Project : Designing a Marketable android Application

Eugenia Almandoz

Paul Ngouchet

Reet Chowdhary

Cameron Fife

EC 327

Douglas Densmore

**Habit Builder Documentation**

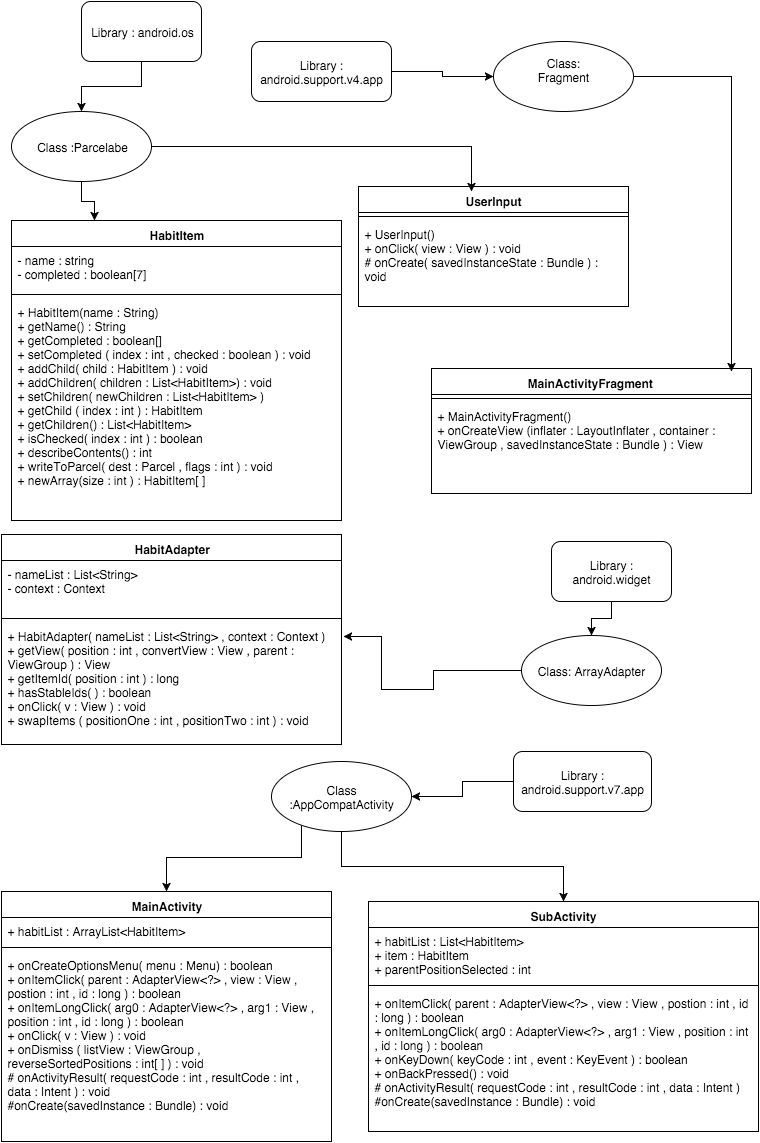
As mentioned in the PowerPoint, the main goal of our app was to provide users with a way to keep track of their daily success based on how many times they did a certain habit/activity. Our first task was simply to get activities and the checkboxes to show up on the screen. This was the most important piece of our app considering this was the main feature. We next wanted to implement list views, in order to get our habit items to scroll up and down in case the user decided to input more HabitItem’s than you were able to see on the screen at once. This was also important due to the fact that we wanted to display a list of items. The next important step was to be able to click on a given HabitItem and create the same list view for each HabitItem. The goal was to implement the same features as that for the original list view. The last thing was to implement features to make it work as we wanted. This included implementing the add button which would allow the user to add a HabitItem of their choice. The next task was to create a pop up screen when the add button was clicked to allow the user to type what habit they wanted to keep track of, if they user changed their mind they could click cancel, else they would click ok and this would be added to that list view. Considering we allowed users to add items we also decided to implement a way to delete a given HabitItem. We did this by swiping items to the right and the item would disappear. Lastly, one of the most important parts of this app was to save our data to a local file in order for the user to be able to access their information once again when the app was opened once again.

**Front End**

* We used list views to display individual habit items.
* We used a library from Niek Haarman called list view animations which allows us to easily animate the swapping of list items and the swipe to dismiss animation for the deletion of list items
* HabitItem view group is created in Habit\_Item.xml and is used to populate the list view.
* In HabitItem.xml we created a relative layout with a text view and 7 checkboxes. This is what each HabitItem displays on the screen in order to keep track of the number of times it is completed.
* Activity\_main contains the toolbar and the floating action button which allows us to add list items.
* Content\_main.xml contains a relative layout for the days of the week and Haarman’s dynamic list view.
* Activity\_Sub and Content\_Sub are the layouts for the sub activity that is called when a user clicks on a list item. They are essentially copies of the activity\_main and content\_main.

Back end

**UML DIAGRAM:**



**MainActivity:**

* Contains a master habit list that keeps track of all of the user’s created habits.
* We create a dynamic list view object and a habit adapter object; the habit adapter acts as a middle-man between the habit list and the dynamic list view by taking raw data from the habit list and populating the dynamic list view with the raw data
* We set an onItemLongClickListener to initiate the swapping of list view items, this way the user can re-arrange the list to suit their needs. For this we used the dynamic list view library.
* Also, using the same library we implemented a swipe to dismiss functionality in the list view such that the user can delete items just by swiping them to the right.
* We used the floating action button to call an alert dialog with an edit text field that the user types into to create a new item. We error checked their entries to ensure the text field is not empty.
* We set a click listener on the list view to check if an item is clicked, if so, we launch an intent to open up a sub activity which also contains a list view. However, the list view is populated with the children of the item that was clicked on.
* When a list item is clicked, we send a HabitItem to the new activity using the Intent.putExtra(Parcelable obj) function, made possible because HabitItem extends parcelable.
* When the user presses back from the new activity the activity sends back an updated version of the habit item which we update the master habit list with. This makes sure the master list is always updated with current information

**subActivity:**

* This class is a near copy of the MainActivity class.
* This class is launched when a list item is clicked on in the main activity.
* It is sent the HabitItem that was clicked on, and populates its list view with the children of that HabitItem
* One difference from the MainActivity is that when the back button is pressed, it sends an updated version of the habit item that it was called with back up to the calling activity.
* When one of its items is clicked on then it makes a call to the sub activity as well.

**HabitItem:**

* We created a class called HabitItem that implements Parcelable. HabitItem contains a name, a boolean array of size 7 which keeps track of the days of the week as well as a list of children habit items.
* The reason we implemented parcelable is so to be able to send. copies of the habit item across activities. This way when the user clicks on a checkbox or adds an item we can update the master habit list by sending the data across the activities.

**Habit Adapter:**

* Acts as a middle-man between the raw data of the Habit list and the front end interface of the list view
* The getView method gets a view at the position of the list item that it constructs from the raw data and returns it to the list view so it can put that view at the specified spot.
* It implements swappable which is used in the dynamic list view in order to re-arrange list items..

**Problems We Encountered:**

Overall we didn’t have too many problems but instead with the time constraint were unable to implement everything we planned to in our app. The only problem we encountered while working on this was how to store and read data. At first, we believed it was a much simpler task than it turned out to be. In order to complete this task we had to look into different data structures to be able to read and write the way we needed to. It became very complicated but it was a priority for our app to work the way it needed to, because without a method to save and write, our app would just lose all the data needed for our app to work again when closed and re-opened. Other than this, our only problem was not having time to make this app as useful and helpful as possible. Some of the things we meant to implement that we weren’t able to was the following:

* Including a calendar in our app to describe our activities throughout a month instead of a week.
* Implementing graphs to display a visual of our success.
* Having user’s input their goals at the beginning of each week to show whether they reached this goal or not.
* We also wanted to add notifications/alarms to remind users to update their information daily.
* Saving to the cloud instead of saving data locally.
* Display what week the user is currently in.

Our Save Format

Our Save Format was one of my personal favorite parts of the project because it implements several data structures and we used our own intuition.

We used a tree, a queue, and a stack. Before I explain their purpose, allow me to explain the problem we ran into that made saving complex.

**The Fundamental Problem:** Normally when we save a simple list of objects, it’s very easy to do - just create a for loop and write out the data for each object. However, we are saving a tree, which means that our objects point to other objects (Similar to the problem in PA4). To get around this, we first instantiate the objects that we are certain don’t point to other objects (the leaves), then we instantiate the objects that only point to the leaves. And then we instantiate the objects that point to the objects that point to the leaves. And this cycle continues until the last object created is the head node.

So if you look at the file format, you’ll notice that the head is always written at the bottom of the text file whereas the leaves are written at the top. And we did that by using a queue and a stack. We basically performed a **Breadth-First Search**. For each item in our search algorithm, we added it to both a queue and a stack. The queue was used to keep track of nodes still unexplored. The stack was used to actually write the file. Why? Because a breadth first search results in us reading from Top-to-bottom and from left-to-right. If I had a tree that looked like this (Pardon poor text drawings)

A

B C

D E F G

H I

J

A breadth first search would neatly organize the tree from right to left, top to bottom.

So A, B, C, D, E, F, G, H, I, J

So by the time I finished adding everything into the stack (Also note that each stack element is added from the back), It would also look like

A, B, C, D, E, F, G, H, I, J

And since Stacks follow LIFO, (Last-in, First-out), using a simple for-loop while popping allows us to print the loop from leaves to head. This ensures that no node in the tree is pointing to a node that doesn’t already exist.

While this was my favorite part of our project, we never got to fully implement it in our assignment. It was the last thing we were working on before it was submitted. Written in pure java, we have a working version. However, we could get FileStream to work on Android. That was the first problem. The second problem is that our front page, isn’t implemented using a single Habit Item, but rather a list of habit items. In other words, we could have more than one header node.

Basically, we could have definitely fixed this, but we just didn’t have enough time (even like a few more hours would’ve sufficed). You can download our original, working algorithm on GitHub though.