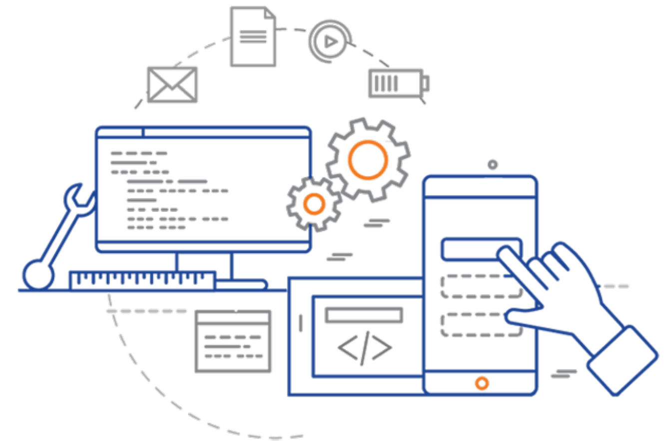


CS 398: Application Development

Building Android Applications

Features; View framework; Layouts.

Features

Mobile Applications

Although we had mobile devices in the 90s and early 2000s, the launch of the iPhone in 2007 really launched the smartphone era.

iOS and Android have become the dominant mobile operating systems.

Features

Smartphones are optimized as portable devices, for ad hoc interaction.

1. Mobile applications use multi-touch as a primary input mechanism. Keyboard input is secondary to touch. No assumption of physical buttons.
2. Run a single foreground application at a time.
3. Applications normally run full-screen, and cannot be resized or moved.
4. Restrictions on what applications can do
   - Low-memory
   - Typically are paused if not in the foreground

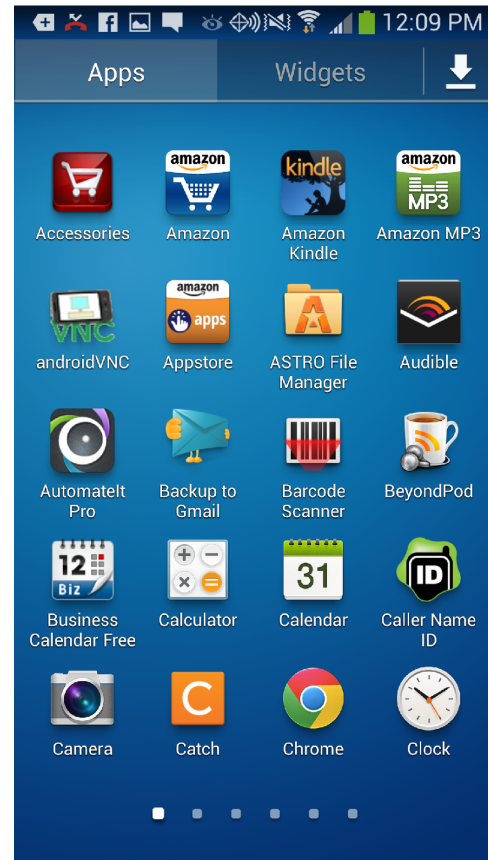
Interaction

Interactive graphical elements: window contents are a combination of txt, images, and interactive elements.

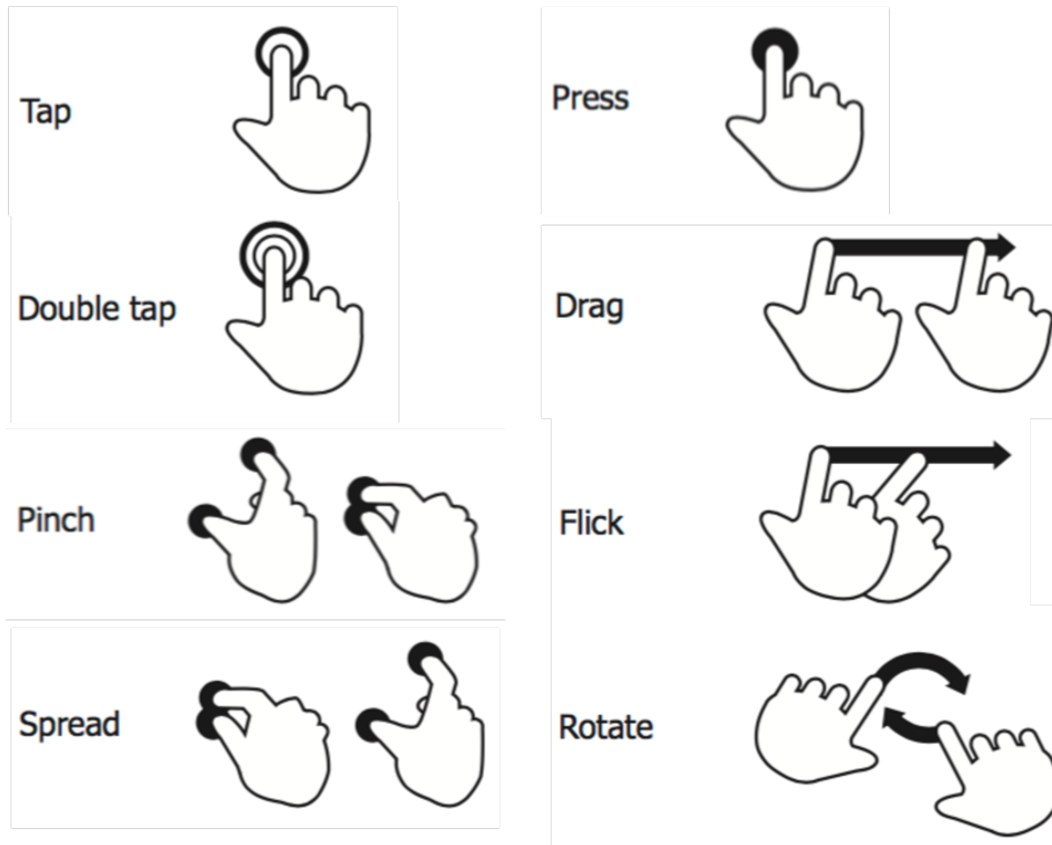
Mobile applications tend to have fewer controls or on-screen widgets compared to desktop.

Interaction is typically by gestures (touch and swipe on regions of the screen). Direct manipulation is emphasized.

Challenges? Screen size and difficulty interacting with small elements by touch.



Gestures



Toolkits

In a desktop OS, we might have a widget or GUI toolkit to provide advanced features for building applications (e.g. creating and managing application windows, providing reusable widgets like buttons, lists, toolbars.

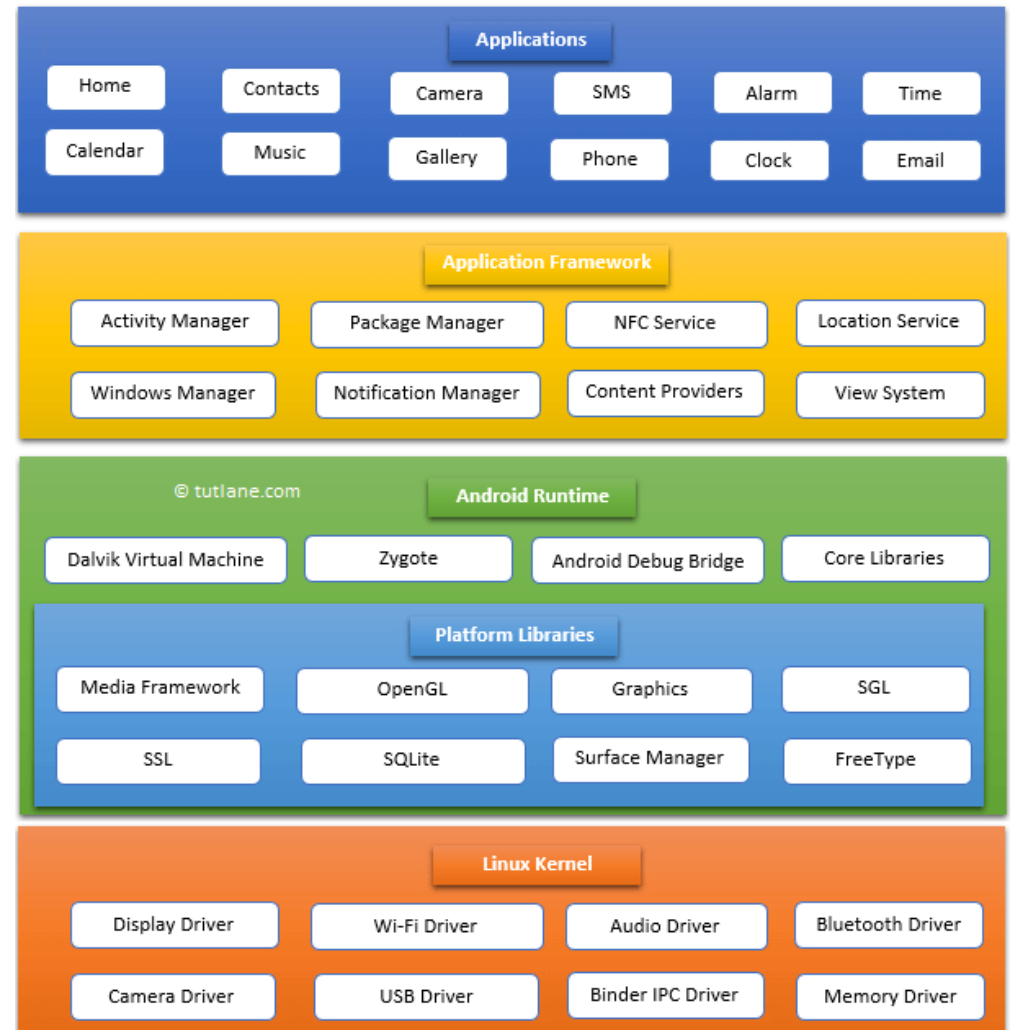
Android has two toolkits both provided by Google:

- **Views and ViewGroups**: the default imperative toolkit.
- **JetPack Compose**: a new innovative declarative toolkit.

Architecture

Android is an open-source, Linux based operating system designed to run across a variety of devices and form-factors.

It's an example of a layered architecture, which increasing levels of abstraction as we move from the low-level hardware to higher-level application APIs. Mid-level components exist to provide services to components further up the stack.



API

The entire feature-set of the Android OS is available through APIs written in Java and/or Kotlin[4]. These APIs form the building blocks you need to create Android apps by providing critical services: :

- A rich and extensible View System you can use to build an app's UI, including lists, grids, text boxes, buttons, and even an embeddable web browser
- A Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files
- A Notification Manager that enables all apps to display custom alerts in the status bar
- An Activity Manager that manages the lifecycle of apps and provides a common navigation back stack
- Content Providers that enable apps to access data from other apps, such as the Contacts app, or to share their own data

Components

There are four different types of core components that can be created in Android. Each represents a different style of application, with a different entry point and lifecycle.

These four component types exist in Android:

- An Activity is an Android class that represent a single screen. It handles drawing the user interface (UI) and managing input events. An application may include multiple activities, where one is the “entry point”.
- A Service is a general-purpose background service, representing some long-running operation that the OS should perform, which does not require a user-interface. e.g. a music playback service.
- Broadcast Receivers: A service that can launch itself in response to a system event, without the need to stay running in the background like a regular service. e.g. an application to pop up a reminder when the user arrives at a destination.
- Content Providers managed shared information that other services or applications can access. e.g. a shared contact database.

Activities

Activities are the most common type of component, since they include user interfaces and visible components. One activity will be the “main” activity that represents the entry point when your application launches.

The system uses the information in the `AndroidManifest.xml` to determine which activity to launch, and how to launch it.

```
<activity
          android:name=".MainActivity"
          android:label="@string/app_name"
          android:theme="@style/Theme.AndroidSandbox.NoActionBar">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Main Activity

The MainActivity is a class that extends `AppCompatActivity`. This is a base class that supports all modern Android features while providing backward compatibility with older versions of Android. For compatibility with older version of Android, you should always use AppCompatActivity as a base class.

Our base class contains a number of methods. The `onCreate()` method is the first method that is called when the MainActivity is instantiated. Here's a basic `onCreate()` method:

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main) // this inflates the activity_main
    // ...
  }
```

Layouts

Activities typically have an associated layout file which describes their appearance.

The activity and the layout are connected by a process known as **layout inflation**. When the activity starts, the views that are defined in the XML layout files are turned into (or "inflated" into) Kotlin view objects in memory. Once this happens, the activity can draw these objects to the screen and also dynamically modify them.

`R.layout.activity_main` in this example corresponds to the `layout/activity_main.xml` file

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">

    <com.google.android.material.appbar.AppBarLayout
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:theme="@style/Theme.AndroidSandbox.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                android:background="?attr/colorPrimary"
                app:popupTheme=
                   "@style/Theme.AndroidSandbox.PopupOverlay"/>

    </com.google.android.material.appbar.AppBarLayout>

    <include layout="@layout/content_main"/>

    <com.google.android.material.floatingactionbutton.FloatingActionButton
            android:id="@+id/fab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="bottom|end"
            android:layout_margin="@dimen/fab_margin"
            app:srcCompat="@android:drawable/ic_dialog_email"/>

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

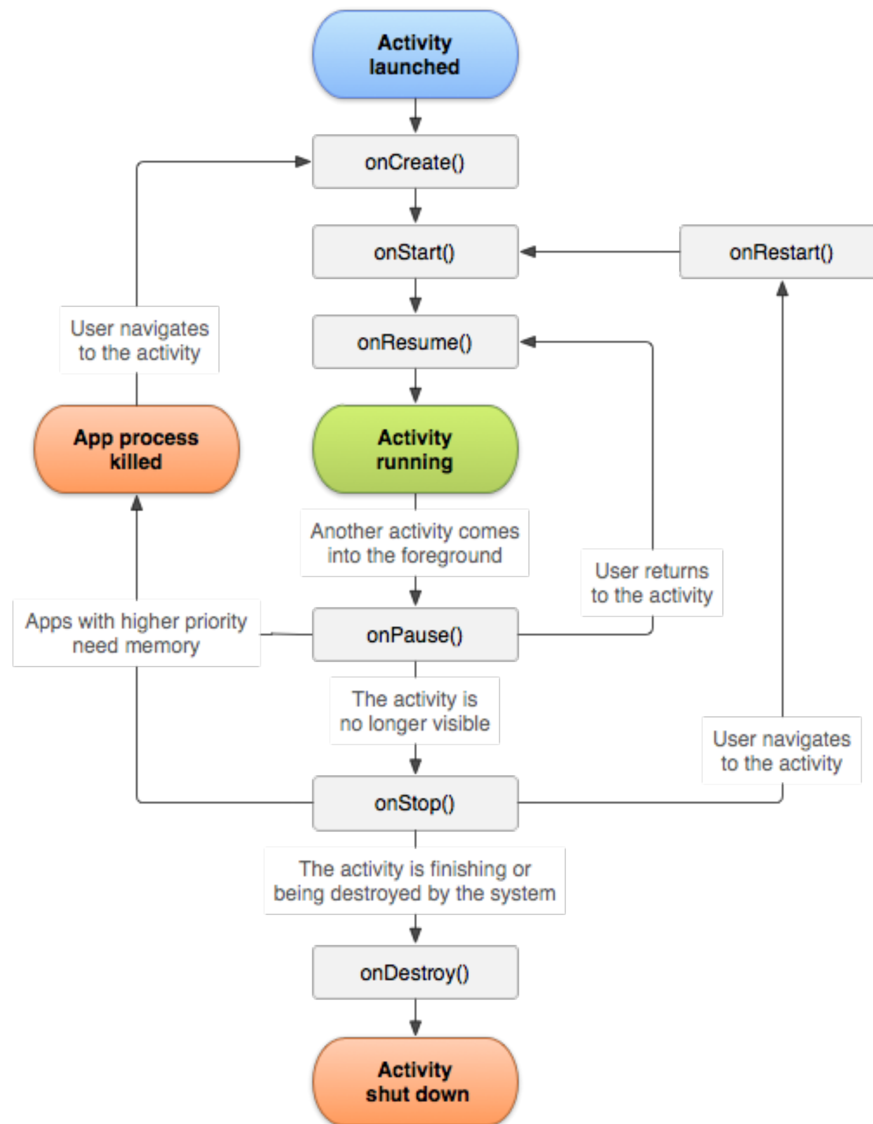
Activity Lifecycle

Applications consist of one or more running activities, each one corresponding to a screen.

Activities in the system are managed as activity stacks. When a new activity is started, it is usually placed on the top of the current stack and becomes the running activity – the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.

An activity can be one of the following running states:

- The activity in the foreground, typically the one that user is able to interact with, is running.
- An activity that has lost focus but can still be seen is visible. It will remain active.
- An activity that is completely hidden, or minimized is stopped. It retains its state (it's basically paused) BUT the OS may choose to terminate it to free up resources.
- The OS can choose to destroy an application to free up resources.


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Activity Lifecycle

There are three key loops that these phases attempt to capture:

- **The entire lifetime of an activity** happens between the first call to `onCreate(Bundle)` through to a single final call to `onDestroy()`. An activity will do all setup of "global" state in `onCreate()`, and release all remaining resources in `onDestroy()`.
- **The visible lifetime of an activity** happens between a call to `onStart()` until a corresponding call to `onStop()`. During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user.
- **The foreground lifetime of an activity** happens between a call to `onResume()` until a corresponding call to `onPause()`. During this time the activity is in visible, active and interacting with the user. An activity can frequently go between the resumed and paused states – for example when the device goes to sleep.

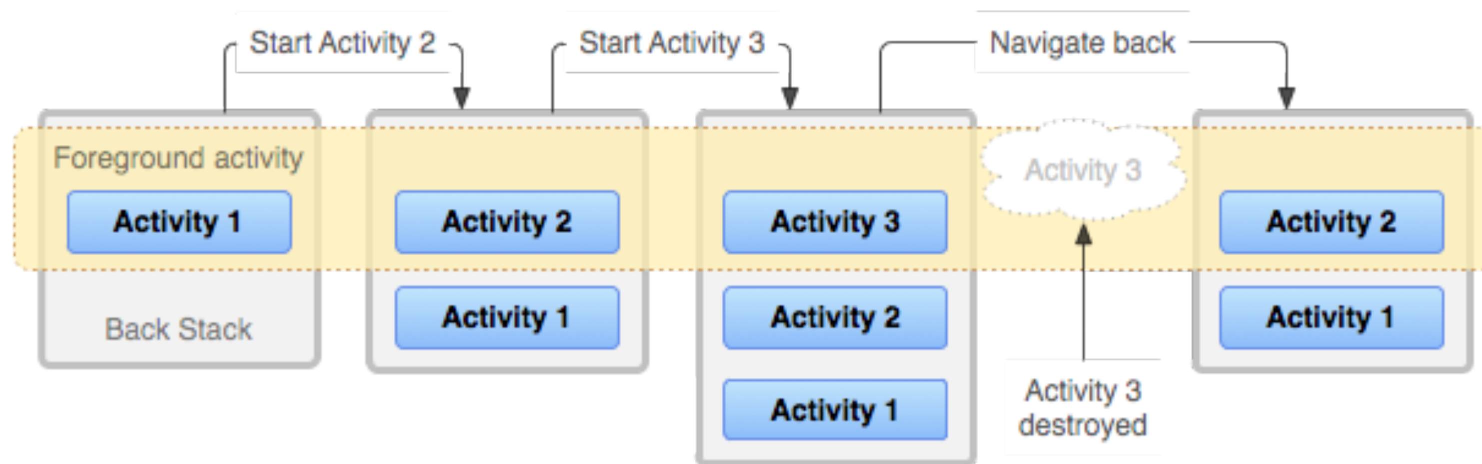
Activity Lifecycle

These phases correspond to the following callback methods. You can override a method in your `Activity` to add code that will get executed when the application enters or exits a particular stage:

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);
    protected void onStart();
    protected void onRestart();
    protected void onResume();
    protected void onPause();
    protected void onStop();
    protected void onDestroy();
}
```

Activity Stack

A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack—the back stack)—in the order in which each activity is opened. This new activity is added to the back stack. If the user presses the Back button, that new activity is finished and popped off the stack.



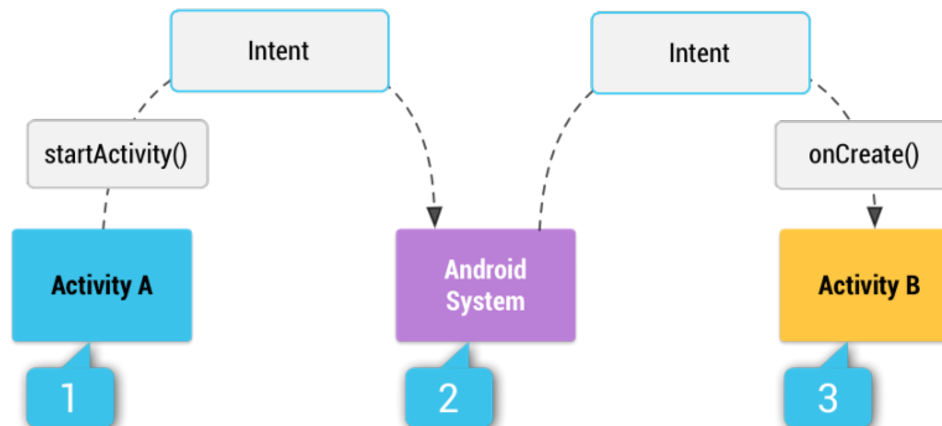
Intents

An **intent** is an asynchronous message, that represents an an operation to be performed. This can include activating components, or activities. An intent is created with an Intent object, which defines a message to activate either a specific component (explicit intent) or a specific type of component (implicit intent).

Similarly, intents can be used to activate an Activity. The `startActivity(Intent)` method is used to start a new activity, which will be placed at the top of the activity stack. It takes a single argument, an `Intent`, which describes the activity to be executed.

Intents to Launch an Activity

Use *startActivity()* method to launch an activity with an intent.



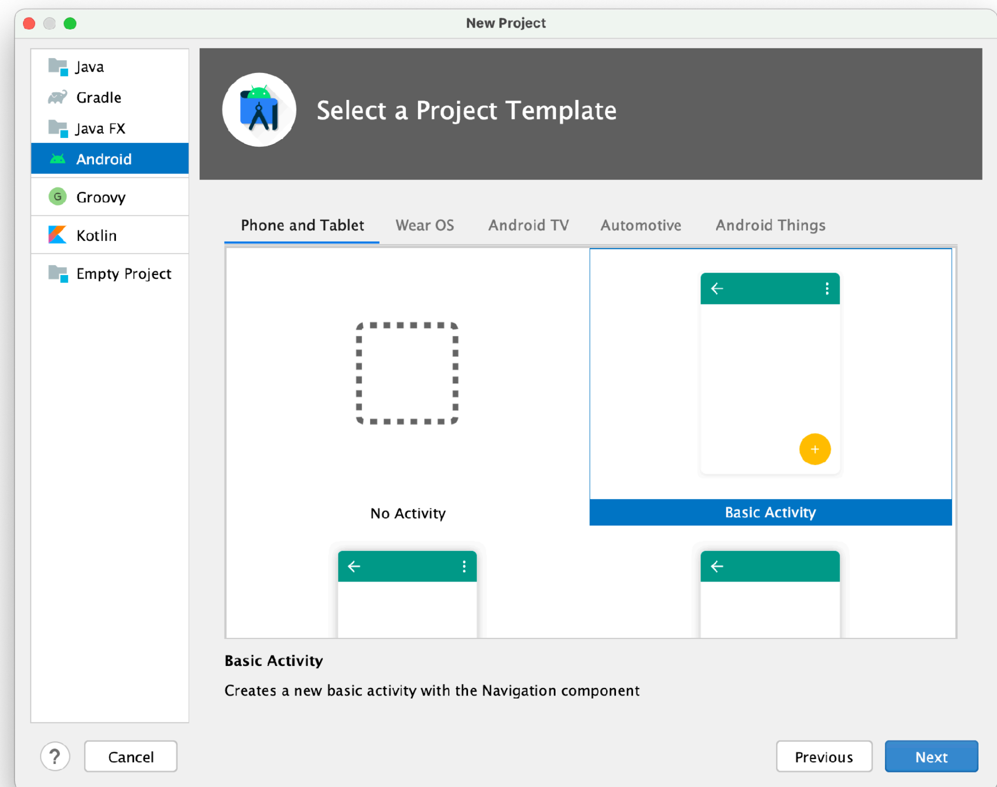
Building Applications

Creating a Project

Android uses Gradle projects. You can create an Android project in IntelliJ IDEA or Android Studio.

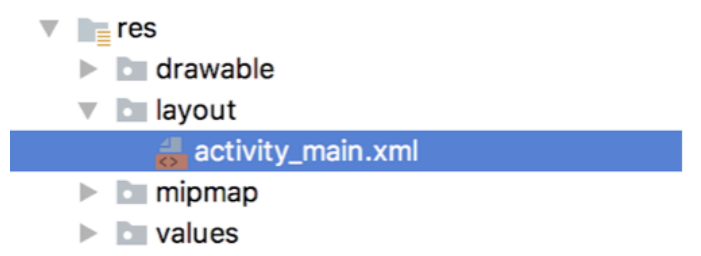
Most project templates will create a simple project with one activity.

The project structure will match a typical gradle project, with `source` and `res` forders.



Project Structures

The res folder contains resources for your project: sounds, images and other useful files that aren't source code.



The following subfolders are used:

- The **drawable** folder contains images that you wish to draw on-screen (directly, or on a widget). It also contains default icons for your application.
- Under the **layout** folder, we have XML files represent a screen layout. These can be loaded dynamically to instantiate a screen at runtime.
- The **values** folder contains XML files with constants: colours, themes, titles etc.

AndroidManifest.xml

The `AndroidManifest.xml` file is generated with your project; one is reproduced below. This file contains settings that tell the application how to present itself on Android e.g. icon, label, theme. The activity element tells is which class to launch when the application launches. Other permissions and settings can be added in here as-needed.

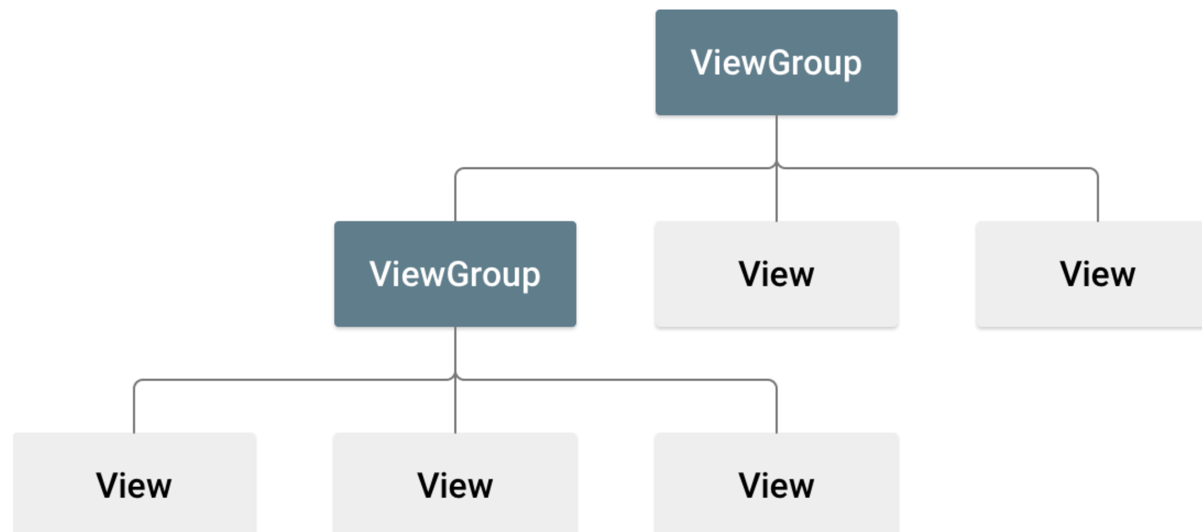
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="net.codebot">

    <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
```

Views and ViewGroups

Android provides a set of prebuilt widgets, called **Views**.

All elements in the user interface are built using a hierarchy of **View** and **ViewGroup** objects. A View usually draws something the user can see and interact with. A ViewGroup is a container that defines the layout structure.



Layout

The **ViewGroup** objects are usually called “layouts” can be one of many types that provide a different layout structure, such as `LinearLayout` or `RelativeLayout`.

You can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
  You can also use IntelliJ or Android Studio’s Layout Editor to build your XML layout using a drag-and-drop interface. This will generate the XML file for you.
- **Instantiate layout elements at runtime**. Your app can instantiate View and ViewGroup objects (and manipulate their properties) programmatically.

Layout Classes

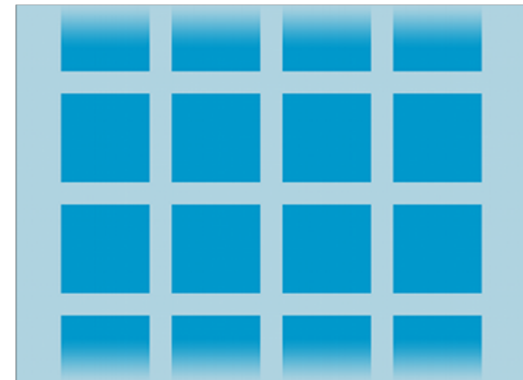
Each subclass of the *ViewGroup* class provides a unique way to display the views you nest within it.

Linear Layout

A layout that organizes its children into a single horizontal or vertical row

Relative Layout

Enables us to specify the location of child objects relative to each other or to the parent.



Grid View

Displays items in a two-dimensional, scrollable grid

Widgets

Widgets are contained in the `android.view.widget package`. Notable widgets include `TextView`, `EditText`, `RadioButton`, `CheckBox`, `Spinners` and others.

The widget can be included in the XML layout file, or instantiated directly in code.

You can also set properties in the XML file or in code.

Properties: Background color, text, font, alignment, size, padding, margin, etc

Event Listeners and Handlers: respond to various events such as: click, long-click, focus change, etc.

Set focus: Set focus on a specific view `requestFocus()` or use XML tag <requestFocus />

Visibility: Hide or show views using `setVisibility()`.

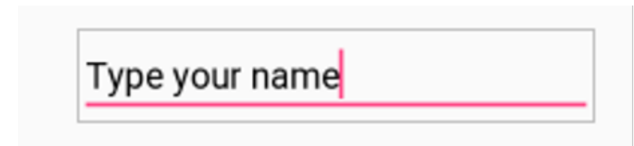
Example: Text View

```
<TextView
    android:id="@+id/txtHello"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />
```

```
    var helloTextView = findViewById(R.id.txtHello) as TextView
    helloTextView.text = "CS349 W19"
```

Example: EditText

```
<EditText
    android:id="@+id/name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:inputType="textPersonName"
    android:text="@string/name" >
    <requestFocus/>
 <EditText/>
```




```
val nameView = findViewById(R.id.name) as EditText
val name = nameView.getText()
```

Layout Files

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Loading a Layout File

When compiled, each layout file is compiled into a `View` resource that can be dynamically loaded.

In your Activity's `onCreate()` method, you should call `setContentView()` to load your starting view. Layout can be changed at anytime by calling `setContentView()` with the new view's ID.

```
fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_layout)
}
```

Events

Event handlers can be associated with widgets (views), much as they are with other toolkits, but are modified to reflect touch interaction over mouse interaction.

For example, we can attach a clickListener to a widget programatically.

```
val button = findViewById(R.id.btnAlarm) as Button
button.setOnClickListener(event -> {
  public void onClick(View v) {
    // Do something in response to button click
  }
})
```

Events

We can also define handlers as layout properties (i.e. in XML layout files).

```
<Button android:id="@+id/btnAlarm"
        android:onClick="sendMessage"/>

// handler function always has view parameter
fun sendMessage(view: View) {
    // Do something in response to button click
}
```

Example: Events

Option 1: Listeners

```
val button = findViewById(R.id.btnAlarm) as Button
button.setOnClickListener(event -> {
    public void onClick(View v) {
        // Do something in response to button click
    }
})
.
```

Option 2: Layout

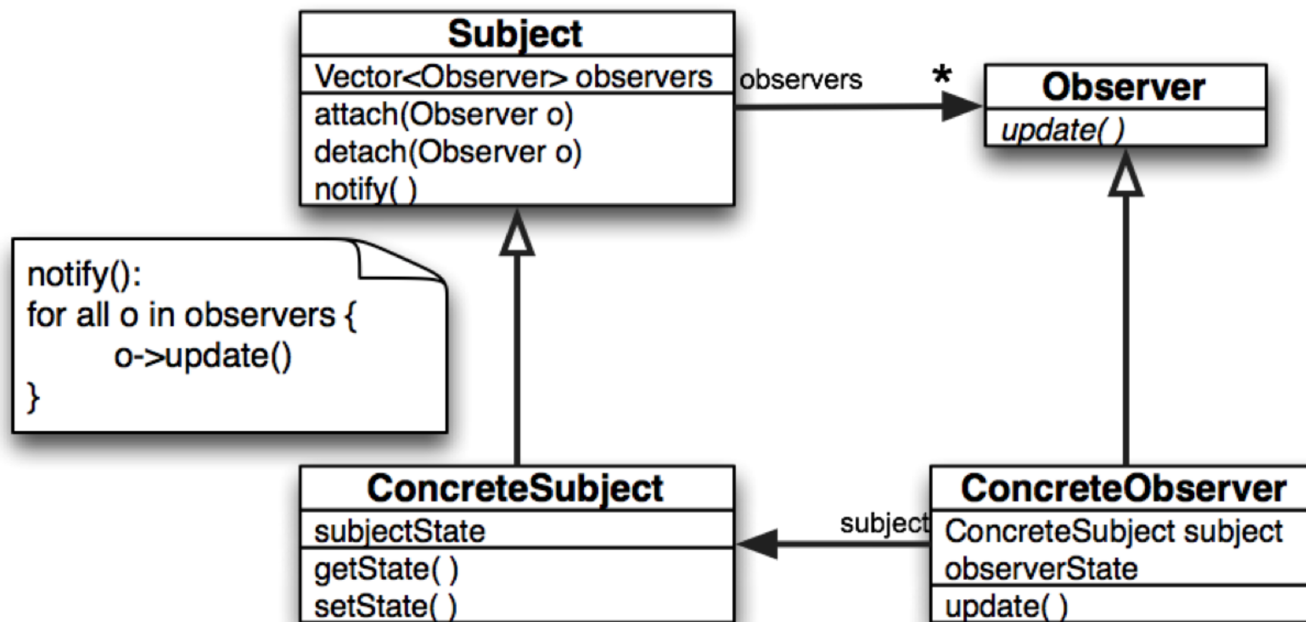
```
<Button
    android:id="@+id/btnAlarm"
    ……
    android:onClick="sendMessage"/>

/** Called in activity when the user touches the button */
fun sendMessage(view: View) {
    // Do something in response to button click
}
```

Structure

MVC Pattern

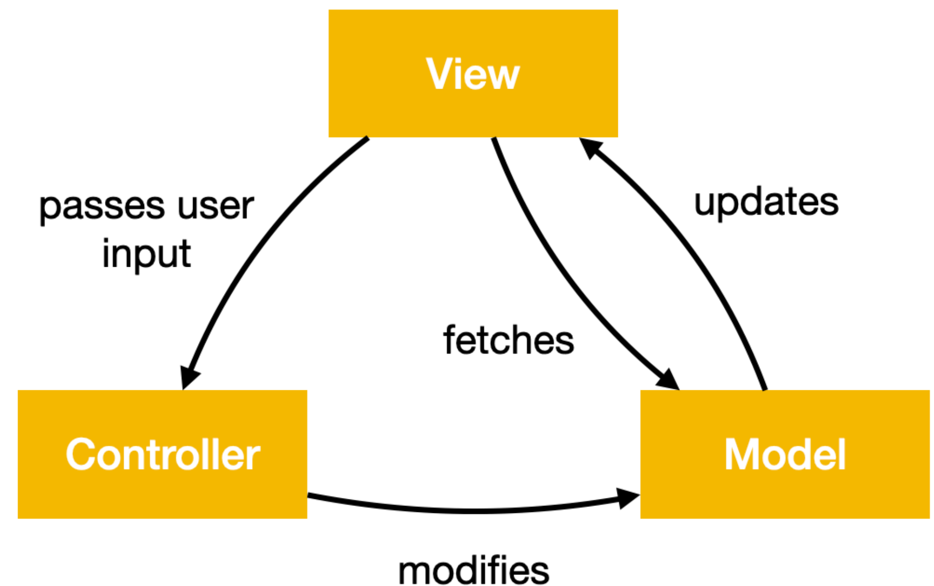
We often choose to model applications using the MVC pattern, which is a specific instance of the observer pattern.



MVC Pattern

MVC divides an application into three distinct parts:

- **Model**: the core component of the application that handles state.
- **View**: a representation of the application state, often as a user-interface ("presentation")
- **Controller**: a component that accepts input, interprets user actions and converts to commands for the model or view ("business logic").



```kotlin
class Main {
  val model = Model()
  val controller = Controller(model)
  val view = View(controller, model)
    model.addView(model)
}
```

```kotlin
class Controller(val model: Model) {
  fun handle(event: Event) {
    // pass event data to model
  }
}
```

```kotlin
class Model {
  val views = listOf()
  fun addView(view: IView) {
    views.add(view)
  }
  fun update() {
    for (view : views) {
      view.update()
    }
  }
}
```

```kotlin
interface IView {
  fun update()
}

class View(
  val controller: Controller, val model: Model): IView
  {
  override fun update() {
    // fetch data from model
  }
}
```