Worksheet 3: Lists and Fragments

Updated: 2nd September, 2018

In this worksheet, you'll implement an app using fragments and `RecyclerView`. We'll also look a bit more carefully at `ConstraintLayout`.

> **Note:** The game from worksheet 2 is a good candidate for the use of fragments and `RecyclerView`, and retrofitting these concepts into it is a good learning exercise.
>
> However, this worksheet actually introduces a completely separate app, in the interests of keeping people from being left behind.

What Are We Making?

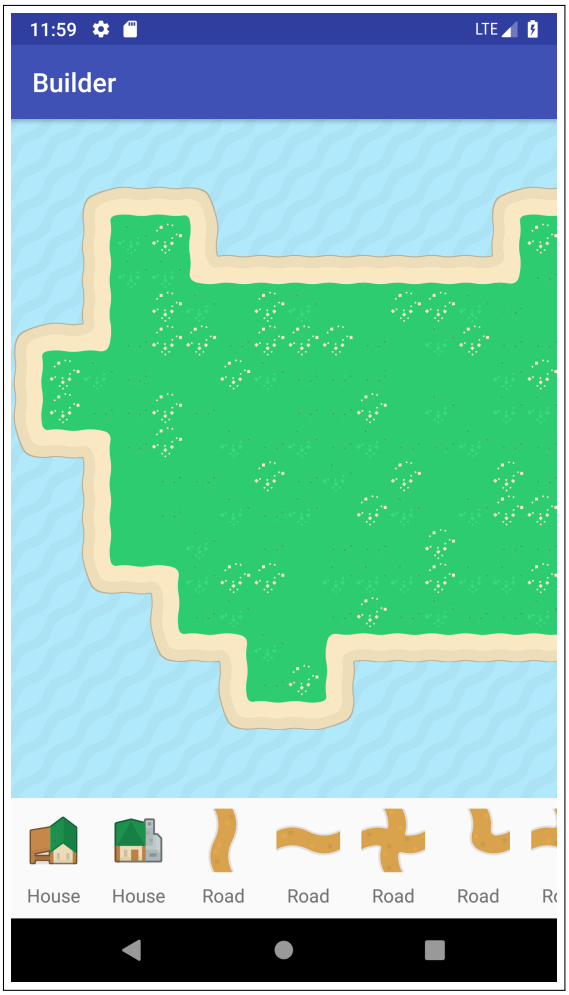
This new app will be a simple map editor – something you might use in conjunction with certain games.

As you can see, there are two parts to the UI, and we're going to put each of them in a fragment:

- The map occupies most of the screen. It actually uses a `RecyclerView`, but with `GridLayoutManager` rather than plain-old `LinearLayoutManager`.

  Unfortunately, this does still limit us to one-directional scrolling, so our map has to be long and thin.

- The selector is the list of items at the bottom. It too uses a `RecyclerView`, this time with `LinearLayoutManager`. The idea is that the user taps/clicks on an item in the selector, and can then place that item by tapping/clicking on points on the map.

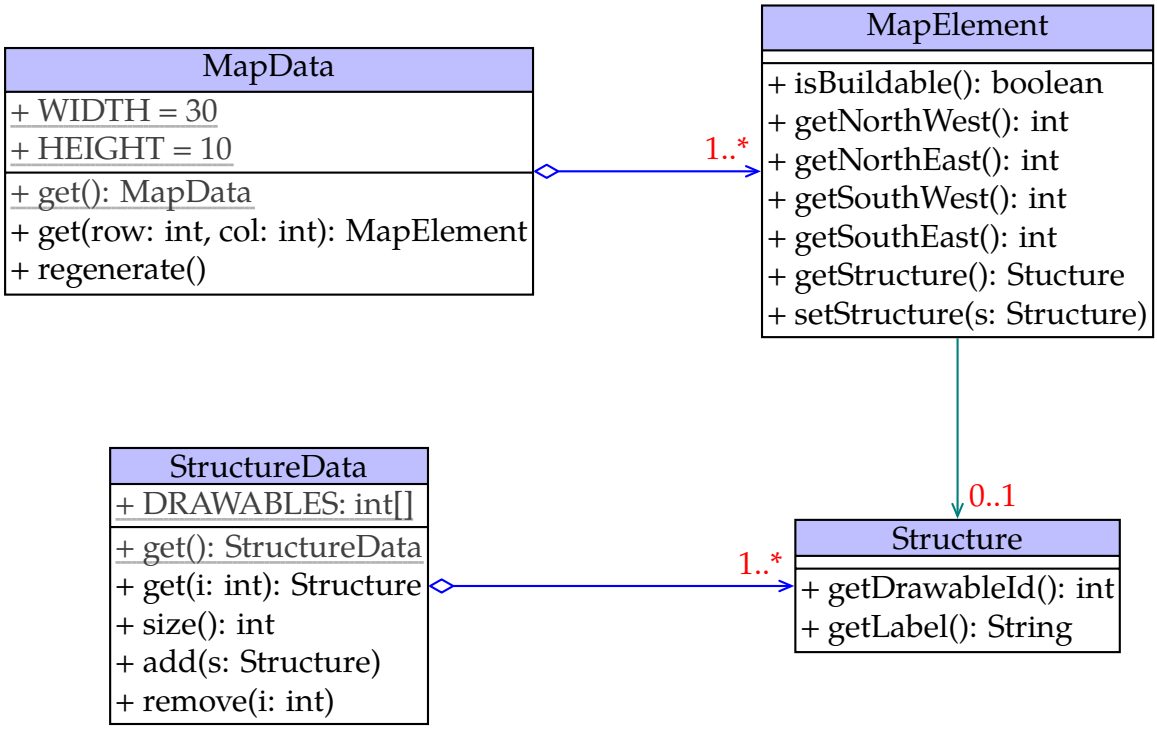


Obtain a copy of `islandbuilder.zip`. It contains all the vector graphics[1] you need (in the form of a series of XML files), along with the some Java model classes.

[1] Courtesy of www.kenney.nl.

Data

This time, you don't have to design or implement the model classes, but you do need to understand them:



You could figure this out from the source code, but let's go through each one anyway:

`MapData` represents the overall map, and contains a 2D array of `MapElement` objects, accessible using the `get(row, col)` method. The two static constants `WIDTH` and `HEIGHT` indicate the size of the map.

There is a static `get()` method to be used to obtain an instance (rather than calling the constructor directly), as follows:

```
MapData md = MapData.get();
```

There is also a `regenerate()` method. The map is randomly-generated, and this method will invoke the algorithm again to replace all the map data with a new randomly-generated grid.

`MapElement` represents a single grid square in the map. Each map element has both *terrain* (i.e. water, land, beach) and an optional *structure* (house, road, tree).

The terrain comes in four pieces, as if each grid square was further divided into its own tiny 2×2 grid (north-west, north-east, south-west and south-east). Each piece of the terrain is represented as an `int`, which is actually a *drawable* reference. That is, if you have both a `ImageView` and a `MapElement`, you can do this:

```
ImageView iv = ...;
MapElement me = ...;
iv.setImageResource(me.getNorthWest());
```

This will cause the `ImageView` to display the grid square's north-western terrain image, whatever it is.

(The terrain is broken up like this because there are a lot of possible combinations of terrain images for each grid square. If we had a single terrain image for each square, we'd need to manually combine all the possible combinations of images, and we'd get a small explosion of image files.)

Meanwhile, the structure is something we want to display over the top of the terrain. Each `MapElement` has either *zero or one* `Structure` objects. For each grid square, we can also change which structure is built on it.

`StructureData` stores the list of possible structures. A bit like `MapData`, this has a static `get()` method for retrieving an instance:

```
StructureData sd = StructureData.get();
```

The remaining methods – `get(int)`, `size()`, `add(Structure)` and `remove(int)` – provide minimalistic list functionality.

There is a static int array called `DRAWABLES`, which stores all the drawable integer references, some of which are not actually used (yet) in a `Structure` object.

`Structure` represents a possible structure to be placed on the map. A structure simply contains a *drawable* int reference, and a string label to be shown in the selector.

Build Dependencies

A quick detail first. Since we're relying on libraries for `ConstraintLayout` and `RecyclerView`, you should check to see that these have been added to your project. Open `app/build.gradle`, and look for these lines (where the version numbers don't really matter):

```
...
dependencies {
    ...
    implementation 'com.android.support.constraint:constraint-layout:1.1.2'
    implementation 'com.android.support:recyclerview-v7:28.0.0-rc02'
}
```

If one or both are not there, you will need to add them. The IDE will then warn you to "Sync Now", so that it can work with the new libraries.

Alternatively, rather than directly editing build.gradle, you can select "File" → "Project Structure", then select the "app" module, and then the "Dependencies" tab. You should see the lists of dependencies. If the required dependencies are not there, you can select "+", and then "Library dependency", and then choose from among the options shown.

UI Structure

We're going to make more careful use of `ConstraintLayout` this time, and we'll be editing XML directly (at least some of the time).

(The drag-and-drop editor may be useful for bits and pieces, such as setting up a basic layout file before you begin editing it. You're welcome to try to use it as much as possible, but it will often be difficult to see what you're doing. For `FrameLayout` and `RecyclerView`, the editor won't know anything about what they actually contain, so they'll appear tiny and invisible.)

Warning: If you get a blank screen when actually running your app, you probably haven't given `ConstraintLayout` enough information to work out what size the various UI elements should be, and so it's making them invisibly small.

If you get stuck on this, try (temporarily) setting a fixed size for various parts of the UI, so that you can move on to the controller logic, and at least get the functionality working.

Let's discuss the structure of the five separate layout files that we'll need:

`activity_main.xml` is attached to the activity, of course, and describes the "big picture". This will be a `ConstaintLayout` containing two `FrameLayout` elements, one for each fragment.

We want the UI to fill the whole screen, which means the `ConstraintLayout` element should have these attributes:

```
<android.support.constraint.ConstraintLayout
    ...
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ...>

    ...

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

"`match_parent`" makes the element occupy the entire space of its "parent" (which for the top-most element is the entire screen). However, for the child elements of `ConstraintLayout`, things are more complicated, and `match_parent` is discouraged (and often unworkable).

Instead, we choose between `0dp` (i.e. zero dp units; also called "`match_constraint`"), which fills up the *available* space, and `wrap_content`, which bases the element's size on its contents. The map simply has to be as big as possible, so its width and height should both be `0dp`:

```
android:layout_width="0dp"
android:layout_height="0dp"
```

The selector must be as wide as possible, but only as high as necessary:

```
android:layout_width="0dp"
android:layout_height="wrap_content"
```

Now we need some constraints to ensure the elements appear in the right places. This is a constraint:

```
<FrameLayout
    ...
    app:layout_constraintLeft_toLeftOf="parent"

    ... />
```

This says to align the left side of the `FrameLayout` to the left side of its parent (i.e. the `ConstraintLayout`). The same applies to the right, top, and bottom of any element. In this case, we need to constrain both `FrameLayouts` to both the left and right edges. We also need to constrain the map to the top, and the selector to the bottom.

And last, but not least, we need to constrain them to one another (so they don't overlap or leave a gap). In particular, we have to constrain the *map* to the *selector*, because the selector will already know its required vertical size, and the map just takes up whatever space is available:

```
<FrameLayout
    android:id="@+id/selector"
    ... />

<FrameLayout
    android:id="@+id/map"
    app:layout_constraintBottom_toTopOf="@id/selector"
    ... />
```

(The IDE knows what all the possible constraints are, so it can help here.)

Notice we've also added IDs here, and we're immediately making a reference to one.

`fragment_map.xml` and `fragment_selector.xml` describe the two fragment-level UIs. The map and selector will each consist of a single `RecyclerView`, and nothing else:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/mapRecyclerView" />
```

Adjust the ID attribute as appropriate.

`grid_cell.xml` describes the contents of each square in the map grid. (The squares are not explicitly shown in the screenshot, but you can tell from the blockiness of the map that it is divided into a grid.)

This is a little tricky. First we need a `ConstraintLayout` containing *five* `ImageViews`: four terrain images in a 2×2 grid pattern, and a fifth structure image occupying the entire area (over the top of the first four).

An `ImageView` is just another GUI element (like `TextView`, etc.), but one that displays an image. If you use the drag-and-drop editor to create the `ImageViews`, it will ask you to specify an image resource to display. This isn't known at compile time, in this case, so instead it's best to select "Color" $\rightarrow$ "android" $\rightarrow$ "transparent".)

Set the left, right, top and bottom constraints as you did before, to fix all five images at their correct locations. For all five, we also want to set the width and height to `0dp` (so that the size will be a fraction of the overall map size).

For the four terrain images, we also need to ensure they all occupy equal space. `ConstraintLayout` has another feature for doing this:

```
<ImageView
    ...
    app:layout_constraintWidth_default="percent"
    app:layout_constraintHeight_default="percent"
    app:layout_constraintWidth_percent="0.5"
    app:layout_constraintHeight_percent="0.5"
    .../>
```

We need one more thing that (as far as I'm aware) isn't easily done within the layout file itself. We want the grid cells to be *square* (not just rectangular). Their *height* can be determined from the amount of screen space available, but this doesn't constrain their *width* (and horizontal scrolling means the total horizontal space is basically unlimited).

So, at the appropriate place (the constructor of your `ViewHolder` subclass), we need the following:

```
int size = parent.getMeasuredHeight() / MapData.HEIGHT + 1;
ViewGroup.LayoutParams lp = itemView.getLayoutParams();
lp.width = size;
lp.height = size;
```

That is, we see what the `RecyclerView`'s height *actually is* at runtime, divide that by the number of cells that must fit in that space (`MapData.HEIGHT`), and add 1 to account for rounding errors that might otherwise leave a 1-pixel gap. This gives us a size value that becomes both the width and height of the grid cell. FYI, `itemView` is a field declared in the superclass, and here it refers to the `ConstraintLayout` object.

(FYI, this means that the declared width and height of the `ConstraintLayout`, in the layout file, are irrelevant.)

`list_selection.xml` describes each element in the selector list. We need `ConstraintLayout` again, containing an `ImageView` and a `TextView`. Since we're dealing with more concrete UI elements, it should be easy enough to use the drag-and-drop editor.

The `ConstraintLayout` this time should have its size determined by its child elements' size (not the other way around):

```
<android.support.constraint.ConstraintLayout
    ...
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    ...>
```

The `ImageView` can have an actual fixed size (because, when it comes right down to it, there's nothing else to base the size on):

```
<ImageView
    ...
    android:layout_width="48dp"
    android:layout_height="48dp"
    ... />
```

(48dp is the value used in the screenshot.)

Note that each constraint can (and in this case probably should) be assigned a margin, by default 8dp – a small gap to ensure that the UI elements are visually separated. In the XML it looks like this:

```
<TextView
    ...
    android:layout_marginLeft="8dp"
    app:layout_constraintLeft_toLeftOf="parent"
    ... />
```

Controller Logic

You can refer to the lecture notes to determine how to implement activites, fragments, and the adapters and view holders needed for `RecyclerView`. However, there are some additional points:

- Setting up a grid:

  The map's `RecyclerView` uses a grid layout:

```
recycView = (RecyclerView) view.findViewById(R.id.mapRecyclerView);
recycView.setLayoutManager(new GridLayoutManager(
    getActivity(),
    MapData.HEIGHT,
    GridLayoutManager.HORIZONTAL,
    false));
```

- Translating rows/columns to positions:

  As discussed in the lecture notes, `RecyclerView` identifies each element by a single "position" integer. How, then, does it deal with grids?

  `GridLayoutManager` has a mapping between rows/columns and positions. When operating horizontally (as in our case), the mapping is "column-major order", which basically means this:

```
int row = position % MapData.HEIGHT;
int col = position / MapData.HEIGHT;
```

  That is, position numbers 0 to $H - 1$ (where H is the map height) map to all the cells in the first column, from top to bottom. Then, positions H to $2H - 1$ map to all the cells in the second column, and so on. (FYI, if we were working vertically, the mapping would instead be "row-major order", and we'd use the width rather than the height.)

  You will need this in order to implement the map's adapter.

- Event handling:

  You'll also need view holders, of course, and this is where the event handling will take place (because the user is going to be tapping/clicking on grid cells and list elements).

  Any UI element can receive a tap/click event, not just buttons:

```
View v = ...; // Any view
v.setOnClickListener(...);
```

  However, when this happens, you'll realise that you're stuck – the view holder won't be able to do anything – *unless* it knows which `MapElement` or `Structure` it's currently representing. (Remember, by the nature of `RecycleView`, view holders get *recycled*, and may represent many different things during their lifetime.)

  So, you need to make it remember. When you implement the "`bind()`" method (or whatever you call it), as well as actually updating the UI, have it take in and store the model object in a field for later reference.

- Communicating between fragments:

  The map and selector are handled by different fragments, but (unlike activities) fragments can easily obtain references to one another and simply call each others' methods.

  I'd recommend the following:

  - Give the selector fragment a field to store the current selected structure, and an accessor to retrieve it. Now its job is done!
  - Give the map a field to reference the selector fragment itself, and a mutator to set it, and call it from the activity (which of course knows about both fragments).

(Note: because fragments have to be automatically re-created, they need zero-argument constructors, so unfortunately you can't use the constructor to pass anything to the fragment.)

- Have the map fragment call the selector fragment to retrieve the selected structure when needed.

- Refreshing the map:

  You need to notify the `RecyclerView` when you make a change to the data that it's displaying. (It can't magically sense changes!)

  To do this (from within a view holder):

```
adapter.notifyItemChanged(getAdapterPosition());
```

  `ViewHolder.getAdapterPosition()` returns the current position of the view holder. There are actually several `notify...()` methods in `Adapter`, depending on what kind of change you've just made to the data. In this case, we've simply changed an item in-place.

Other Considerations

- There are stylistic aspects to this app that could be improved. We'd probably like to highlight the currently-selected structure, for instance. You can certainly play around with the theme too.
- We'd probably like to actually create a drag-and-drop effect, rather than (or in addition to) the tap/click arrangement. If you'd like to experiment with this, see the following: https://developer.android.com/guide/topics/ui/drag-drop.
- We haven't given the user a way to *remove* a structure, but this could be easily enough done.
- We haven't done anything yet with the `MapData.regenerate()` method, but of course you could implement a button to invoke this. If you do so, you'll need to notify the map that everything has changed:

```
adapter.notifyItemRangeChanged(0, MapData.WIDTH * MapData.HEIGHT);
```

- We might like to allow the user to customise the list of possible structures. To do this, we could (for instance) have a second activity that re-uses the same selector fragment, but provides buttons to add new structures, and edit or delete existing ones.

End of Worksheet