Mobile Applications (420-P84-AB)

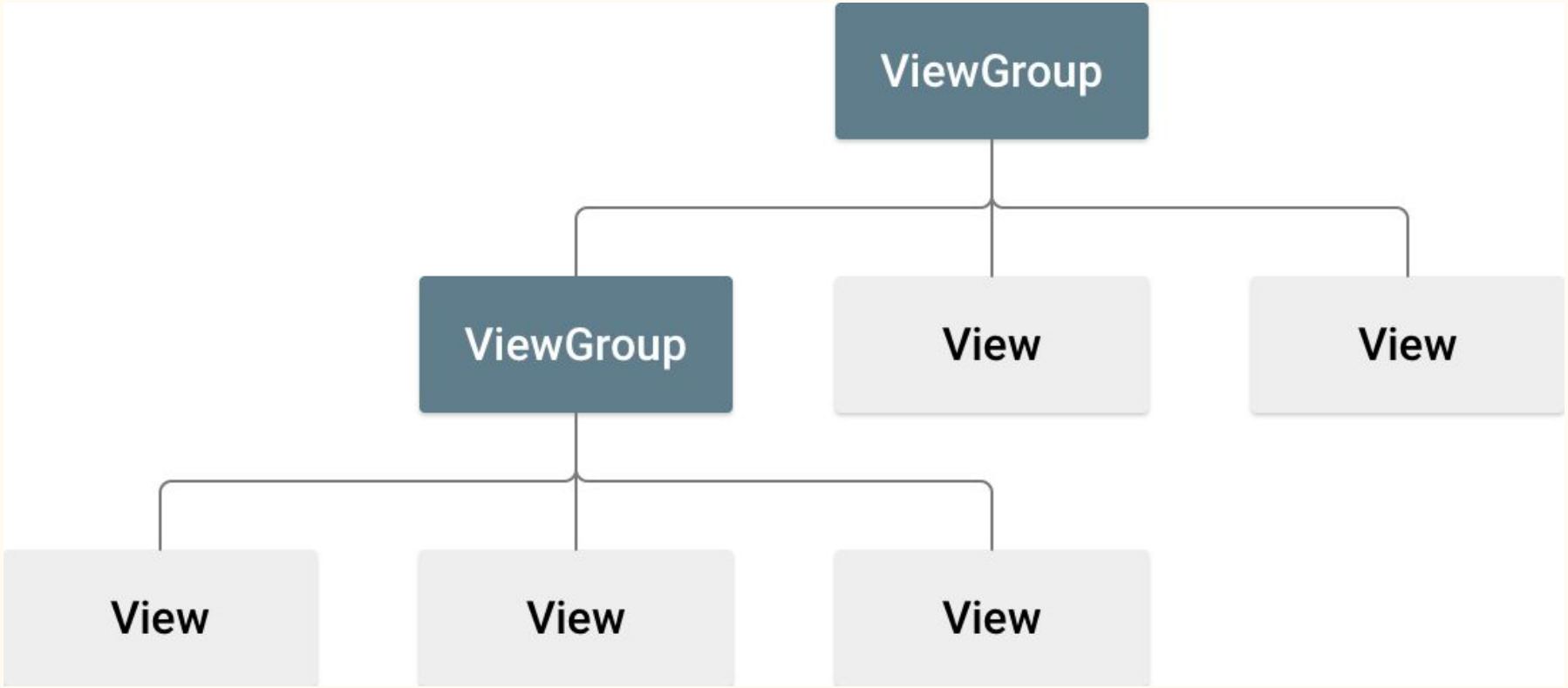
John Abbott College

Layouts

Aakash Malhotra

Overview

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of `View` and `ViewGroup` objects. A `View` usually draws something the user can see and interact with. Whereas a `ViewGroup` is an invisible container that defines the layout structure for `View` and other `ViewGroup` objects, as shown in figure on next slide.


ViewGroup
ViewGroup
View
View
View
View
View

Overview

The `View` objects are usually called "widgets" and can be one of many subclasses, such as `Button` or `TextView`. The `ViewGroup` objects are usually called "layouts" can be one of many types that provide a different layout structure, such as `LinearLayout` or `ConstraintLayout`.

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 Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.
- **Instantiate layout elements at runtime**. Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

Overview

Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations.

The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical `LinearLayout` to hold a `TextView` and a `Button`:

Write the XML

```
<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>
```

Write the XML

After you've declared your layout in XML, save the file with the `.xml` extension, in your Android project's `res/layout/` directory, so it will properly compile.

Load XML Resources

When you compile your app, each XML layout file is compiled into a `View` resource. You should load the layout resource from your app code, in your `Activity.onCreate()` callback implementation. Do so by calling `setContentView()`, passing it the reference to your layout resource in the form of:`R.layout.layout_file_name`. For example, if your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched

Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the `textSize` attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the `id` attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the `id` attribute. This is an XML attribute common to all View objects (defined by the `View` class) and you will use it very often. The syntax for an ID, inside an XML tag is:

Attributes

```
android:id="@+id/my_button"
```

The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the `R.java` file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the `android` package namespace, like so:

```
android:id="@android:id/empty"
```

With the `android` package namespace in place, we're now referencing an ID from the `android.R` resources class, rather than the local resources class.

Attributes

In order to create views and reference them from the app, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

2. Then create an instance of the view object and capture it from the layout (typically in the `onCreate()` method):

```
Button myButton = (Button) findViewById(R.id.my_button);
```

Attributes

Defining IDs for view objects is important when creating a `RelativeLayout`. In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

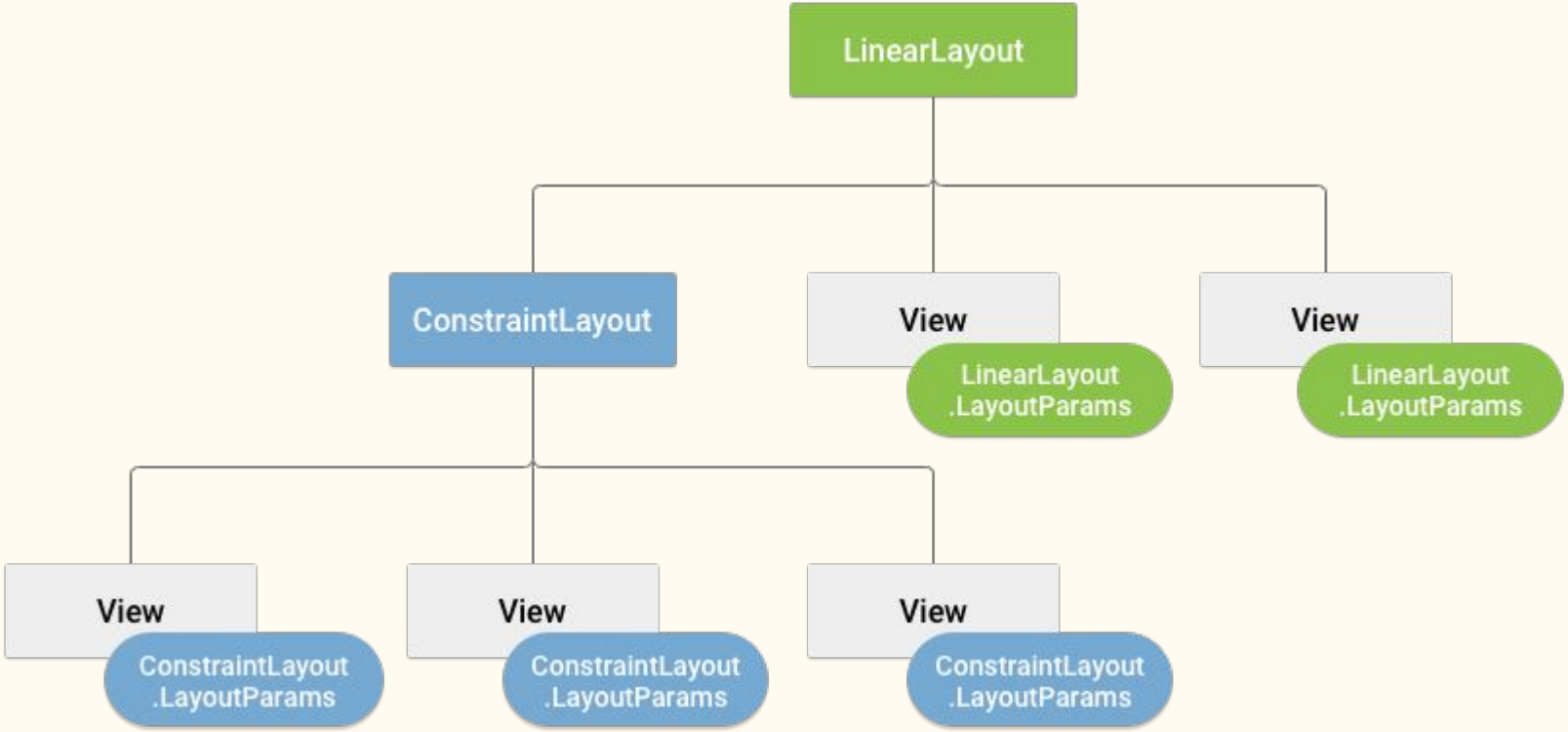
An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

Layout Parameters

XML layout attributes named `layout_something` define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends `ViewGroup.LayoutParams`. This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure below,, the parent view group defines layout parameters for each child view (including the child view group).

Layout Parameters



Layout Parameters

Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (`layout_width` and `layout_height`), and each view is required to define them. Many LayoutParams also include optional margins and borders.

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

- **wrap_content** tells your view to size itself to the dimensions required by its content.
- **match_parent** tells your view to become as big as its parent view group will allow.

Layout Parameters

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (**dp**), **wrap_content**, or **match_parent**, is a better approach, because it helps ensure that your app will display properly across a variety of device screen sizes. The accepted measurement types are defined in the Available Resources document.

Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of *left* and *top* coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods `getLeft()` and `getTop()`. The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when `getLeft()` returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely `getRight()` and `getBottom()`. These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling `getRight()` is similar to the following computation: `getLeft() + getWidth()`.

Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possess two pairs of width and height values.

The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling `getMeasuredWidth()` and `getMeasuredHeight()`.

The second pair is simply known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling `getWidth()` and `getHeight()`.

Size, Padding and Margins

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the `setPadding(int, int, int, int)` method and queried by calling `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` and `getPaddingBottom()`.

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support. Refer to `ViewGroup` and `ViewGroup.MarginLayoutParams` for further information.

Common Layouts

Each subclass of the `ViewGroup` class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

Note: Although you can nest one or more layouts within another layout to achieve your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).

- Linear Layout
- Relative Layout
- Web View
- Constraint Layout

Linear Layout

`LinearLayout` is a view group that aligns all children in a single direction, vertically or horizontally. You can specify the layout direction with the `android:orientation` attribute.

All children of a `LinearLayout` are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). A `LinearLayout` respects *margin*s between children and the *gravity* (right, center, or left alignment) of each child.

Layout Weight

Equally weighted children

To create a linear layout in which each child uses the same amount of space on the screen, set the `android:layout_height` of each view to `"0dp"` (for a vertical layout) or the `android:layout_width` of each view to `"0dp"` (for a horizontal layout). Then set the `android:layout_weight` of each view to `"1"`.

Layout Weight

`LinearLayout` also supports assigning a *weight* to individual children with the `android:layout_weight` attribute. This attribute assigns an "importance" value to a view in terms of how much space it should occupy on the screen. A larger weight value allows it to expand to fill any remaining space in the parent view. Child views can specify a weight value, and then any remaining space in the view group is assigned to children in the proportion of their declared weight. Default weight is zero.

For example, if there are three text fields and two of them declare a weight of 1, while the other is given no weight, the third text field without weight will not grow and will only occupy the area required by its content. The other two will expand equally to fill the space remaining after all three fields are measured. If the third field is then given a weight of 2 (instead of 0), then it is now declared more important than both the others, so it gets half the total remaining space, while the first two share the rest equally.

Example

```
<?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:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
```

Example

```
<EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
```

Example

```
<Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```

Relative Layout

`RelativeLayout` is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent `RelativeLayout` area (such as aligned to the bottom, left or center).

A `RelativeLayout` is a very powerful utility for designing a user interface because it can eliminate nested view groups and keep your layout hierarchy flat, which improves performance. If you find yourself using several nested `LinearLayout` groups, you may be able to replace them with a single `RelativeLayout`.

Relative Layout - Positioning Views

`RelativeLayout` lets child views specify their position relative to the parent view or to each other (specified by ID). So you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on. By default, all child views are drawn at the top-left of the layout, so you must define the position of each view using the various layout properties available from `RelativeLayout.LayoutParams`.

Some of the many layout properties available to views in a `RelativeLayout` include:

`android:layout_alignParentTop`

> If `"true"`, makes the top edge of this view match the top edge of the parent.

Relative Layout - Positioning Views

`android:layout_centerVertical`

If `"true"`, centers this child vertically within its parent.

`android:layout_below`

Positions the top edge of this view below the view specified with a resource ID.

`android:layout_toRightOf`

Positions the left edge of this view to the right of the view specified with a resource ID.

Relative Layout - Positioning Views

These are just a few examples. All layout attributes are documented at `RelativeLayout.LayoutParams`.

The value for each layout property is either a boolean to enable a layout position relative to the parent `RelativeLayout` or an ID that references another view in the layout against which the view should be positioned.

In your XML layout, dependencies against other views in the layout can be declared in any order. For example, you can declare that "view1" be positioned below "view2" even if "view2" is the last view declared in the hierarchy. The example below demonstrates such a scenario.

Relative Layout - Example

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
```

Relative Layout - Example

```
    <Spinner
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
```

Relative Layout - Example

```
<Spinner
        android:id="@id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
```

Relative Layout - Example

```
<Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```

Web View

If you want to deliver a web application (or just a web page) as a part of a client application, you can do it using `WebView`. The `WebView` class is an extension of Android's `View` class that allows you to display web pages as a part of your activity layout. It does *not* include any features of a fully developed web browser, such as navigation controls or an address bar. All that `WebView` does, by default, is show a web page.

A common scenario in which using `WebView` is helpful is when you want to provide information in your app that you might need to update, such as an end-user agreement or a user guide. Within your Android app, you can create an `Activity` that contains a `WebView`, then use that to display your document that's hosted online.

Web View

Another scenario in which `WebView` can help is if your app provides data to the user that always requires an Internet connection to retrieve data, such as email. In this case, you might find that it's easier to build a `WebView` in your Android app that shows a web page with all the user data, rather than performing a network request, then parsing the data and rendering it in an Android layout. Instead, you can design a web page that's tailored for Android devices and then implement a `WebView` in your Android app that loads the web page.

This document shows you how to get started with `WebView` and how to do some additional things, such as handle page navigation and bind JavaScript from your web page to client-side code in your Android app.

Adding a Web View

To add a `WebView` to your app, you can either include the `<WebView>` element in your activity layout, or set the entire Activity window as a `WebView` in `onCreate()`.

Adding a WebView in the activity layout

To add a `WebView` to your app in the layout, add the following code to your activity's layout XML file:

Adding a Web View

```
<WebView
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
```

To load a web page in the `WebView`, use `loadUrl()`

Adding a Web View

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.loadUrl("http://www.example.com");
```

Adding a Web View in onCreate

To add a `WebView` to your app in an activity's `onCreate()` method instead, use logic similar to the following:

```
WebView myWebView = new WebView(activityContext);
setContentView(myWebView);
```

Then load the page with:

```
myWebView.loadUrl("https://www.example.com");
```

Adding a Web View

Before this will work, however, your app must have access to the Internet. To get Internet access, request the `INTERNET` permission in your manifest file. For example:

```
<manifest ... >
    <uses-permission android:name="android.permission.INTERNET" />
    ...
</manifest>
```

More with Web View

That's all you need for a basic `WebView` that displays a web page. Additionally, you can customize your `WebView` by modifying the following:

- Enabling fullscreen support with `WebChromeClient`. This class is also called when a `WebView` needs permission to alter the host app's UI, such as creating or closing windows and sending JavaScript dialogs to the user.
- Handling events that impact content rendering, such as errors on form submissions or navigation with `WebViewClient`. You can also use this subclass to intercept URL loading.
- Enabling JavaScript by modifying `WebSettings`.
- Using JavaScript to access Android framework objects that you have injected into a `WebView`.

Using JavaScript in WebView

If the web page you plan to load in your `WebView` uses JavaScript, you must enable JavaScript for your `WebView`. Once JavaScript is enabled, you can also create interfaces between your app code and your JavaScript code.

Enabling JavaScript

JavaScript is disabled in a `WebView` by default. You can enable it through the `WebSettings` attached to your `WebView`.You can retrieve `WebSettings` with `getSettings()`, then enable JavaScript with `setJavaScriptEnabled()`.

Using JavaScript in WebView

Example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

`WebSettings` provides access to a variety of other settings that you might find useful. For example, if you're developing a web application that's designed specifically for the `WebView` in your Android app, then you can define a custom user agent string with `setUserAgentString()`, then query the custom user agent in your web page to verify that the client requesting your web page is actually your Android app.

Building layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses `AdapterView` to populate the layout with views at runtime. A subclass of the `AdapterView` class uses an `Adapter` to bind data to its layout. The `Adapter` behaves as a middleman between the data source and the `AdapterView` layout—the `Adapter` retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the `AdapterView` layout.

Building layouts with an Adapter

Common layouts backed by an adapter include:

- List View
- Grid View

You can populate an `AdapterView` such as `ListView` or `GridView` by binding the `AdapterView` instance to an `Adapter`, which retrieves data from an external source and creates a `View` that represents each data entry.

Android provides several subclasses of `Adapter` that are useful for retrieving different kinds of data and building views for an `AdapterView`. The two most common adapters are:

Building layouts with an Adapter

`ArrayAdapter`

Use this adapter when your data source is an array. By default, `ArrayAdapter` creates a view for each array item by calling `toString()` on each item and placing the contents in a `TextView`.

For example, if you have an array of strings you want to display in a `ListView`, initialize a new `ArrayAdapter` using a constructor to specify the layout for each string and the string array:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,

        android.R.layout.simple_list_item_1, myStringArray);
```

Building layouts with an Adapter

The arguments for this constructor are:

- Your app `Context`
- The layout that contains a `TextView` for each string in the array
- The string array

Then simply call `setAdapter()` on your `ListView`:

```
ListView listView = (ListView) findViewById(R.id.listview);

listView.setAdapter(adapter);
```

Building layouts with an Adapter

To customize the appearance of each item you can override the `toString()` method for the objects in your array. Or, to create a view for each item that's something other than a `TextView`(for example, if you want an `ImageView` for each array item), extend the `ArrayAdapter` class and override `getView()` to return the type of view you want for each item.

`SimpleCursorAdapter`

Use this adapter when your data comes from a `Cursor`. When using `SimpleCursorAdapter`, you must specify a layout to use for each row in the `Cursor` and which columns in the `Cursor`should be inserted into which views of the layout. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a `Cursor` containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the `Cursor` you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

Building layouts with an Adapter

```
String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME,
                        ContactsContract.CommonDataKinds.Phone.NUMBER};
int[] toViews = {R.id.display_name, R.id.phone_number};
```

When you instantiate the `SimpleCursorAdapter`, pass the layout to use for each result, the `Cursor` containing the results, and these two arrays:

```
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
        R.layout.person_name_and_number, cursor, fromColumns, toViews, 0);
ListView listView = getListView();
listView.setAdapter(adapter);
```

Building layouts with an Adapter

The `SimpleCursorAdapter` then creates a view for each row in the `Cursor` using the provided layout by inserting each `fromColumns` item into the corresponding `toViews` view.

If, during the course of your app's life, you change the underlying data that is read by your adapter, you should call `notifyDataSetChanged()`. This will notify the attached view that the data has been changed and it should refresh itself.

Handling Click Events

You can respond to click events on each item in an `AdapterView` by implementing the `AdapterView.OnItemClickListener` interface. For example:

```
// Create a message handling object as an anonymous class.
private OnItemClickListener mMessageClickedHandler = new
OnItemClickListener() {
    public void onItemClick(AdapterView parent, View v, int position, long
id) {
        // Do something in response to the click
    }
};

listView.setOnItemClickListener(mMessageClickedHandler);
```

References:

- https://developer.android.com/guide/topics/ui/declaring-layout
- https://developer.android.com/guide/topics/ui/layout/linear
- https://developer.android.com/studio/debug/layout-inspector
- https://developer.android.com/training/multiscreen/screensizes
- https://developer.android.com/guide/topics/ui/layout/relative