



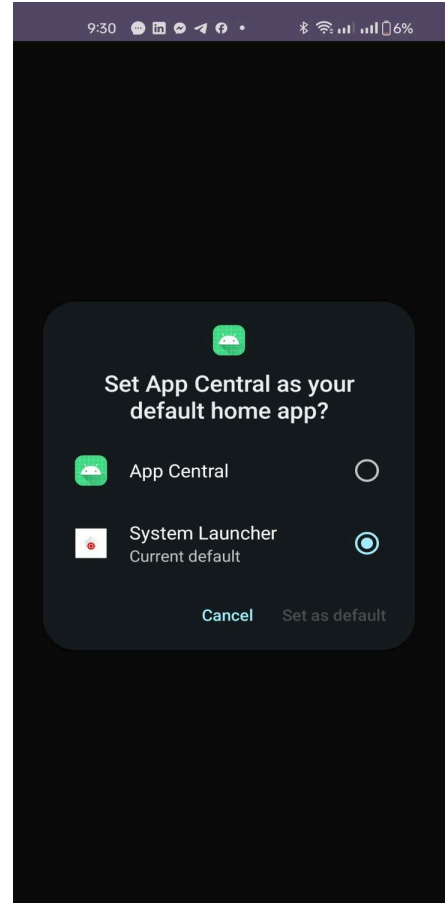
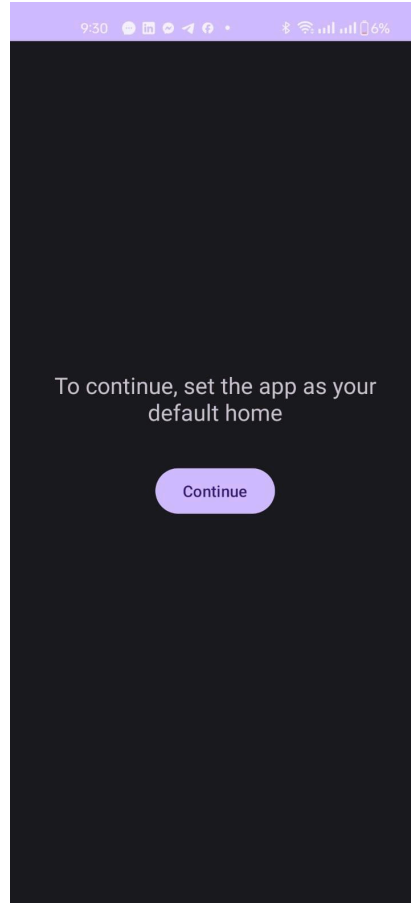
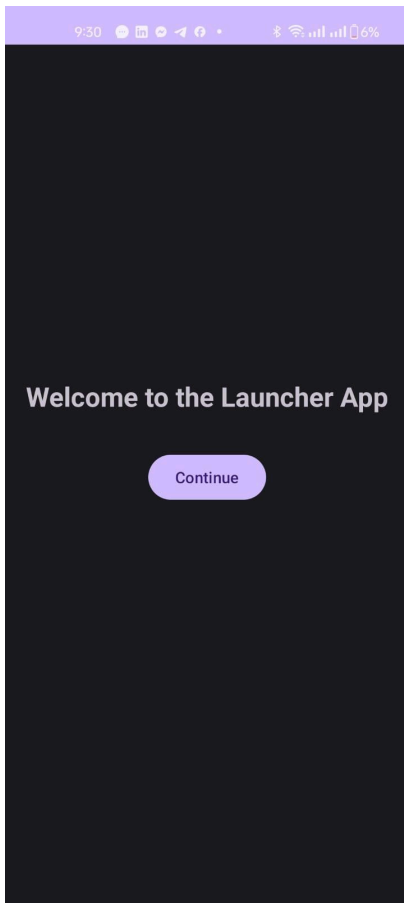
# James Nyakundi Take Home Test Solution

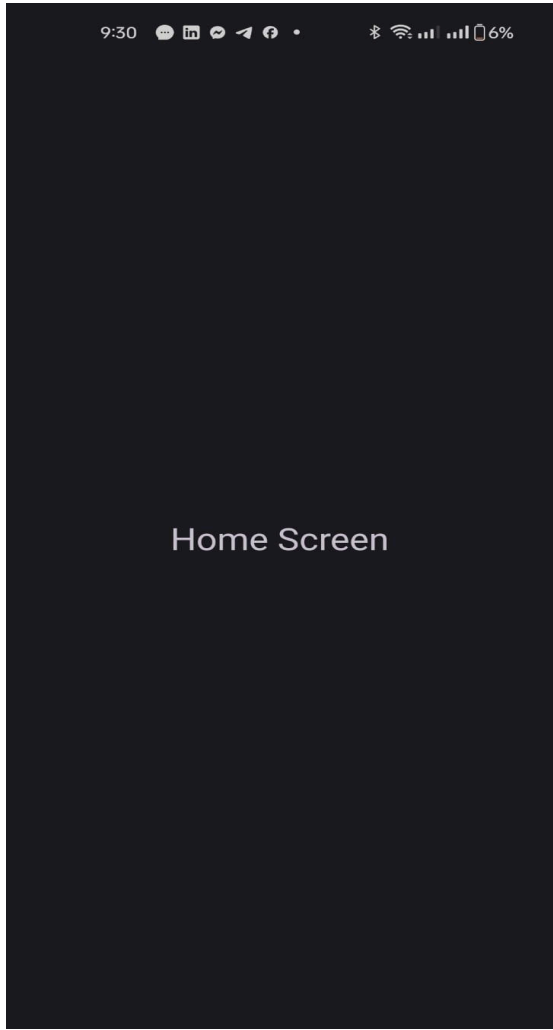
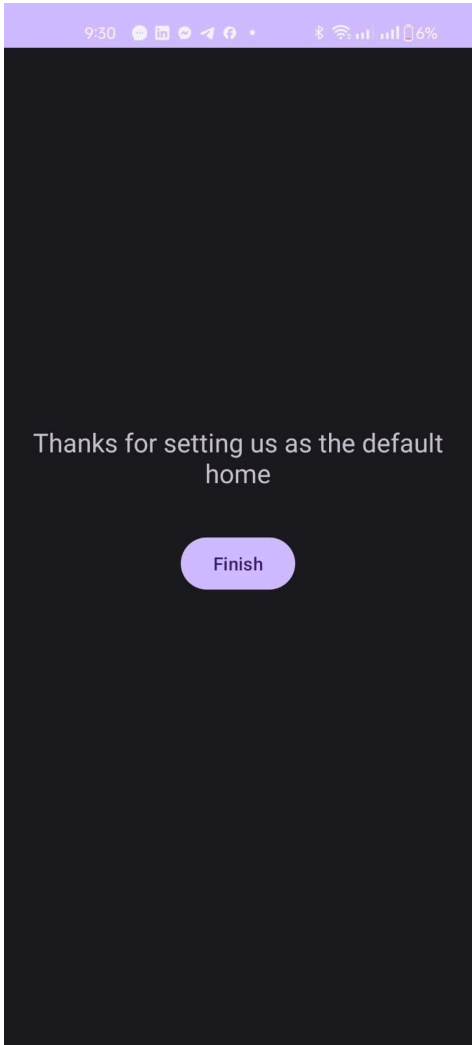
 **Email:** [jnyakush99@gmail.com](mailto:jnyakush99@gmail.com)

 **Phone:** +254 746 445 198

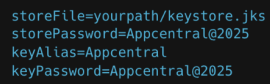
 **Location:** Nairobi, Kenya

## App screenshots





Add keystore configs on keystore.properties inside the directory.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays four lines of text in a light blue font, representing keystore configuration properties.

```
storeFile=yourpath/keystore.jks  
storePassword=Appcentral@2025  
keyAlias=Appcentral  
keyPassword=Appcentral@2025
```

## Timber Initialization in Application Class

The AppCentral class extends the Application class and is responsible for initializing the Timber logging framework across the app. By centralizing the setup in the application's lifecycle, it ensures that logging is consistently configured from the start. In debug builds, a custom DebugTree is planted, enhancing log readability by including line numbers with each tag—making it easier to trace issues during development. In production builds, the plan is to replace this with a CrashlyticsTree to route critical logs and errors to Firebase Crashlytics, while avoiding verbose debug output. This approach ensures a clear separation between development and production logging behaviors.

```
class AppCentral : Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
        initTimber()  
    }  
  
    private fun initTimber() {  
        if (BuildConfig.DEBUG) {  
            Timber.plant(object : Timber.DebugTree() {  
                override fun createStackElementTag(element: StackTraceElement): String {  
                    return super.createStackElementTag(element) + ":" + element.lineNumber  
                }  
            })  
        } else {  
            // Timber.plant(CrashlyticsTree())  
        }  
    }  
}
```

The project includes two main activities:

- **HomeActivity:** This activity uses `activity_home.xml` as its layout, which is based on a `ConstraintLayout` containing a single `TextView`.
- **OnboardingActivity:** This activity uses `activity_onboarding.xml` as its layout. The layout contains a `FrameLayout` that serves as a **fragment container** for managing the different onboarding steps (e.g., `Step1Fragment`, `Step2Fragment`, `Step3Fragment`).

## My Solution

Inside `HomeActivity`, I have a function named `checkOnboardingStatus` that accepts an optional `intentFlags` parameter. This function checks whether the onboarding process has been completed.

If onboarding **has not been completed**, it redirects the user to `OnboardingActivity`, optionally applying the provided intent flags.

```
/**
 * Checks if onboarding is completed and redirects to OnboardingActivity if not
 * @param intentFlags Optional flags to add to the intent
 */
private fun checkOnboardingStatus(intentFlags: Int = 0) {
    val sharedPrefs = getSharedPreferences("home_app_prefs", MODE_PRIVATE)
    val onboardingCompleted = sharedPrefs.getBoolean("onboarding_completed", false)
    val defaultLauncherRequested = sharedPrefs.getBoolean("default_launcher_requested", false)

    if (!onboardingCompleted) {
        Timber.tag(TAG).d("Onboarding not completed, redirecting")
        val onboardingIntent = Intent(this, OnboardingActivity::class.java)
        onboardingIntent.putExtra("FROM_DEFAULT_LAUNCHER_SETTING", defaultLauncherRequested)

        if (intentFlags != 0) {
            onboardingIntent.addFlags(intentFlags)
        }

        startActivity(onboardingIntent)
        return
    }
}
```

## HomeActivity Overview

In HomeActivity, I'm using **ViewBinding** to manage the layout efficiently. Inside the onCreate() method, I call the checkOnboardingStatus() function **without** any intent flags.

This setup ensures that when the app is launched and the onboarding has not been completed, the user is redirected to OnboardingActivity. When onboarding is completed (e.g., after choosing the app as the default launcher in step 2), the user is seamlessly taken to **step 3**, and only **after completing step 3**, they return to HomeActivity. This ensures step 3 is not skipped or bypassed.

## Handling Existing Instances with

### onNewIntent

Additionally, I override the onNewIntent() method in HomeActivity to handle scenarios where the activity is brought to the foreground from the back stack.

In this method, I call checkOnboardingStatus() again, but this time I pass the Intent.FLAG\_ACTIVITY\_CLEAR\_TOP flag. This flag ensures that if HomeActivity is already running in the task stack, it is brought to the front, and any intermediate activities (such as onboarding fragments or steps) above it are cleared. This helps maintain a clean and predictable navigation experience.

```
class HomeActivity : AppCompatActivity() {  
    private val TAG = "HomeActivity"  
    private lateinit var binding: ActivityHomeBinding  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
  
        binding = ActivityHomeBinding.inflate(layoutInflater)  
        setContentView(binding.root)  
  
        checkOnboardingStatus()  
    }  
  
    override fun onNewIntent(intent: Intent) {  
        super.onNewIntent(intent)  
        Timber.tag(TAG).d("onNewIntent called")  
  
        checkOnboardingStatus(Intent.FLAG_ACTIVITY_CLEAR_TOP)  
    }  
}
```

## OnboardingActivity Overview

### Role Request Launcher Function Explanation

This code defines an Activity Result Launcher that handles the result of requesting the HOME role (default launcher) permission from the user. Here's what it does step by step:

1. **registerForActivityResult()** creates a callback that will be triggered when the user responds to the default launcher permission request.
2. When the user makes a selection (either setting your app as default launcher or declining):

- It saves a preference `default_launcher_requested` as `true` to indicate that the user has gone through the default launcher request flow

- It checks if the app is now set as the default launcher using `isDefaultLauncher()`

3. If the app is successfully set as the **default launcher**:

- It logs a success message
- It saves the current onboarding step as 3
- It navigates to step 3 of the onboarding process

4. If the app is not set as the default launcher (**user declined**):

- It simply logs that the app is not set as default launcher

This launcher is used in the **requestDefaultLauncher()** method, which is called when the user clicks the continue button in Step 2 of the onboarding process. The launcher handles the transition from Step 2 to Step 3 when the user successfully sets the app as the default launcher.

In the context of your previous request about handling manual default launcher settings, this launcher is part of the "normal case" flow where the user explicitly goes through the onboarding process.

```
private val roleRequestLauncher = registerForActivityResult(  
    ActivityResultContracts.StartActivityForResult()  
) { result ->  
    savePreference("default_launcher_requested", true)  
  
    if (isDefaultLauncher()) {  
        Timber.tag(TAG).d("App is now set as default launcher")  
        savePreference("current_onboarding_step", 3)  
        navigateToStep(3)  
    } else {  
        Timber.tag(TAG).d("App is not set as default launcher")  
    }  
}
```



## isDefaultLauncher Function Explanation

This function checks whether the app is currently **set as the default launcher (home app)** on the device. It works differently depending on the Android version:

### For Android 10 (Q) and above:

1. It gets the RoleManager system service.
2. It calls **roleManager.isRoleHeld(RoleManager.ROLE\_HOME)** to check if the app has been granted the HOME role.
3. It logs the result with Timber for debugging purposes.
4. It returns true if the app holds the HOME role, false otherwise.

### For Android 9 (Pie) and below:

1. It creates an Intent with **ACTION\_MAIN** and **CATEGORY\_HOME** (this is the intent that launches when the user presses the home button).
2. It uses **packageManager.resolveActivity()** to find which app would handle this intent.
3. It compares the package name of the resolved activity with this app's package name.
4. It logs detailed information including the resolved package name and the app's package name.
5. It returns true if this app's package name matches the resolved package name, indicating it's the default launcher.

This function is crucial for your app's onboarding flow, particularly for determining whether to advance from Step 2 to Step 3 after the user has set the app as the default launcher. It's called in several places including:

- In the **roleRequestLauncher** callback to check if setting the default launcher was successful.
- In **Step2Fragment.onResume()** to check if the app became the default launcher while in the background.
- In **OnboardingActivity.onResume()** to check if the app should advance to Step 3.
- In **checkAndHandleDefaultLauncherReturn()** to handle returning to the onboarding flow after setting the app as default.

```

fun isDefaultLauncher(): Boolean {
    return if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {

        val roleManager = getSystemService(ROLE_SERVICE) as RoleManager
        val isRoleHeld = roleManager.isRoleHeld(RoleManager.ROLE_HOME)

        Timber.tag(TAG).d("isDefaultLauncher check: isRoleHeld = $isRoleHeld")

        isRoleHeld
    } else {
        val intent = Intent(Intent.ACTION_MAIN)
        intent.addCategory(Intent.CATEGORY_HOME)

        val resolveInfo = packageManager.resolveActivity(intent, 0)
        val isDefault = resolveInfo?.activityInfo?.packageName == packageName

        Timber.tag(TAG).d("isDefaultLauncher check: isDefault = $isDefault, package = ${resolveInfo?.activityInfo?.packageName}, our package = $packageName")
        isDefault
    }
}

```

## requestDefaultLauncher Function Explanation

This function initiates the process of requesting the user to **set the app as the default launcher (home app)**. Here's what it does step by step:

1. First, it saves a preference `default_launcher_requested` as true to track that the app has initiated the default launcher request process
2. Then it handles the request differently based on the Android version:

**For Android 10 (Q) and above:**

- It gets the RoleManager system service
- It checks if the HOME role is available on the device using **`roleManager.isRoleAvailable(RoleManager.ROLE_HOME)`**.
- If available, it creates an intent to request the HOME role using **`roleManager.createRequestRoleIntent(RoleManager.ROLE_HOME)`**.
- It launches this intent using the previously defined **`roleRequestLauncher` (which is an `ActivityResultLauncher`)**.
- The result of this request will be handled by the `roleRequestLauncher` callback we saw earlier.
- It logs that the role request was launched.

### For Android 9 (Pie) and below:

- It creates an Intent with **ACTION\_MAIN** and **CATEGORY\_HOME**.
- It calls **startActivity(intent)** which shows the system's app chooser dialog for the home action.
- The user can select the app and optionally set it as default.
- It logs that the home intent chooser was launched.

This function is called when the user clicks the continue button in Step 2 of the onboarding process. It's the trigger that starts the process of requesting the user to set the app as their default launcher.

### After this function completes:

1. The system UI for selecting a default launcher appears to the user.
2. The user makes a selection.
3. The system then returns control to the app, with the result being handled by the **roleRequestLauncher** callback.

```
fun requestDefaultLauncher() {
    savePreference("default_launcher_requested", true)

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
        val roleManager = getSystemService(ROLE_SERVICE) as RoleManager
        if (roleManager.isRoleAvailable(RoleManager.ROLE_HOME)) {
            val intent = roleManager.createRequestRoleIntent(RoleManager.ROLE_HOME)
            roleRequestLauncher.launch(intent)
            Timber.tag(TAG).d("Launched role request for home")
        }
    } else {
        val intent = Intent(Intent.ACTION_MAIN)
        intent.addCategory(Intent.CATEGORY_HOME)
        startActivity(intent)
        Timber.tag(TAG).d("Launched home intent chooser")
    }
}
```

## Function Explanation: navigateToStep

The navigateToStep function manages navigation between different steps in an onboarding flow. Here's what it does:

1. **Updates Current Step:** Sets the currentStep variable to the provided step number
2. **Saves Preference:** Stores the current step number in SharedPreferences with the key "current\_onboarding\_step" for persistence across app sessions
3. **Creates Fragment :** Creates the appropriate fragment based on the step number:
  - Step 1 → Step1Fragment()
  - Step 2 → Step2Fragment()
  - Step 3 → Step3Fragment()
  - Any other value → defaults to Step1Fragment() as a fallback
4. **Performs Fragment Transaction:** Replaces whatever fragment is currently in the **fragment\_container** with the new fragment using the FragmentManager
5. **Adds Animation:** Applies a fade transition effect between fragments for a smoother user experience
6. **Commits Transaction:** Finalizes the fragment transaction

This function is a key part of the onboarding flow, allowing the app to navigate between different onboarding steps while maintaining the current state.

```
fun navigateToStep(step: Int) {
    currentStep = step

    savePreference("current_onboarding_step", step)

    val fragment = when (step) {
        1 -> Step1Fragment()
        2 -> Step2Fragment()
        3 -> Step3Fragment()
        else -> Step1Fragment()
    }

    supportFragmentManager.beginTransaction()
        .replace(R.id.fragment_container, fragment)
        .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE)
        .commit()
}
```

### **checkAndHandleDefaultLauncherReturn()**

This function checks if the user is returning to the activity after setting the app as the default launcher and handles the navigation accordingly:

1. It checks two conditions:

- If the intent contains an extra boolean flag **FROM\_DEFAULT\_LAUNCHER\_SETTING** set to true.
- If the app is currently set as the default launcher ( **isDefaultLauncher() returns true**).

2. If both conditions are met:

- It logs a debug message using Timber.
- Resets the default\_launcher\_requested preference to false.
- Navigates to step 3 of the onboarding process.
- Returns true to indicate the return was handled.

3. If the conditions aren't met, it returns false, indicating no special handling was needed.

This function is crucial for the app's flow when returning from the system's default launcher selection screen.

### **savePreference(key: String, value: Boolean)**

This is a helper method that saves a boolean value to SharedPreferences:

1. It takes a key (**String**) and a boolean value.
2. Uses the prefs SharedPreferences object with the Kotlin extension function edit to store the value.

### **savePreference(key: String, value: Int)**

This is an overloaded helper method that saves an integer value to SharedPreferences:

1. It takes a key (String) and an integer value.
2. Uses the prefs SharedPreferences object with the Kotlin extension function edit to store the value.

## finishOnboarding()

This function completes the onboarding process:

1. It saves a boolean preference **onboarding\_completed** as **true** to indicate the user has completed onboarding.
2. Calls **finish()** to close the current activity (likely the OnboardingActivity)

This would typically be called when the user completes all onboarding steps and is ready to use the main app.

```
/**
 * Checks if we're returning from setting the app as default launcher
 * @return true if we've handled the return and navigation
 */
private fun checkAndHandleDefaultLauncherReturn(): Boolean {
    if (intent.getBooleanExtra("FROM_DEFAULT_LAUNCHER_SETTING", false) && isDefaultLauncher()) {
        Timber.tag(TAG).d("Coming from setting as default launcher, navigating to step 3")
        savePreference("default_launcher_requested", false)
        navigateToStep(3)
        return true
    }
    return false
}

/**
 * Helper method to save a boolean preference
 */
private fun savePreference(key: String, value: Boolean) {
    prefs.edit { putBoolean(key, value) }
}

/**
 * Helper method to save an integer preference
 */
private fun savePreference(key: String, value: Int) {
    prefs.edit { putInt(key, value) }
}

fun finishOnboarding() {
    savePreference("onboarding_completed", true)
    finish()
}
```

### **onNewIntent(intent: Intent)**

This function is called when the activity receives a new intent while it's already running:

1. It first calls the parent class implementation with `super.onNewIntent(intent)`
2. Logs a debug message using Timber with the tag "TAG" and message "onNewIntent called"
3. Calls `checkAndHandleDefaultLauncherReturn()` to check if the user is returning from setting the app as the default launcher

This method is important for handling cases where the activity is already running and receives a new intent, such as when the user sets the app as the default launcher and the system redirects back to the app.

### **onSaveInstanceState(outState: Bundle)**

This function is called before an activity might be killed so that its state can be saved:

1. It saves the current onboarding step (**currentStep**) to the outState Bundle with the key **"current\_step"**.
2. Calls the parent class implementation with **`super.onSaveInstanceState(outState)`**.

This ensures that if the activity is destroyed and recreated (**e.g., due to a configuration change like screen rotation**), the app can restore the correct onboarding step.

### **onResume()**

This function is called when the activity becomes visible to the user after being paused:

1. It first calls the parent class implementation with **`super.onResume()`**.
2. Checks if the current onboarding step is 2 (**likely the step where the user is prompted to set the app as default launcher**) AND if the app is now **set as the default launcher** (**`isDefaultLauncher()` returns true**).
3. If both conditions are met, it logs a debug message indicating the app became the default launcher while in the background.
4. Then navigates to step 3 of the onboarding process.

This handles the scenario where the user manually sets the app as the default launcher from system settings while the app is in the background, ensuring the onboarding flow continues appropriately when the user returns to the app.

```
override fun onNewIntent(intent: Intent) {
    super.onNewIntent(intent)
    Timber.tag(TAG).d("onNewIntent called")
    checkAndHandleDefaultLauncherReturn()
}

override fun onSaveInstanceState(outState: Bundle) {
    outState.putInt("current_step", currentStep)
    super.onSaveInstanceState(outState)
}

override fun onResume() {
    super.onResume()

    if (currentStep == 2 && isDefaultLauncher()) {
        Timber.tag(TAG).d("App became default launcher while in background")
        navigateToStep(3)
    }
}
```

## Fragment Class Explanation

The Step1Fragment, Step2Fragment and Step3Fragment class represents the first , second and third step in an app onboarding process. Here's what it does:

### Class Structure

- Extends the Android Fragment class, which is a modular section of an activity's UI
- Uses View Binding to interact with UI elements efficiently

### Key Components

#### Properties

- **\_binding**: Holds the binding reference to the fragment's layout (FragmentStep1Binding)
- **binding**: A property delegate that provides safe access to the binding
- **TAG**: A constant used for logging purposes



## Lifecycle Methods

### 1. onCreateView

- Inflates the fragment's layout using View Binding
- Returns the root view of the inflated layout

### 2. onViewCreated :

- Sets up the UI interactions after the view is created
- Configures a click listener for the continue button
- When the button is clicked:
- Logs a debug message using Timber
- Casts the parent activity to OnboardingActivity
- Calls **navigateToStep()** to proceed to the second step of onboarding

### 3. onDestroyView

- Cleans up resources when the fragment's view is destroyed
- Sets the binding reference to null to prevent memory leaks

```
class Step1Fragment : Fragment() {  
  
    private var _binding: FragmentStep1Binding? = null  
    private val binding get() = _binding!!  
  
    private val TAG = "Step1Fragment"  
  
    override fun onCreateView(  
        inflater: LayoutInflater, container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View {  
        _binding = FragmentStep1Binding.inflate(inflater, container, false)  
        return binding.root  
    }  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        binding.continueButton.setOnClickListener {  
            Timber.tag(TAG).d("Continue button clicked, navigating to step 2")  
            (activity as OnboardingActivity).navigateToStep(2)  
        }  
    }  
  
    override fun onDestroyView() {  
        super.onDestroyView()  
        _binding = null  
    }  
}
```

```

class Step2Fragment : Fragment() {

    private var _binding: FragmentStep2Binding? = null
    private val binding get() = _binding!!

    private val TAG = "Step2Fragment"

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentStep2Binding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.continueButton.setOnClickListener {
            Timber.tag(TAG).d("Continue button clicked, requesting default launcher")
            // Save that we're in the middle of setting default launcher
            activity?.getSharedPreferences("home_app_prefs", Context.MODE_PRIVATE)?.edit()
                ?.putBoolean("default_launcher_requested", true)?.apply()

            (activity as OnboardingActivity).requestDefaultLauncher()
        }
    }

    override fun onResume() {
        super.onResume()

        val isDefault = (activity as OnboardingActivity).isDefaultLauncher()
        Timber.tag(TAG).d("onResume: isDefaultLauncher check returned: $isDefault")

        if (isDefault) {
            Timber.tag(TAG).d("App is default launcher, navigating to step 3")
            // Save the current step in case we get interrupted
            activity?.getSharedPreferences("home_app_prefs", Context.MODE_PRIVATE)?.edit()
                ?.putInt("current_onboarding_step", 3)?.apply()
            (activity as OnboardingActivity).navigateToStep(3)
        } else {
            Timber.tag(TAG).d("App is NOT default launcher, staying on step 2")
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}

```

```
class Step3Fragment : Fragment() {

    private var _binding: FragmentStep3Binding? = null
    private val binding get() = _binding!!

    private val TAG = "Step3Fragment"

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentStep3Binding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.continueButton.setOnClickListener {
            Timber.tag(TAG).d("Continue button clicked, finishing onboarding")
            (activity as OnboardingActivity).finishOnboarding()
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

## Reference

**Timber** is a lightweight and extensible logging library for Android, developed by Jake Wharton. It wraps Android's native Log class and provides a cleaner, more flexible API for logging.

<https://github.com/JakeWharton/timber>

The **Android Build Configuration Catalogs** provide a centralized way to manage dependencies, plugins, and versions in Android projects. Introduced in Gradle 7.0+, catalogs use a libs.versions.toml file to simplify and standardize dependency management across modules.

<https://developer.android.com/build/migrate-to-catalogs>

**ViewBinding** in Android provides a modern, safer, and more efficient way to interact with XML views in your Kotlin or Java code. Here are the **main benefits** of using ViewBinding:

<https://developer.android.com/topic/libraries/view-binding>

The Android **SharedPreferences** API provides a simple way to store small amounts of key-value data persistently. It's ideal for saving user preferences, app settings, or flags (like whether onboarding is completed). The data is stored in an XML file and remains available even after the app is closed or the device is restarted. You can access SharedPreferences in either private mode (only accessible by your app) or shared across processes. It's lightweight, fast, and easy to use, making it perfect for storing simple, non-sensitive data.

<https://developer.android.com/training/data-storage/shared-preferences>