Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Android Components and Resource handling

Study Guide

Mr. Akash Suresh Patil
Assistant Professor
CSE, Parul Institute of Technology
Parul University

Parul® University NAAC GRADE A++
Vadodara, Gujarat

Contents

1. Android Components and resource handling

1.1 Components:Context

The four main Android application components are: Activities, Services, Broadcast Receivers, and Content Providers. All of these components rely on Context, though not all of them inherit from it directly:

- Activities and Services are subclasses of Context (specifically ContextThemeWrapper and ContextWrapper, respectively). This means they inherently possess a context and can perform operations that require one.
- Broadcast Receivers and Content Providers are not Context subclasses, but they receive a Context object (or have one associated with them) when the system calls their lifecycle methods (e.g., onReceive(Context, Intent)).

Role of Context
Context is essential for various application-level operations:

- Accessing Resources: Retrieving application-specific resources like strings (getString()), images, layouts (getLayout()), and themes.
- Interacting with Other Components: Launching new Activities (startActivity()), starting Services (startService()), and sending Broadcasts (sendBroadcast()) using Intents.
- Accessing System Services: Interacting with system-level services like LocationManager, NotificationManager, or ConnectivityManager via getSystemService().
- File and Database Access: Performing operations related to the application's file system, databases (SQLite), and shared preferences.

Types of Context and Resource Handling
There are two primary types of Context used in Android, each with a different scope and lifecycle, which is critical for proper resource management:

Context Type	Description	Lifecycle	Primary Use Cases
Activity Context	Tied to the lifecycle of a specific Activity	Exists only while the Activity is alive (between onCreate() and onDestroy())	UI-related operations like inflating layouts, showing dialogs, or displaying Toast messages, as it includes theme information
Application Context	A singleton instance tied to the entire application's lifecycle	Exists as long as the app process is running	Long-lived, non-UI operations such as database connections, network

			operations, or singletons, to avoid memory leaks

Resource Handling Best Practices:

- Use the correct context: Using an Activity Context for long-lived operations that outlast the activity can cause memory leaks. The Application Context is safer for such cases.
- Avoid static references: Do not store an Activity Context in a static variable, as this prevents the garbage collector from freeing the associated activity's memory.
- UI operations require a UI context: Operations involving UI elements (like showing a dialog or inflating a layout) must use an Activity context, as the Application context lacks theme and window information and will cause a crash.

1.2 Activity

An **Activity** is a fundamental Android application component that provides a single screen with a user interface for users to interact with. It is the entry point for user interaction and plays a central role in displaying content, handling input, and managing the flow of an application.

Key Characteristics

- **User Interface (UI):** Every activity is given a window in which to draw its UI, typically defined using XML layout files.
- **Context:** The Activity class is a subclass of Context and provides the necessary context for UI-related operations, as it is tied to the screen's lifecycle and theming.
- **Back Stack Management:** Activities are managed in a "back stack" (a Last-In-First-Out stack). When a new activity starts, it's pushed onto the stack and takes focus. Pressing the Back button finishes the current activity and pops it off the stack, resuming the previous one.
- **Manifest Declaration:** All activities must be declared in the application's AndroidManifest.xml file using the <activity> element.

Activity Lifecycle

The Activity component has a well-defined lifecycle managed by the Android system, with a core set of callback methods that you can override to manage the activity's state and resources effectively:

Method	Description	Resource Handling
onCreate()	Called when the activity is first created. This is where you perform most of your initial setup, such as calling setContentView() to inflate the UI layout and initializing variables.	Initialize UI elements, create objects, bind data to lists.
onStart()	Called when the activity is becoming visible to the user.	Start background tasks, register listeners that need to run while visible but not in the foreground.
onResume()	Called when the activity is in the foreground and ready to interact with the user (it has user focus).	Resume animations, start camera previews, acquire exclusive user-interactive resources.
onPause()	The first indication that the user is leaving the activity. It is called when the activity loses focus.	Pause ongoing operations, stop animations, commit unsaved changes that should be persisted (lightweight operations only). **Do not** use for heavy data saving or network calls.
onStop()	Called when the activity is no longer visible to the user (e.g., a new activity fully covers it).	Release heavy resources like database connections or sensor handlers to conserve power and memory.

onRestart()	Called after the activity has been stopped, just prior to it being started again.	Restore state from when it was stopped; always followed by onStart().
onDestroy()	The final call before the activity instance is completely destroyed and removed from memory.	Release all remaining resources not released by previous callbacks, unregister any remaining listeners.

1.3 Intent

An **Intent** is a messaging object used to request an action from another app component in Android. It is the primary mechanism for inter-component communication and provides a powerful way to decouple components while enabling dynamic application flow.

Key Characteristics

- **Inter-Component Communication:** Intents facilitate communication between Activities, Services, and Broadcast Receivers. Content Providers are not started via Intents.
- **Decoupling:** Components don't need to know about the implementation details of the components they are launching; they just send an intent describing the action they need.
- **Resource Handling:** Intents themselves are lightweight objects and do not manage heavy resources. They primarily act as a data carrier and communication bridge, indirectly leading to resource loading (e.g., when a new Activity is started, it loads its layout resources).

Types of Intents

There are two main types of intents:

1. **Explicit Intents:**

1. Specify the exact component name (class name) to start.
2. Used typically for starting components within the same application.
3. *Example:* Starting a SettingsActivity from the MainActivity.

2. **Implicit Intents:**

   0. Declare a general action to be performed (e.g., "view a map", "send an email", "take a picture").
   1. Rely on the Android system to find and launch an appropriate component (potentially in a different app) that can handle that action.
   2. *Example:* Launching a map application to show a specific location using the ACTION_VIEW action and a geographic URI.

Anatomy of an Intent

An Intent object typically contains several key pieces of information:

- **Component Name (Explicit Intents only):** The fully qualified class name of the target component.
- **Action:** A string that names the action to be performed (e.g., Intent.ACTION_VIEW, Intent.ACTION_SEND).
- **Data:** A URI pointing to the data to be acted upon (e.g., a phone number to dial, a URL to view).
- **Category (Optional):** Additional information about the kind of component that should handle the intent (e.g., Intent.CATEGORY_LAUNCHER).
- **Extras (Optional):** Key-value pairs in a Bundle that carry supplementary data required for the action (e.g., the subject and body of an email).
- **Flags (Optional):** Instructions for the Android system about how to launch the activity (e.g., whether the new activity should belong to the current task stack).

Using Intents in Resource Handling

Intents are the mechanism by which components transition and manage their lifecycles, which inherently involves resource management:

- **Launching Components:**
  - startActivity(Intent): Starts a new Activity and triggers its lifecycle (onCreate(), onStart(), etc., which load layout resources).
  - startService(Intent): Starts a Service (which may load non-UI resources, like starting a background music player).

- sendBroadcast(Intent): Sends a broadcast message to registered Broadcast Receivers.

- **Passing Data and Resources:**
  - Intents carry resource identifiers or data references (Extras and Data URI) from one screen to the next. The receiving component then uses this information and its local Context to fetch the actual resources (e.g., reading a specific image file path from an Extra and displaying it).

1.4 Service

A **Service** is an application component that can perform long-running operations in the background and does not provide a user interface. Services are primarily used for tasks that need to run continuously, even when the application's user interface is not visible or the user switches to a different application.

Key Characteristics

- **No UI:** A Service runs in the background and does not interact with the user directly.
- **Background Tasks:** Ideal for operations such as playing music, fetching data over the network, handling database transactions, or running operations that should persist across activity switches.
- **Lifecycle:** Like Activities, Services have a defined lifecycle managed by the Android system.
- **Context:** Services are a subclass of ContextWrapper (which inherits from Context) and have access to application resources and system services.
- **Main Thread Warning:** By default, a service runs in the main thread of its host application's process. Heavy, blocking operations (like network calls) must be performed in a separate worker thread within the service to prevent the *Application Not Responding* (ANR) error.

Types of Services

Android services are broadly categorized into three types:

1. **Foreground Services:**

1. Perform operations noticeable to the user (e.g., a music player playing a song, a navigation app providing directions).
2. They must display a Notification to the user, preventing the system from terminating them easily when memory is low.
3. Started using startForeground().

2. **Background Services:**
    0. Perform operations that are not directly noticed by the user (e.g., syncing data in the background).
    1. The system is more likely to terminate a background service if it needs memory for foreground processes.
3. **Bound Services:**
    0. Offer a client-server interface that allows components (like Activities) to interact with the service, send requests, get results, and even do inter-process communication (IPC).
    1. Multiple components can bind to a service simultaneously.
    2. Exist only as long as another application component is bound to them.

Service Lifecycle and Resource Handling

The service lifecycle callbacks are fundamental for managing resources efficiently:

Method	Description	Resource Handling
onCreate()	Called when the service is first created.	Initialize variables and set up worker threads.
onStartCommand()	Called when a component calls startService(). The service starts running and can be explicitly stopped later.	Begin background tasks, acquire necessary resources (e.g., network connection, media player).

onBind()	Called when a component calls bindService(). It must return an IBinder interface for communication.	Set up the communication channel/interface for binding clients.
onUnbind()	Called when all clients have disconnected from the service.	Release communication resources associated with the binder.
onDestroy()	The final call before the service is shut down and reclaimed by the system.	**Crucial:** Release all remaining resources, stop all worker threads, unregister receivers, and close network connections.

Resource Handling

Effective resource handling in a Service involves:

- **Thread Management:** Heavy operations must use worker threads or an IntentService (deprecated but simpler for single tasks) or modern alternatives like WorkManager to prevent blocking the UI thread.
- **Notification for Foreground:** For long-running, user-visible tasks, use startForeground() and its associated Notification resources to ensure the service persists and is not killed by the system.
- **Prompt Cleanup:** Releasing resources immediately in onDestroy() or onUnbind() is vital to avoid memory leaks and ensure the system runs smoothly.

1.5 BroadcastReceiver Resources:String,Colour,Drawable,Styles

A **BroadcastReceiver** is an Android component that listens for system-wide or application-specific broadcast announcements (Intents). These broadcasts are messages sent by the system (e.g., "battery is low", "Wi-Fi connected") or by applications (e.g., "data download complete").

Broadcast Receivers do not have a user interface and are designed to be short-lived, typically just processing the incoming broadcast message and perhaps starting a Service or displaying a Notification.

Key Characteristics

- **Event-Driven:** They only run when an event matching their registered IntentFilter occurs.
- **Short Lifecycle:** onReceive() is the only main method. The system terminates the receiver shortly after this method finishes (within ~10 seconds).
- **No UI, but can trigger UI:** Receivers cannot display dialogs directly but can launch an Activity or use the NotificationManager to alert the user.
- **Context Access:** The onReceive(Context context, Intent intent) method provides a Context which can be used to access application resources and system services.

Broadcast Receiver Lifecycle

The lifecycle is extremely simple:

1. The receiver is created when the broadcast is sent.
2. onReceive() is called.
3. The process is killed shortly after onReceive() returns.

Resource Handling

Due to their brief lifecycle and lack of UI, Broadcast Receivers handle resources differently than Activities or Services. They use the provided Context to fetch resources needed for follow-up actions (like building a Notification).

Here is how a BroadcastReceiver interacts with common Android resource types:

1. Strings

Strings are often used in the actions defined in the Intent filters or in the messages displayed in notifications triggered by the receiver.

- **Access Method:** context.getString(R.string.resource_id)
- **Example Use:** Getting a notification title from the string resources to display after a successful broadcast event.

2. Colors

Colors might be used to customize status bar icons or notification backgrounds when the receiver decides to show a notification.

- **Access Method:** ContextCompat.getColor(context, R.color.resource_id) or context.getColor(R.color.resource_id) (API 23+)
- **Example Use:** Setting the color of a notification icon.

3. Drawables

Drawables are commonly used for icons in the status bar notifications launched by the receiver.

- **Access Method:** ContextCompat.getDrawable(context, R.drawable.resource_id) or context.getDrawable(R.drawable.resource_id) (API 21+)
- **Example Use:** Setting the small icon (setSmallIcon()) for a Notification Builder.

4. Styles and Themes

Broadcast Receivers themselves do not use themes or styles because they have no UI window. Themes are a concept tied specifically to UI components like Activities (which inherit from ContextThemeWrapper).

- **Usage Context:** If a Broadcast Receiver *launches* an Activity, that newly launched Activity will inherit its defined theme and apply its styles when it inflates its layout resources. The receiver just initiates the process.

1.6 Theme Localization:Prepare Application for Localization

To prepare an Android application for localization (internationalization), the core strategy is to **separate all localizable content from the core application logic** using the Android resource framework. This allows the system to load appropriate resources based on the user's locale settings without requiring changes to the compiled code.

Core Steps for Preparation

1. Externalize All String Resources

The most crucial step is moving every user-facing text string out of the Java/Kotlin code and XML layouts into a default strings.xml file located in the res/values/ directory.

- **Avoid Hardcoding:** Never hardcode strings in your code or layout files. Android Studio provides inspections and tools to help extract hardcoded strings into resource files.
- **Provide Context:** Add comments within the strings.xml file to provide context for translators (e.g., whether a word is used as a verb or a noun, or character limits for a button).
- **Handle Plurals:** Use <plurals> tags in strings.xml to handle quantity strings correctly, as different languages have different pluralization rules (e.g., 0 cats, 1 cat, many cats).
- **Mark Non-Translatables:** Use the translatable="false" attribute or an <xliff:g> placeholder tag for text that should not be translated (e.g., code snippets, proper names, or special symbols).

2. Design Flexible Layouts

Translated text often expands or contracts (e.g., German words can be longer than English ones, while East Asian languages can be shorter).

- **Use Flexible Units:** Avoid using fixed widths and heights for UI elements. Use wrap_content, match_parent, weights, and constraint layouts to allow UI elements to adjust their size dynamically to fit the translated content.
- **Consider RTL Languages:** Prepare for Right-to-Left (RTL) languages like Arabic and Hebrew from the start. Use start and end attributes instead of left and right for padding and margins, and set android:supportsRtl="true" in your manifest to allow the layout to flip automatically.

3. Use Locale-Aware APIs for Formats

Date, time, currency, and number formats vary globally. Avoid hardcoding these formats.

- **Use System APIs:** Use standard Java/Kotlin formatters (DateFormat, NumberFormat) which automatically use the device's current locale settings to display information correctly.

4. Prepare Alternative Resources

Beyond strings, other resources might need localization (e.g., an image containing text).

- **Use Resource Qualifiers:** Create alternative resource directories using qualifiers (e.g., res/values-fr/ for French strings, res/drawable-de/ for German-specific drawables, or res/layout-ar/ for Arabic layouts).
- **Provide Defaults:** Ensure a complete set of **default resources** (in folders without qualifiers like res/values/ and res/drawable/) exists. If Android cannot find a locale-specific resource, it falls back to the default. A missing default resource can cause an app to crash.

5. Consider Cultural Nuances in Design

Localization is more than translation; it involves cultural adaptation.

- **Culturally Appropriate Images:** Avoid imagery or icons that might be offensive or irrelevant in certain cultures. If necessary, provide localized images using resource qualifiers.
- **Font Choice:** Use Unicode fonts that support the diverse characters required by multiple languages.

Parul® University
Vadodara, Gujarat
NAAC GRADE A++

https://paruluniversity.ac.in/