|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04 MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS. 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. **TRANSFORM MOCK-UP TO MOBILE APP.** 4. INTEGRATE MOBILE APP WITH DATA SOURCE. 5. VERIFY SUCCESSFUL API INTEGRATION 6. VERIFY DEVELOPED MOBILE APP. 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/IS(5/15) | Page: 1 of 18 |

**TITLE**:

**NAVIGATION FLOW AND PAGE LIFE CYCLE**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of navigation flow and page life cycle.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of navigation flow and page life cycle.

# **NAVIGATION FLOW**

Navigation refers to the interactions that allow users to navigate across, into, and back out from the different pieces of content within your app. For example, Android Jetpack's Navigation component helps you implement navigation, from simple button clicks to more complex patterns, such as app bars and the navigation drawer. The Navigation component also ensures a consistent and predictable user experience by adhering to an established set of principles.

## Navigation component

The Navigation component consists of three key parts that are described below:

* **Navigation graph:** An XML resource that contains all navigation-related information in one centralized location. This includes all the individual content areas within your app, called destinations, as well as the possible paths that a user can take through your app.
* **NavHost:** An empty container that displays destinations from your navigation graph. The Navigation component contains a default NavHost implementation, NavHostFragment, that displays fragment destinations.
* **NavController:** An object that manages app navigation within a NavHost. The NavController orchestrates the swapping of destination content in the NavHost as users move throughout your app.

As you navigate through your app, you tell the NavController that you want to navigate either along a specific path in your navigation graph or directly to a specific destination. The NavController then shows the appropriate destination in the NavHost.

1. Creating navigation graph

Navigation occurs between your app's destinations—that is, anywhere in your app to which users can navigate. These destinations are connected via actions. A navigation graph is a resource file that contains all your destinations and actions. The graph represents all your app's navigation paths.

**Figure 1** shows a visual representation of a navigation graph for a sample app containing six destinations connected by five actions. Each destination is represented by a preview thumbnail, and connecting actions are represented by arrows that show how users can navigate from one destination to another.

* + 1. Destinations are the different content areas in your app.
    2. Actions are logical connections between your destinations that represent paths that users can take.

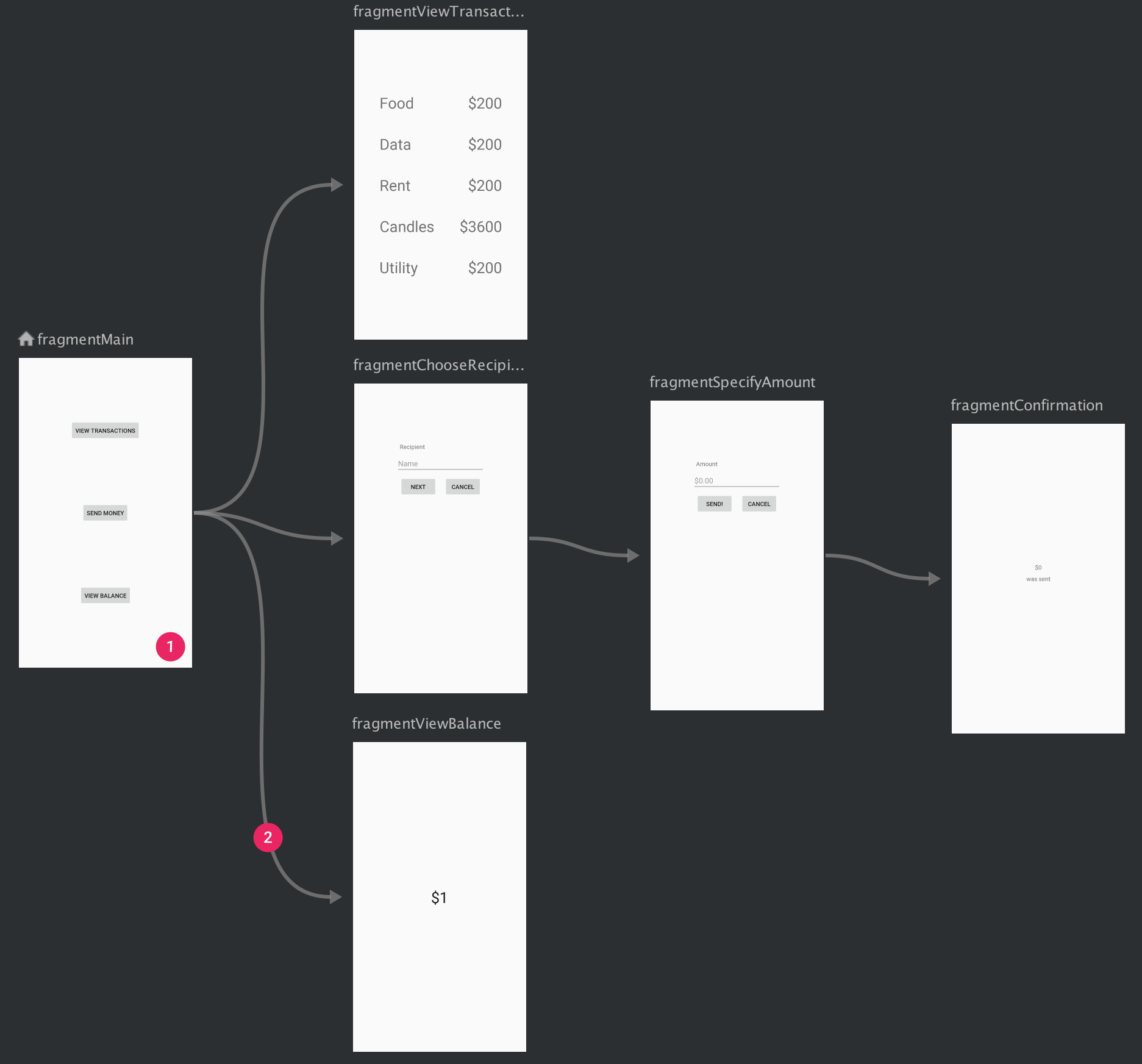


Figure 1: A Navigation Graph That Shows Previews Of Six Different Destinations That Are Connected Via Five Actions

1. Add a NavHost to an activity

One of the core parts of the Navigation component is the navigation host. The navigation host is an empty container where destinations are swapped in and out as a user navigates through your app. A navigation host must derive from NavHost. The Navigation component's default NavHost implementation, NavHostFragment, handles swapping fragment destinations.

1. NavController

NavController manages app navigation within a NavHost. Apps will generally obtain a controller directly from a host, or by using one of the utility methods on the Navigation class rather than create a controller directly.

Navigation flows and destinations are determined by the navigation graph owned by the controller. These graphs are typically inflated from an Android resource, but, like views, they can also be constructed or combined programmatically or for the case of dynamic navigation structure. For example, if the navigation structure of the application is determined by live data obtained' from a remote server.

The Navigation component provides several other benefits, including the following:

* Handling fragment transactions.
* Handling Up and Back actions correctly by default.
* Providing standardized resources for animations and transitions.
* Implementing and handling deep linking.
* Including Navigation UI patterns, such as navigation drawers and bottom navigation, with minimal additional work.
* Safe Args - a Gradle plugin that provides type safety when navigating and passing data between destinations.
* ViewModel support - you can scope a ViewModel to a navigation graph to share UI-related data between the graph's destinations.

## Principle of navigation

Navigation between different screens and apps is a core part of the user experience. The following principles set a baseline for a consistent and intuitive user experience across apps. The Navigation component is designed to implement these principles by default, ensuring that users can apply the same heuristics and patterns in navigation as they move between apps.

1. Fixed start destination

Every app you build has a fixed start destination. This is the first screen the user sees when they launch your app from the launcher. This destination is also the last screen the user sees when they return to the launcher after pressing the Back button. **Figure 2** shows Sunflower app as an example.

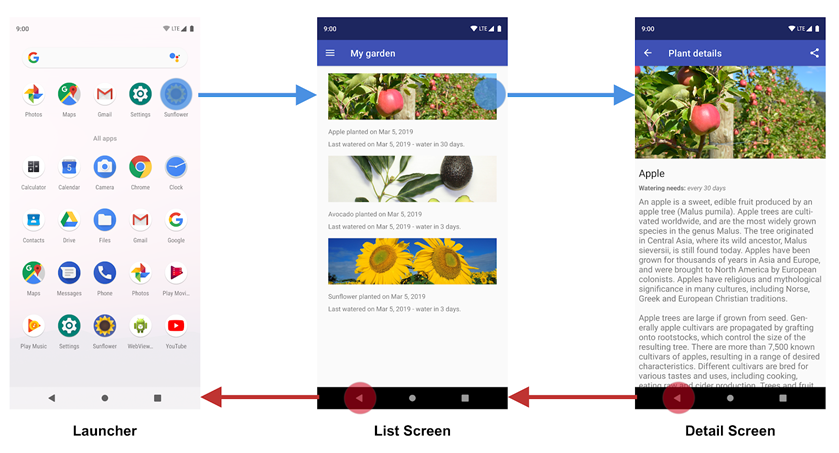


Figure 2: The List Screen is the Sunflower App’s Start Destination

1. Navigation state is represented as a stack of destinations

When launching the Sunflower app from the launcher, the first screen that a user sees is the List Screen, the list of plants in their garden. This is also the last screen they see before exiting the app. If they press the Back button from the list screen, they navigate back to the launcher.

When your app is first launched, a new task is created for the user, and the app displays its start destination. This becomes the base destination of what is known as the back stack and is the basis for your app’s navigation state. The top of the stack is the current screen, and the previous destinations in the stack represent the history of where you have been. The back stack always has the start destination of the app at the bottom of the stack.

Operations that change the back stack always operate on the top of the stack, either by pushing a new destination onto the top of the stack or popping the top-most destination off the stack. Navigating to a destination pushes that destination on top of the stack.

The Navigation component manages all your back-stack ordering for you, though you can also choose to manage the back stack yourself.

1. Up and Back are identical within your app's task

The Back button appears in the system navigation bar at the bottom of the screen and is used to navigate in reverse-chronological order through the history of screens the user has recently worked with. When you press the Back button, the current destination is popped off the top of the back stack, and you then navigate to the previous destination.

The Up button appears in the app bar at the top of the screen. Within your app's task, the Up and Back buttons behave identically.



Figure 3: The Up and Back Buttons

1. The Up button never exits your app

If a user is at the app's start destination, then the Up button does not appear, because the Up button never exits the app. The Back button, however, is shown and does exit the app. When your app is launched using a deep link on another app's task, Up transitions users back to your app’s task and through a simulated back stack and not to the app that triggered the deep link. The Back button, however, does take you back to the other app.

1. Deep linking simulates manual navigation

Whether deep linking or manually navigating to a specific destination, you can use the Up button to navigate through destinations back to the start destination.

When deep linking to a destination within your app’s task, any existing back stack for your app’s task is removed and replaced with the deep-linked back stack.

Using the Sunflower app again as an example, assume that the user had previously launched the app from the launcher screen and navigated to the detail screen for an apple. Looking at the recent screen would indicate that a task exists with the topmost screen being the detail screen for the Apple.

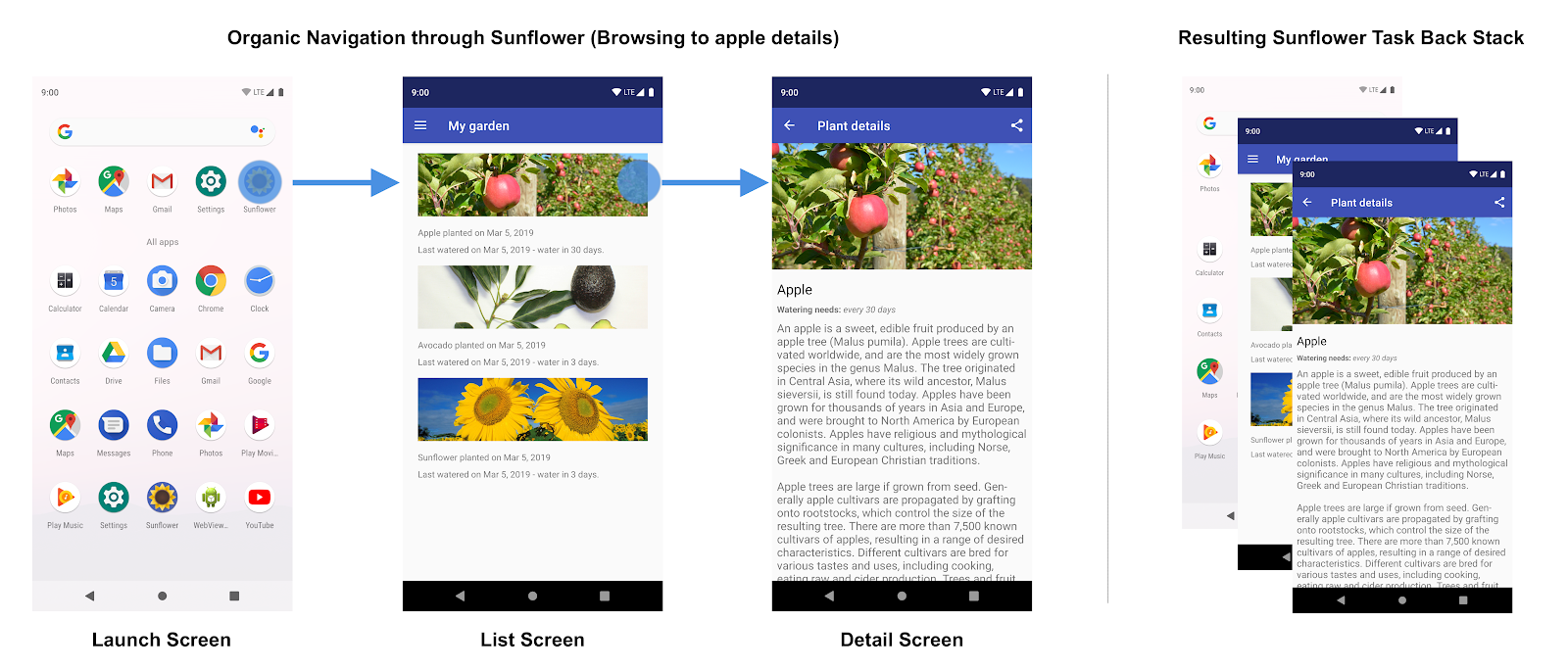


Figure 4: User Navigation Through The Sunflower App And The Resulting Back Stack

At this point, the user can tap the Home button to put the app in the background. Next, let’s say this app has a deep link feature that allows users to launch directly into a specific plant detail screen by name. Opening the app via this deep link completely replaces the current Sunflower back stack shown in figure 3 with a new back stack, as shown in figure 5:

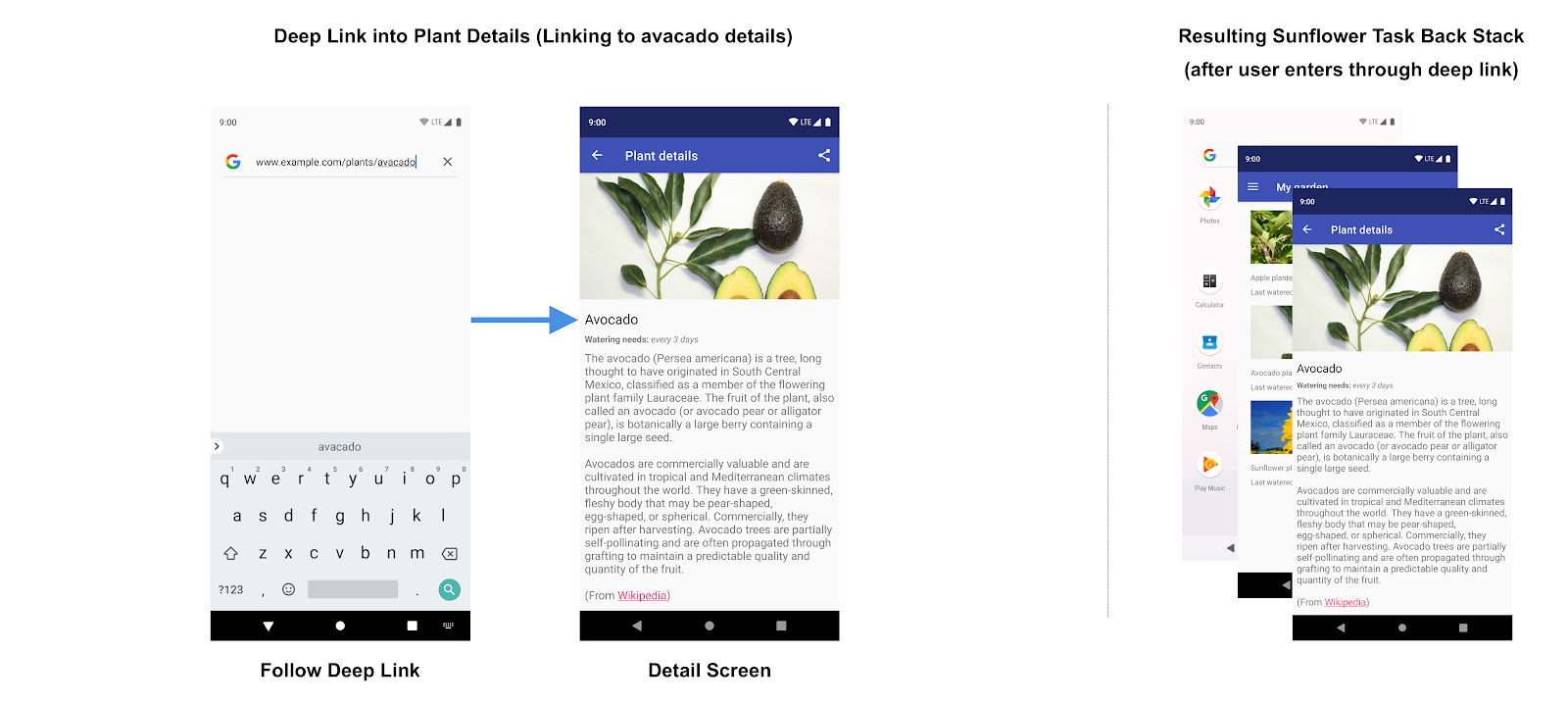


Figure 5: Following A Deep Link Replaces The Existing Back Stack For The Sunflower App

Notice that the Sunflower back stack is replaced by a synthetic back stack with the avocado detail screen at the top. The My Garden screen, which is the start destination, was also added to the back stack. This is important because the synthetic back stack must be realistic. It should match a back stack that could have been achieved by organically navigating through the app. The original Sunflower back stack is gone, including the app's knowledge that the user was on the Apple details screen before.

The Navigation component supports deep linking and recreates a realistic back stack for you when linking to any destination in your navigation graph.

# **PAGE LIFE CYCLE**

The concept of Page refers to the screen instances that are presented on the screen by the app. The term is varied from platform to platform, Activity in Android, and View in iOS. As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you are building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot. In other words, each callback allows you to perform specific work that is appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant. For example, good implementation of the lifecycle callbacks can help ensure that your app avoids:

* Crashing if the user receives a phone call or switches to another app while using your app.
* Consuming valuable system resources when the user is not actively using it.
* Losing the user's progress if they leave your app and return to it later.
* Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

## Activity-lifecycle concepts (Android)

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). The system invokes each of these callbacks as an activity enters a new state. Figure 6 presents a visual representation of this paradigm.

Diagram

Description automatically generated

Figure 6: A Simplified Illustration Of The Activity Lifecycle

As the user begins to leave the activity, the system calls methods to dismantle the activity. In some cases, this dismantlement is only partial; the activity still resides in memory (such as when the user switches to another app) and can still come back to the foreground. If the user returns to that activity, the activity resumes from where the user left off. With a few exceptions, apps are restricted from starting activities when running in the background.

The system’s likelihood of killing a given process - along with the activities in it - depends on the state of the activity at the time. Activity state and ejection from memory provides more information on the relationship between state and vulnerability to ejection.

Depending on the complexity of your activity, you probably do not need to implement all the lifecycle methods. However, it is important that you understand each one and implement those that ensure your app behaves the way users expect.

## Lifecycle callbacks

Over the course of its lifetime, an activity goes through several states. You use a series of callbacks to handle transitions between states. Table 1 shows the details of each callback.

Table 1: Callbacks And Its Details Explanation

|  |  |
| --- | --- |
| Callbacks | Description |
| onCreate() | You must implement this callback, which fires when the system first creates the activity. On activity creation, the activity enters the Created state. In the onCreate() method, you perform basic application startup logic that should happen only once for the entire life of the activity. For example, your implementation of onCreate() might bind data to lists, associate the activity with a ViewModel, and instantiate some class-scope variables.  This method receives the parameter savedInstanceState, which is a Bundle object containing the activity’s previously saved state. If the activity has never existed before, the value of the Bundle object is null. |
| onStart() | When the activity enters the Started state, the system invokes this callback. The onStart() call makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive. For example, this method is where the app initializes the code that maintains the UI. When the activity moves to the started state, any lifecycle-aware component tied to the activity’s lifecycle will receive the ON\_START event.  The onStart() method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state. Once this callback finishes, the activity enters the Resumed state, and the system invokes the onResume() method. |
| onResume() | When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the onResume() callback. This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user’s navigating to another activity, or the device screen’s turning off.  When the activity moves to the resumed state, any lifecycle-aware component tied to the activity’s lifecycle will receive the ON\_RESUME event. This is where the lifecycle components can enable any functionality that needs to run while the component is visible and, in the foreground, such as starting a camera preview. When an interruptive event occurs, the activity enters the Paused state, and the system invokes the onPause() callback.  If the activity returns to the Resumed state from the Paused state, the system once again calls onResume() method. For this reason, you should implement onResume() to initialize components that you release during onPause(), and perform any other initializations that must occur each time the activity enters the Resumed state. |
| onPause() | The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed); it indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode). Use the onPause() method to pause or adjust operations that should not continue (or should continue in moderation) while the Activity is in the Paused state, and that you expect to resume shortly. There are several reasons why an activity may enter this state. For example:   * Some event interrupts app execution, as described in the onResume() section. This is the most common case. * In Android 7.0 (API level 24) or higher, multiple apps run in multi-window mode. Because only one of the apps (windows) has focus at any time, the system pauses all of the other apps. * A new, semi-transparent activity (such as a dialog) opens. As long as the activity is still partially visible but not in focus, it remains paused.   When the activity moves to the paused state, any lifecycle-aware component tied to the activity’s lifecycle will receive the ON\_PAUSE event. This is where the lifecycle components can stop any functionality that does not need to run while the component is not in the foreground, such as stopping a camera preview.  You can also use the onPause() method to release system resources, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them. However, as mentioned above in the onResume() section, a Paused activity may still be fully visible if in multi-window mode. As such, you should consider using onStop() instead of onPause() to fully release or adjust UI-related resources and operations to better support multi-window mode. |
| onStop() | When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback. This may occur, for example, when a newly launched activity covers the entire screen. The system may also call onStop() when the activity has finished running, and is about to be terminated.  When the activity moves to the stopped state, any lifecycle-aware component tied to the activity’s lifecycle will receive the ON\_STOP event. This is where the lifecycle components can stop any functionality that does not need to run while the component is not visible on the screen.  In the onStop() method, the app should release or adjust resources that are not needed while the app is not visible to the user. For example, your app might pause animations or switch from fine-grained to coarse-grained location updates. Using onStop() instead of onPause() ensures that UI-related work continues, even when the user is viewing your activity in multi-window mode.  You should also use onStop() to perform relatively CPU-intensive shutdown operations. For example, if you can’t find a more opportune time to save information to a database, you might do so during onStop().  When your activity enters the Stopped state, the Activity object is kept resident in memory: It maintains all state and member information but is not attached to the window manager. When the activity resumes, the activity recalls this information. You do not need to re-initialize components that were created during any of the callback methods leading up to the Resumed state. The system also keeps track of the current state for each View object in the layout, so if the user entered text into an EditText widget, that content is retained so you do not need to save and restore it.  From the Stopped state, the activity either comes back to interact with the user, or the activity is finished running and goes away. If the activity comes back, the system invokes onRestart(). If the Activity is finished running, the system calls onDestroy(). |
| onDestroy() | onDestroy() is called before the activity is destroyed. The system invokes this callback either because:   1. the activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity), or 2. the system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)   When the activity moves to the destroyed state, any lifecycle-aware component tied to the activity’s lifecycle will receive the ON\_DESTROY event. This is where the lifecycle components can clean up anything it needs to before the Activity is destroyed.  Instead of putting logic in your Activity to determine why it is being destroyed you should use a ViewModel object to contain the relevant view data for your Activity. If the Activity is going to be recreated due to a configuration change the ViewModel does not have to do anything since it will be preserved and given to the next Activity instance. If the Activity is not going to be recreated then the ViewModel will have the onCleared() method called where it can clean up any data it needs to before being destroyed.  You can distinguish between these two scenarios with the isFinishing() method. If the activity is finishing, onDestroy() is the final lifecycle callback the activity receives. If onDestroy() is called as the result of a configuration change, the system immediately creates a new activity instance and then calls onCreate() on that new instance in the new configuration. The onDestroy() callback should release all resources that have not yet been released by earlier callbacks such as onStop(). |

**QUESTIONS:**

1. Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA. As it is a native category IDE, describes the (any five) features provided by Android studio.

**Answer:**

* 1. A flexible Gradle-based build system
  2. A fast and feature-rich emulator
  3. A unified environment where you can develop for all Android devices.
  4. Apply Changes to push code and resource changes to your running app without restarting your app.
  5. Code templates and GitHub integration to help you build common app features and import sample code.

**(5 marks)**

**REFERENCE:**

1. https://developer.android.com/guide/navigation/navigation-getting-started