CHAPTER 0 – DOING

Preliminaries

D0.1 – THE TOOLBOX

D0.1.a – The IDE

Android Studio is the official Integrated Development Environment (IDE) for the Android platform. It includes all the tools needed to create, edit, build, test, and deploy Android apps. You will therefore need to install it on your development machine before attempting any of the projects in the "Doing" parts of this book.

The IDE is available on all the major platforms and can be download for free from:

```
https://developer.android.com/studio/
```

This single download has all the needed components, including Java, so no other download is needed. Follow the installation process and accept all the defaults.

There are a multitude of tools bundled within this IDE. In particular:

- **Java Tools**
  To develop the Java components of the app.
- **User Interface Tools**
  To develop the user interface components of the app.
- **Build Tools**
  To translate Java code to Dalvik (the code that Android devices can execute) and then to package all components and resources in one **APK** (Android Package).

You will learn how to use the IDE in the remainder of this chapter.

D0.1.b – The Device

The IDE has tools that test each and every component of your app. For example, there are tools that check your Java code as you type, and ones that verify your user interface as you build it. But even though each component is tested thoroughly on its own, you still need to ensure that the components work together when combined into one app. This type of integration testing is done by **deploying** the app on an Android device.

App deployment can be done on a real device or a virtual device. In the former, you connect the device to the computer using **ADB** (the **A**ndroid **D**ebug **B**ridge) and transfer the APK to it (a process known as **sideloading**). In the latter, you use an **AVD** (**A**ndroid **V**irtual **D**evice) to select any emulating device you want and run your app on it.

D0.2 – A QUICK TOUR OF THE IDE

D0.2.a – Creating a Project

We will create a new project for the *Doing* part of each chapter. Follow these steps to create Project Zero (the prompts and their order may differ slightly based on IDE version):

1. Launch Android Studio and start a new project.
2. Accept *all* defaults except for the ones indicated below:
   - Select *Phone/Tablet* for the target platform.
   - Select *Empty Activity* for the default screen.
   - Select *Java* for the programming language.
   - Name the project *Zero*.
3. Click Finish.

When Studio completes its project setup, it displays its main screen.

D0.2.b – The Main Screen

Spend some time to become comfortable with the content of this screen. Yes, it can be overwhelming at first but by focusing only on few aspects, you will gradually overcome the initial jolt. These aspects are highlighted in the remainder of this section.

The screen consist of many *panes*, windows that can be minimized, maximized, and docked. It is recommended that you keep only the following two panes open and minimize the rest: The **Project Pane** on the left and the bigger, multi-tab **Editor Pane** on the right.

Note that the Project tab (on the left margin of the project pane) is selected by default. If you accidently de-selected it, or if you find the project pane empty, simply click on it.

The project pane shows the components of your project and it does so in a number of views. By default, the **Android view** is selected (as shown in a drop-down tab that appears just above the project pane). In this view, the components appear in a tree outline, which you can expand or collapse by clicking the arrows. Two sections of this view are noteworthy:

- The subtree under **app**, **java** contains files that control the *behaviour* of the app, and is divided into three **packages**: one holds our Java classes, including the *activity* (verify and note the activity name), and we will call it the **main package**. The other two are used for testing, and we will use the one labeled *Test* to unit-test our code.

- The subtree under **app**, **res** contains files that control the *look-and-feel* of the app, and is divided into several **resource** folders. The folder named **layout** holds the user interface (verify and note that its name has an *xml* extension).

In addition to the Android view, the project components can be viewed via the **Project view**. Drop down the list of views (the tab above the project pane) and select Project. The resulting tree outline contains two sections that are noteworthy:

- The subtree under **app**, **src**, **main**, **java** contains the main package—the one that holds our project's Java classes, including the activity. Verify this.
- The subtree under **app**, **src**, **main**, **res**, **layout** contains the layout folder of our project. Verify this.

Hence, you can reach these two important components of our project via either view. We will find later that the Android view (the default) is best suited for development work, while the Project view is best suited for file-based operations, such as backup and restore, because it mirrors the directory structure in which the component files are stored on disk.

The Editor Pane to the right of the Project Pane is where we enter our code and do our user interface design. Note that our activity and layout names appear *in tabs* above the pane. Click the two tabs and verify.

D0.2.c – User Interface Tools

Click on the layout tab of the editor pane to bring the design editor to the front. By default, the editor shows two design surfaces: *Design* and *Blueprint*, but we only need the former. Hence, select the design-only surface using the toolbar above the design editor as shown in Fig. D0.1 (click the box marked ❶ and select *Design*).

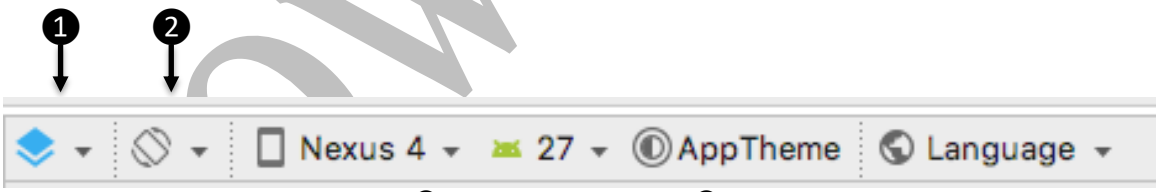

Figure D0.1 Configuring the design editor: ❶ choose the design surface, ❷ choose the device orientation.

The design surface shows the app as it will appear on the device. You can pick the device to be used (replacing *Nexus 4* on the toolbar of Fig. D0.1 with another device) but there is no need to change the default. More importantly, the box marked ❷ in Fig. D0.1 enables you to change the device **orientation** from *Portrait* to *Landscape*. Try it out. This feature is extremely useful as it allows us to verify the correctness of our design. Good user interface design should tolerate orientation changes and this button helps us test this.

The *user interface* (UI) of an app involves a number of widgets that allow the app and its user to communicate with each other. The three most-used widgets are:

- **Label**
  This is a box in which the app can display text for the user to see. The user cannot write into this box (i.e. it cannot receive the focus). In Android lingo, labels are called text views because the name of their Java class is **TextView**.

- **Textbox**
  This is a box in which the user can type text in; i.e. make an input. In Android lingo, it is known as an **EditText** because this is the name of its Java class.

- **Button**
  This is a button that the user can tap in order to ask the app to perform some action. The button has a *caption* written on it to let the user know the action it performs. In Android lingo, it is known as a **Button** because this is the name of its Java class.

Each widget has a unique **id** that uniquely identifies. It also has a large number of **attributes** (aka properties) that you can control manually during the design and programmatically while the app is running. One of the most frequently used attributes is called **text**. It holds the text displayed in a label; the text entered by the user in a textbox; and the caption of a button.

The Android library comes with a large number of ready-made widgets that allow you to create almost any required UI. These widgets appear in the **Palette** window that appears to the left of the design editor. Click *All* in the palette window to see all the available widgets. Note that the EditText widget (i.e. the textbox) is listed under the name **Plain Text**, so whenever you are asked to insert a textbox in this book, pick the Plain Text from the palette.

Our Zero app comes with a default label with "*Hello World*" as its text. Click on it to see its attributes displayed in the **attributes pane** that appears to the right of the design editor. The pane lists the most-often used attributes such as the id, layout, text, and text format (font, colour, style, etc.). You would normally set the attributes specified in the requirement document and leave the rest at their default values.

In addition to viewing the UI graphically as a collection of widgets, which is useful at the design stage, you can also view it textually as XML code, which is useful for editing. By clicking the three toolbar icons (Fig. D0.2), you switch from code, to design, to split view (which shows both). Try it.


Figure D0.2

We now turn our attention to how widgets are laid out relative to each other and relative to the borders of the device. This task is rather challenging because we don't know the size of the device on which our app will be deployed or how the user will hold it.

Note that the default *Hello World* label appears in the centre of the screen. Flip the orientation to landscape (as was explained in Fig. D0.1) and note that it is still in the centre. You can also change the emulated device from the default phone to a small watch or to a big tablet, yet the label will remain in the centre.

The placement of widgets should not be done via absolute dimensions but rather through **layout managers** that reposition the widgets dynamically (i.e. when the app is running) based on device size and orientation.

Android comes with several layout managers such as:

- **ConstraintLayout** which is explained below.
- **LinearLayout** which arranges the widgets vertically or horizontally.
- **RelativeLayout** which arranges the widgets relative to each other.
- **Grid** and **Table** layouts which place widgets in rows and columns.

The **constraint layout** is the default—the IDE auto-selects it when an activity is created. We will use this layout for all the apps of this textbook because it is very versatile. One of its key advantages is that it can accommodate most UI requirements with a *flat* hierarchy; i.e. all widgets would reside inside it without having to put a layout inside another layout.

In order to familiarize ourselves with the key features of the Constraint Layout, let us build the UI shown in Fig. D0.3. This UI has the following features:

- The first row has two buttons, with captions `0` and `1`, residing in the upper-left and upper-right corners of the screen.
- The second row has a button with caption `2` centered horizontally in the screen.
- The third row has two buttons, with captions `3` and `4`, with equal distances between them and the screen edges.
- The fourth row has two buttons, with captions `5` and `6`, stretched across the screen.
- Finally, there are two buttons, with captions `7` and `8`, residing in the lower-left and lower-right corners of the screen.

Note that adding a button to the design surface is easy: you simply drag the button widget from the palette to the device and then set its *text* attribute (in the attributes pane) based on the required caption. Laying out the button to match Fig. D0.3, however, is not as easy. If you do it manually, by visually placing the button where it belongs, it may look fine at first, but once you switch the orientation to landscape, you will immediately realize that it is not. Instead, we should let the constraint layout place the buttons as explained below.

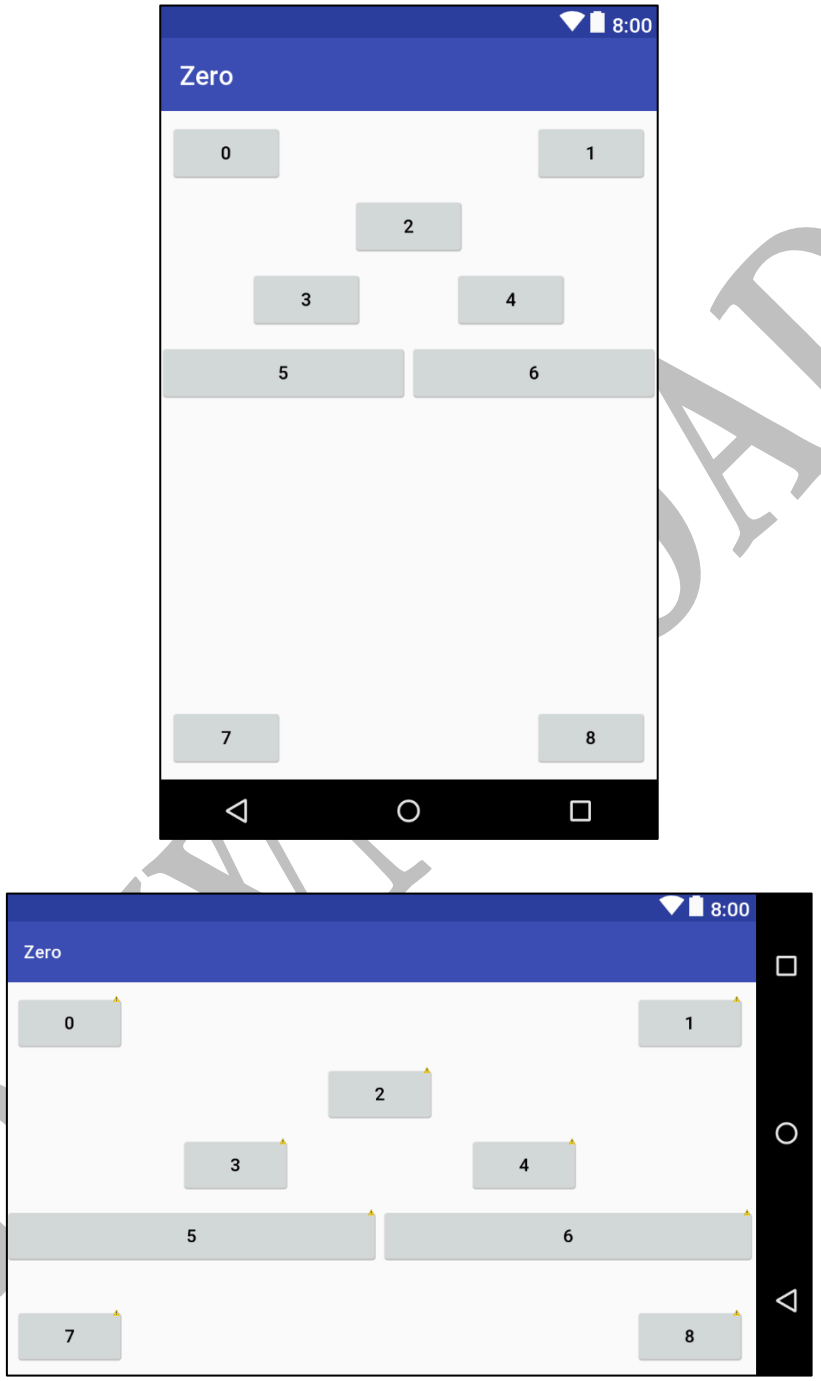

Figure D0.3 UI of the Zero app in portrait (top) and landscape (bottom).

Follow these steps:

1. Delete the Hello World label which was added by default.
2. Drag a button from the palette and place it somewhere in the middle of the screen.
3. Click the button and set its text attribute in the attributes pane.
4. Note that the button has **circular constraint handles** on each side.
5. Click and hold the constraint handle on the left side and drag it to the left edge of the screen. This forces the button to hug the left edge (in any orientation).
6. Click and hold the constraint handle on the top side and drag it to the top edge of the screen. This forces the button to hug the upper edge (in any orientation).

The `0`-button is now anchored to the upper left corner. Perform similar steps to anchor the `1`-button to the upper right corner.

7. Drag another button to the middle for the `2`-button.
8. Click and hold the constraint handle on the top side and drag it to the bottom handle of the `0`-button. This forces the button to be under the first row.
9. Click and hold the constraint handle on the left side and drag it to the left edge of the screen. This forces the button to hug the left edge. Then, click and hold the constraint handle on the right side and drag it to the right edge of the screen.

The `2`-button has to satisfy two horizontal constraints, one pulling it to the left edge of the screen and one to the right edge, so it centres itself across the screen.

10. Drag two more button widgets to create the `3` and `4` button.
11. Select both buttons and right-click the selection. Choose **Chain** and then *Create Horizontal Chain*.

The `3` and `4` buttons are now properly laid out horizontally. To anchor them vertically, simply drag the top handle of each and drag it to the bottom handle of the `2`-button.

12. For the `5` and `6` buttons, do the same thing (i.e. chain them) but change their **layout_width** attributes from **wrap_content** to **match_constraint**.

Finally, add the `7` and `8` buttons and constrain them to the bottom left and right corners. Verify your design (ideally after every placement) by switching the device orientation. The constraints of the top four rows are shown in Fig. D0.4.

It should be noted that we have covered only a small subset of the capabilities of this layout manager. In particular, the attributes pane allows you to control the margins and the horizontal bias of every widget.

As you add widgets, the IDE keeps track of their types and id's and depicts these in a pane known as the **component tree**. A sample of this tree is shown in Fig. D0.5 for our app.

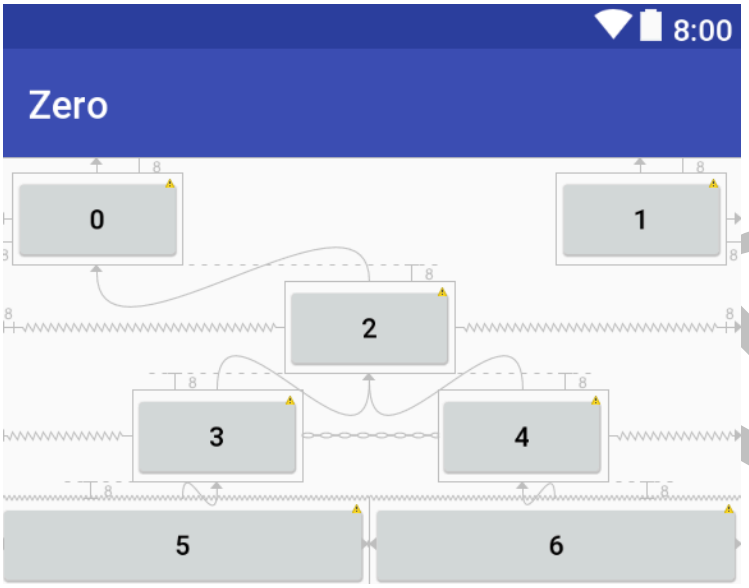

Figure D0.4 The top components of the Zero app with constraints shown.

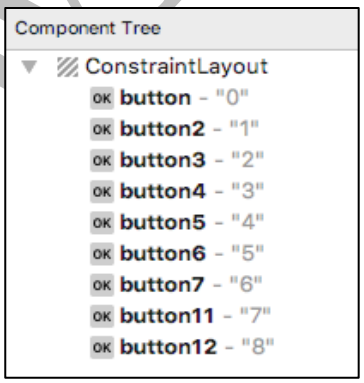

Figure D0.5 The component tree of the Zero app.

D0.2.d – Saving Your Work

Whenever you work on a project, it is important that you save your work frequently so that it can be recovered should your development environment become unusable (perhaps due to a software bug or a hardware malfunction).

There are several options for saving your work:

- **Use a Version Control System**
  This is by far the most professional approach. A system such as *git* allows you not only to save your work both locally and to the cloud, but also to keep track of changes made by you or by others in your team. Moreover, it integrates seamlessly with our IDE.

- **Zip and Save the Project Folder**
  This quick-and-dirty method uses the operating system's zip utility to compress the project folder (e.g. *AndroidStudioProjects/Zero* for our Zero app) into a file (e.g. *Zero.zip*) which can then be moved to cloud storage or a USB flash drive. This method has a high overhead as it is not automatic and the zipped file is rather large (~30MB). Moreover, it has a dependency on the particular version of the IDE.

- **Save Only Your Work**
  For most projects, your work is captured in very few files, typically one XML file and a couple of Java files. You can find these files in these folders:

  ```
  AndroidStudioProjects/[app]/app/src/main/java/[package]
  AndroidStudioProjects/[app]/app/src/main/res/layout
  ```

  where [app] is your app name (e.g. *Zero*). These *text* files are tiny (~kilobytes) and can easily be saved; in fact, you can simply copy-and-paste their contents directly from the editor. Most beginners like this method because it allows them to actually see and study the copied files, and easily reuse them in future projects.

D0.2.e – Running a Project

To run your project on a virtual device, click *AVD Manager* in the Tools menu and create an AVD (e.g. a phone or a tablet) and download any image (e.g. Q). This takes some time but it is only done once. Once created, you simply select it in the Run menu and run.

To run on a physical Android device, turn on debugging in its setting, and then connect it to your computer. It will appear in the list along with your virtual devices so you can select it in the Run menu and run your app on it.

D0.3 – EXERCISES

1. Modify the layout of the *Zero* app so that Button 3 becomes as in Fig. D0.6 below.

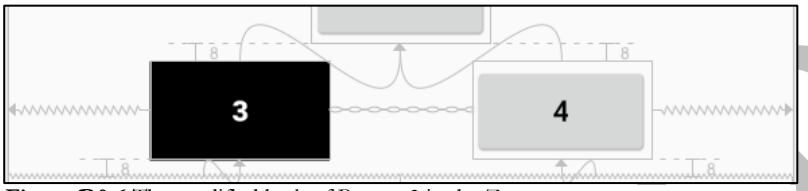

Figure D0.6 The modified look of Button 3 in the Zero app.

As you can see, the button's background has become black and its caption's colour has become white.
Hint: Click on the brackets (or ellipsis) to the right of the attribute to access the available colours.

2. Add a ninth button to the Zero app with layout and caption as shown in Fig. D0.7. Its location must be such that:

   - It divides the screen horizontally by three quarters to one (75% of the screen width to its left and 25% to its right).

   - It divides the distance between the fourth row (i.e. Buttons 5 and 6) and the bottom edge of the screen by one quarter to three; i.e. 25% of that distance above it and 75% below it.

   Verify your design by changing the orientation of the emulated device from portrait to landscape and by changing the device type.

   Hint:
   The attribute window shows an outline of the button widget above the partial attribute list. That outline has two so-called bias slider that allow you to adjust the left-right and top-bottom ratios. The default is 50-50 but you slide the bias to any desired percentage.

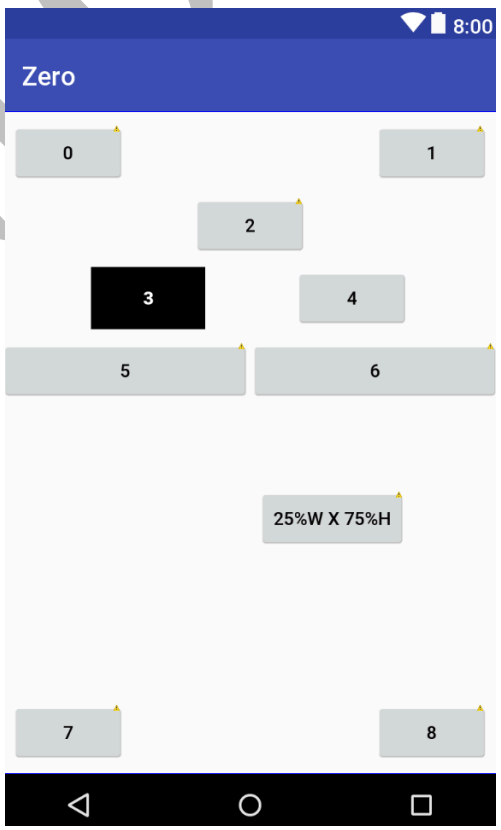

Figure D0.7 The Zero app with an added button.

3. We viewed our design on the screen but have not yet deployed our *Zero* app on a device. Let us do that now. Click the *Run* menu of the IDE and select Run… and then app. To run on an emulator, pick any emulator you like (or create a new virtual device). After a delay, the emulated device should appear in a window of its own (separate from the IDE). It will boot up like a real device and your app will be downloaded on it (i.e. it will be stored in its *Downloads* folder). The app will be launched automatically and you will be able to interact with the app using your mouse. You can even rotate the device or trigger its sensors by using its side toolbar.

4. We will now deploy the *Zero* app on a *real* device rather than an emulated one. You will need an actual Android device for this. By default, Android allows only signed apps to be installed on the device (for security reasons) so you need to change the settings of your device to allow our *unsigned* app to be installed. Once done, connect your device to your computer's USB port and re-run the app. The IDE will now ask you to select the device to run the app on and it should show the emulated device and the real device just connected. Select the real device. After a brief delay, the app will be side-loaded and it will appear on your device screen.

❖ ❖ ❖ ❖

5. Change the captions of the eight buttons of the Zero app from digits to names of digits; i.e. `ZERO` instead of `0`, `ONE` instead of `1`, and so on. But rather than hard-coding the names, we will learn a more professional, best-practice approach.

   To change the `text` attribute of Button 0 click the brackets (or ellipsis) to the right of the attribute in the attribute window and click + to *Add new resources* in the window that pops and then add a *string Value*. Enter `s0` as the *Resource name* and `ZERO` as its *value*. Click ok and note that the `text` attribute of this button has become: **`@string/s0`**.

   Repeat this process for the remaining buttons to make the final layout is in Fig. D0.8 (ignoring the button added in Exercise 2).

   Referring to the values `ZERO`, `ONE`, … `EIGHT` via names (such as `s0`, `s1`, … `s8`) creates a level of indirection between the attribute and its value. For example, the `text` attribute of Button 5 is now `@string/s5` rather than a fixed value. The advantage of this indirection is that it makes our app scalable and easier to maintain. A case in point is shown in Exercise 6 where the indirection enables the app to become sensitive to the device locale thereby changing the caption from `FIVE` to `CINQ` to خمسه automatically depending on the device's language.

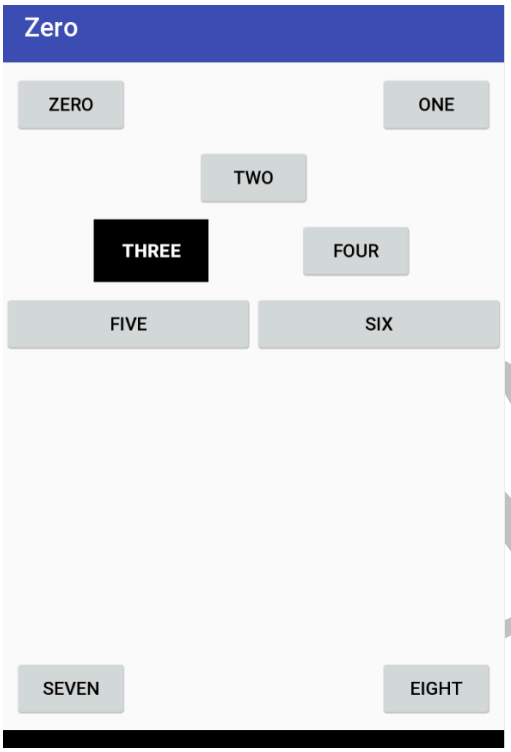

Figure D0.8 Digits replaced with digit names.

As you will discover, indirection is a recurring theme in Computer Science. By separating the concerns and creating intermediate abstraction levels, we can confront complexity and architect robust and scalable solutions.

Note: The name-value pairs that you have just created are now recorded in `strings.xml` *(a file that resides in app/res/values in the Project Pane) as shown below. In fact, you could have created these resources yourself by editing the file directly. Go ahead and double click this file to see it.*

```
<resources>
    <string name="app_name">Zero</string>
    <string name="s0">ZERO</string>
    <string name="s1">ONE</string>
    <string name="s2">TWO</string>
    ...
    <string name="s7">SEVEN</string>
    <string name="s8">EIGHT</string>
</resources>
```

6. Let us make our Zero app English/French bilingual by making the captions locale sensitive; i.e. they automatically switch language based on the language of the device the app is running on. Follow these steps:

   - Double-click the `strings.xml` file that was built in the previous exercise.
   - Click on *Open editor* in the upper right.
   - Click the globe in the upper left and select French (Canada).
   - For each of our eight resources, click the English name and then enter its corresponding French translation in the bottom. You can see the translations in Fig. D0.9.

   To test our work, go back to the layout tab (or double-click the layout in the project pane). The toolbar above the device has a **globe** that allows you to select the language. Switch to French and observe.

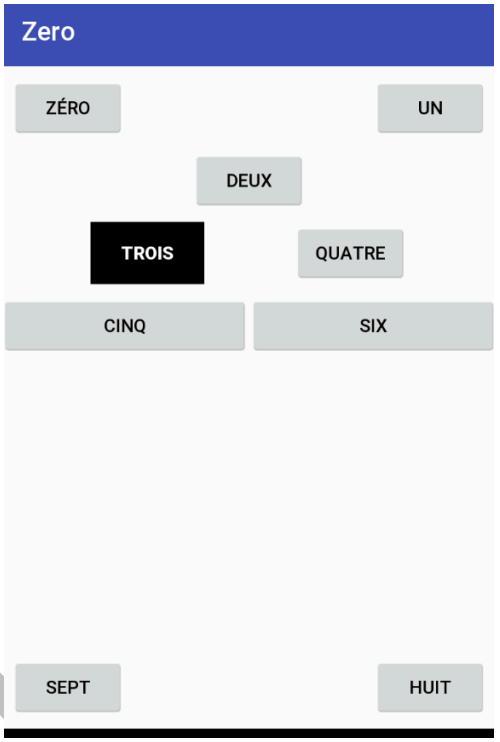

Figure D0.9 The Zero app in a French locale.

Notes:

– *To test your work on an emulator, run the app as usual and when the emulator window boots up, go to the emulated device settings and change the default language to French. You may have to add the language if it is not already present.*

– *Similarly, you can test your work on a real device by deploying the app on it and then changing its default language.*

– *Go back to the resource editor and click the globe again. Select a new language that you know and enter the translations. Verify that your app is now tri-lingual.*

– *The translations that you have entered get stored in a file that is also named* `strings.xml` *and you can find it also under app/res/values in the Project Pane. On disk, it resides under res in a folder parallel to values named* `values-fr-rCA`.

– *As before, you could have created the translations by editing this file directly. In general, anything that the IDE does can also be done manually by creating files.*

7. In Exercise 1 you were asked to change a couple of attributes for one of the buttons. In this exercise, you are asked to make the same change to *all eight* buttons, but rather than doing so one button at a time, you need to style all buttons in one shot. Specifically:

   - Double click the `styles.xml` file under `app/res/values`.
   - The file already contains a default style. Add a new style named `"i2c"` by appending the file (but before the closing `</resoueces>` tag) with the following:

```
<style name="i2c">

    <item name="android:textColor">#ff0000</item>

    <item name="android:background">#0000ff</item>

</style>
```

   - This style sets the text colour to `#ff0000`. This hex number specifies the red-green-blue components of the colour, with each component's intensity ranging between `00` and `ff`. Hence, the text colour is set to maximum red, no green, and no blue.
   - The style also sets the background colour to `#0000ff` or blue.
   - Change the style of all eight buttons to this new custom style. For each button, click it and type `i2c` in its `style` attribute in the attribute window.

   Did all buttons change as expected? If so, modify the `i2c` style so that the background colour becomes black and the text colour becomes white. Again, you should see the change immediately taking effect in the layout tab.

   Finally, elaborate on the following statements by arguing for or against them:

   - This styling methodology introduces a level of indirection between a widget and its attributes.
   - This styling methodology makes our app easier to maintain.
   - This styling methodology enables us to give our app a consistent look and feel.
   - This styling methodology separates presentation from the formatting.
   - If you are familiar with *html* (hyper-text markup language) and *css* (cascading style sheets), then compare and contrast this styling methodology with css.

8. In Exercise 7, we created a custom style for all our buttons. In this exercise, we create a second style as follows:

   - Double click the `styles.xml` file under `app/res/values`.
   - Add a new style named "`i2c.big`" after the first and before the closing `</resoueces>` tag as follows:

```
<style name="i2c.big">

    <item name="android:textSize">20sp</item>

</style>
```

   - The name of the new style is prefixed (with a dot separator) by the name of the first style. This means the new style inherits all the items in the first style but it can add or override some items.
   - The new style adds an item relating to the size of the text font. It sets it to 20sp.
   - The unit **sp** (scalable pixel) allows device independence because it is converted to actual screen pixels based on the resolution of the device. It is similar to **dp** (which is used for widget size and layout) but is intended specifically for font size.
   - Change the style of some of the buttons (e.g. the even ones) to the new custom style.

   Did the captions change as expected? If so, modify the `i2c.big` style so that it overrides the background colour so it becomes different from the one set in the parent `i2c` style.