result and uses the device's built-in GPS receiver.
2. NETWORK_PROVIDER, which determines the location using data collected from the phone network or WiFi access points. This provider is not as accurate as the previous one, but for most applications exactly enough.
3. PASSIVE_PROVIDER, which means that the device itself does not want to determine the location, but instead uses locations determined by other applications or services. The goal is to save power.

The application *TheLocation* uses the first two of the above three providers to determine the location of the device (see the window below). The program opens a window that determines the coordinates of the device location (longitude and latitude). The window has two buttons, both from start showing the text *Start*. Clicking on the top button changes the text to *Stop*, and the program will periodically update the values for *Longitude* and *Latitude* with the current location using the GPS provider. The bottom button works the same way, but instead, the network provider is used.

Regarding the application's user interface, there is nothing new and the layout consists of 8 widgets placed using a *RelativeLayout* and I do not want to display the code here. In order for an application to use the device's GPS, it must be allowed in the manifest:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

and actually only the first permission element is required.

Then there is the Java code, which immediately fills something more and where there is a lot of new:

```
public class MainActivity extends AppCompatActivity {
 private static final int REQUEST_PERMISSION = 1;
 private boolean permission = false;
 private LocationManager locationManager;
 private TextView long1;
 private TextView lati1;
 private TextView long2;
 private TextView lati2;

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

```
 setContentView(R.layout.activity_main);
 locationManager = (LocationManager)
  getSystemService(Context.LOCATION_SERVICE);
 long1 = findViewById(R.id.txtLong1);
 lati1 = findViewById(R.id.txtLati1);
 long2 = findViewById(R.id.txtLong2);
 lati2 = findViewById(R.id.txtLati2);
 requestPermission();
}

private void requestPermission()
{
 if (ContextCompat.checkSelfPermission(this,
  android.Manifest.permission.ACCESS_FINE_LOCATION) !=
  PackageManager.PERMISSION_GRANTED)
 {
  if (ActivityCompat.shouldShowRequestPermissionRationale(this,
   android.Manifest.permission.ACCESS_FINE_LOCATION)) {}
  else ActivityCompat.requestPermissions(this, new String[]
   {android.Manifest.permission.ACCESS_FINE_LOCATION}, REQUEST_PERMISSION);
 }
 permission = ContextCompat.checkSelfPermission(this,
  android.Manifest.permission.ACCESS_FINE_LOCATION) ==
```

```
   PackageManager.PERMISSION_GRANTED;
}

public void gpsOnOff(View view) {
 if(!permission) return;
 Button button = (Button) view;
 if(button.getText().equals("Stop")) {
  locationManager.removeUpdates(gpsListener);
  button.setText("Start");
 }
 else {
  try {
   locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
    30 * 1000, 10, gpsListener);
   button.setText("Stop");
  }
  catch (SecurityException ex)
  {
  }
 }
}

public void netOnOff(View view) {
 …
}

private final LocationListener netListener = new LocationListener() {
 …
};

private final LocationListener gpsListener = new LocationListener() {
 @Override
 public void onLocationChanged(Location location) {
  final double longitude = location.getLongitude();
  final double latitude = location.getLatitude();
  runOnUiThread(new Runnable() {
   @Override
   public void run() {
   long1.setText(String.format("%1.6f", longitude));
   lati1.setText(String.format("%1.6f", latitude));
   Toast.makeText(MainActivity.this, "GPS coordinates updated",
    Toast.LENGTH_SHORT).show();
   }
  });
 }

 @Override
 public void onStatusChanged(String s, int i, Bundle bundle) {
 }
```

```
  @Override
  public void onProviderEnabled(String s) {
  }

  @Override
  public void onProviderDisabled(String s) {
  }
 };
}
```

The class has in addition to a single constant 6 variables, and here are the last 4 references to the 4 *TextView* components in the user interface that need to be updated. The variable *permission* is used to test whether the program is entitled to determine the location, and finally there is the last one that has the type *LocationManager* and represent an object that is used to process locations, and of course it is the use of this object as is the most important thing.

In *onCreate()*, nothing happens when the *LocationManager* object is created, and the 4 variables to refer to the components in the user interface are initialized. The manifest has defined that the program may use the GPS system, but for security reasons, the user must explicitly give permission for it and it happens the same as I have shown in the previous program, where the necessary statements have been placed in a method *requestPermission()*. Viewed from this program, it is important that the variable *permission* is initialized.

The method *gpsOnOff()* is the event handler for the top button. First, the variable *permission* is tested and if it's *true*, the button's text is used to determine if you want to start reading GPS locations, or the opposite. Eventually, an event handler is removed from the *LocationManager* object, and otherwise is *requestLocationUpdates()* called to start reading locations. Here, first, it is said that a GPS provider must be used, and then the minimum time (at least 30 seconds) between each reading. The next parameter indicates the minimum distance in meters for a new reading to be done and finally the last parameter is the event handler that is to be performed for each reading. It is called *gpsListener* and is defined as an object created based on an anonymous class whose type is *LocationListener*, which is an interface that defines 4 methods. In this case, only one is called for each read. The method's parameter is a *Location* object, which indicates, among other things, longitude and latitude, but the object actually contains some other values. The method is simple, but you should note the use of *runOnUiThread()* as the method is called from another thread.

There is similar to an event handler for the other button, and an associated listener class, but I do not want to display the code here as it is in principle identical.

EXERCISE 3: GOOGLE PLAY SERVICES

In this exercise, you must write a program that, in principle, performs the same as the above, thus displaying the user's location, but this time it should be done using *Google Play Services*. The goal is to show a little about what it takes to use Google Play Services and what it is, as it may be important in practice.

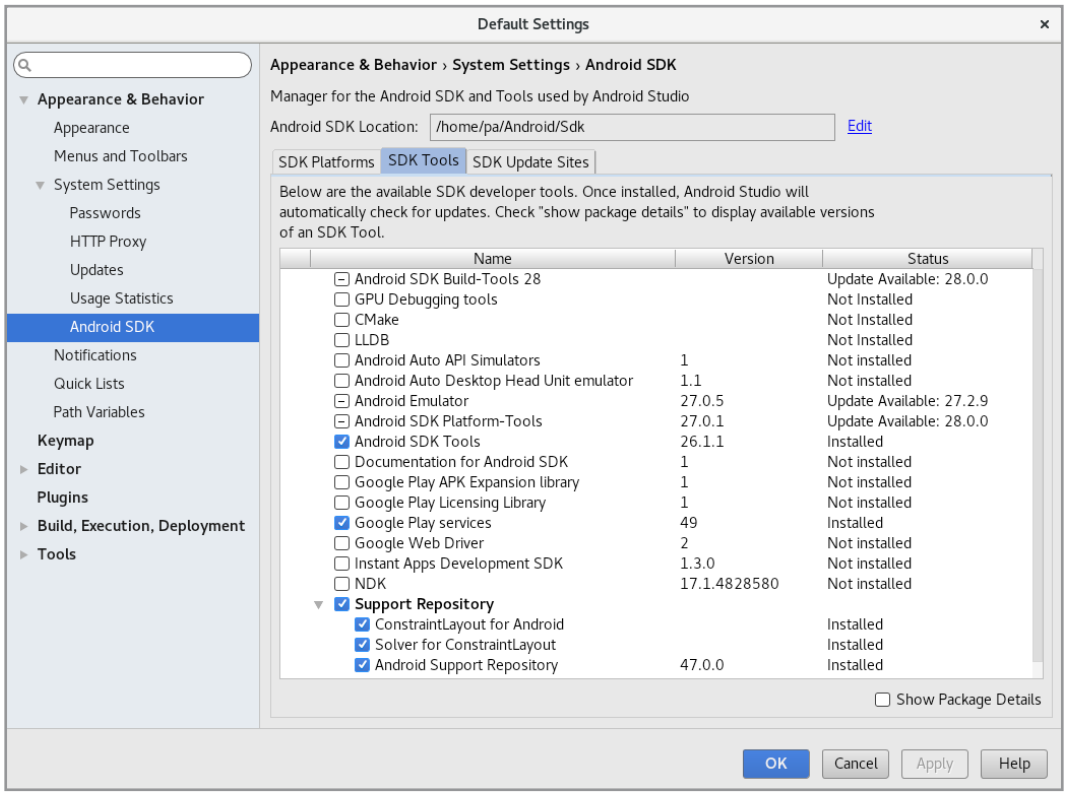
Start by creating a new project that you can call *TheLocation*. Translate the project and test in the emulator that you have a running *Hello World* project. In *activity_main.xml*, add the following ID to your *TextView* component:

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

If you have not already installed Google Play Services, do it. In Android Studio, choose

Tools | Android | SDK Manager

and click on the middle tab:



Here set a check mark next to Google Play Services and click OK, after which Google Play Services is installed on the machine.

As a next step, you must tell Android Studio that this API must be used, and you do that under *Gradle Scripts* in the file *build.gradle (Module: app)*:

```
apply plugin: 'com.android.application'
android {
  compileSdkVersion 26
  defaultConfig {
    applicationId "dk.data.torus.thelocation"
    minSdkVersion 24
    targetSdkVersion 26
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
  }
  buildTypes {
    release {
      minifyEnabled false
      proguardFiles getDefaultProguardFile('proguard-android.txt'),
       'proguard-rules.pro'
    }
  }
}
```

```
dependencies {
  implementation fileTree(dir: 'libs', include: ['*.jar'])
  implementation 'com.google.android.gms:play-services-location:15.0.1'
  implementation 'com.android.support:appcompat-v7:26.1.0'
  implementation 'com.android.support.constraint:constraint-layout:1.1.2'
  testImplementation 'junit:junit:4.12'
  androidTestImplementation 'com.android.support.test:runner:1.0.2'
  androidTestImplementation 'com.android.support.test.espresso:espresso-
   core:3.0.2'
}
```

Finally, in the same way as in the previous example, update the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.thelocation">
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
  <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">
    <meta-data android:name="com.google.android.gms.version"
     android:value="@integer/google_play_services_version" />
    <activity android:name=".MainActivity">
      <intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Then you are ready to use Google Play Services to determine your location and thus you need to update the class *MainActivity*. As a first step, *MainActivity* must implement two Google Play Services interfaces:

```
package dk.data.torus.thelocation;

import android.location.Location;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
```

```
import com.google.android.gms.common.api.GoogleApiClient;
import com.google.android.gms.location.*;

public class MainActivity extends AppCompatActivity
 implements LocationListener, GoogleApiClient.ConnectionCallbacks
{
 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
 }

 @Override
 public void onConnected(Bundle bundle) {
 }

 @Override
 public void onConnectionSuspended(int i) {
 }

 @Override
 public void onLocationChanged(Location location) {
 }
```

```
 @Override
 public void onRequestPermissionsResult(int code, String[] permissions,
  int[] grants) {
 }
}
```

Note which *import* statements it requires. Translate the program and check if it still can run. That should be possible. Next, add the following method so that the user may allow the application to get the permission *ACCESS_FINE_LOCATION* (note that it is in the same way as in the same method in the previous example):

```
private void requestPermission()
{
 if (ContextCompat.checkSelfPermission(this,
  android.Manifest.permission.ACCESS_FINE_LOCATION) !=
  PackageManager.PERMISSION_GRANTED)
 {
  if (ActivityCompat.shouldShowRequestPermissionRationale(this,
   android.Manifest.permission.ACCESS_FINE_LOCATION)) {}
  else ActivityCompat.requestPermissions(this, new String[]
   {android.Manifest.permission.ACCESS_FINE_LOCATION}, REQUEST_PERMISSION);
 }
 if (ContextCompat.checkSelfPermission(this,
  android.Manifest.permission.ACCESS_FINE_LOCATION) !=
  PackageManager.PERMISSION_GRANTED)
 {
  AlertDialog.Builder builder = new AlertDialog.Builder(this);
  builder.setTitle("Error");
  builder.setMessage("You have no permissions to show locations");
  builder.setPositiveButton("Close", new DialogInterface.OnClickListener() {
   public void onClick(DialogInterface dialog, int which) {
    finish();
   }
  });
  builder.show();
 }
}
```

The method requires adding multiple *import* statements and a single static *int* constant *REQUEST_PERMISSION*. Add another *import* statement

```
import com.google.android.gms.common.*;
```

and add another method:

```
private void showError(int code) {
 GoogleApiAvailability.getInstance().showErrorDialogFragment(this, code, 10,
  new DialogInterface.OnCancelListener() {

  @Override
  public void onCancel(DialogInterface dialogInterface) {
   finish();
  }
 });
}
```

This method is used if connection to Google Play Services can not be established. As a next step, add 4 variables to the class and expand the method *onCreate()* as shown below:

```
public class MainActivity extends AppCompatActivity
 implements LocationListener, GoogleApiClient.ConnectionCallbacks
{
 private static final int REQUEST_PERMISSION = 101;
 private GoogleApiClient client;
 private LocationRequest request;
 private Location location;
 private TextView txtLocation;

 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  txtLocation = findViewById(R.id.txtLocation);
  requestPermission();
  int code =
   GoogleApiAvailability.getInstance().isGooglePlayServicesAvailable(this);
  if (code != ConnectionResult.SUCCESS) showError(code);
  client = new GoogleApiClient.Builder(this).addApi(LocationServices.API).
   addConnectionCallbacks(this).build();
  request = LocationRequest.create().setPriority(
   LocationRequest.PRIORITY_HIGH_ACCURACY).setInterval(15000).
    setFastestInterval(2000);
 }
```

Translate the program and check if it still can run. *MainActivity* also needs to implement *onResume()* and additionally you must add a method *display()* that can update your *TextView* component in the layout:

```
@Override
public void onResume() {
 super.onResume();
 client.connect();
}

private void display() {
 if (location == null) txtLocation.setText("");
 else txtLocation.setText(String.format("Latitute: %.4f\n\nLongitude: %.4f",
  location.getLatitude(), location.getLongitude()));
}
```

Then you must implement the four methods defined by the two interfaces:

```
@Override
public void onConnected(Bundle bundle)
{
 fetchLocation();
}
```

```
@SuppressWarnings("MissingPermission")
private void fetchLocation() {
 location = LocationServices.FusedLocationApi.getLastLocation(client);
 LocationServices.FusedLocationApi.
  requestLocationUpdates(client, request, this);
}

@Override
public void onConnectionSuspended(int i) {
}

@Override
public void onLocationChanged(Location location) {
 this.location = location;
 display();
}

@Override
public void onRequestPermissionsResult(int code,
String[] permissions, int[] grants) {
 super.onRequestPermissionsResult(code, permissions, grants);
 if (code == REQUEST_PERMISSION) {
  for (int i = 0; i < grants.length; ++i) {
   int result = grants[i];
   String permission = permissions[i];
   if (result != PackageManager.PERMISSION_GRANTED) {
    finish();
    return;
   }
  }
 }
}
```

6 PREFERENCES

A phone (and any other Android device) has an application called Settings, which is used for the phone's settings. The settings you choose are still valid after the device is restarted and must therefore be stored somewhere. This could be done in a database or a file, but it is small amounts of data, and Android saves these kind of information as key / value pairs using a special system called *preferences*. In the same way as for the device itself, you often need to be able to save settings for your own applications, and instead of using files or databases, you can use the preference system.

Preferences can in principle be used for other than settings, as it is just a matter of saving key / value pairs, and the only limitation is that the value must be a string or a value of a simple data type. Preferences, however, are more than just a way to store data, but it also includes components that are used to maintain this data, as illustrated in the following example.

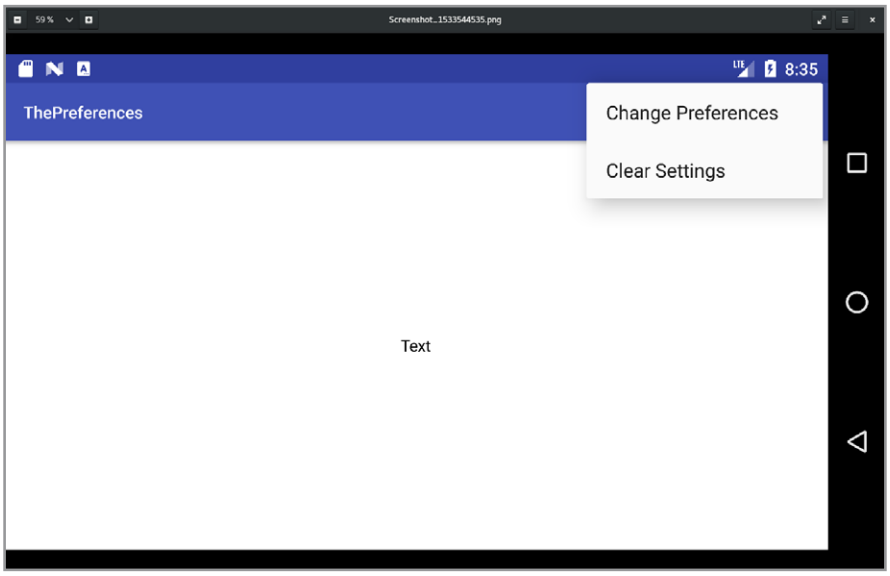
Preferences can be assigned to an *Activity*, and an *Activity* object has a method *getPreferences()*, which returns a *SharedPreference* object. Preferences can also be associated with the application, and with the method *getSharedPreferences()*, you can get a *SharedPreferences* object identified by a name, so there may be several preferences attached to the same application. Finally, there are preferences pertaining to the entire device and maintained using a *PreferenceManager* object, where you can get a *SharedPreferences* object using a method *getDefaultSharedPreferences()*. The method has as a parameter a *Context* object and returns preferences for this object. You can use which of the three ways you find the most appropriate, but in most cases you apply the last way.

A *SharedPreferences* object has *get* methods that can be used to determine a value, that is, methods like

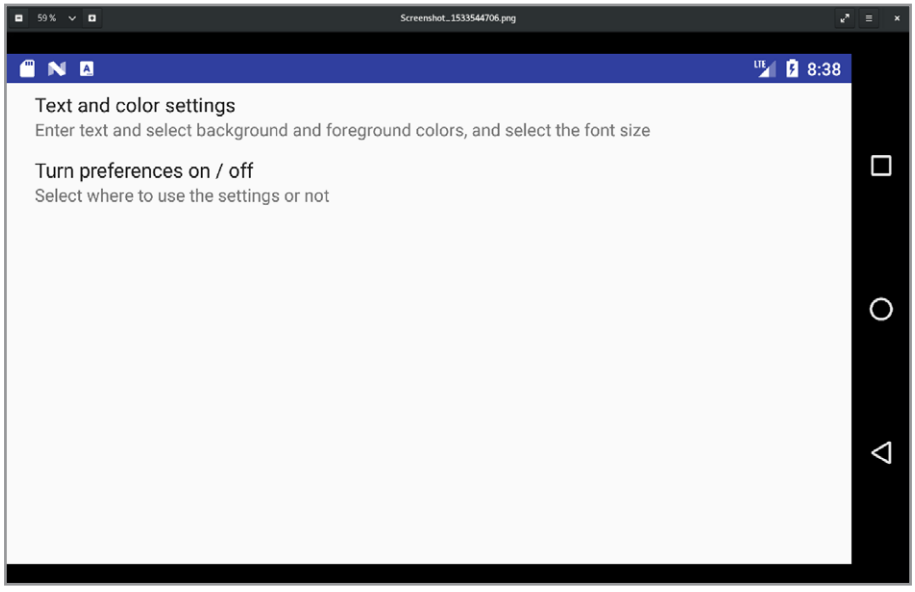
- *getString()*
- *getInt()*

and others, and the methods have two parameters, where the first is the key, while the other is a default value. You can (naturally) also create and modify values for a *SharedPreference* object, which occurs with similarly *set* methods, where the parameters are the key and the value. However, it takes a bit different from normal, since a *SharedPreference* object has a method *edit()* that returns an *Editor* object, and the methods are assigned to this object. For the changes to have effect, finish the changes by the *Editor* object and execute the method *commit()* (or the method *apply()*). The *Editor* object also has a method *remove()* that can be used to remove a single preference, as well as a method *clear()* that removes all preferences.

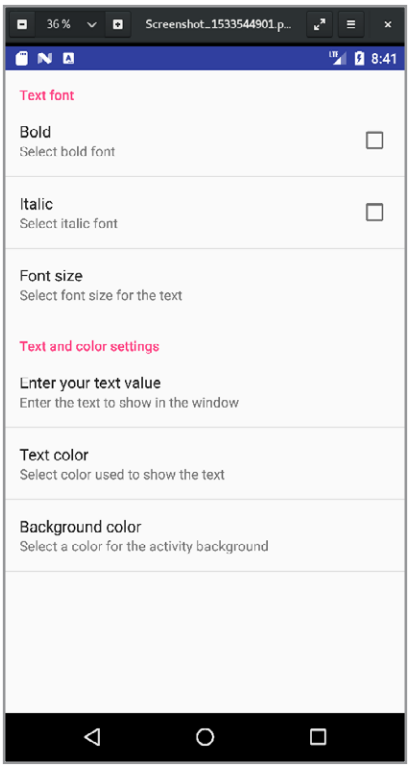
If an application needs to maintain settings, you can of course define your own activities for the purpose, which can be comprehensive. For that reason, Android offers a special activity *PreferenceActivity* and a fragment *PreferenceFragment* that can be used to implement settings for an application, and in addition to making it easier, the most important is, that settings in this way work the same to every application. In the following, I will with an example *ThePreferences* show you how to do. If you open the application, you get the following window:



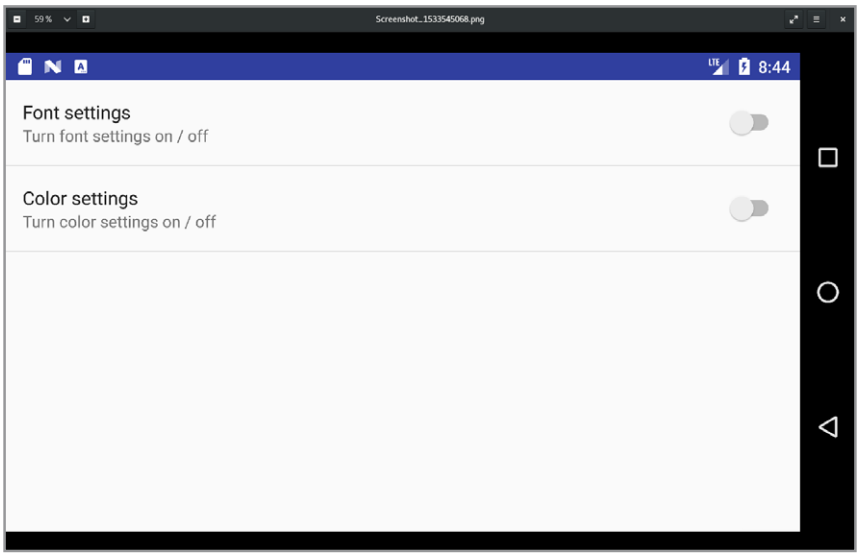
which is nothing but a simple Hello World activity, but with a menu with two menu items. If you select the top one, you will receive a window that shows the headings for the program's settings:



There are two things to set, where the first setting concerns colors and how the text should appear (see the window below), while the other option is to turn the settings on and off. In the first case, you can choose whether that the text should be bold or italic (or both), and you can choose the font size. If you select the font size, you get a dialog box with radio buttons, and you can click the button for the desired font size. You can also choose to enter the text, and the result is a dialog box for text entering. Finally, you can choose text color and background color, and in both cases you get a dialog box with radio buttons.



In the last setting window, two switch components enable you to set the two settings on and off:



It does not make much sense in this case beyond as an example of typical settings. Below is a window after both switch components are on and after selecting values in the first setting window:

In particular, note that if you close the program and reopen it, then the settings will still apply, leaving them saved somewhere – in the device's repository for preferences.

The starting point is a usual project called *ThePreferences*. There is added more resources, and here is *string.xml* updated with two strings:

```
<resources>
  <string name="app_name">ThePreferences</string>
  <string name="action_settings">Clear Settings</string>
  <string name="change_preferences">Change Preferences</string>
</resources>
```

which will be used for texts in the menu. To the directory *res*, a folder has been added with the name *menu* and here again a XML document called *main.xml*, which defines a menu with two menu items:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <item
    android:id="@+id/change_preferences"
    android:title="@string/change_preferences" />
  <item
    android:id="@+id/clear_settings"
    android:title="@string/action_settings" />
</menu>
```

To the directory *values*, a file *arrays.xml* has been added:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="sizes">
    <item>6</item>
    <item>8</item>
    <item>10</item>
    <item>12</item>
    <item>14</item>
    <item>18</item>
    <item>24</item>
    <item>36</item>
    <item>48</item>
    <item>72</item>
    <item>144</item>
  </string-array>
  <string-array name="colors">
    <item>White</item>
    <item>Red</item>
    <item>Orange</item>
```

```
    <item>Yellow</item>
    <item>Green</item>
    <item>Blue</item>
    <item>Indigo</item>
    <item>Violet</item>
    <item>Black</item>
  </string-array>
  <string-array name="color_codes">
    <item>#FFFFFF</item>
    <item>#FF0000</item>
    <item>#FFA500</item>
    <item>#FFFF00</item>
    <item>#00FF00</item>
    <item>#0000FF</item>
    <item>#4B0082</item>
    <item>#EE82EE</item>
    <item>#000000</item>
  </string-array>
</resources>
```

It defines three arrays as resources. The first array defines values to be used as the size of a font. The next two defines colors as the color name and color code, respectively, and they are used to select text color and background color.

The layout *activity_main.xml* consists solely of a *TextView* component and is not changed from what Android Studio has created except that I have changed the name of the component to *txtLine* and then the layout element has a name so I can refer it from Java code. The Java parts fill a part and are as follows:

```
public class MainActivity extends AppCompatActivity {
 private TextView txtLine;

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

 @Override
 public void onResume() {
  super.onResume();
  refreshActivity();
 }
```

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
 getMenuInflater().inflate(R.menu.main, menu);
 return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
 int id = item.getItemId();
 if (id == R.id.change_preferences) {
  startActivity(new Intent(this, ChangePreferences.class));
  return true;
 }
 if (id == R.id.clear_settings) {
  setDefault();
  return true;
 }
 return super.onOptionsItemSelected(item);
}
```

```
 private void setDefault()
 {
  SharedPreferences prefs = PreferenceManager.
   getDefaultSharedPreferences(this);
  SharedPreferences.Editor edit = prefs.edit();
  edit.putBoolean("colorswitch", false);
  edit.putBoolean("fontswitch", false);
  edit.putString("bgcolor", "#FFFFFF");
  edit.putString("fgcolor", "#000000");
  edit.putString("text", "Text");
  edit.putBoolean("italic", false);
  edit.putBoolean("bold", false);
  edit.putString("size", "6");
  edit.commit();
  refreshActivity();
 }

 private void refreshActivity()
 {
  SharedPreferences prefs = PreferenceManager.
   getDefaultSharedPreferences(this);
  boolean colorOn = prefs.getBoolean("colorswitch", false);
  boolean fontOn = prefs.getBoolean("fontswitch", false);
  String bg = colorOn ? prefs.getString("bgcolor", "#FFFFFF") : "#FFFFFF";
  String fg = colorOn ? prefs.getString("fgcolor", "#000000") : "#000000";
  ConstraintLayout layout = (ConstraintLayout)
   this.findViewById(R.id.background);
  layout.setBackgroundColor(Color.parseColor(bg));
  txtLine.setTextColor(Color.parseColor(fg));
  txtLine.setText(prefs.getString("text", "Text"));
  boolean italic = fontOn ? prefs.getBoolean("italic", false) : false;
  boolean bold = fontOn ? prefs.getBoolean("bold", false) : false;
  int style = Typeface.NORMAL;
  if (bold && italic) style = Typeface.BOLD_ITALIC;
  else if (bold) style = Typeface.BOLD;
  else if (italic) style = Typeface.ITALIC;
  txtLine.setTypeface(Typeface.create(txtLine.getTypeface(), style));
  txtLine.setTextSize(TypedValue.COMPLEX_UNIT_PT, fontOn ?
   Integer.parseInt(prefs.getString("size", "6")) : 6);
 }
}
```

The class has only one variable that refers to the component *txtLine*, which is initially initialized in *onCreate()*. You should note that the class also overrides *onResume()*, which calls a method *refreshActivity()*. As a result, this method is performed every time the window must be initialized, for example, if the phone is rotated. Otherwise, there are methods that create the menu (based on *main.xml*) and process events from the menu.

If you select the lower menu *Clear Preferences*, the method *setDefault()* is performed. It starts by determining a *SharedPreference* object for this activity, and for this object, a reference is made to the object's *Editor*, which is subsequently used to initialize 8 preferences. This means that if that preference does not exist, it will be created and otherwise overwritten. Before the method is terminated, a *commit()* is executed, so all preferences are saved and finally the method *refreshActivity()* is called. This method also determines a *SharedPreferences* object with preferences for this activity, and it uses these preferences to update the user interface with respect to the background color, which text to display, and in that context the font, size, and color of the text. Here you should especially note how the two *Switch* settings are used to test whether the other settings are to be used or if default values are to be used instead. The method *refreshActivity()* fills a part, but apart from reading preferences, there is not much new, and the bottom line is that the lower menu item *Clear Preferences* resets all settings to default.

Then there is the first menu item, and it opens a new activity defined by the class *ChangePreferences*. This activity is derived from the class *PreferencesActivity*:

```
public class ChangePreferences extends PreferenceActivity {
 @Override
 public void onBuildHeaders(List<Header> headers) {
  loadHeadersFromResource(R.xml.pref_headers, headers);
 }

 @Override
 protected boolean isValidFragment(String fragmentName) {
  return DefPrefFragment.class.getName().equals(fragmentName);
 }
}
```

The class does not fill much, as it is all stored in the base class. When the activity is opened, the method *onBuildHeaders()* is performed, which initializes a list of *Header* objects by loading a resource called *pref_headers.xml* that is a XML document created in a directory *xml*:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
  <header
    android:fragment="dk.data.torus.thepreferences.DefPrefFragment"
    android:title="Text and color settings"
    android:summary="Enter text and select background and foreground colors,
     and select the font size">
    <extra android:name="fragresource" android:value="preference_1" />
  </header>
  <header
    android:fragment="dk.data.torus.thepreferences.DefPrefFragment"
    android:title="Turn preferences on / off"
```

```
    android:summary="Select where to use the settings or not">
    <extra android:name="fragresource" android:value="preference_2" />
  </header>
</preference-headers>
```

In this case, two *Header* objects (headings) are defined and the result is the window of the settings list. Clicking on one of these headlines performs the method *isValidFragment()*, which validates if the name is a legitimate fragment, here to be *DefPrefFragment*:

```
public class DefPrefFragment extends PreferenceFragment {
 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  int res = getActivity().getResources().getIdentifier(
   getArguments().getString("fragresource"), "xml",
   getActivity().getPackageName());
  addPreferencesFromResource(res);
 }
}
```

If this is the case, this fragment is inserted. Note that the class inherits *PreferenceFragment*, and corresponding to which heading is clicked, one of two layouts is used, either *xml / preference_1.xml* or *xml / preference_2.xml*. The first is the following:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="Text font">
    <CheckBoxPreference
      android:key="bold"
      android:title="Bold"
      android:summary="Select bold font" />
    <CheckBoxPreference
      android:key="italic"
      android:title="Italic"
      android:summary="Select italic font" />
    <ListPreference
      android:key="size"
      android:title="Font size"
      android:summary="Select font size for the text"
      android:entries="@array/sizes"
      android:entryValues="@array/sizes"
      android:dialogTitle="Select font size" />
  </PreferenceCategory>
  <PreferenceCategory android:title="Text and color settings">
    <EditTextPreference
      android:key="text"
      android:title="Enter your text value"
      android:summary="Enter the text to show in the window"
      android:dialogTitle="Enter text" />
    <ListPreference
      android:key="fgcolor"
      android:title="Text color"
      android:summary="Select color used to show the text"
      android:entries="@array/colors"
      android:entryValues="@array/color_codes"
      android:dialogTitle="Select text color" />
    <ListPreference
      android:key="bgcolor"
      android:title="Background color"
      android:summary="Select a color for the activity background"
      android:entries="@array/colors"
      android:entryValues="@array/color_codes"
      android:dialogTitle="Select background color" />
  </PreferenceCategory>
</PreferenceScreen>
```

Here, two *PreferenceCategory* elements are defined that have no purpose other than grouping preferences under a heading. Otherwise, note the items that are used. A *CheckBoxPreference* appears as a *CheckBox*, but it updates the reference to which it is associated, such as the preference with the key *bold*. A *ListPreference* opens a dialog box with a list of radio buttons, so you can choose a value, and the same applies if you choose OK to update the preference. Note how to *listPreference* with a resource from *arrays* defines the text that the radio buttons should display and the value they should have. Finally, there is a *EditTextPreference*, which opens a dialog box where you can enter a text.

xml / preference_2-xml is as follows:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <SwitchPreference
    android:key="fontswitch"
    android:title="Font settings"
    android:summary="Turn font settings on / off" />
  <SwitchPreference
    android:key="colorswitch"
    android:title="Color settings"
    android:summary="Turn color settings on / off" />
</PreferenceScreen>
```

where only a *SwitchPreference* is used, and in principle it works as the other preference elements. It should be mentioned that there are other preference elements than those used in this example.

Then the example is complete and there is nothing more than noting the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.thepreferences">
  <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>
```

```
    <activity
      android:name=".ChangePreferences">
    </activity>
  </application>
</manifest>
```

PROBLEM 1: PUZZLEAPP

In the previous book problem 2, you wrote a program called *PuzzleApp*. The program is a simple game where you need to solve a puzzle with 9, 25, 49 or 81 pieces. Create a copy of the *PuzzleApp* project.

In this task you must implement a single change in the program. The program maintains a high score list – in fact 4 high score lists, as there is a high score list for each of the 4 difficulty levels. In the solution in the book Java 16, the high score lists are stored in a file. You must now change the solution so that the lists are instead saved as preferences. Note that it has nothing to do with settings, so you should not use either *PreferenceActivity* or *PreferenceFragment*. In fact, it is only necessary to change the class *HighscoreList*.

7 EXTERNAL STORAGE

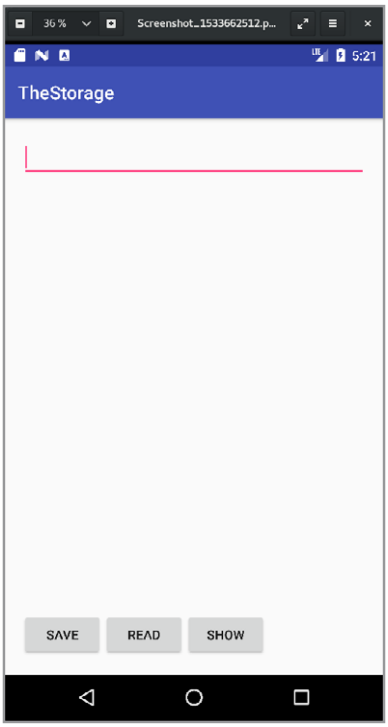
In the previous book on files, I only showed examples of using internal storage, which is storage associated with the single application. In this chapter, I will show an example that shows how to use external storage, which is storage that may contain data that can be used by multiple applications.

There are actually two types of external storage

1. *Primary External Storage*, which physically is part of the phone and whose size is determined by the phone's producer.
2. *Secondary External Storage*, which is typically a SD card, but may also be a device connected to the phone with a cable.

The following example relates only to the first type of external storage, as it is the most effective form of external storage, and since the device does not necessarily have a SD card. From Android version 6, it is quite simple to use external storage on a SD card, but the supplier may not have opened up the possibility. The reason is lacking security, and Android actually recommends not using a SD card as external storage.

When I in the following refer to external storage, it means primary external storage, but otherwise, the use of external storage is in principle similar to that shown for internal storage in the previous book. The project *TheStorage* opens a window



where at the top there is a *EditText* component for entering a text. Below is a *TextView* component to display a text and finally there are three buttons in which the first button is used to save the input text into a file in external storage, while the next is used to load the content of the file and show it in the *TextView* component. Finally, the last button is used to display an information about the file system.

First step is the manifest, where you has to give the right to use external storage:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.thestorage">
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  <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>
  </application>
</manifest>
```

There are two *uses-permission* elements listed. Here, *WRITE_EXTERNAL_STORAGE* permission automatically provides *READ_EXTERNAL_STORAGE* permission, so the last one is only necessary if you just want to read external storage.

Regarding the layout, I will not display the XML code here as only 5 components are placed using a *RelativeLayout,* but in return, the Java code fills a lot:

```
public class MainActivity extends AppCompatActivity {
 private static final int REQUEST_READ_PERMISSION = 100;
 private static final int REQUEST_WRITE_PERMISSION = 101;
 private EditText txtEdit;
 private TextView txtView;
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 txtEdit = findViewById(R.id.txtEdit);
 txtView = findViewById(R.id.txtView);
 findViewById(R.id.cmdSave).setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
   askWritePermission();
  }
 });
 findViewById(R.id.cmdRead).setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
   askReadPermission();
  }
 });
```

```
 findViewById(R.id.cmdShow).setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
   showStorage();
  }
 });
}

private void askWritePermission() {
 if (askPermission(REQUEST_WRITE_PERMISSION,
  Manifest.permission.WRITE_EXTERNAL_STORAGE)) writeFile();
}

private void askReadPermission() {
 if (askPermission(REQUEST_READ_PERMISSION,
  Manifest.permission.READ_EXTERNAL_STORAGE)) readFile();
}

private boolean askPermission(int requestId, String permission) {
 if (android.os.Build.VERSION.SDK_INT >= 23) {
  if (ActivityCompat.checkSelfPermission(this, permission) !=
   PackageManager.PERMISSION_GRANTED) {
   requestPermissions(new String[] { permission }, requestId);
   return false;
  }
 }
 return true;
}

@Override
public void onRequestPermissionsResult(int requestCode, String permissions[],
 int[] grantResults) {
 super.onRequestPermissionsResult(requestCode, permissions, grantResults);
 if (grantResults.length > 0) {
  switch (requestCode) {
   case REQUEST_READ_PERMISSION:
    if (grantResults[0] == PackageManager.PERMISSION_GRANTED) readFile();
    return;
   case REQUEST_WRITE_PERMISSION: {
    if (grantResults[0] == PackageManager.PERMISSION_GRANTED) writeFile();
    return;
   }
  }
 } else Toast.makeText(getApplicationContext(), "Permission denied",
   Toast.LENGTH_SHORT).show();
}
```

```
 private void writeFile() {
  String path =
   Environment.getExternalStorageDirectory().getAbsolutePath() + "/note.txt";
  String data = txtEdit.getText().toString();
  try {
   File file = new File(path);
   file.createNewFile();
   OutputStreamWriter writer =
    new OutputStreamWriter(new FileOutputStream(file));
   writer.append(data);
   writer.close();
   Toast.makeText(getApplicationContext(), "Data saved",
    Toast.LENGTH_LONG).show();
  } catch (Exception ex) {
   Toast.makeText(getApplicationContext(), ex.getMessage(),
    Toast.LENGTH_LONG).show();
  }
 }

 private void readFile() {
  String path =
   Environment.getExternalStorageDirectory().getAbsolutePath() + "/note.txt";
  StringBuilder builder = new StringBuilder();
  try {
   File file = new File(path);
   BufferedReader reader =
    new BufferedReader(new InputStreamReader(new FileInputStream(file)));
   for (String str = reader.readLine(); str !=
    null; str = reader.readLine()) {
    builder.append(str);
    builder.append("\n");
   }
   reader.close();
   Toast.makeText(getApplicationContext(), "Data loaded",
    Toast.LENGTH_LONG).show();
   txtView.setText(builder.toString());
  } catch (IOException ex) {
   Toast.makeText(getApplicationContext(), ex.getMessage(),
    Toast.LENGTH_LONG).show();
  }
 }

 private void showStorage() {
  StringBuilder builder = new StringBuilder();
  builder.append(ExternalTools.getPublicBaseDir());
  builder.append("\n");
  builder.append(ExternalTools.getStorageSize());
  builder.append("\n");
  txtView.setText(builder.toString());
 }
}
```

Initially, two constants are defined which are used in connection with a callback method. Next, there are two variables for the two text components in the user interface. These variables are initialized in *onCreate()*, but else *onCreate()* only consists of associating event handlers with the three buttons where each handler calls a method.

The event handler for the *SAVE* button calls the method *askWritePermission()*. From before Android version 23, an application had write permission for external storage, as long as it was stated in the manifest, but from version 23 and later, the requirements have been sharpened so that the user must now actively grant permission in a simple dialog box (this is only the first time, where the user performes the program and see also chapter 11). The method *askWritePermission()* calls the method *askPermission()* with two parameters, the first being one of the two constants defined at the start of the program while the other is the permission that the user may allow.

In *askPermission()*, first the Android version is tested, and if it is less than 23, nothing happens and the method returns *true*. Otherwise, the following method is used

```
ActivityCompat.checkSelfPermission(this, permission)
```

to test if you have the required permission and is it not the case, then the method

```
requestPermissions(new String[] { permission }, requestId)
```

is called and *askPermission()* returns *false*. It is the method *requestPermissions()* that opens the dialog where the user must actively assign the desired permission. The method *requestPermissions()* calls the callback method

```
onRequestPermissionsResult(…)
```

which returns the result of the user's choice. In this case, if the user has granted permission, so the method *writeFile()* is called, which writes to the file, and otherwise a simple *Toast* appears indicating that no permission was granted.

If the method *askPermission()* method returns *true*, the method *askWritePermission()* will call *writeFil()*.

Then there's the method *writeFile()* that does not really add so much new to what I've previously treated with internal storage, and the most important thing is the call:

```
Environment.getExternalStorageDirectory().getAbsolutePath() + "/note.txt"
```

which determines the name of a file in external storage. You should note that the file is stored in the root of external storage, which obviously does not have to be the case. The file name is hard-coded to make it simple, and in practice it is of course not very appropriate.

The method *askReadPermission()* is called from the event handler for the middle button, and it works in principle in exactly the same way as *askWritePermission()*, but with the difference that it instead calls the method *readFile()*. In this method, there is also not much to explain, and in particular, note that the name of the file is determined in the same way as above.

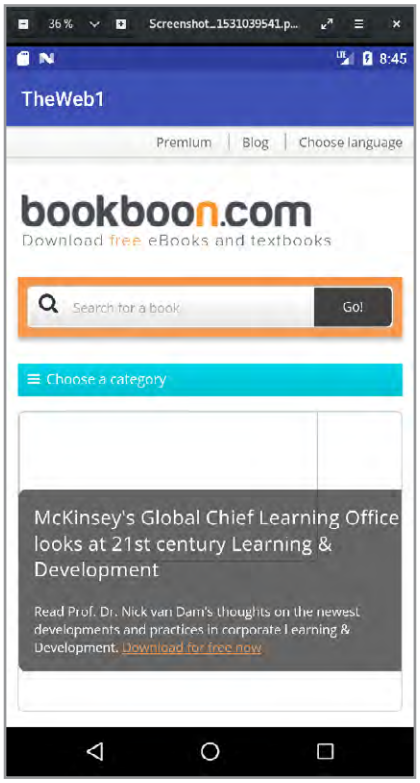
The last event handler calls the method *showStorage()*. The method shown some information about the file system, and android has several methods for that purpose. I have encapsulated some of these methods as static methods in a helper class *ExternalTools*, and the goal is that the class can be expanded continuously as needed. You are encouraged to study the class, which is quite simple.

8 THE WEB

A mobile phone is not much more than a small computer, but with special features like phone calling, sending SMS, built-in GPS, camera, and more specifically, you can also go to the Internet. Therefore, Android also has special components for that purpose, and the following is an introduction. This is primarily about

1. *WebView*, which is a component that can display HTML
2. *HttpURLConnection*, which is a class to manipulate web services
3. *DownloadManager*, which is a class to download data

As a simple start, I will show a program that uses the component *WebView* to display a web page. The project is called *TheWeb1*, and if you executes the program, it opens the following window:



The layout is as follows:

```
<?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.theweb1.MainActivity">

  <WebView
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:id="@+id/webView" />
</android.support.constraint.ConstraintLayout>
```

and as you can see, it's only a matter of having a *WebView* component. Then there is *MainActivity*:

```
package dk.data.torus.theweb1;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.webkit.*;

public class MainActivity extends AppCompatActivity {
 private WebView webView;
```

```
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  webView = (WebView)findViewById(R.id.webView);
  webView.loadUrl("https://bookboon.com/");
 }
}
```

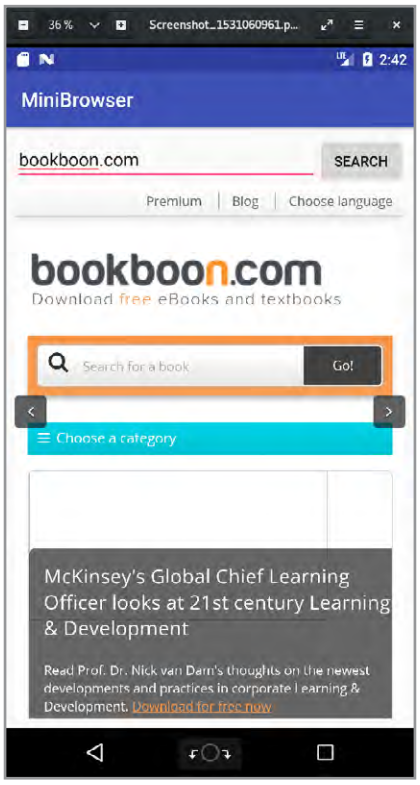
where there is nothing to explain, it's almost impossible to write it easier. There is, however, one more detail as a permission must be inserted in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.theweb1">
  <uses-permission android:name="android.permission.INTERNET" />
  <application
    …
  </application>
</manifest>
```

EXERCISE 4: MINIBROWSER

In the example above, the web address is hard-coded, and it is of course not so often that it is needed. You must therefore write a simple app, which you can call *MiniBrowser*, where you can enter the address instead, and the program should subsequently display that web page with a *WebView* (see the window below). There may be a problem where as the phone for a newer Android will automatically redirect to your phone's browser. If your *WebView* is called *webView*, you can solve the problem by adding the following code (in *onCreate()*):

```
webView.setWebViewClient(new WebViewClient() {
 @Override
 public boolean shouldOverrideUrlLoading(WebVie
  w view, WebResourceRequest request)
 {
  view.loadUrl(request.getUrl().toString());
  return true;
 }
});
```



8.1 LOADDATA()

The component *WebView* can generally be used to display HTML, and it does not need to be the result of a web page from the Internet as above. The applications are more and, for example, you can display the content of a file containing HTML, and in principle you could define the entire user interface as HTML and display it with a *WebView* component. As an example, the project *TheWeb2* shows how. The layout is the same as *TheWeb1*. In addition, I have added the two classes *King* and *Kings* from the project *TheKings* in the book Java 16, and I have modified the method *toString()* in the class *King*. With these two classes available, I've changed *MainActivity* to:

```
public class MainActivity extends AppCompatActivity {
 private WebView webView;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  webView = (WebView)findViewById(R.id.webView);
  webView.loadData(buildHtml(), "text/html", "UTF_8");
 }
```

```
 private String buildHtml()
 {
  List<King> kings = (new Kings()).getKings();
  StringBuilder builder = new StringBuilder();
  builder.append("<html><p><ol>");
  for (King king : kings)
  {
   builder.append("<li>");
   builder.append(king.toString());
   builder.append("</li>");
  }
  builder.append("</ol></p></html>");
  return builder.toString();
 }
}
```

The only thing to note is the method *buildHtml()*, which dynamically builds a string with HTML, and that the *WebView* component displays this text using the method *loadData()*. If you performs application the result is:

8.2 CONNECT TO AN URL

A *WebView* component can be used to display a website, but seen in relation to apps on a phone it is far from the typical use of the Internet. Far more often, you are interested in retrieving data to your phone by connecting to an URL. It could be a web service that returns data in some format, and there are actually many of those kinds of services that are public and directly available to anyone who wants to apply them. Many of these services return JSON data, which does not matter a lot, and partly there are standard parsers, so they are easy to use. In this section I will show an application that connects to such a service and retrieves the service's data. The application does not use the data in question (does not interpret data), but after the data is retrieved, a *Toast* tells how many characters are retrieved.

The service in question has the address:

```
http://portal.opendata.dk/api/3/action/datastore_
search?resource_id=c3097987-c394-4092-ad1d-ad86a81dbf37
```

which refers to data from an organization that provides Danish data for free. The project is called *TheWeb3*, and the layout is trivial with a single button. Clicking the button creates a connection to the above address, and the application reads the relevant JSON data, but without interpreting them. The code for *MainActivity* is:

```
package dk.data.torus.theweb3;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.AsyncTask;
import android.widget.*;
import android.view.*;
import java.io.*;
import java.net.*;

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

 public void onClick(View v) {
  new InternetTask().execute("");
 }
 private String fetchUrl(String addr) {
  StringBuilder builder = new StringBuilder();
  try {
   URL url = new URL(addr);
   HttpURLConnection conn = (HttpURLConnection)url.openConnection();
   conn.setRequestProperty("User-Agent", "");
   conn.setRequestMethod("GET");
   conn.setDoInput(true);
   conn.connect();
   BufferedReader reader =
    new BufferedReader(new InputStreamReader(conn.getInputStream()));
   for (String content = reader.readLine(); content != null;
    content = reader.readLine()) builder.append(content);
  } catch (IOException e) {
  }
  return builder.toString();
 }

 private class InternetTask extends AsyncTask<String, Void, String> {
  @Override
  protected void onPreExecute() {}
  @Override
  protected String doInBackground(String… params) {
   return fetchUrl("http://portal.opendata.dk/api/3/action/datastore_search?
    resource_id=c3097987-c394-4092-ad1d-ad86a81dbf37");
  }
```

```
  @Override
  protected void onPostExecute(String result) {
   Toast.makeText(getApplicationContext(), result.length() +
    " characters transfered", Toast.LENGTH_LONG).show();
  }

  @Override
  protected void onProgressUpdate(Void… values) {}
 }
}
```

The Event handler for the button creates an *InternetTask* and executes the method *execute()*. The class *InternetTask* is an inner class that inherits the Android class *AsyncTask*, which is a generic class that is parameterized with three parameters. The goal of the class is to start an operation that is performed in a background thread, and the class will automatic update the UI thread. You must override 4 methods, where the first *onPreExecute()* is performed by the UI thread before the operation itself is performed, which occurs in the method *doInBackground()*. The first parameter type for the class indicates the type of parameters for *doInBackground()*. The third method to override is *onProgressUpdate()*, which also performs on the UI thread and can be indirectly called from *doInBackground()* to illustrate the progress of the operation. The other parameter type for the class indicates the type of the

parameters for this method. The last method to override is called *onPostExecute()*, and it is also executed by the UI thread, but after the method *doInBackground()* is performed. The third parameter type indicates the type of the parameter for this method, which is often the return value from *doInBackground()*. The operation in question starts with the method *execute()* which executes *onPreExecute()*, creates a thread that performs *doInBackground()* and after the thread terminates *executes()* performs the method *onPostExcecute()*.

In this case, there are only two of the methods that are not trivial. The class *MainActivity* has a method called *fetchUrl()*, and *doInBackground()* calls this method and returns the result. That is, that *fetchUrl()* is executed in a background thread, and after the method is completed, a *Toast* (the method *onPostExecute()*) appears, showing how many characters *doInBackground()* has returned.

Then there is *fetchUrl()* which is where everything happens. It has a parameter, which is a web address transferred from *doInBackground()* – here hard-coded, and in practice it would probably happen in a different way, and at least the address could be moved to a resource. The method creates an URL for that address and uses this URL to create a *HttpURLConnection* object, which is an object representing a connection for the current URL. For this connection, it is defined to be used as an HTTP request, after which the connection is created:

```
conn.connect();
```

Next, a *BufferedReader* is opened to the connection's input stream, and the result is read in principle in the same way as another stream and is filled into a *StringBuilder*. Finally, the method returns the content of the builder.

The important thing in this example is the use of the class *HttpURLConnection*, which represents a connection to an URL.

8.3 DOWNLOAD A FILE

As the last example of programs for the web, I will show a program that downloads a file and saves it in external storage. The program is called *TheWeb4*. The program's manifest has two permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

The application's user interface consists only of a button that can start downloading a file. Downloading a file is done using a *DownloadManager* class that offers what is needed. I do not want to display the code for *MainActivity* here, but you are encouraged to check the code. In this case, the file to be downloaded is hard-coded, but of course it does not have to be the case.

9 MORE ON SERVICES

In the previous book I have treated services, but there is more to add, and especially how an activity can communicate with a service. This is the subject of this chapter.

If a component starts a service by calling *startService()*, the service will run until it stops itself with *stopSelf()* or another component stops it by calling *stopService()*. If a component instead starts a service by calling *bindService()*, the service will run as long as there are components that are bound to it and only when the service is no longer bound to clients, Android will shut it down. However, a service can also be stopped if the system is missing memory, and it is necessary to obtain resources for the activity that the user is working with.

If a service is started with *startService()*, there is no direct communication between the starting activity and the service. The service carries out its work, and it is not possible to send a message to it (in addition to stopping it) and the service can not call a method on the starting activity, but you can only get to know about it in the form of a toast or a notification. Should there be a direct communication, it is necessary to deposit an intermediary called a receiver. If the service has to be able to receive messages from the activity, the service must instantiate a receiver object, which is a *BroadcastReceiver* object and register this object as a receiver who wishes to receive messages with a particular ID. The same applies to the activity that wishing to receive messages from the service, it must also create and register a receiver object.

The application *TheServices1* opens the following window:

If you click the START button, a service is started as every second sends a message (a counter is counted up with 1), and that means that the activity every second receives a message from the service. Clicking the STOP button stops the service. Finally, the last button is used to send a message to the service, which is the text entered in the input field, after which the service sends a message back to the activity with the text converted to uppercase letters. This means that the activity must also instantiate a receiver object so that it can receive messages from the service. When the service converts a received message to uppercase letters, it is only to show that the service has done something. Below is the application's window after clicking the START button and after a text has been entered and clicked on the SEND button:

With regard to the layout, there is nothing new, and it is only about placing 6 components in a window. The service is called *TheService*, and the code is as follows:

```
package dk.data.torus.theservices1;

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

```
public class TheService extends Service {
 public final static String KEY_COUNTER_CLIENT = "KEY_COUNTER_CLIENT";
 public final static String KEY_RECEIVE_CLIENT = "KEY_RECEIVE_CLIENT";
 public final static String GET_COUNTER_CLIENT = "GET_COUNTER_CLIENT";
 public final static String GET_MESSAGE_CLIENT = "GET_MESSAGE_CLIENT";
 public final static String KEY_MESSAGE_SERVICE = "KEY_MESSAGE_SERVICE";
 public final static String GET_MESSAGE_SERVICE = "GET_MESSAGE_SERVICE";
 private TheReceiver theReceiver;
 private TheThread theThread;
 private int counter = 0;

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

 @Override
 public void onCreate() {
  Toast.makeText(getApplicationContext(),"Service created",
   Toast.LENGTH_LONG).show();
  theReceiver = new TheReceiver();
  super.onCreate();
 }

 @Override
 public int onStartCommand(Intent intent, int flags, int startId) {
  Toast.makeText(getApplicationContext(), "Service started",
   Toast.LENGTH_LONG).show();
  IntentFilter intentFilter = new IntentFilter();
  intentFilter.addAction(GET_MESSAGE_SERVICE);
  registerReceiver(theReceiver, intentFilter);
  theThread = new TheThread();
  theThread.start();
  return super.onStartCommand(intent, flags, startId);
 }

 @Override
 public void onDestroy() {
  Toast.makeText(getApplicationContext(),"Service destroyed",
   Toast.LENGTH_LONG).show();
  theThread.terminate();
  unregisterReceiver(theReceiver);
  super.onDestroy();
 }

 public class TheReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
```

```
   String action = intent.getAction();
   if(action.equals(GET_MESSAGE_SERVICE)) {
    String msg = intent.getStringExtra(KEY_MESSAGE_SERVICE);
    Intent tosend = new Intent();
    tosend.setAction(GET_MESSAGE_CLIENT);
    tosend.putExtra(KEY_RECEIVE_CLIENT, msg.toUpperCase());
    sendBroadcast(tosend);
   }
  }
}

private class TheThread extends Thread {
 private boolean running;
 public void terminate() {
  running = false;
 }
 @Override
 public void run() {
  running = true;
  while (running){
   try {
    Thread.sleep(1000);
    Intent intent = new Intent();
```

```
      intent.setAction(GET_COUNTER_CLIENT);
      intent.putExtra(KEY_COUNTER_CLIENT, counter);
      sendBroadcast(intent);
      ++counter;
     } catch (InterruptedException e) {
     }
    }
   }
  }
}
```

Please first note that it is a service as the class inherits *Service*. Then note that the class *TheService* has two inner classes. The class starts by defining 6 constants, which are used to define parameters associated with *Intent* objects, as well as for the operations that the service must perform. The only requirement is that you have strict identities, but the actual values do not really matter – only in connection with debug.

After the 6 constants, three variables are defined, the first representing a receiver – has the type *TheReceiver* – so the application can send messages to the service. The next represents a thread that every second increments a counter with 1 and the last variable is the counter.

The receiver object is created in *onCreate()* and is an object whose type is *TheReceiver*, which is an inner class. You should note that the class inherits *BroadcastReceiver* and therefore you must implement the method *onReceive()*. It has two parameters, where the first is the *Context* object that sent a message while the last is the message packed into an *Intent* object. The method determines the value of *getAction()* from the Intent object, and if this value is one of the 6 constants, a parameter is determined with the key *KEY_MESSAGE_SERVICE*. The value of this parameter is converted to uppercase letters and packed into an *Intent* object named *GET_MESSAGE_CLIENT*, after which it is sent using the method *sendBroadcast()*. You should note that it is a method in the class *Context*. The relevant message is sent to the receiver objects, which are registered as listeners for messages with this name.

onStartCommand() creates an *IntentFilter* and registers *theReceiver* that listens for messages named *GET_MESSAGE_SERVICE*. Then, the method creates a *TheThread* object (also an inner class) that implements and starts a thread, which every second updates the variable *counter*. In addition, the thread creates an *Intent* identified by *GET_COUNTER_CLIENT* and adds the counter as a parameter, and that *Intent* is sent with *sendBroadcast()* to any registered activity. You should note that the service does not know if there are registered listeners for the actual action, but it is also not the idea that the service should know it.

Then there is the class *MainActvity*:

```
public class MainActivity extends AppCompatActivity {
 private EditText txtTosend;
 private TextView txtReceived;
 private TextView txtCounter;
 TheReceiver theReceiver = null;
 Intent theIntent = null;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  txtTosend = findViewById(R.id.txtTosend);
  txtReceived = findViewById(R.id.txtReceived);
  txtCounter = findViewById(R.id.txtCounter);
  findViewById(R.id.cmdStart).setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
    startService();
   }
  });
  findViewById(R.id.cmdStop).setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
    stopService();
   }
  });
  findViewById(R.id.cmdSend).setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
    String message = txtTosend.getText().toString();
    Intent intent = new Intent();
    intent.setAction(TheService.GET_MESSAGE_SERVICE);
    intent.putExtra(TheService.KEY_MESSAGE_SERVICE, message);
    sendBroadcast(intent);
   }
  });
 }

 private void startService(){
  if (theIntent == null)
   startService(theIntent = new Intent(MainActivity.this, TheService.class));
 }

 private void stopService(){
  if (theIntent != null) stopService(theIntent);
  theIntent = null;
 }
```

```
@Override
protected void onStart() {
 theReceiver = new TheReceiver();
 IntentFilter intentFilter = new IntentFilter();
 intentFilter.addAction(TheService.GET_COUNTER_CLIENT);
 intentFilter.addAction(TheService.GET_MESSAGE_CLIENT);
 registerReceiver(theReceiver, intentFilter);
 super.onStart();
}

@Override
protected void onStop() {
 unregisterReceiver(theReceiver);
 super.onStop();
}

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

```
 private class TheReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
   String action = intent.getAction();
   if (action.equals(TheService.GET_COUNTER_CLIENT)) {
    int value = intent.getIntExtra(TheService.KEY_COUNTER_CLIENT, 0);
    txtCounter.setText(String.valueOf(value));
   } else if (action.equals(TheService.GET_MESSAGE_CLIENT)) {
    String message = intent.getStringExtra(TheService.KEY_RECEIVE_CLIENT);
    txtReceived.setText(String.valueOf(message));
   }
  }
 }
}
```

The class starts by defining 3 variables used to refer to text fields in the user interface. In addition, there is a variable for a receiver whose type is *TheReceiver*, which is an inner class. Finally, there is an *Intent* variable that is used to start a service. In *onCreate()*, the three variables are initialized to components, and event handlers are associated to the three buttons.

The program also implements *onStart()*, where a receiver is triggered. An *IntentFilter* defines two actions that the receiver must handle after which it is registered. Also note the implementation of *onStop()*, which removes the registration. The receiver itself is again derived from *BroadcastReceiver*, and it works in principle as in the class *TheService*, just updating this time a component in the user interface, depending on what action it is.

The event handler for the START button calls the method *startService()*, which, in the same way as in the previous book, starts a service of the type *TheService*. The service in question is stopped in the event handler to the STOP button. Finally, there is the last event handler that sends the entered text in the entry field. This happens in the same way as shown above with *sendBroadcast()*. Here you should note that if the service is not started, nothing happens, because it simply means there is no listener for the action.

In order for it all to work, an element must be added to the manifest, which defines the service, but then the application can be tested by either the emulator or on a physical device:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.theservices1">
  <application
    …
    <activity android:name=".MainActivity">
      …
```

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

9.1 AN INTENTSERVICE

In the above example, the service performs its work using a thread defined as an inner class. Instead, you can define the service as an *IntentService* (see possibly the previous book) that automatically handouts the work to a thread. The project *TheServices2* is the same example as the above, but the service is instead an *IntentService*. The service basically works in the same way, but instead of starting the service and running it until it stops, the service is started after which it performs a job, but stops after running a number of seconds where the value is a parameter. The service's user interface is basically the same, but there is a button less. The code for the service is as follows:

```
package dk.data.torus.theservices2;

import android.app.IntentService;
import android.content.*;
import android.widget.Toast;

public class TheService extends IntentService {
 public final static String KEY_COUNTER_CLIENT = "KEY_COUNTER_CLIENT";
 public final static String KEY_RECEIVE_CLIENT = "KEY_RECEIVE_CLIENT";
 public final static String GET_COUNTER_CLIENT = "GET_COUNTER_CLIENT";
 public final static String GET_MESSAGE_CLIENT = "GET_MESSAGE_CLIENT";
 public final static String KEY_MESSAGE_SERVICE = "KEY_MESSAGE_SERVICE";
 public final static String GET_MESSAGE_SERVICE = "GET_MESSAGE_SERVICE";
 private TheReceiver theReceiver;
 private int counter;

 public TheService() {
  super("TheService");
 }

 @Override
 public void onCreate() {
  Toast.makeText(getApplicationContext(),"Service created",
   Toast.LENGTH_LONG).show();
  theReceiver = new TheReceiver();
  super.onCreate();
 }
```

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
 Toast.makeText(getApplicationContext(),"Service started",
  Toast.LENGTH_LONG).show();
 IntentFilter intentFilter = new IntentFilter();
 intentFilter.addAction(GET_MESSAGE_SERVICE);
 registerReceiver(theReceiver, intentFilter);
 return super.onStartCommand(intent, flags, startId);
}

@Override
public void onDestroy() {
 Toast.makeText(getApplicationContext(),"Service destroyed",
  Toast.LENGTH_LONG).show();
 unregisterReceiver(theReceiver);
 super.onDestroy();
}

@Override
protected void onHandleIntent(Intent intent) {
 counter = intent.getIntExtra("WORK_TO_DO", 10);
 while (counter >= 0){
  try {
```

```
    Thread.sleep(1000);
    Intent tosend = new Intent();
    tosend.setAction(GET_COUNTER_CLIENT);
    tosend.putExtra(KEY_COUNTER_CLIENT, counter);
    sendBroadcast(tosend);
    --counter;
   } catch (InterruptedException e) {
   }
  }
 }

 public class TheReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
   String action = intent.getAction();
   if(action.equals(GET_MESSAGE_SERVICE)){
    String msg = intent.getStringExtra(KEY_MESSAGE_SERVICE);
    Intent tosend = new Intent();
    tosend.setAction(GET_MESSAGE_CLIENT);
    tosend.putExtra(KEY_RECEIVE_CLIENT, msg.toUpperCase());
    sendBroadcast(tosend);
   }
  }
 }
}
```

The class now inherits *IntentService*, and otherwise the main difference is that the class *TheThread* has now been removed and instead replaced by the method *onHandleIntent()*. Here you should note that it has an *Intent* parameter, that in this case is used to transfer a parameter as an *int*, which indicates how many times the loop should iterate and thus the amount of work that the method should perform. For each iteration, the method performs in the same way as in the previous example and broadcast the counter's value, to a client (and in this case, *MainActivity*).

MainActivity is almost identical to the previous example (a bit simpler) and I do not want to display the code here.

9.2 A NEVER ENDING SERVICE

As the last thing about services, I will show an example consisting of two programs:

1. A program, that start a service end then stops. The service is still running and generates the primes, and for every prime the service broadcast the prime.
2. A program, that is registered as listener for the primes, and shows every new prime on the screen.

In principle, it is quite simple, but there is a simple challenge, which consists in keeping the service alive after the starting activity is completed. The service is part of the project *TheService3*, and there are three Java files that you should notice.

The first is *PrimesRestarter* and it is a quite simple receiver class:

```
package dk.data.torus.theservices3;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class PrimesRestarter extends BroadcastReceiver {
 @Override
 public void onReceive(Context context, Intent intent) {
  context.startService(new Intent(context, PrimesService.class));;
 }
}
```

As you can see, nothing happens in *onReceive()* than a service is started. For the time being, it's not easy to see what's going on with this class, but it should solve the problem that if an application starts a service, and afterwards the application closes, then it terminates the service.

The service class is called *PrimesService*:

```
package dk.data.torus.theservices3;

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

public class PrimesService extends Service {
 private long prime = 0;
 private TheThread theThread;
```

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
 super.onStartCommand(intent, flags, startId);
 theThread = new TheThread();
 theThread.start();
 return START_STICKY;
}

@Override
public void onDestroy() {
 super.onDestroy();
 sendBroadcast(new Intent("dk.data.torus.RestartPrimes"));
 theThread.terminate();
}

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

```
 private class TheThread extends Thread {
  private boolean running;
  public void terminate() {
   running = false;
  }
  @Override
  public void run() {
   running = true;
   while (running){
    try {
     while (!isPrime(prime))
     {
      ++prime;
      if (prime == Long.MAX_VALUE) prime = 0;
     }
     Intent tosend = new Intent();
     tosend.setAction("NEXT_PRIME");
     tosend.putExtra("PRIME", prime);
     sendBroadcast(tosend);
     ++prime;
    } catch (Exception e) {
    }
   }
  }

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

onStartCommand() creates a thread and starts it. The thread's type is an inner class and consisting of a loop that runs for as long as the variable *running* is *true* (until the method *terminate()* is performed). For each iteration, the next prime is determined and a broadcast of that number is performed. The idea is, of course, that if the service is allowed to run for a long time, it starts taking time to determine the next prime numbers, and from a client you will find that there is time between each number is displayed.

Then there is *onDestroy()* and it sends a broadcast with the value *dk.torus.data.RestartPrimes*. This means that the method *onReceive()* in the above class *PrimesRestarter* is performed and restarts the service. That is what ensures. that the service is still running after the starting activity is closed.

Then there is *MainActivity*, which this time is trivial:

```
package dk.data.torus.theservices3;

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

public class MainActivity extends AppCompatActivity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  startService(new Intent(this, PrimesService.class));
  finish();
 }
}
```

onCreate() starts the service, after which the activity closes itself. You should note that there is no content view – the application does not have a user interface. The project has an *activity_main.xml* and it could easily be deleted. It is automatically created by Android Studio and may be useful for testing.

Then the application that creates and starts the service is almost complete. All that is missing is the manifest that will tie it all together:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="dk.data.torus.theservices3">
  <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.theservices3.PrimesService"
      android:enabled="true" >
    </service>
```

```
    <receiver
      android:name="dk.data.torus.theservices3.PrimesRestarter"
      android:enabled="true"
      android:exported="true"
      android:label="RestartServiceWhenStopped">
      <intent-filter>
        <action android:name="dk.data.torus.RestartPrimes"/>
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

Here is the the most important the receiver element that tells *onDestroy()* in the class *PrimesService* what it is for receiver used to restart the service.

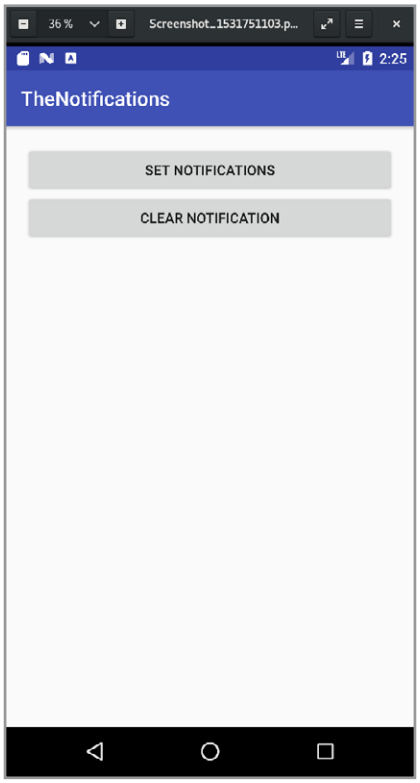
Finally, there is the client who is a normal app. The layout is just the default layout created by Android Studio. *MainActivity* is:

```
package dk.data.torus.theclient;

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

public class MainActivity extends AppCompatActivity {
 private TextView txtView;
 private TheReceiver theReceiver = null;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  txtView = findViewById(R.id.txtView);
  theReceiver = new TheReceiver();
  IntentFilter intentFilter = new IntentFilter();
  intentFilter.addAction("NEXT_PRIME");
  registerReceiver(theReceiver, intentFilter);
 }

 private class TheReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
   String action = intent.getAction();
   if(action.equals("NEXT_PRIME")){
    long value = intent.getLongExtra("PRIME", 0);
    txtView.setText("" + value);
   }
  }
 }
}
```

Here is not much new and you should primarily note the receiver class, where the method *onReceive()* is performed every time the service generates a new prime number – provided that the service is running. If it's first started (by the application *TheServices3*), it will in principle run forever or until the phone goes out, but you can of course always manually stop services.

10 NOTIFICATIONS

As the last thing about developing applications for mobile phones, I want to mention notifications where a program wishes to make the user aware that an event has occurred. This happens by invoking the user's attention and can happen, for example, by playing a sound or letting the phone vibrate, and often combining it by displaying an icon in the phone's notification bar. The program *TheNotifications* shows how to do that and it's actually quite simple as Android has a class *NotificationManager* that providing everything needed available. The program opens the following window:



The top button sends a notification while the other bottom removes the notification (so the icon disappears from the application's notification bar). The layout is quite simple and consists only of a *RelativeLayout* with two buttons. In addition to the layout, the program only consists of the class *MainActivity*:

```
package dk.data.torus.thenotifications;

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

```
public class MainActivity extends AppCompatActivity {
 private static final int NOTIFICATION_ID = 12345;
 private int count = 0;
 private NotificationManager notiMgr = null;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  notiMgr = (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
  findViewById(R.id.cmdNotify).setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
    notifyUser(v);
   }
  });
  findViewById(R.id.cmdClear).setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
    clearNotificaiton(v);
   }
  });
 }
```

```
 public void notifyUser(View view) {
  Notification noti = new Notification.Builder(MainActivity.this)
       .setContentText("Notice This")
       .setSmallIcon(R.drawable.alarm)
       .setWhen(System.currentTimeMillis())
       .setContentTitle("From Poul")
       .build();
  noti.sound =
    Uri.parse("android.resource://" + getPackageName() + "/" + R.raw.sound);
  noti.vibrate = new long[] {1000L, 500L, 1000L, 500L, 1000L};
  noti.number = ++count;
  noti.flags |= Notification.FLAG_AUTO_CANCEL;
  notiMgr.notify(NOTIFICATION_ID, noti);
 }

 public void clearNotificaiton(View view) {
  notiMgr.cancel(NOTIFICATION_ID);
 }
}
```

In *onCreate()*, an object of the type *NotificationManager* is created, which is the object used for notifications. Additionally, event handlers are associated to the two buttons, the first being the most important and calls the method *notifyUser()*. The class *Notification* has a static method *builder()* that is used to instantiate a *Notification* object. Values are what should be used for the application's notification bar, and you should especially note the assignment of an icon, which is a *drawable* named *alarm.png*. After the object *noti* is created, an audio file is assigned, which is also a resource and has the name *sound.mp3* and is stored in the directory *raw*. Final is defined that the device should vibrate twice. You are instructed to read the documentation for the class *Notification*, including the details of the individual values. After the object *noti* is initialized, the *NotificationManager* object is used to raise the notification.

11 ANDROID AND SECURITY

When writing programs, security plays an increasing role, and it also applies to applications for mobile phones and other Android devices, and even more than is the case is with PC applications, as you constantly download and install apps on your device. Android has a relatively short history and the security issue has been a part of Android right from the start, but nevertheless, the security of Android devices has always been challenged and that has constantly caused Google to strengthen the security requirements, and they has thus all the time changed the requirements for the applications that users can download and install on the phone. Therefore, I would like to close this book on developing mobile phone applications with some remarks about the security requirements and thus what an application can do.

As long as it's all about developing apps as it has been done in this and the previous book, where an app is tested in the emulator, or possibly installed and tested on a physical device using an USB cable, there are not many problems, and at least it is the developer who takes responsibility for his own programs. If, on the other hand, downloading apps from an app store (as is what ordinary users do), then there's a whole different case, because here you allow programs that you do not know how is made and what they do in the hidden, being installed and running on your own machine. Here is trust not enough, but procedures and controls are required to ensure that these apps do not damage the device and the user's data and possibly steal the user's data – if that is possible and if not, at least minimize the risk. All that's not simple and it complicates the development of programs considerably.

11.1 THE ANDROIDMANIFEST.XML

As already shown in many examples, an app can not immediately use many of the device's features, such as making a phone call, writing to a file, opening a website, and more. Here it is the developer who should give permission using a permission element in the manifest:

```
<uses-permission android:name="<name>" android:maxSdkVersion="<number>"
```

and you can have all the permission elements that are needed. The name is a constant defined in a package *android.permission*, and an example could be:

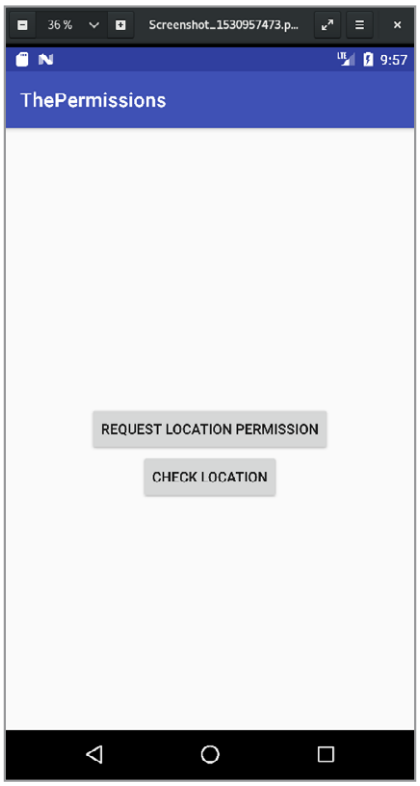
```
android.permission.INTERNET
```

The version number specifies a threshold for the Android version, which is tested when the application is installed and an Android release larger than *masSdkVersion* will simply ignore a permission request. The intention is not to ignore the permission requirements, but to solve the problem that the issue of the permission check for requirements change from version to version. For example, an app from version 19 onwards will no longer require a *WRITE_EXTERNAL_STORAGE* to write to a file as security is resolved in another way.

11.2 MARSHMALLOW AND BEYOND

From Android 6.0 Mashmallow, you can solve security in a more flexible way, where the user explicitly allows permission to use a particular feature such as the phone's camera, GPS, and so forth. Earlier it happened to all features when the application was installed based on the information in the manifest, but it is not very flexible. Now, if it is the first time that the feature in question is used, where you get a simple dialog box where you can allow or reject it for each feature, and you can go back to Settings under Settings | Apps and remove that permission. It requires a bit more of the programmer, and although I have already shown how, I will illustrate how to do with an example.

The application *ThePermissions* opens the following window:



Clicking on the bottom button lets you know that you do not have rights:

You have no permission

which means that you are not allowed to read the GPS coordinates. If you click on the top button, you get this right, that is, *ACCESS_FINE_LOCATION*, but the first time this feature is used, you will be prompted to give the permission, but else you will only get a *Toast* that tells that you to already has the permission.

After clicking on the top button, you get a *Toast* with the GPS coordinates every time you click the bottom button.

The program's code is as follows:

```
public class MainActivity extends AppCompatActivity {
 private static final int ACCESS_FINE_CODE = 123;
 private LocationManager locationManager;
 private Button cmdGrant;
 private Button cmdCheck;
 private boolean permission = false;
 private boolean waitingGps = false;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 locationManager = (LocationManager)
  getSystemService(Context.LOCATION_SERVICE);
 cmdGrant = findViewById(R.id.cmdGrant);
 cmdCheck = findViewById(R.id.cmdCheck);
 cmdGrant.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
  if(permission || hasPermission()) {
   Toast.makeText(MainActivity.this,"You already have the permission",
    Toast.LENGTH_LONG).show();
   permission = true;
  }
  else requestPermission();
  }
 });
 cmdCheck.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
   if (waitingGps) return;
   if(!permission)
   {
    Toast.makeText(MainActivity.this,
     "You have no permission",Toast.LENGTH_LONG).show();
    return;
   }
   try {
    waitingGps = true;
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
     10 * 1000, 1, gpsListener);
   }
   catch (SecurityException ex)
   {
   }
  }
 });
}

private boolean hasPermission() {
 return ContextCompat.checkSelfPermission(this,
  Manifest.permission.ACCESS_FINE_LOCATION) ==
  PackageManager.PERMISSION_GRANTED;
}
private void requestPermission(){
 if (ActivityCompat.shouldShowRequestPermissionRationale(this,
```

```
  Manifest.permission.ACCESS_FINE_LOCATION)) {
  Toast.makeText(MainActivity.this,"This permission is very dangerous",
   Toast.LENGTH_LONG).show();
 }
 ActivityCompat.requestPermissions(this, new String[]
  { Manifest.permission.ACCESS_FINE_LOCATION }, ACCESS_FINE_CODE);
}

@Override
public void onRequestPermissionsResult(int requestCode,
 @NonNull String[] permissions, @NonNull int[] grantResults) {
 if(requestCode == ACCESS_FINE_CODE) {
  if (grantResults.length > 0 && grantResults[0] ==
   PackageManager.PERMISSION_GRANTED) {
   Toast.makeText(this,"Yoy now have permission for check location",
    Toast.LENGTH_LONG).show();
  }
  else {
   Toast.makeText(this,
    "You are denied the permission",Toast.LENGTH_LONG).show();
  }
 }
 permission = ContextCompat.checkSelfPermission(this,
```

```
    android.Manifest.permission.ACCESS_FINE_LOCATION) ==
     PackageManager.PERMISSION_GRANTED;
  }

  private final LocationListener gpsListener = new LocationListener() {
   @Override
   public void onLocationChanged(Location location) {
    final double longitude = location.getLongitude();
    final double latitude = location.getLatitude();
    runOnUiThread(new Runnable() {
     @Override
     public void run() {
      Toast.makeText(MainActivity.this,
       String.format("Longitude: %1.4f\nLatitude: %1.4f", longitude,
       latitude), Toast.LENGTH_SHORT).show();
      locationManager.removeUpdates(gpsListener);
      waitingGps = false;
     }
    });
   }

   @Override
   public void onStatusChanged(String s, int i, Bundle bundle) {
   }
   @Override
   public void onProviderEnabled(String s) {
   }
   @Override
   public void onProviderDisabled(String s) {
   }
  };
}
```

Initially, there are in addition to a constant defined 5 variables, where the first one is to a *LocationManager*, while the next two are references to the two buttons. The variable *permission* is used to control whether the top button is clicked and the user has been given permission to read the location. The variable *waitingGps* is set to true when clicking on the bottom button and tells that a location reading has started. The goal is to avoid starting a reading again before the first has returned.

onCreate() fills a lot, but it is primarily due to the event handlers of the two buttons. The event handler for the top button tests the variable *permission* and calls the method *hasPermission()* to see if the user already has the permission. The method *hasPermission()* uses

```
ContextCompat.checkSelfPermission(this, Manifest.
permission.ACCESS_FINE_LOCATION)
```

to test if the application already has *ACCESS_FINE_LOCATION* permission and it will have it if it is not the first time the application is used. If the user already has the desired permission, a *Toast* is displayed and the variable *permission* is set to *true*. Otherwise, the method *requestPermission()* is called. Here is first called

```
ActivityCompat.shouldShowRequestPermissionRationale(this,
 Manifest.permission.ACCESS_FINE_LOCATION))
```

to test for *ACCESS_FINE_LOCATION* and the meaning is that you can do something to tell the user what this permission means. In this case, a simple Toast appears (which does not make much sense, since you can not reach to read it before the next *Toast*). Then the method

```
ActivityCompat.requestPermissions(this,
 new String[] { Manifest.permission.ACCESS_FINE_LOCATION }, ACCESS_FINE_CODE)
```

is called which opens a dialog box where the user must actively allow or reject the requested permission. The dialog performs a callback to *onRequestPermissionsResult()*, where the program with a *Toast* shows the result and the variable *permission* is initialized.

The event handler for the second button reads the GPS coordinates, if the application is otherwise allowed. First, it is tested whether a reading is already started, and if so, nothing happens other than the method returns. Otherwise, the variable *permission* is tested, and if it is *false*, showing a *Toast*. Otherwise, the variable *watingGps* is set to *true*, and the object *locationManager* is used to send a request for reading the coordinates. This happens in the same way as shown in the program *TheLocation* and with the object *gpsListener* that is created at the end of the program. When the coordinates are returned, the method *onLocationChanged()* is called on the *gpsListener* object, which displays a *Toast* with the coordinates and uses the *locationManager* object to remove the listener.

11.3 AN APP STORE

Once an application has been completed and tested, it should usually be placed on an app store like *Google Play* for others to download and apply the application. I do not want to show here how, as it depends on the current store, but for Google Play, I can mention that you need a *Google Play Developer Acount*, which costs a smaller start amount, after which you can upload apps by a defined procedure. However, the following is generally and applies in practice to any app that needs to be uploaded to a store.

Before you can publish an app to an app store so it can be downloaded and installed on an Android device, it must be signed with a certificate. Such a certificate must be purchased from a PKI (such as *Verisign* or *Thawte*), and this certificate allows you to sign the project's .apk file (which contains everything the app consists of). A certificate is stored in a *keystore*, and for that purpose you can use a tool called *keytool* that comes with JDK. If you've developed an app and tested it on your phone, you've actually already used a certificate and a keystore, as Android Studio creates a keystore (*Debug Keystore*) and signs your apps with a so-called self-signed certificate. Such a certificate is, of course, a certificate that no one trusts and therefore it is necessary to purchase a certificate from an authority (Verisign or Thawte), which others trust.

Once you have the certificate, you can sign your apps (that is your .apk files), and for that you can use another tool called *jarsigner*. Then your app is ready to be uploaded to an app store such as Google Play.

12 A FINAL EXAMPLE: THE WINEAPP

The final example of the book Java 6 is a program for managing a private wine cellar. The program is in principle excellent in the light of its functions and what it can, but for practical use, there is a big problem, namely that it is a PC program which means that you have to sit at the computer for access (purchase) of the wines, and correspondingly every time you use a bottle of wine, and especially the last is a problem. You will not get it done. You postpone it for a time when you are at the computer. It is therefore natural to think of a solution like an app for a mobile phone, which would mean that you always have the program "on you". Thus, the chance of using the program would be far greater, and the following is about implementing a version of the program *WineApp* as an app for a mobile phone. You can therefore think of the task of migrating a program from one platform to another.

The program should basically be the same and work in the same way as the previous version, but the program must use SQLite instead of MySQL. In addition, it should be considered whether there are functions that need to be changed and as the last especially with a view on how to simplify the application's use. In addition, you should use the phone's camera so that you can save a picture of a bottle.

One of the challenges of migrating the application to a phone is exactly the small screen. The application's user interface must be customized for a phone and a tablet, respectively, but since the user dialog largely consists of data entry, and since a phone does not have an appropriate user interface (due to the virtual keyboard), it has been decided that the program should not support landscape mode.

12.1 DEVELOPMENT PLAN

The development of the application is scheduled to include the following iterations, where the result of each iteration is an extension of the result of the previous iteration.

MainActivity. What should be on the start activity, and especially how to navigate between the application's features.	2
Program architecture and design and implementation of the database.	2
Maintenance of wine properties: Producers.	1
Maintenance of wine properties: Suppliers.	1

Maintenance of wine properties: Countries.	1
Maintenance of wine properties: Districts.	1
Maintenance of wine properties: Classifications.	1
Maintenance of wine properties: Grapes.	1
Maintenance of wine properties: Wine types.	1
Maintenance of wine properties: Categories.	1
Maintenance of wine properties: Bottles / packing.	1
Create wine. Also includes taking a picture.	4
Maintenance of wines. Also includes maintenance of the picture.	3
Purchase wines.	3
Consumption of wines.	3
Search wines.	4
Watch for wines ready to drink.	3
Refactoring and test.	4
Deployment	2

The numbers in the right column indicate the complexity of the iteration. They do not specify exact time estimates, but express the extent and where to expect special challenges. Assuming 1 = a half working day, it means that the expected workload is about 20 working days. Experience shows that there is always something that takes longer and it is always recommended to extend the estimated time by for example 25%, and if so I will estimate the time to approximate 25 working days, and if the program is to be written by one developer, it will say about 1 month.

Finally, new features may come in during the course of development and if so, of course, it is necessary to extend the estimate.

Looking at the above time estimates, they are not very precise and one can expect that some of the estimated tasks may take longer than expected, but there may also be others where you spend less than the estimated time, among other things because several of the iterations are similar and Therefore, to some extent, will be repetitions of each other. If it turns out, you save time and get ahead of the schedule, it is in principle good, but you should be careful about adjusting your expected time consuming early in the project, as

there is a high risk that other iterations may be underestimated and if other iterations have resulted in a time reserve is good, as you later in the process has something to do with.

Time estimation is one of the hardest dictations in software development, and software developers tend to underestimate (or do so for political / business reasons). The method is to break down a task in many small sub tasks, each of which is so small and has such a complexity that you can assess how long the task takes. Even if you get there, the timing is associated with great uncertainty, as practice shows that there are always tasks that for some reason take longer than expected. The reasons are many and range from simple tasks that suddenly take longer than are expected to that during the development process new demands arise. Particular attention should be paid to the latter, where new demands must be estimated for the time consumed and included in the project plan.

In addition to estimating the time to solve the individual tasks, one should also pay attention to how much time each project participant can add to the project. The time estimate and project participants are ultimately important for the delivery date and are a project participant assigned to the project on a full-time basis, many development organizations plan that a project participant can deliver a work that is 75% of a full-time job. Experience shows that a project participant must also solve other (and minor tasks) during a project period,

that the participant may be attend for a course or participate in other events and that the participant may become ill.

All these things are in favor of when a project can be completed and the product is delivered to the customer, but practice shows that it is far from always. As mentioned above, there may be many reasons, but in the vast majority of cases, the reason is that there are too many activities / tasks that are not estimated. Therefore, the key to time estimation is problem decomposition where a task is broken down into small, clear sub-tasks and focusing that all tasks being included.

12.2 THE PROJECT

The project is called *WineApp*, but I do not want to describe the project process here. I just want to mention that the project has been completed corresponding to the above breakdown of the task. When I do not want to describe the development, it is because there will only be a lot of words that, in fact, will not tell you a lot. On the other hand, you are encouraged to study the project and try it out, if not in the emulator, but on the phone. The program is relative to what is shown in this and the previous book, a major program, and more things about Android are used than what is otherwise mentioned.

With what this project adds of new things (and maybe it's even a part), the two books provide a foundation for mobile apps development, but there's actually a lot before it's all along, and especially you should be aware that the developments in the mobile phone area are still moving fast, and new things are constantly evolving and new ways of doing things are being developed. The previous has been solely devoted to the development of apps written in Java, and with practice you can be quite as effective as apps developer for Android devices. However, there are other tools, and if you are going to work with these tools, there is a good reason to be interested in these tools, which can sometimes be more effective than writing the applications in Java. In general, you should expect that in a mobile phone application development organization, you will meet other tools, and it's only good, because after all, the differences are not so big.