Android Platform Overview

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Objectives

After this module, you will be able to:

- List the four levels of the Android technology stack and describe the principal characteristics and responsibilities of each level
- Design the high-level architecture of an Android application using the primary Android application components
- Select appropriate Android message types to activate application components and pass information between them

1 The Android Technology Stack

Objectives After this section, you will be able to:

- List the four levels of the Android technology stack and describe the principal characteristics and responsibilities of each level

1.1 Android Stack Overview

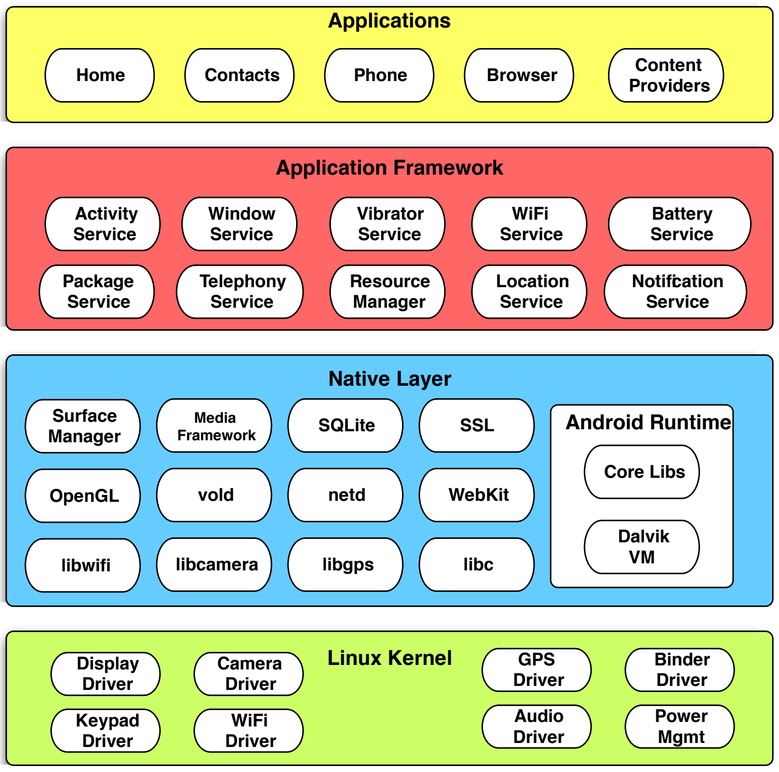

Figure 1: Android Stack

The Android technology stack can be divided into four layers. The following sections provide an overview of each layer and the role it plays in the Android platform.

1.2 Linux Kernel Layer

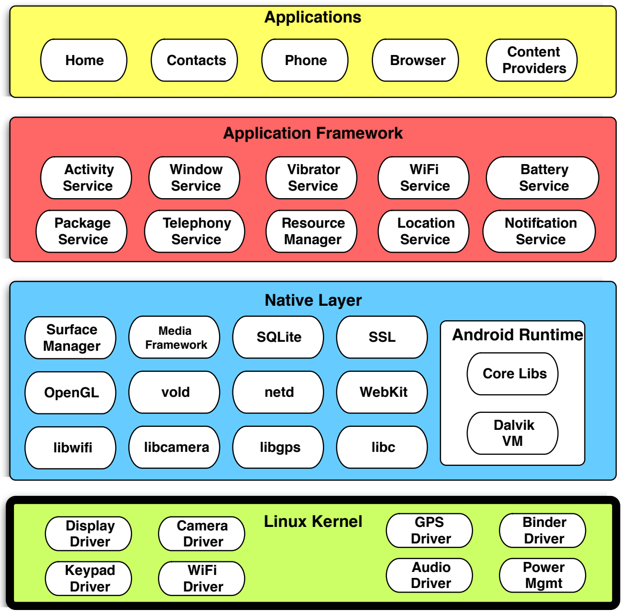

Figure 2: Android Stack: Linux Kernel

Android is based on a modified Linux kernel, which provides features such as:

- A permission-based security model
- Memory management
- Process management
- A network stack
- A device driver model

The lowest layer of the Android software stack is a Linux kernel customized by Google for the Android platform. Application developers don't interact directly with the kernel layer, but instead access system features through the higher levels of the Android platform.

One aspect of the Linux core that does affect application developers is the Linux permission-based security model. An Android device assigns a unique user ID to each application installed. Each application runs as a separate Linux process with the effective user ID assigned to that application. Linux file system permissions then determine the directories, files, and devices that an application's process can access.

We'll explore Android security model in more depth throughout this course.

1.3 Native Layer

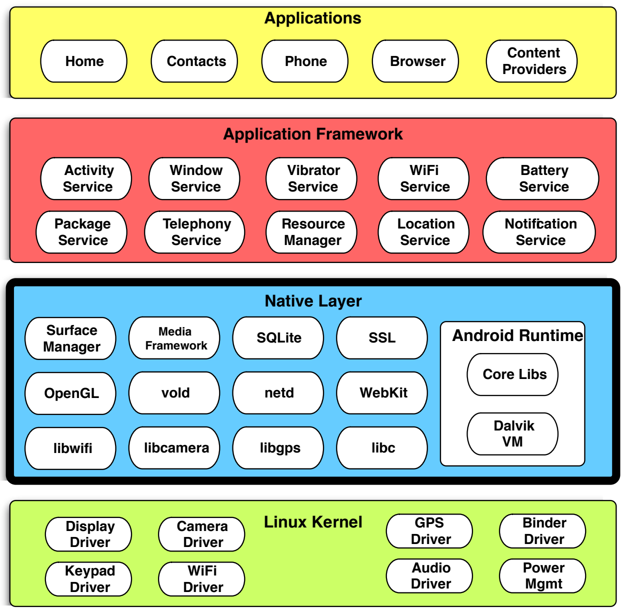

Figure 3: Android Stack: Native Libraries

The Android native layer contains:

- A custom `libc` implementation called *Bionic*
- Custom libraries for Android features
- Third-party open-source libraries such as WebKit, SQLite, OpenGL, etc.
- Linux daemons managing system features such as telephony, networking, sensors, etc.
- Auxiliary Linux executables

Note
Android exposes only a small subset of native libraries to application developers through Android's Native Development Kit (NDK).

The native layer is Android's Linux user-space component written in C/C++ and compiled to native machine code. It includes many third-party open-source libraries in addition to the custom code implementing and managing Android-specific features.

Application developers don't interact directly with the native layer in most cases. As we'll see in the next topic, Android supports the Java programming language as the primary application development language. However, Android's Native Development Kit (NDK) exposes a small subset of native libraries to application developers, including access to native OpenGL and audio libraries. See the documentation for Android's NDK for more detailed information on incorporating native code in an application.

Aside from some third-party libraries, almost all of the code implementing the native layer and above is subject to the Apache Software License, 2.0.

1.4 Android Runtime Environment: Java and Dalvik

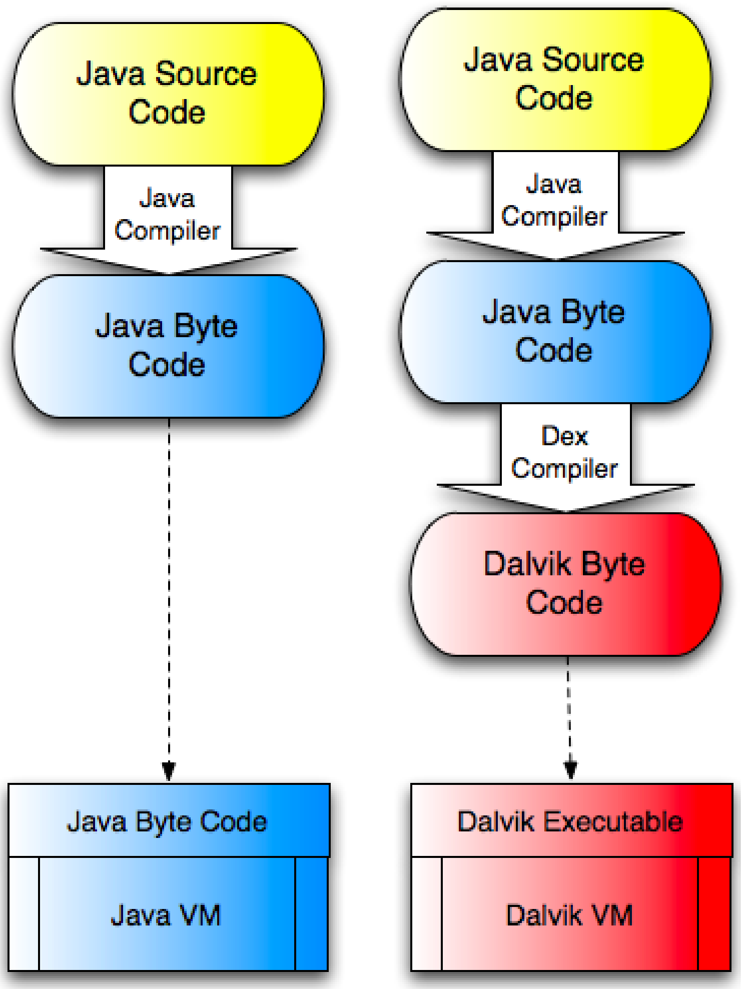

Figure 4: Generating Dalvik Bytecode

Android applications execute in a *Dalvik* virtual machine (VM).

- The Dalvik VM was custom designed for the Android environment; it is not based on the Java VM.
- The Dalvik VM executes files in the Dalvik Executable (`.dex`) format.
- Each application process hosts its own Dalvik VM, ensuring strict sandboxing of applications.

Android supports Java as the primary application development language.

- The Android SDK includes the `dx` tool, which compiles Java bytecode to Dalvik bytecode.
- The Android Development Tool (ADT) plugin to Eclipse is pre-configured to generate Dalvik bytecode from your Java source.

Dalvik is the name of the virtual machine (VM) which runs Android applications. This VM executes Dalvik bytecode, which is compiled from programs written in the Java language. Note that the Dalvik VM is not a Java VM (JVM).

Every Android application runs in its own process, with its own instance of the Dalvik VM. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (`.dex`) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the `.dex` format by the `dx` tool included in the Android SDK.

Note
Although you can invoke the `dx` tool directly to generate Dalvik bytecode from compiled Java classes, the Android Development Tool (ADT) plugin to Eclipse is pre-configured to handle this for you.

As of version 2.2 (Froyo), Dalvik includes a Just-In-Time (JIT) compiler. The Dalvik JIT is a "trace-granularity JIT," which means that it compiles individual code fragments that it discovers at runtime to be hot spots (that is, it does not compile entire methods). Google reports the memory overhead of the JIT compiled code to be between 100-200KB per application.

1.5 Application Framework Layer

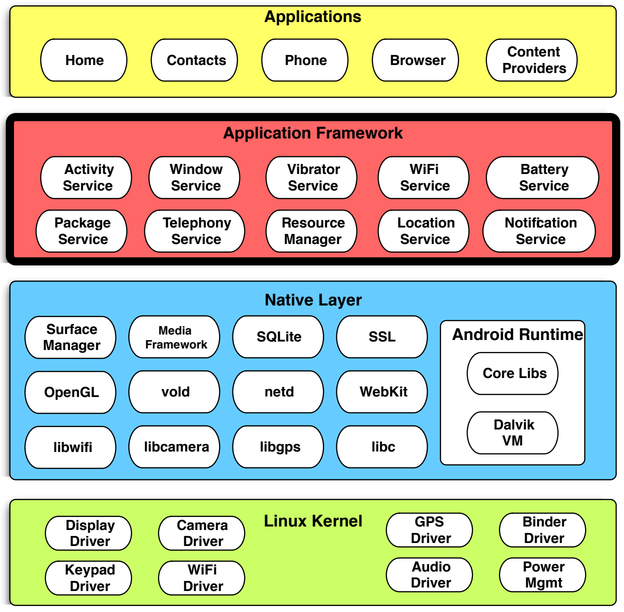

Figure 5: Android Stack: Application Framework

The application framework layer implements Android's Java language API for application development. It includes:

- Classes for implementing application components
- "Manager" classes for accessing Android system services such as Bluetooth, cameras, sensors, WiFi, etc.
- "Wrapper" classes for native libraries such as OpenGL and SQLite
- Most standard Java SE packages
- Auxiliary packages such as JSON and XML parsers and the Apache HTTP Client

---

The application framework layer is a rich environment that provides numerous services to you, the application developer. As you explore Android application development, most of your focus will be on this part of the stack.

In the application framework layer, you will find numerous Java libraries specifically built for Android. It includes:

- The core application component classes for implementing user interfaces and performing background processing
- *Manager* classes that access Android's system services, such as location services, sensors, WiFi, telephony, and so on
- Java language wrappers for native libraries like OpenGL and SQLite
- The Apache HTTP Client packages for writing web service clients
- Parsers for JSON and XML data
- Most of the packages found in Java SE, based on the Apache Harmony open source project

You can find full documentation for the application framework APIs at http://developer.android.com/reference/packages.html

1.6 Applications Layer

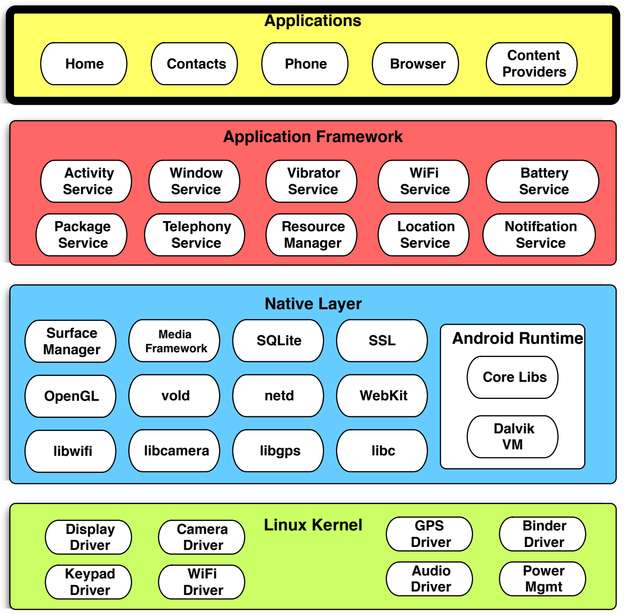

Figure 6: Android Stack: Applications

The application layer consists of pre-installed *system applications* and user-installed applications.

- System and user-installed applications have the same structure and are built on the same application framework layer.
- System applications have access to some private framework APIs not exposed to third-party application developers.
- System applications can't be uninstalled by the user.

User-installed applications can provide the same functionality as most system applications.

- For example, there are several third-party browser applications available.
- Android allows the user to select one of these "replacement" applications as a default for actions like viewing a web page or sending an email.

1.7 Application Structure

You distribute an Android application as a single file, known as an *application package* or *APK*, containing:

- Dalvik executable code compiled from your Java source code
- Resources, such as images, audio/video clips, as well as numerous XML files describing layouts, language packs, and so on
- Optional native shared libraries, compiled from C/C++ source code using Android's NDK.

APKs must be signed with a security certificate before they can be installed on a device.

- During development, you'll use an automatically-generated debug certificate.
- For production, you'll use a self-signed certificate that you generate.

Note
Google recommends a validity period of at least 25 years.

An application is a single file, known as an *application package* or *APK*. APK files are ZIP file formatted packages based on the JAR file format, with `.apk` file extensions. They contain:

- Dalvik executable code: This is all your Java source code compiled down to Dalvik executable. This is the code that runs your application.
- Resources: Resources are everything that is not code. Your application may contain number of images, audio/video clips, as well as numerous XML files describing layouts, language packs, and so on. Collectively, these items are resources.
- Native libraries: Optionally, your application may include some native shared libraries, compiled from C/C++ source code using Android's NDK.

Android applications must be signed before they can be installed on a device. For development purposes, you'll sign applications with a debug key, which the ADT plugin to Eclipse creates for you automatically when you first use it to create an Android application. However, when you want to distribute your application commercially, you'll want to sign it with your own key. The details how to do that are at Signing Your Application page.

1.8 Topic Summary

You should now be able to:

- List the four levels of the Android technology stack and describe the principal characteristics and responsibilities of each level

2 Android Application Components

Objectives After this section, you will be able to:

- List the four types of application components provided by the Android framework and describe their roles in an Android application
- Design the high-level architecture of an Android application using the primary Android application components

2.1 Application Components Overview

The Android framework defines four types of application components that serve as the building blocks of an Android application:

Activities
An *activity* represents a single screen with a user interface. An activity is implemented as a subclass of `Activity`.

Services
A *service* performs some long-running operation in the background without a user interface. A service is implemented as a subclass of `Service`.

Broadcast Receivers
A *broadcast receiver* responds to system-wide broadcast announcements; they are the subscribers in Android's *publish-subscribe* messaging pattern. A broadcast receiver is implemented as a subclass of `BroadcastReceiver`.

Content Providers
A *content provider* exposes a structured set of data to other applications through a database-like interface. A content provider is implemented as a subclass of `ContentProvider`.

2.2 Template-Based Application Components

The component base classes follow the *template method* design pattern.

- The base classes define methods invoked by the system to interact with the component.
- Most base class methods implement some default functionality.
- You'll create subclasses of these base classes to define your application's components, overriding the base class methods to implement custom functionality.

The system creates and destroys instances of your application components as needed.

- You never explicitly invoke the constructor for any of your application components.
- The system invokes lifecycle methods on the components to notify them of their creation, destruction, and other lifecycle events.

2.3 Components in the Application Process

Android applications don't have a single entry point (that is, there is no `main()` method).

- The Android systems starts a Linux process for the application when it needs to launch one of that application's components.
  - For example, when the user taps your application's icon in Android's app launcher, the system launches your app's *main activity*.
  - If a Linux process does not already exist for your app, the system starts one, initializes a Dalvik VM, and creates an instance of your app's main activity class.
- Multiple components can exist simultaneously within the application process.
- Each application component has its own independent lifecycle driven by the system.

Even if the system destroys all of an app's components, it might keep the process in memory.

- Even destroying the main activity does not cause the process to terminate automatically.
- The system might cache the application process to decrease the start-up time on subsequent launches of app components.

- The system terminates the application process only if it needs to reclaim memory for use in other processes.

Programming for Android is conceptually different than programming for some other environments. In Android, you find yourself more responding to certain changes in the state of your application rather than driving that change yourself. It is a managed, container-based environment similar to programming for Java applets or servlets. So for example, when it comes to an activity's lifecycle, you don't get to say what state the activity is in but you do get to say what happens on transitions from state to state by overriding the appropriate lifecycle method.

2.4 Activity Lifecycle Methods

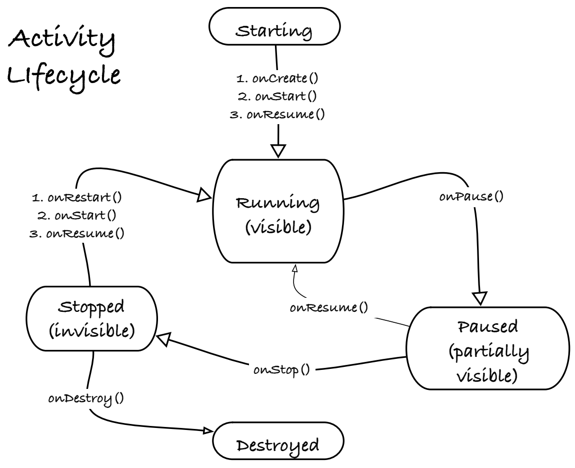

Figure 7: Activity Lifecycle

When the system launches an activity, it invokes a set of callback methods on the activity that you as a developer can override and implement. Eventually, the activity will be in *Running State*.

Keep in mind that this transition from *Starting State* to *Running State* is one of the most expensive operations in terms of computing time needed. The computing time directly affects the battery life of the device as well. This is why the system doesn't automatically destroy activities that are no longer shown. The user may want to come back to them, so the system keeps them around for awhile.

An activity in a *Running State* is the one that is currently on the screen interacting with the user. We also say this activity is in the *foreground* or has *focus*, meaning that all user interactions, such as typing, touching screen, clicking buttons, are directed to this one activity. As such, there is only one foreground activity in the system at any one time.

The running activity is the one that has all the priorities in terms of getting memory and resources needed to run as fast as possible. This is because Android wants to make sure the running activity is zippy and responsive to user.

When an activity is not in the foreground but still visible on the screen, we say it's in *Paused State*. This is not a very typical scenario since the screen is usually small and an activity is either taking the whole screen or not at all. We often see this case with dialog boxes that come up in front an activity causing it to become *Paused*. *Paused State* is a state that all activities go through it en route to being stopped.

Paused activities still have high priority in terms of getting memory and other resources. This is because they are visible and cannot be removed from the screen without making it look very strange to the user.

When an activity is not visible, but still in memory, we say it's in *Stopped State*. A stopped activity could be brought back to front to become a *Running* activity again. Or, it could be destroyed and removed from memory.

The system keeps activities around in *Stopped State* because it is likely that the user will still want to get back to those activities at some time. Restarting a stopped activity is far cheaper than starting an activity from scratch because the activity object is still in memory and simply needs to be brought to the foreground.

Stopped activities can also, an any point, be removed from memory.

A destroyed activity is no longer in memory. The Activity Manager decided that this activity is no longer needed, and as such has removed it. Before the activity is destroyed, you, the developer, have an opportunity to perform certain actions, such as save any unsaved information. However, there's no guarantee that your activity will be *Stopped* prior to being *Destroyed*. It is possible that a *Paused* activity gets destroyed as well. For that reason, it is better to do important work, such as saving unsaved data, en route to being *Paused* rather than *Destroyed*.

Note
The fact that an activity is in the *Running State* doesn't mean it's doing much. It could be just sitting there and waiting for user input. Similarly, an activity in *Stopped State* is not necessarily doing nothing. The state names mostly refer to how active the activity is with respect to user input — in other words, whether an activity is visible, has focus, or is not visible at all.

2.5 Service Lifecycle Methods

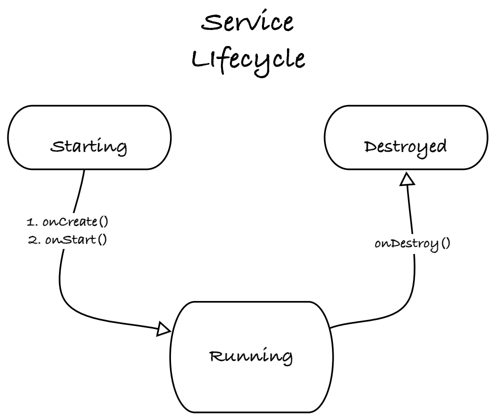

Figure 8: Service Lifecycle

Services have a much simpler lifecycle than activities. Also, you typically have more control as to when to start and stop a service. So, you should be mindful to run your services so that they don't unnecessarily consume shared resources, such as CPU and battery.

IMPORTANT

The fact that a service runs in the background doesn't mean it runs on a separate thread. The system runs all lifecycle methods of all your application components in a single thread in your application, known as the *main* thread or *UI* thread.

If one lifecycle method takes a while to complete, the system cannot execute any other lifecycle methods in your application. In this case, your application can appear frozen as far as the user is concerned. User interactions such as touch events, key presses, and screen redraws are queued for processing while a lifecycle method is blocked.

If the system detects that events have been queued and not processed for longer than 5 seconds, it determines that the application is not responding (also known as an *ANR condition*). The system automatically posts a dialog alerting the user of an ANR condition, and gives the user the opportunity to force close the application, terminating the application process.

Obviously, any operation taking longer than 5 seconds should be executed in a worker thread, but usability studies have shown that users can perceive delays as short as 0.1 second. Therefore, in any situation where you anticipate an operation to take any significant time to execute (for example, network access, database queries, file I/O, etc.), you should create a worker thread to perform the operation and not block the main thread. We will examine strategies and mechanisms for thread management in Android later in this course.

2.6 Broadcast Receiver Lifecycle Methods

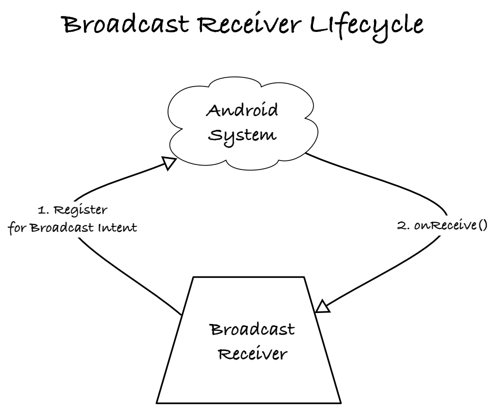
Broadcast Receiver Lifecycle
Android System
1. Register for Broadcast Intent
2. onReceive()
Broadcast Receiver

Figure 9: Broadcast Receiver Lifecycle

Broadcast receivers are the subscribers in Android's *publish-subscribe* messaging pattern (more precisely, this is an Observer pattern). The receiver is simply an object that the system activates once an event occurs that the receivers is subscribed to. A broadcast receiver does not have any visual representation.

The system itself broadcasts events all the time. For example, when an SMS arrives, or call comes in, or battery runs low, or system gets booted, all those events are broadcast and any number of receivers could be triggered by them.

You can also send your own broadcasts from one part of your application to another, or a totally different application.

Broadcast receivers do not exist in memory until the event they are subscribed to occurs. When that happens, the system instantiates the broadcast receiver and executes its `onReceive()` method (in the main thread of the application process). Once the `onReceive()` method returns, the system marks the broadcast receiver object for garbage collection.

Note

If an application process for the receiver does not exist at the time the subscribed event takes place, the system creates an application process and then instantiates and invokes the receiver.

2.7 Content Provider Lifecycle Methods

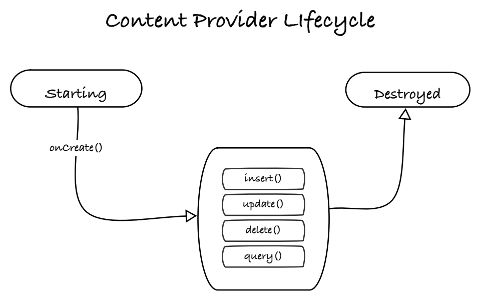

Figure 10: Content Provider Lifecycle

Content providers are interfaces for sharing data between applications. Android by default runs each application in its own sandbox so that all data that belongs to an application is totally isolated from other applications on the system. While small amounts of data can be passed between applications via a type of message called an `Intent`, content providers are much better suited for sharing large datasets.

Content providers are a relatively simple interface with the standard `insert()`, `update()`, `delete()`, and `query()` methods. These methods look a lot like standard database methods, so it is relatively easy to implement a content provider as a proxy to the database. Having said that, you are much more likely to use content providers than write your own.

The Android system uses this mechanism all the time. For example, the `ContactsContract` provider is a content provider that exposes all users contacts data to various applications. `Settings` provider exposes system settings to various applications including the built-in Settings application. `MediaStore` is responsible for storing and sharing all various media, such as photos, and music across various applications.

This separation of data storage and the actual user interface application offers greater flexibility to mash up various parts of the system. For example, a user could install an alternative address book application that uses the same data as the default contacts app. Or, install widgets on the home screen that allow for easy changes in the system settings, such as turning on or off the Wifi, Bluetooth or GPS features.

2.8 The Application Class

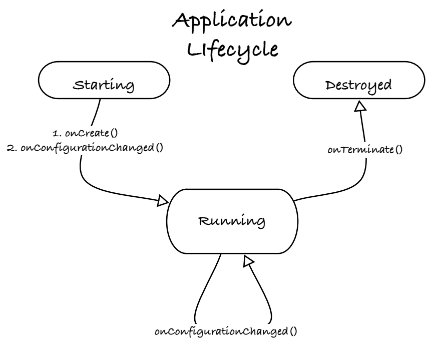

Figure 11: Application Object Lifecycle

In addition to the four types of application components just described, the Android framework includes an `Application` class. When the Android system creates a Linux process for your application, it first creates an instance of the `Application` class before instantiating any other of your application components. This instance of the `Application` class continues to exists for the lifetime of the process.

The purpose of this `Application` instance is to provide a singleton object through which you can access your application's environment, including:

- Application resources (images, strings, layouts, etc.) bundled into your APK
- Application *assets*, such as fonts, that you can also include in your APK
- Files, databases, and preferences in your application's private storage area in the file system

The methods providing access to the environment are defined in an ancestor class of `Application` named `Context`. `Context` is also an ancestor of the `Activity` and `Service` classes, so you can access your application's environment through instances of those classes as well. However activities and services are transient, whereas the `Application` object persists through the life of your application process. If your code need a long-lived reference to an object providing a `Context`, the `Application` object is the best choice to use; holding references to activities and services after the system tries to destroy those objects can lead to memory leaks in your application.

You can retrieve a reference to the `Application` object by invoking `Context.getApplication()`. If you need the `Application` object solely for the purpose of accessing the application environment, you can invoke `Context.getApplication` which returns a reference of type `Context`.

As we'll see later in this class, you also have the option of extending the `Application` class and requesting the system to instantiate your subclass instead of the default `Application` class. Doing so allows you to:

- Maintain application state
- Transfer objects between application components
- Manage resources used by several application components

2.9 Processes and Memory Management

The Android system retains application processes in memory for as long as possible.

- Retaining the processes increases system responsiveness for the user.
- Under most circumstances, the system doesn't need to recreate an app process when the user returns to a previously launched app.

The system kills application processes automatically if system memory is low and the system needs to reclaim memory.

- Application processes are prioritized, and the system kills lower-priority processes until it has freed a sufficient amount of memory.

Note
The heap space allocated to an application process is strictly limited by the system. (The exact amount is determined by the vendor or creator of a custom system image.) Therefore, a normal application process can't "go rogue" and consume all available system memory.

2.10 Application Process Prioritization

For memory management, the priority of a process depends active application components it contains and their current state.

The prioritization of application processes is shown below, in descending order:

Foreground process
A process hosting:

- the foreground activity
- a service marked as running in the foreground (for example, for music playback)
- a service executing one of its lifecycle methods
- a broadcast receiver executing it `onReceive()` method

Visible process
A process without a foreground component (as defined above), but with one or more visible, paused activities

Service process
A process with a running service and no foreground or visible components

Background process
A process with no foreground or visible components, and no running services

Empty processes
A process with no active application components

A configuration file in the system images specifies the low-memory thresholds that trigger the system to kill different types of processes. Vendors and creators of custom system images can adjust these settings if desired.

2.11 Topic Summary

You should now be able to:

- List the four types of application components provided by the Android framework and describe their roles in an Android application
- Design the high-level architecture of an Android application using the primary Android application components

3 Intents: Messages and Component Activation

Objectives After this section, you will be able to:

- Select appropriate Android message types to activate application components and pass information between them

3.1 Intents

Three of the four component types (activities, services, and broadcast receivers) are activated by an asynchronous message called an *intent*.

- Intents are instances of the `Intent` class.
- Invoking `Context.startActivity(Intent)` starts an activity and passes the `Intent` to the activity.
- Invoking `Context.startService(Intent)` starts a service and passes the `Intent` to the activity.
- Invoking `Context.stopService(Intent)` stops a service identified by the `Intent`.
- Invoking `Context.sendBroadcast(Intent)` delivers an `Intent` as a broadcast message to any broadcast receivers registered to receive it.

3.2 Intent Properties

There are several properties on an `Intent` object that you can set to provide information and select the recipient component. The intent properties are:

Component name
The target of the intent expressed as a `ComponentName`, which is a combination of the application package name and the fully-qualified class name of the component.

Action
A `String` specifying the requested action to perform. The system defines several standard actions, and you can define your own custom actions.

Data
A `Uri` object identifying the data to act upon.

Categories
One or more `String` values providing additional information about the kind of component that should handle the intent.

Extras
Key-value pairs for additional information that should be delivered to the component handling the intent.

When creating an `Intent` object:

- No single property is required.
- You can set the commonly used properties on creation through the overloaded constructor.
- The `Intent` class has setter methods to set any property after creation.

Consult the `Intent` class documentation for more information about these properties and examples of their use.

3.3 Explicit Intents

Explicit intents designate the target component by its name.

- They are used to start activities or start or stop services.
- The request is successful if the indicated target exists on the system; otherwise, the request raises an exception.

You typically use explicit intents for intra-application communication.

3.4 Implicit Intents

Implicit intents do not name a target.

- They also are used to start activities or start or stop services.
- The system uses the action, data, and/or categories to identify a suitable recipient.
- Components can provide *intent filters* to specify the types of intents they can handle.
- The request is successful if at least one suitable target exists on the system; otherwise, the request raises an exception.
- If there is more than one suitable target, the system automatically displays a dialog asking the user to choose a target application.

You typically use implicit intents to activate components in other applications.

- An application designed to handle implicit intents should provide documentation of the intent types it supports.
- The `Intent` class documentation describes some of the standard action/data values supported by system applications.

We will see later in this course how to create and use different types of intents and intent filters.

3.5 Broadcast Intents

Broadcast intents serve as broadcast messages delivered to all broadcast receivers registered to handle them.

- They must have their action properties set.
- You can also set data and/or extras on a broadcast intent.
- Broadcast receivers provide intent filters to specify the broadcast intent(s) they can handle.

A single broadcast message can activate multiple broadcast receivers.

- By default, the system activates the receivers in an undefined order — often concurrently — and provides a separate copy of the broadcast intent to each receiver.
- It’s possible (though rare) to send an *ordered broadcast*, in which case the system delivers the broadcast intent to one receiver at a time. Each receiver has the option to propagate the intent to the next receiver or abort the broadcast.

3.6 Tasks and the Back Stack

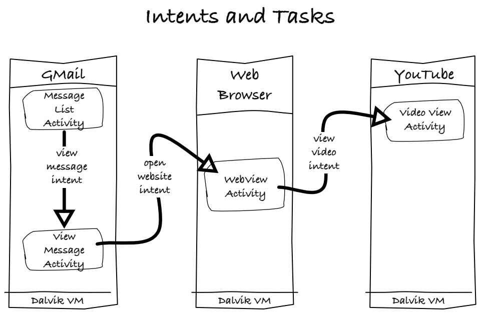

Figure 12: Tasks

A *task* is a sequence of activities a user follows while performing some action.

- Launching the main activity of an app from the Home application launcher starts a new task if no existing task exists associated with that app.
- Whenever the user starts a new activity within a task, the system adds the activity to the top of the task's *back stack*.
- A task can include activities from multiple applications.
- By default, multiple instances of the same `Activity` subclass can exist in the same task or in multiple tasks.
- The default behavior of the "Back" button is to destroy the top activity on the back start, bringing the previous activity on the stack to the foreground.

If the user presses the "Home" button, the system saves the entire task's back stack.

- If the user later resumes the task by selecting the launcher icon that began the task, the task comes to the foreground and resumes the activity at the top of the stack.

It's possible to override the default behavior of tasks. For example, you can indicate that launching a particular activity always starts a new task. For more information on tasks and managing the back stack, see the *Developer's Guide* section on "Tasks and Back Stack".

3.7 Topic Summary

You should now be able to:

- Design the high-level architecture of an Android application using the primary Android application components

4 Module Summary

You should now be able to:

- List the four levels of the Android technology stack and describe the principal characteristics and responsibilities of each level
- Design the high-level architecture of an Android application using the primary Android application components
- Select appropriate Android message types to activate application components and pass information between them