Advanced Programming

Ricard Solé Casas

February 11, 2018

Foreword

The source code for this report and app can be found on Github[1]. A production version of the application can be installed from Google's Play Store at http://bit.ly/ada-zooify. The minimum Android version required is 5.0 (Lollipop).

Some of the decisions taken in building this application will appear to not follow the suggested guidelines. A detailed explanation is provided in the *design choices* chapter.

Declaration

[1]https://github.com/rcsole/zoo-coursework

Contents

[2]https://android.com

[3]https://www.wikiwand.com/en/JavaFX

Chapter 1

Design Choices

Some of the choices I have made when building this project deviate a little from the initially suggested approach and can therefore benefit from some extra explanation.

1.1 Android[1], not JavaFX[2]

The design of this assignment seems to be testing more of UI development than it does OOP and other concepts covered in the *Advanced Programming* module. While one lecture did brie y touch on what *JavaFX* is, and how it might be and improvement on Java's Swing[3] the lecture was not nearly in-depth enough to justify using a framework that has been put on maintenance mode[4].

Instead, the more sensible choices seemed to use a web-framework with a UI built on HTML, CSS, and JavaScript, like Spark[5]; or the most popular mobile platform's SDK, *Android*. I would argue an even better choice for the given requirements (building a UI) would have been a 100% TypeScript[6] (or plain *JavaScript*) application using the browser's LevelDB[7], LocalStorage[8], or Firebase[9] as a persistence layer but one of the requirements commanded *Java* as the main language so they were not valid options. My most recent professional experience in *Java* included building *Android* applications so that was, from my point view, the most sensible choice given all the previously stated arguments was to use the *Android Platform*:

1. Existing knowledge
2. Industry standards
3. Assignment requirements

1.2 Pens

The assignment's specification was vague. it was specially so when describing what a *pen* is and its characteristics. I will unpack the challenges presented in the specification and explain my interpretation, in an attempt to get the reader of this report and myself on the same page:

A single pen can only contain animals of the same type.

[1]https://android.com
[2]https://www.wikiwand.com/en/JavaFX
[3]https://www.wikiwand.com/en/Swing_(Java)
[4]Last version is JavaFX 8, released nearly 4 years ago in March of 2014.
[5]http://sparkjava.com/
[6]https://www.typescriptlang.org/
[7]http://leveldb.org/
[8]https://html.spec.whatwg.org/multipage/webstorage.html
[9]https://firebase.google.com/

It's unclear what the specification refers to here when it comes to *type*. An animal type could be one of the provided *environments* (water, dry, hybrid, and air), or it could also mean a species (Sloth, Cat, Dog, etc.). For this implementation the assumption made is that type refers to *environment*, not *species*. Using the *species* as the type would be a trivial change with the existing implementation.

Each pen has a length, width, and temperature. Aquariums and aviaries also have a height.

There are two things worth noting in this particular part of the specification:

1. There is no need to store the length, the width, or the height. If a pen is dry its are will be land in m^2, if it's an aviary it'll be m^3 in air space, and in water for aquariums. If it's hybrid, to accommodate, for example, a Hippo, the pen will have two measurements, land in m^2, and water in m^3.
2. The temperature is irrelevant, at least with the examples provided in the spec. The person writing the spec did not specify why or how the temperature should be used. No animals need specific temperature, making that data useless. Were this requirement to change, storing temperature data would be a trivial change with this implementation.

Pens including water have a water volume and can be either salt or fresh water.

As with to the temperature data whether it is fresh or salt water is irrelevant. Examples only specify water, dry, petting, air, or part-dry, party-water. Were this requirement changed, storing water type data would be a trivial change with this implementation.

1.3 Species class as opposed to multiple Animal classes

The assignment presented looks like a more lengthy example of the canonical concrete `Dog` class extends from `abstract Animal` class. The difference being the need for a UI and some other entities, like `Keeper`, and `Pen`. If that were the case it would indicate that the coursework designer expected or suggested we create one `class` for each sample animal provided.

While that approach works, I believe it would be severely defficient in a real-world application when managing a zoo. The most obvious short-coming of the aforementioned architecture is that any new species registered by the zoo would require actual source code change. Instead I propose this: treat each animal entity as having a relationship to a *species* entity. Akin to how in the game of *Pokémon* one would have a *Pokédex* with all known *Pokémon types* (*species*) and a separate compartment with all the actual Pokémon. One wouldn't create a new `class` for every new *Pokémon type* that appeared. Instead, one would treat each *species* as a simply data. This would allow new discoveries about the *species* to be changed as needed from a UI and without changing the source code itself.

1.4 Keepers don't have a type

The module specification makes each of the initial Keepers be responsible for a given type of pen. I have chosen not to do that. It seems to be a silly choice. To change that it's a trivial modification.

Chapter 2

Overview

In this chapter I will attempt to provide a medium-level overview of the application architecture. I will also mention the few libraries that are used on top of the already existing *Android Platform* framework.

2.1 Libraries

2.1.1 Retrofit/OkHttp

> *Type-safe HTTP client for Android and Java by Square, Inc.*
>
> https://github.com/square/retrofit

Square's retrofit is built on top of another Square library: OkHttp[1]. It comes with a set of Java annotations that make it straight forward and safe to declare a remote service in Java and Android. Here it is used to model the OpenWeatherMap API data.

2.1.2 Lightweight-Stream-API

> *Stream API from Java 8 rewritten on iterators for Java 7 and below.*
>
> https://github.com/aNNiMON/Lightweight-Stream-API

To allow for backward compatibility on older Android devices (back to Lollipop in this application) and still be able to use the nifty Java Stream API[2] I have included a library that backports the functionality to Android friendly idioms.

2.1.3 Gson

> *Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of.*
>
> https://github.com/google/gson

It is widely known that the default *org.json* is a terrible implementation of a JSON parsing library. It is also widely known the Gson and Jackson are great implementations solving the same problem. Due to my familiarity with it I have chosen the former.

I use it to deserialize the Weather data from the API as well as load the initial seed data from a JSON file.

[1] http://square.github.io/okhttp/

[2] https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

2.1.4 Guava

> *Guava is a set of core libraries that includes new collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection, string processing, and much more!*
>
> https://github.com/google/guava

I don't think Guava needs much explanation due to its widespread use in the industry. In the particular case of this application it is used to have better futures for asynchronous and non-blocking operations, as well as immutable data structures.

2.1.5 Dagger

> *Dagger 2 is a compile-time evolution approach to dependency injection. Taking the approach started in Dagger 1.x to its ultimate conclusion, Dagger 2.x eliminates all reflection, and improves code clarity by removing the traditional ObjectGraph/Injector in favor of user-specified @Component interfaces.*
>
> https://github.com/google/dagger

Like Guava, Dagger 2 is a very common library to use dependency injection. Even more so when it comes to Android applications, due to its performance. *Dependency Injection* solves many problems and is out of the scope of explanation in this report.

2.2 Data

2.2.1 Local entities

2.2.2 Local services

2.2.3 Remote services

2.3 UI

2.3.1 Base classes

2.3.2 Main

2.3.3 Create

Chapter 3

Critical Evaluation

1. Use firebase for the persistance layer
2. Auto-update weather, display timestamp
3. Unit tests
4. Provide sensible empty states

Chapter 4

Test Plan

- The program has three main screens:
    1. A list of all animals, displaying key information about each animal.
    2. A list of all pens, displaying key information about each pen.
    3. A list of staff, displaying key information about each staff member.
- The user must be able to add new animals
    - Add new animal
- Entering all the information regarding animal requirements.
    - Add new species
- The user must be able to add new pens, entering all the information about the pen.
    - Add new pens
- The user must be able to assign animals to pens.
    - Assign animal to a pen.
- The user must be able to assign staff to pens.
    - Assign a staff member to a pen.
- If a pen is full or otherwise unable to accommodate the animal, the user should see an error message explaining why.
    - The UI only displays pens that can accomodate the animal.
- If a staff member is assigned to an unsuitable pen, the user should see an error message explaining why.
    - Decided not to implement this requirement.
- Any animals that have not been assigned a pen should be indicated clearly.
    - Navigate to animal list with unassigned animals.
- Any pens that have not had staff assigned should generate an alert of some kind.
    - Navigate to pen list with unassigned animals.
- The program must display the current weather, using data queried from https://openweathermap.org/api
    - Highlight weather area.
- The user must be able to refresh the weather data. This should not block the UI whilst the request is made.
    - Refresh weather data, showing notification.
- Automatic mode, which automatically tries to allocate animals to the available pens without input from the user/Automatic mode, which automatically tries to allocate staff to the available pens without input from the user.
    - Create animals and pens, run auto-allocator.

Chapter 5

Appendix A: UML Diagram