Worksheet 2: Adaptive UIs and Activities

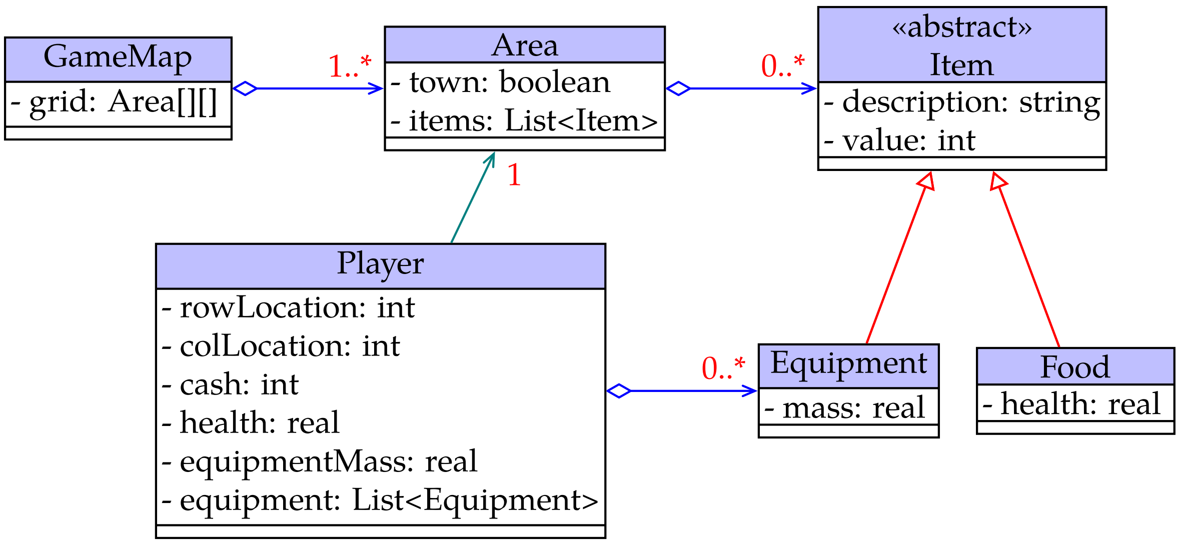
Updated: 13th August, 2018

In this worksheet, you'll write a simple Android-based game. It will use a grid-based map, where the player starts off in a given grid square ("area"), and can move north, south, east or west to other areas. The player can also buy, sell, pick up or drop certain items, and the ultimate objective is to find a particular combination of three items: (1) the jade monkey, (2) the roadmap, and (3) the ice scraper [1].

Fair warning: we will revisit and build on this scenario in future practical worksheets.

1 Data

All initial data can be hard-coded (as a set of constants). However, you will have to create some simple classes for holding it, as follows:



In other words:

- `GameMap` contains a 2D array, where each element is an `Area`.
- Each `Area` object either represents a town, or a wilderness area.
- Within each `Area` contains a set of zero or more `Items` (which can change, depending on what the player does).
- Each `Item` has a description (i.e. what it is), a value (price). It can either be `Food`, which increases the player's health, or a piece of `Equipment` that the player carries.
- The `Player`, at any given point in time, is at a particular area, has a certain amount of cash, has a health rating (0–100), and is carrying a certain amount of equipment.

[1]The significance of these are left as an exercise for the reader.

There are some slightly different ways to represent this arrangement. For instance, the player's equipment mass [2] could be calculated on-the-fly from the `Equipment` objects, rather than being stored and updated. You're free to make such tweaks as you see fit, as long as the result is functionally equivalent.

2 Structure and Logic

The game must consist of three activities. In all three activities, there should be a "status bar" at the bottom that shows the values of `cash`, `health` and `equipmentMass`.

In the case of the navigation activity (below), the status bar should *also* contain:

- A "Restart" button that restarts the game, resetting all data to initial values.
- A message indicating whether the player has won or lost, if either of those have happened.

2.1 Navigation Activity

In the main "navigation" activity, we show where the player is on the grid; i.e. which `Area` object we're at. (We only show the current area / grid square, not the whole grid at once.)

The game must show four buttons around the edge of the screen: "North" (at the top), "South" (at the bottom), "East" (on the right), and "West" (on the left). The player can use these buttons to move from one area to an adjacent another. However, the player cannot move off the edge of the map.

Travelling one square reduces the player's health, as follows:

```
health = Math.max(0.0, health - 5.0 - (equipmentMass / 2.0))
```

If the player's health reaches zero, the game ends.

The centre of the screen shows three things: (1) the description for the current area, (2) the string "Town" if `town == true`, or "Wilderness" otherwise, and (3) a button labelled "Options". When the options button is pressed, one of the other two activites must be started.

2.2 Market Activity

If the current area is a town, the options button will start the market activity, showing all the items that might be bought or sold. I haven't shown you how to properly implement a list in the UI, so for now you can either:

- Hard code a fixed set of (say) ten `TextViews` and corresponding `Buttons`, OR

[2] I could say "weight" and you might more easily understand what I mean, but that would be technically incorrect, and no self-respecting physicist would talk to me again.

- Show one item at a time, and have "Next" and "Previous" buttons to move between items.

You will also need a "Leave" button to return to the navigation activity.

Meanwhile, the actual buying and selling works as follows:

Buying items: The items available to be bought are those contained in the relevant `Area`. As you would expect, purchasing an item can only be done if the player has enough `cash`, and will subtract the item's value from the `cash` amount, and remove it from the `Area`.

If the `Item` is an instance of `Food`, the player will eat it, and `health` will increase by the specified amount (up to a maximum of 100.0). Food could also be poisonous, and decrease the player's health. They could lose the game at this point if their health drops to zero.

If the `Item` is an instance of `Equipment`, the player will add it to their inventory, and `equipmentMass` will increase by the specified amount (with no specific upper limit). Also, if the item being purchased is the last of the three being sought, the player has won the game. The game should return to the navigation activity, which should display a victory message in the status bar.

Selling equipment: The player may also sell any equipment in their inventory. Selling an equipment item is always possible. However, the player will only receive 75% of the item's value. (The player cannot sell food, as that never gets added to the player's inventory in the first place.)

Selling an item will also cause it to be added to the relevant `Area` object, and will reduce the equipment mass being carried by the player.

2.3 Wilderness Activity

If the current area is not a town, the options button will instead start the wilderness activity. This will actually work much like the market, except that there is no cash involved. That is, the player can simply pick up any items present, and/or drop any from their inventory.

Note: You may be able to use inheritance (or other OO design methods) to avoid code duplication.

3 Android UI Layouts

Use `ConstraintLayout` for the navigation activity's UI, and `LinearLayout` for the market and wilderness activities. You can use the drag-and-drop interface in Android Studio for both. *However*, you must be able to identify the relevant parts of the XML that cause everything to appear in the place that it does.

You will also need to arrange for all three activities to be rotatable. In portait mode, the status bar should appear at the bottom. In landscape mode, it should instead appear on the right.

4 How to Proceed?

You now have all the information you need to write the game. However, if you need a head start, I'd advise tackling the problem as follows:

(a) Write the model classes first: `Player`, `Area`, etc. This is basic OOPD/DSA class design.

> **Note:** It shouldn't be too hard to find the option to create a new Java class. While Android Studio does a lot of Android-specific things, it still also helps you write ordinary Java code.

(b) Hard code some map data inside `GameMap`, by creating instances of `Area` and assigning them to 2D array elements. Limit your map size, so you don't have to spend too long here. Don't worry about the `Items` yet.

(c) Implement the navigation activity. This will involve creating a `Player` and a `GameMap` (and associated `Area` objects).

You will need callbacks for each of the five buttons, and you will need to update the various text fields as the player moves around and loses health.

(d) Implement the restart button. (Don't worry about winning/losing yet.)

(e) Hard code some food and equipment items. Add them to various places on the map, including both towns and wilderness areas, and perhaps also to the player's initial inventory.

(f) Decide how to represent items, and the player's data, using `Intent` "extras". Unfortunately you cannot simply pass/return instances of your own classes to/from another activity, so you need to find a way to bundle the information into arrays, strings and primitives, each identified by a string "key".

> **Note:** See developer.android.com/reference/android/content/Intent. In particular, look at all the overloaded `putExtra()` and `get...Extra()` methods to get a sense of what's possible.
>
> Actually, a more elegant solution would be to make your classes implement the `Parcelable` interface, but that may be a steeper learning curve. You can use your own discretion here.

(g) Implement the market activity. This will need to return a result, because buying and selling things will have an impact on what happens back in the navigation activity.

You will need to:

- Reconstruct the area's items, the player's inventory, and the player's cash, health, and equipment mass from the `Intent` object.
- Use this information to populate the UI fields (i.e. show what's available to be bought and sold).
- Implement event handlers to handle the actual buying and selling.
- When the leave button is pressed, put the updated item/player information back in the result `Intent` object.

(You can implement another restart button here too, but that's optional, and slightly more complicated than the first one.)

(h) Implement the wilderness activity. You can just copy and paste your market activity code at first, but you should then find a way to eliminate the code duplication.

> **Note:** While you may not be used to the idea of activities, remember that standard Java principles still apply. You can create an *abstract* activity class to represent the commonalities among several different activities.

(i) Implement the win and lose conditions.

(j) Handle UI rotation. This will involve:

- Creating the alternate layouts, and saving them in the appropriate directories.
- Preserving each activity's data when it is destroyed and re-created, by the use of `Bundle` objects. This is very closely related to what you've already done to send/receive information to/from the market and wilderness activities. In those cases, you may even be able to re-use some of your existing code.

End of Worksheet