- With the possible exception of listening to streaming audio, a user's interaction with an Android device is primarily visual and tactile in nature.

- All of this interaction takes place through the user interfaces of the applications installed on the device, including both the built-in applications and any third party applications installed by the user.

- It should come as no surprise, therefore, that a key element of developing Android applications involves the design and creation of user interfaces.

- Within this chapter, the topic of Android user interface structure will be covered, together with an overview of the different elements that can be brought together to make up a user interface; namely Views, View Groups and Layouts.

Designing for Different Android Devices

- The term "Android device" covers a vast array of tablet and smartphone products with different screen sizes and resolutions.

- As a result, application user interfaces must now be carefully designed to ensure correct presentation on as wide a range of display sizes as possible.

- A key part of this is ensuring that the user interface layouts resize correctly when run on different devices.

- This can largely be achieved through careful planning and the use of the layout managers outlined in this chapter.

- It is also important to keep in mind that the majority of Android based smartphones and tablets can be held by the user in both portrait and landscape orientations.

- A well-designed user interface should be able to adapt to such changes and make sensible layout adjustments to utilize the available screen space in each orientation.

Views and View Groups

- Every item in a user interface is a subclass of the Android `View` class (to be precise `android.view.View`).

- The Android SDK provides a set of pre-built views that can be used to construct a user interface.

- Typical examples include standard items such as the `Button`, `CheckBox`, `ProgressBar` and `TextView` classes.

- Such views are also referred to as widgets or components.

- For requirements that are not met by the widgets supplied with the SDK, new views may be created either by subclassing and extending an existing class, or creating an entirely new component by building directly on top of the `View` class.

- A view can also be comprised of multiple other views (otherwise known as a composite view).
- Such views are subclassed from the Android `ViewGroup` class (`android.view.ViewGroup`) which is itself a subclass of View.
- An example of such a view is the `RadioGroup`, which is intended to contain multiple `RadioButton` objects such that only one can be in the "on" position at any one time.
- In terms of structure, composite views consist of a single parent view (derived from the `ViewGroup` class and otherwise known as a container view or root element) that is capable of containing other views (known as child views).
- Another category of `ViewGroup` based container view is that of the layout manager.

Android Layout Managers

- In addition to the widget style views discussed in the previous section, the SDK also includes a set of views referred to as layouts.
- Layouts are container views (and, therefore, subclassed from `ViewGroup`) designed for the sole purpose of controlling how child views are positioned on the screen.

The Android SDK includes the following layout views that may be used within an Android user interface design:

`ConstraintLayout`

- Introduced in Android 7, use of this layout manager is recommended for most layout requirements. `ConstraintLayout` allows the positioning and behavior of the views in a layout to be defined by simple constraint settings assigned to each child view.
- The flexibility of this layout allows complex layouts to be quickly and easily created without the necessity to nest other layout types inside each other, resulting in improved layout performance.
- `ConstraintLayout` is also tightly integrated into the Android Studio Layout Editor tool.
- Unless otherwise stated, this is the layout of choice for the majority of examples in this book.

`LinearLayout`

- Positions child views in a single row or column depending on the orientation selected.
- A weight value can be set on each child to specify how much of the layout space that child should occupy relative to other children.

`TableLayout`

- Arranges child views into a grid format of rows and columns.

- Each row within a table is represented by a `TableRow` object child, which, in turn, contains a view object for each cell.

`FrameLayout`

- The purpose of the `FrameLayout` is to allocate an area of screen, typically for the purposes of displaying a single view.

- If multiple child views are added they will, by default, appear on top of each other positioned in the top left-hand corner of the layout area.

- Alternate positioning of individual child views can be achieved by setting gravity values on each child.

- For example, setting a `center_vertical` gravity value on a child will cause it to be positioned in the vertical center of the containing `FrameLayout` view.

`RelativeLayout`

- The `RelativeLayout` allows child views to be positioned relative both to each other and the containing layout view through the specification of alignments and margins on child views.

- For example, child `View` A may be configured to be positioned in the vertical and horizontal center of the containing `RelativeLayout` view.

- `View` B, on the other hand, might also be configured to be centered horizontally within the layout view, but positioned 30 pixels above the top edge of `View` A, thereby making the vertical position relative to that of `View` A.

- The `RelativeLayout` manager can be of particular use when designing a user interface that must work on a variety of screen sizes and orientations.

`AbsoluteLayout`

- Allows child views to be positioned at specific X and Y coordinates within the containing layout view.

- Use of this layout is discouraged since it lacks the flexibility to respond to changes in screen size and orientation.

`GridLayout`

- A `GridLayout` instance is divided by invisible lines that form a grid containing rows and columns of cells.

- Child views are then placed in cells and may be configured to cover multiple cells both horizontally and vertically allowing a wide range of layout options to be quickly and easily implemented.

- Gaps between components in a `GridLayout` may be implemented by placing a special type of view called a `Space` view into adjacent cells, or by setting margin parameters.

`CoordinatorLayout`

- Introduced as part of the Android Design Support Library with Android 5.0, the `CoordinatorLayout` is designed specifically for coordinating the appearance and behavior of the app bar across the top of an application screen with other view elements.

- When creating a new activity using the Basic Activity template, the parent view in the main layout will be implemented using a `CoordinatorLayout` instance.

- This layout manager will be covered in greater detail in a later chapter.

- When considering the use of layouts in the user interface for an Android application it is worth keeping in mind that, as will be outlined in the next section, these can be nested within each other to create a user interface design of just about any necessary level of complexity.

The View Hierarchy

- Each view in a user interface represents a rectangular area of the display.

- A view is responsible for what is drawn in that rectangle and for responding to events that occur within that part of the screen (such as a touch event).

- A user interface screen is comprised of a view hierarchy with a root view positioned at the top of the tree and child views positioned on branches below.

- The child of a container view appears on top of its parent view and is constrained to appear within the bounds of the parent view's display area. Consider, for example, the user interface illustrated in Figure 8-1:

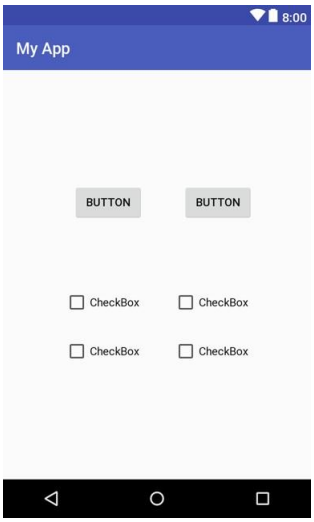

Figure 8-1

- In addition to the visible button and checkbox views, the user interface actually includes a number of layout views that control how the visible views are positioned.

- Figure 8-2 shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

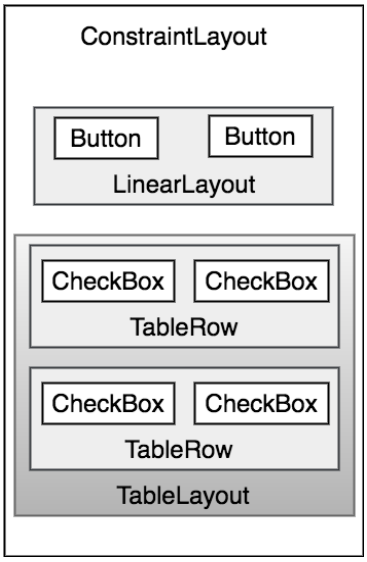

Figure 8-2

- As was was previously discussed, user interfaces are constructed in the form of a view hierarchy with a root view at the top.

- This being the case, we can also visualize the above user interface example in the form of the view tree illustrated in Figure 8-3:

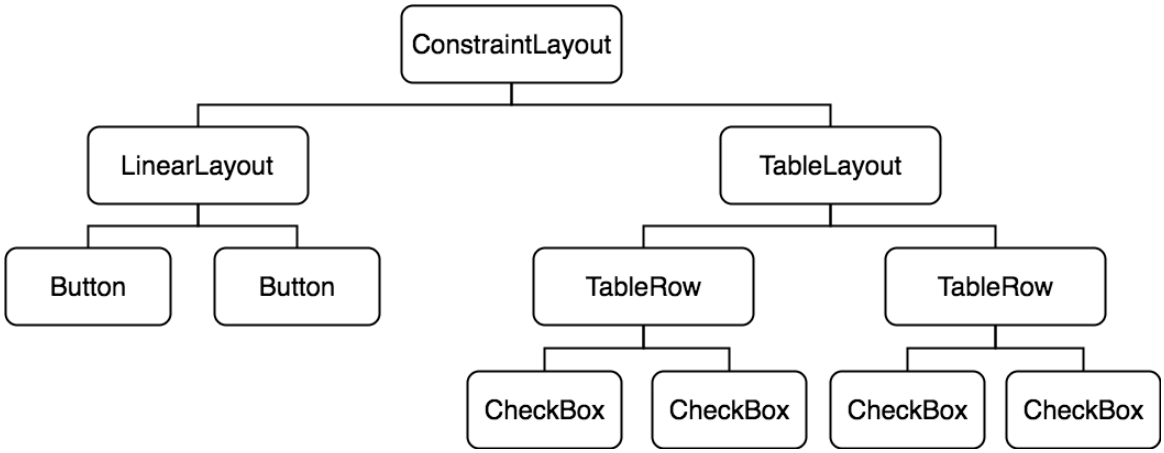

Figure 8-3

- The view hierarchy diagram gives probably the clearest overview of the relationship between the various views that make up the user interface shown in Figure 8-1.

- When a user interface is displayed to the user, the Android runtime walks the view hierarchy, starting at the root view and working down the tree as it renders each view.

Creating User Interfaces

With a clearer understanding of the concepts of views, layouts and the view hierarchy, the following few chapters will focus on the steps involved in creating user interfaces for Android activities.

In fact, there are three different approaches to user interface design: using the Android Studio Layout Editor tool, handwriting XML layout resource files or writing Java code, each of which will be covered.

A Guide to the Android Studio Layout Editor Tool

It is difficult to think of an Android application concept that does not require some form of user interface.

Most Android devices come equipped with a touch screen and keyboard (either virtual or physical) and taps and swipes are the primary form of interaction between the user and application.

Invariably these interactions take place through the application's user interface.

A well designed and implemented user interface, an important factor in creating a successful and popular Android application, can vary from simple to extremely complex, depending on the design requirements of the individual application.

Regardless of the level of complexity, the Android Studio Layout Editor tool significantly simplifies the task of designing and implementing Android user interfaces.

Basic vs. Empty Activity Templates

As outlined before, Android applications are made up of one or more activities.

An activity is a standalone module of application functionality that usually correlates directly to a single user interface screen.

As such, when working with the Android Studio Layout Editor we are invariably working on the layout for an activity.

When creating a new Android Studio project, a number of different templates are available to be used as the starting point for the user interface of the main activity.

The most basic of these templates are the Basic Activity and Empty Activity templates.

Although these seem similar at first glance, there are actually considerable differences between the two options.

To see these differences within the layout editor, use the View Options menu to enable Show System UI as shown in Figure 8-4 below:

(continued)

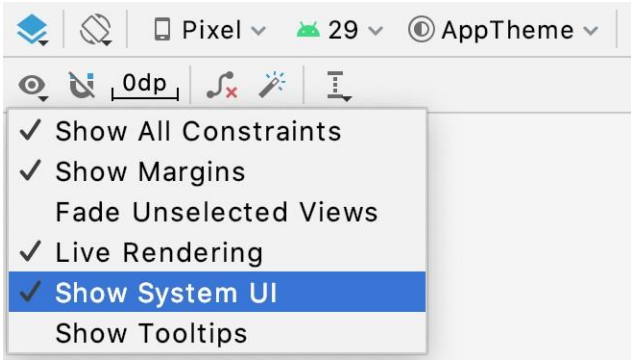

Figure 8-4

- The Empty Activity template creates a single layout file consisting of a `ConstraintLayout` manager instance containing a `TextView` object as shown in Figure 8-5:

Figure 8-5

- The Basic Activity, on the other hand, consists of multiple layout files.

- The top level layout file has a `CoordinatorLayout` as the root view, a configurable app bar (also known as an action bar) that appears across the top of the device screen (marked A in Figure 8-6) and a floating action button (the email button marked B).

- In addition to these items, the `activity_main.xml` layout file contains a reference to a second file named `content_main.xml` containing the content layout (marked C):

(continued)

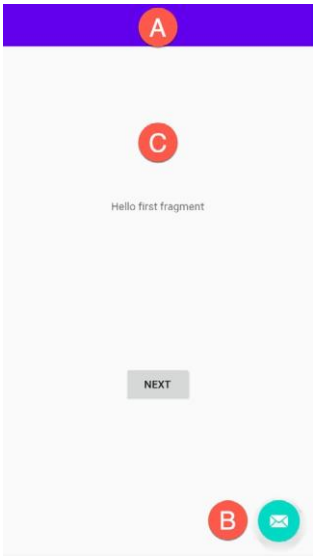

Figure 8-6

- The Basic Activity contains layouts for two screens, both containing a button and a text view.
- The purpose of this template is to demonstrate how to implement navigation between multiple screens within an app.
- If an unmodified app using the Basic Activity template were to be run, the first of these two screens would appear (marked A in Figure 8-7).
- Pressing the Next button, would navigate to the second screen (B) which, in turn, contains a button to return to the first screen:

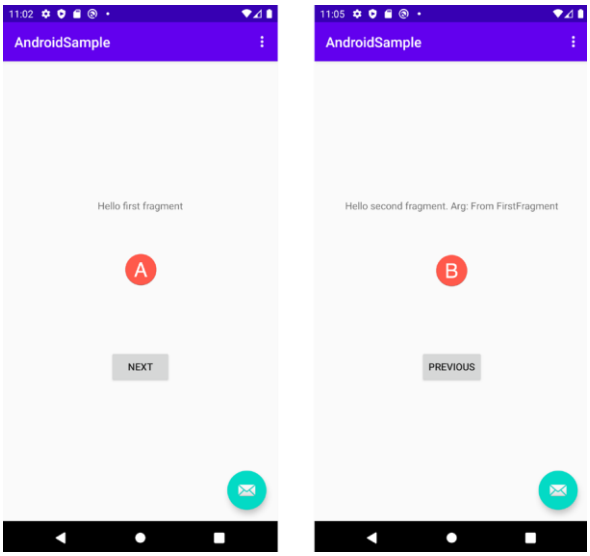

Figure 8-7

- This app behavior makes use of two Android features referred to as fragments and navigation, both of which will be covered later.

- The `content_main.xml` file contains a special fragment known as a Navigation Host Fragment which allows different content to be switched in and out of view depending on the settings configured in the res > layout nav_graph.xml file.

- In the case of the Basic Activity template, the `nav_graph.xml` file is configured to switch between the user interface layouts defined in the `fragment_first.xml` and `fragment_second.xml` files based on the Next and Previous button selections made by the user.

- Clearly the Empty Activity template is useful if you need neither a floating action button nor a menu in your activity and do not need the special app bar behavior provided by the `CoordinatorLayout` such as options to make the app bar and toolbar collapse from view during certain scrolling operations (a topic covered later).

- The Basic Activity is useful, however, in that it provides these elements by default.

- In fact, it is often quicker to create a new activity using the Basic Activity template and delete the elements you do not require than to use the Empty Activity template and manually implement behavior such as collapsing toolbars, a menu or floating action button.

- Since not all of the examples in this book require the features of the Basic Activity template, however, most of the examples in this chapter will use the Empty Activity template unless the example requires one or other of the features provided by the Basic Activity template.

- For future reference, if you need a menu but not a floating action button, use the Basic Activity and follow these steps to delete the floating action button:

- Double-click on the main `activity_main.xml` layout file located in the Project tool window under app > res > layout to load it into the Layout Editor.

- With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard Delete key to remove the object from the layout.

- Locate and edit the Java code for the activity (located under app > java > <package name> > <activity class name> and remove the floating action button code from the `onCreate` method as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  Toolbar toolbar = findViewById(R.id.toolbar);
  setSupportActionBar(toolbar);

  FloatingActionButton fab = findViewById(R.id.fab);

  fab.setOnClickListener(new View.OnClickListener()
  {
    @Override
    public void onClick(View view)
    {
```

```
        Snackbar.make(view, "Replace with your own action",
                    Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show();
      }
    });
}
```

- If you need a floating action button but no menu, use the Basic Activity template and follow these steps:
  - Edit the activity class file and delete the `onCreateOptionsMenu` and `onOptionsItemSelected` methods.
  - Select the res > menu item in the Project tool window and tap the keyboard Delete key to remove the folder and corresponding menu resource files from the project.
- If you need to use the Basic Activity template but need neither the navigation features nor the second content fragment, follow these steps:
  - Within the Project tool window, navigate to and double-click on the app > res > navigation > nav_graph.
  - xml file to load it into the navigation editor.
  - Within the editor, select the SecondFragment entry in the Destinations panel and tap the keyboard delete key to remove it from the graph.
  - Locate and delete the SecondFragment.java (app > java > <package name> > SecondFragment) and `fragment_second.xml` (app > res > layout > fragment_second.xml) files.
  - The final task is to remove some code from the `FirstFragment` class so that the `Button` view no longer navigates to the now non-existent second fragment when clicked.
- Locate the `FirstFragment.java` file, double click on it to load it into the editor and remove the code from the `onViewCreated()` method so that it reads as follows:

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
super.onViewCreated(view, savedInstanceState);

view.findViewById(R.id.button_first).setOnClickListener(new View.
OnClickListener() {
    @Override
    public void onClick(View view) {
    NavHostFragment.findNavController(FirstFragment.this)
    .navigate(R.id.action_FirstFragment_to_SecondFragment);
    }
    });
}
```

(continued)

The Android Studio Layout Editor

- As has been demonstrated in previous chapters, the Layout Editor tool provides a "what you see is what you get" (WYSIWYG) environment in which views can be selected from a palette and then placed onto a canvas representing the display of an Android device.

- Once a view has been placed on the canvas, it can be moved, deleted and resized (subject to the constraints of the parent view).

- Further, a wide variety of properties relating to the selected view may be modified using the Attributes tool window.

- Under the surface, the Layout Editor tool actually constructs an XML resource file containing the definition of the user interface that is being designed.

- As such, the Layout Editor tool operates in three distinct modes referred to as Design, Code and Split modes.

Design Mode

- In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout.

- Figure 8-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:

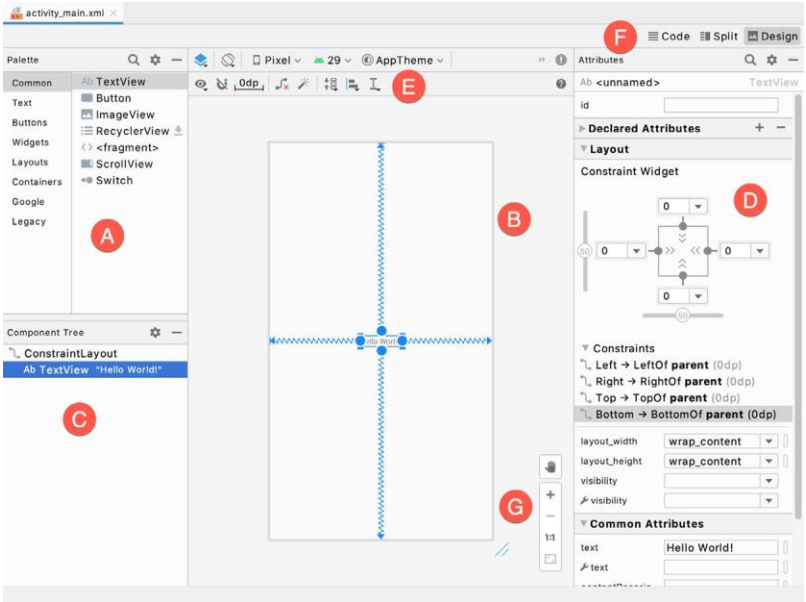

Figure 8-5

- A – Palette – The palette provides access to the range of view components provided by the Android SDK.

  These are grouped into categories for easy navigation. Items may be added to the layout by dragging a view component from the palette and dropping it at the desired position on the layout.

- B – Device Screen – The device screen provides a visual "what you see is what you get" representation of the user interface layout as it is being designed.

  This layout allows for direct manipulation of the design in terms of allowing views to be selected, deleted, moved and resized.

  The device model represented by the layout can be changed at any time using a menu located in the toolbar.

- C – Component Tree – As outlined previously, user interfaces are constructed using a hierarchical structure.

  The component tree provides a visual overview of the hierarchy of the user interface design.

  Selecting an element from the component tree will cause the corresponding view in the layout to be selected.

  Similarly, selecting a view from the device screen layout will select that view in the component tree hierarchy.

- D – Attributes – All of the component views listed in the palette have associated with them a set of attributes that can be used to adjust the behavior and appearance of that view.

  The Layout Editor's attributes panel provides access to the attributes of the currently selected view in the layout allowing changes to be made.

- E – Toolbar – The Layout Editor toolbar provides quick access to a wide range of options including, amongst other options, the ability to zoom in and out of the device screen layout, change the device model currently displayed, rotate the layout between portrait and landscape and switch to a different Android SDK API level.

  The toolbar also has a set of context sensitive buttons which will appear when relevant view types are selected in the device screen layout.

- F – Mode Switching Controls – These three buttons provide a way to switch back and forth between the Layout Editor tool's Design, Code and Split modes.

- G - Zoom and Pan Controls - This control panel allows you to zoom in and out of the design canvas and to grab the canvas and pan around to find areas that are obscured when zoomed in.

The Palette

- The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design.

- The category panel (marked A in Figure 8-6) lists the different categories of view components supported by the Android SDK.

- When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

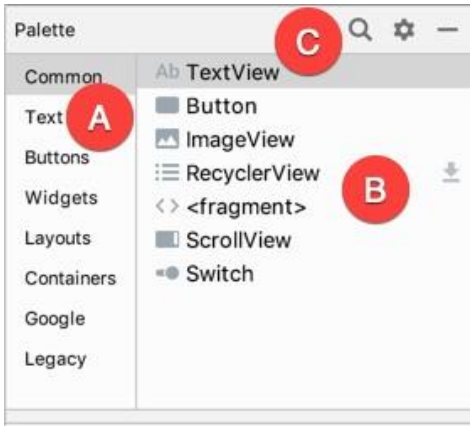

Figure 8-6

- To add a component from the palette onto the layout canvas, simply select the item either from the component list or the preview panel, drag it to the desired location on the canvas and drop it into place.

- A search for a specific component within the currently selected category may be initiated by clicking on the search button (marked C in Figure 8-6 above) in the palette toolbar and typing in the component name.

- As characters are typed, matching results will appear in real-time within the component list panel.

- If you are unsure of the category in which the component resides, simply select the All category either before or during the search operation.

Design Mode and Layout Views

- By default, the layout editor will appear in Design mode as is the case in Figure 8-5 above.

- This mode provides a visual representation of the user interface.

- Design mode can be selected at any time by clicking on the rightmost mode switching control has shown in Figure 8-7:

Figure 8-7

- When the Layout Editor tool is in Design mode, the layout can be viewed in two different ways.

- The view shown in Figure 8-5 above is the Design view and shows the layout and widgets as they will appear in the running app.

- A second mode, referred to as the Blueprint view can be shown either instead of, or concurrently with the Design view.

- The toolbar menu shown in Figure 8-8 provides options to display the Design, Blueprint, or both views.

- A fourth option, Force Refresh Layout, causes the layout to rebuild and redraw.

- This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

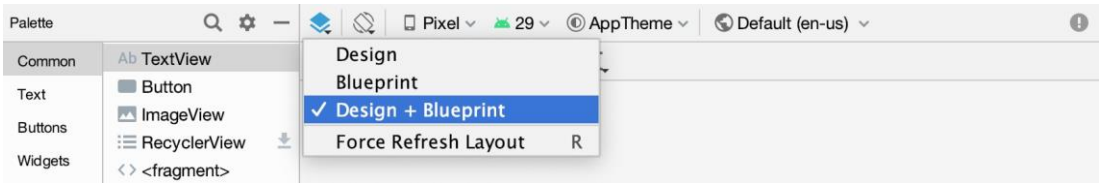

Figure 8-8

- Whether to display the layout view, design view or both is a matter of personal preference. A good approach is to begin with both displayed as shown in Figure 8-9:

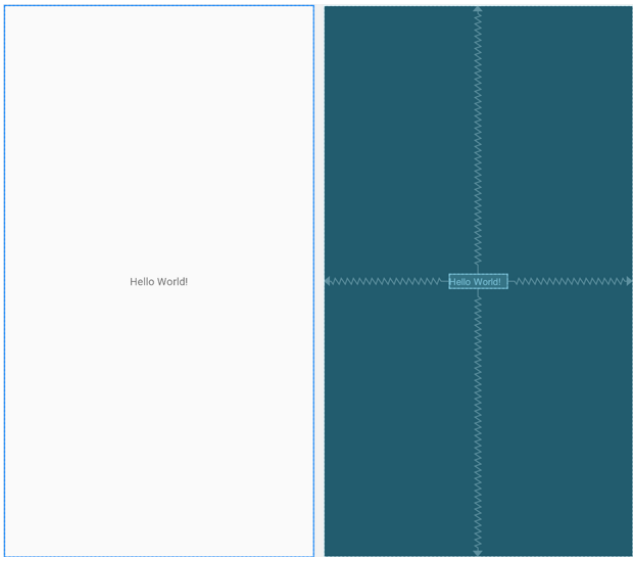

Figure 8-9

Code Mode

- It is important to keep in mind when using the Android Studio Layout Editor tool that all it is really doing is providing a user friendly approach to creating XML layout resource files.

- At any time during the design process, the underlying XML can be viewed and directly edited simply by clicking on the Code button located in the top right-hand corner of the Layout Editor tool panel as shown in Figure 8-10:

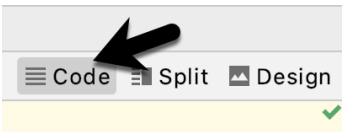

Figure 8-10

(continued)

Figure 8-11 shows the Android Studio Layout Editor tool in Code mode, allowing changes to be made to the user interface declaration by making changes to the XML:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout 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">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figure 8-11

Split Mode

- In Split mode, the editor shows the Design and Code views side by side allowing the user interface to be modified both visually using the design canvas and by making changes directly to the XML declarations.

- To enter Split mode, click on the middle button shown in Figure 8-12 below:

Figure 8-12

- Any changes to the XML are automatically reflected in the design canvas and vice versa as shown below:

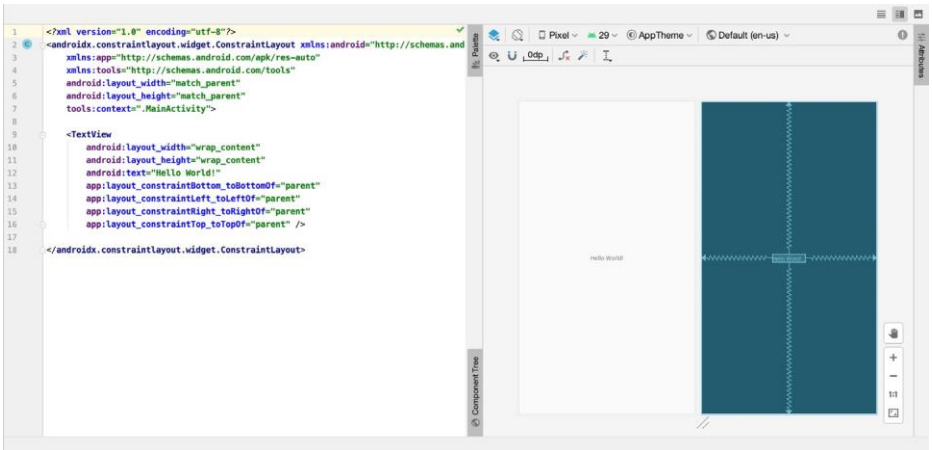

Figure 8-13

Setting Attributes

The Attributes panel provides access to all of the available settings for the currently selected component.

Figure 8-14, for example, shows the attributes for the `TextView` widget:

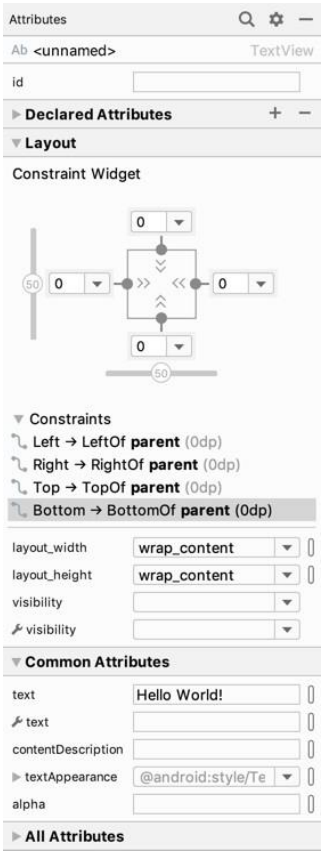

Figure 8-14

- The Attributes tool window is divided into the following different sections:
  - id - Contains the id property which defines the name by which the currently selected object will be referenced in the source code of the app.
  - Declared Attributes - Contains all of the properties which have already been assigned a value.
  - Layout - The settings that define how the currently selected view object is positioned and sized in relation to the screen and other objects in the layout.
  - Common Attributes - A list of attributes that commonly need to be changed for the class of view object currently selected.
  - All Attributes - A complete list of all of the attributes available for the currently selected object.
- A search for a specific attribute may also be performed by selecting the search button in the toolbar of the attributes tool window and typing in the attribute name.

- Some attributes contain a narrow button to the right of the value field.
- This indicates that the Resources dialog is available to assist in selecting a suitable property value.
- To display the dialog, simply click on the button.
- The appearance of this button changes to reflect whether or not the corresponding property value is stored in a resource file or hardcoded.
- If the value is stored in a resource file, the button to the right of the text property field will be filled in to indicate that the value is not hard coded as highlighted in Figure 8-15 below:

Figure 8-15

- Attributes for which a finite number of valid options are available will present a drop down menu (Figure 8-16) from which a selection may be made.

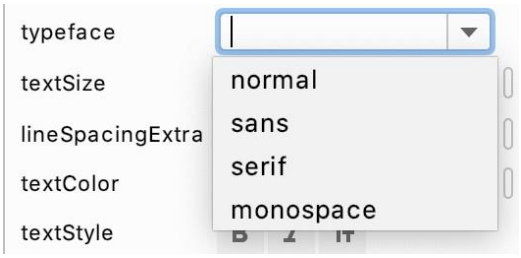

Figure 8-16

- A dropper icon (as shown in the backgroundTint field in Figure 8-15 above) can be clicked to display the color selection palette.
- Similarly, when a flag icon appears in this position it can clicked to display a list of options available for the attribute, while an image icon opens the resource manager panel allowing images and other resource types to be selected for the attribute.

Converting Views

- Changing a view in a layout from one type to another (such as converting a `TextView` to an `EditText`) can be performed easily within the Android Studio layout editor simply by right-clicking on the view either within the screen layout or Component tree window and selecting the Convert view... menu option (Figure 8-17):

(continued)

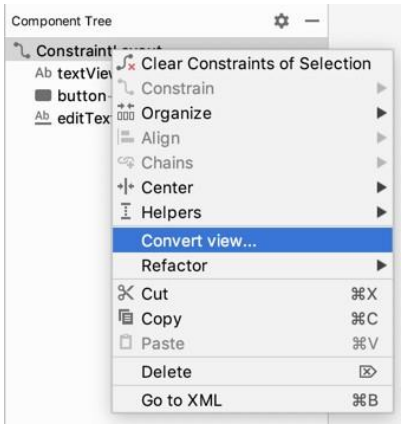

Figure 8-17

- Once selected, a dialog will appear containing a list of compatible view types to which the selected object is eligible for conversion.

- Figure 8-18, for example shows the types to which an existing `TextView` view may be converted:

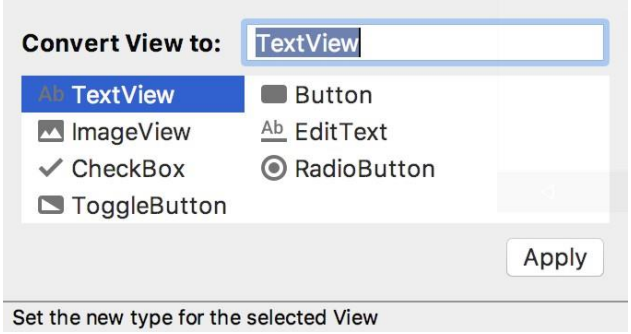

Figure 8-18

- This technique is also useful for converting layouts from one type to another (for example converting a `ConstraintLayout` to a `LinearLayout`).

Displaying Sample Data

- When designing layouts in Android Studio situations will arise where the content to be displayed within the user interface will not be available until the app is completed and running.

- This can sometimes make it difficult to assess from within the layout editor how the layout will appear at app runtime.

- To address this issue, the layout editor allows sample data to be specified that will populate views within the layout editor with sample images and data.

- This sample data only appears within the layout editor and is not displayed when the app runs.

- Sample data may be configured either by directly editing the XML for the layout, or visually using the design-time helper by right-clicking on the widget in the design area and selecting the Set Sample Data menu option.

- The design-time helper panel will display a range of preconfigured options for sample data to be displayed on the selected view item including combinations of text and images in a variety of configurations.

- Figure 8-19, for example, shows the sample data options displayed when selecting sample data to appear in a `RecyclerView` list:

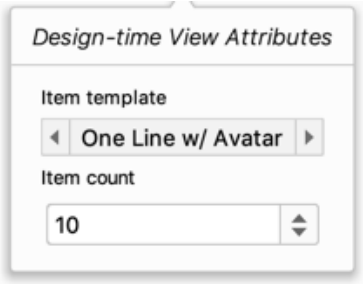

Figure 8-19

- Alternatively, custom text and images may be provided for display during the layout design process.

- An example of using sample data within the layout editor is included in a later chapter.

Creating a Custom Device Definition

- The device menu in the Layout Editor toolbar (Figure 8-20) provides a list of preconfigured device types which, when selected, will appear as the device screen canvas.

- In addition to the pre-configured device types, any AVD instances that have previously been configured within the Android Studio environment will also be listed within the menu.

- To add additional device configurations, display the device menu, select the Add Device Definition… option and follow the steps outlined in an earlier.

Figure 8-20

Changing the Current Device

- As an alternative to the device selection menu, the current device format may be changed by selecting the Custom option from the device menu, clicking on the resize handle located next to the bottom right-hand corner of the device screen (Figure 8-21) and dragging to select an alternate device display format.

- As the screen resizes, markers will appear indicating the various size options and orientations available for selection:

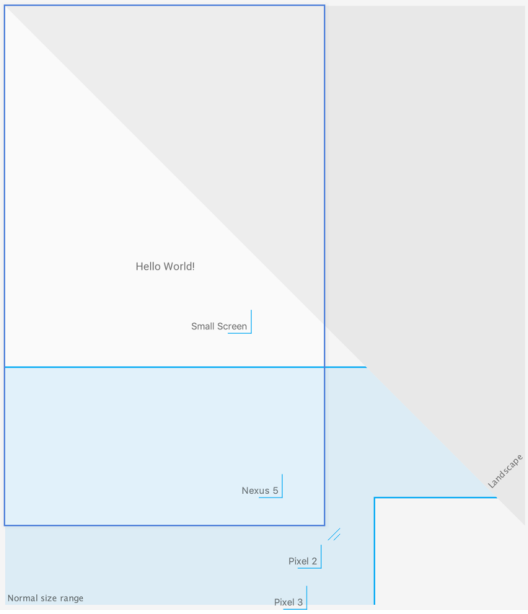

Figure 8-21

Layout Validation (Multi Preview)

- The layout validation (also referred to as multi preview) option allows the user interface layout to be previewed on a range of Pixel-sized screens simultaneously.

- To access multi preview, click on the tab located near the top right-hand corner of the Android Studio main window as indicated in Figure 8-22:

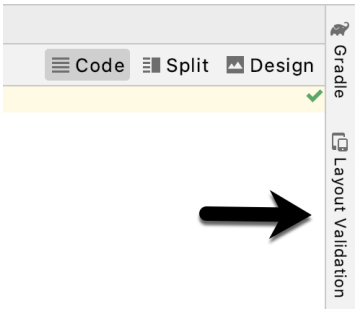

Figure 8-22

- Once loaded, the panel will appear as shown in Figure 8-23 with the layout rendered on multiple Pixel device screen configurations:

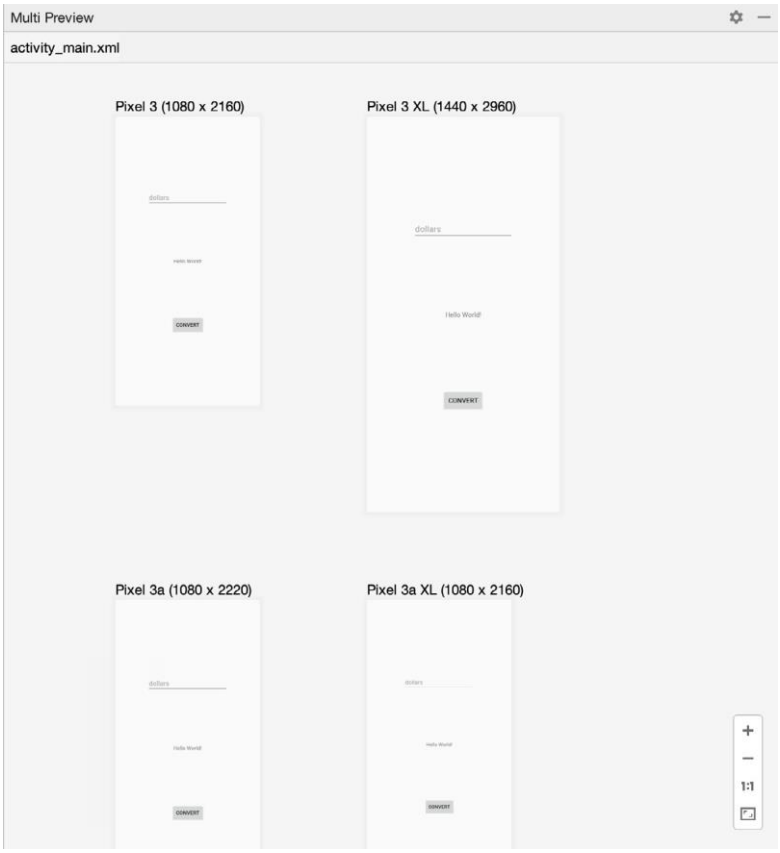

Figure 8-23

Summary

- Each element within a user interface screen of an Android application is a view that is ultimately subclassed from the `android.view.View` class.
- Each view represents a rectangular area of the device display and is responsible both for what appears in that rectangle and for handling events that take place within the view's bounds.
- Multiple views may be combined to create a single composite view.
- The views within a composite view are children of a container view which is generally a subclass of `android.view.ViewGroup` (which is itself a subclass of `android. view.View`).
- A user interface is comprised of views constructed in the form of a view hierarchy.
- The Android SDK includes a range of pre-built views that can be used to create a user interface.
- These include basic components such as text fields and buttons, in addition to a range of layout managers that can be used to control the positioning of child views.
- In the event that the supplied views do not meet a specific requirement, custom views may be created, either by extending or combining existing views, or by subclassing `android.view.View` and creating an entirely new class of view.

- User interfaces may be created using the Android Studio Layout Editor tool, handwriting XML layout resource files or by writing Java code.

- Each of these approaches will be covered in the chapters that follow.

A key part of developing Android applications involves the creation of the user interface. Within the Android Studio environment, this is performed using the Layout Editor tool which operates in three modes.

In Design mode, view components are selected from a palette and positioned on a layout representing an Android device screen and configured using a list of attributes. In Code mode, the underlying XML that represents the user interface layout can be directly edited. Split mode, on the other hand allows the layout to be created and modified both visually and via direct XML editing. These modes combine to provide an extensive and intuitive user interface design environment.

The layout validation panel allows user interface layouts to be quickly previewed on a range of different device screen sizes.