Jacob Taggart
Week 1 Day 4 Research

Week 1 Day 4 Research

1. Define the following terms: View, ViewGroup, View Hierarchy.

A: In the most basic of terms, a View is the display on the screen being used. View is also the base class for widgets. Thus, Views are the building blocks for User Interface components such as buttons, radio buttons, text fields, etc. ViewGroup, on the other hand, is the specific container which holds a View(s) or even other ViewGroups. ViewGroups must have the layout of the container specified via developer-defined properties of these ViewGroups. A view hierarchy is a tree structure which starts with the entire window the user is looking at on the top of the structure with branches that include other individual Views and/or ViewGroups. The branching itself is wholly based on the parent/child relationship of ViewGroups and any contained Views.

2. Explain in detail how the following layouts render, a what unique items each has that must be implemented: Constraint, Linear, Coordinator, Grid and Relative?

A: Constraint layouts only have one rendering process as there are no nested ViewGroups. Thus, the ViewHierarchy is completely streamlined and doesn't require more than the one rendering. Constraint layouts are based on developer specifications (toStartOf, toEndOf, toTopOf, toBottomOf) in relation to the parent View.

Linear layouts are ViewGroups that align child Views/ViewGroups in either a horizontal or vertical direction. Linear layout members always require a layout_width and layout_height. For a linear layout list, the keyword "orientation" being set to vertical or horizontal decides how the layout list looks in conjunction. To set child ViewGroup(s) in sequence in a manner out of step with the overall orientation, a nested ViewGroup is the way to achieve this. For example if there is a layout list with name, address, city, state and ZIP, but you want city and state side by side while the rest of the layout list continues in the direction of the list orientation (vertically), a nested ViewGroup is how you'd achieve this in linear layouts.

CoordinatorLayouts are fairly different from the others in that it is considered a highly powered FrameLayout. The developer should use CoordinatorLayouts in two situations: to act as the highest level of design in the overall layout and also to act as a ViewGroup for a specific interaction with one or multiple child ViewGroup(s). CoordinatorLayouts allow the developer to interact with child classes via the "Behavior" keyword. The underlying

CoordinatorLayout class has plenty of tools in the toolbelt already which makes it special compared to the rest of the common layouts. Generally this type of layout can carry out the most complex customization out of the others mentioned in this question. The developer can create his/her own custom constraint behaviors and has the freedom to finely tune the child ViewGroup(s) via the use of child anchors and the ability to change layout parameters.

Grid layouts are used to place its children into a set of horizontal and vertical lines. More specifically, the children are placed in the cells that are made by the intersection of these horizontal and vertical lines. Each grid line can also be called via its indices. If we are talking vertical lines, the vertical line on the very far left of a screen would have an index of 0 while the far right vertical line would have an index of one less than how many vertical lines total there are. Thus, one can place a View or ViewGroup into a grid layout by specifying the indices of the lines if one so wanted to. Children View/ViewGroups always occupy at least one or more connected cells via the use of Grid.Layout.LayoutParams#rowSpec and Grid.Layout.LayoutParams$columnSpec. When the developer sets the orientation, row count, and column count but does not use the Grid.Layout.LayoutParams call, the children of the Grid layout will populate into cells by default depending on the number of lines both ways and the orientation.

All layouts need some form of width and height specification whether it's based on relation to other objects, its own size, or a custom specification. Otherwise, it wouldn't be a 2-dimensional drawable which layouts generally are.

Relative layouts are very similar to constraint layouts in that the positioning of child elements are wholly based on parent ViewGroups' positions. However, the difference from a constraint layout is that even though relative layouts are designed to minimize nested ViewGroups like constraint layouts, to finely tune how far away a child ViewGroup may be from its parent and how well centered that child may be in relation to its parent, it oftentimes requires nested ViewGroups. This makes relative layout more useful for more basic UI windows while constraint layouts are perfect for detailed UI layouts.

3. What are Listeners?

Event listeners are exactly what they sound like. In Android, whenever an event happens - such as a button being clicked - the program is made aware of what has occurred due to the interaction with the View on the UI. Technically an Event Listener is actually an interface contained View's class which is why listener methods like onClick() are so easily called without defining much. The developer only needs to make sure to define the button object, for example, and make it equal to the ID of the button in the View file and then needs to call setOnClick() and pass the variable that represents the UI button. After doing this in activity_main.xml, now the java file can use the listener.

4. How does Java garbage collection work?
Firstly, Java garbage collection is the process of scanning heap memory to search for unused objects, variables, etc. that takes up unnecessary space. The process of clearing this garbage is a few-step process that begins with the phase of marking. Like what it sounds, this is when the system sifts through the heap memory looking for unused/redundant allocated memory. More specifically, it looks for unreferenced objects allocated in memory. This step can take a long time depending on the efficiency of the code and the amount of objects to scan through. The next step is known as normal deletion. Normal deletion deletes the unreferenced objects allocated in memory and leaves blank spaces between the referenced objects as opposed to the previously unreferenced objects. There is also deletion with compaction which is the same thing except after deleting the unreferenced objects, the memory spaces that would be blank memory space in normal deletion are now moved to the end of the memory heap meaning that all objects that are referenced are now allocated "shoulder to shoulder", so to speak, leaving no blanks in between allocation.

5. Explain the software development lifecycle.

The software development lifecycle is a process by which almost all software development projects follow. It includes six steps: the first step is planning (analysis), the second is design, third is development (design and development together are sometimes compacted as "creation") and testing/debugging, the fourth is implementation (putting the program to its intended use), fifth is documentation (mostly a business-minded step in order to keep close track of the process that was used in the development cycle as well as the creation of licenses, etc.), and sixth is evaluation (looking at the finished product and trying to find potential improvements).