15. Understanding Android Views, View Groups and Layouts

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.

15.1 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.

15.2 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.

15.3 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 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** – The GridLayout is a relatively new layout manager that was introduced as part of Android 4.0. 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 starting with the chapter entitled *Working with the Floating Action Button and Snackbar*.

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.

15.4 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 15-1:

Figure 15-1

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 15-2 shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

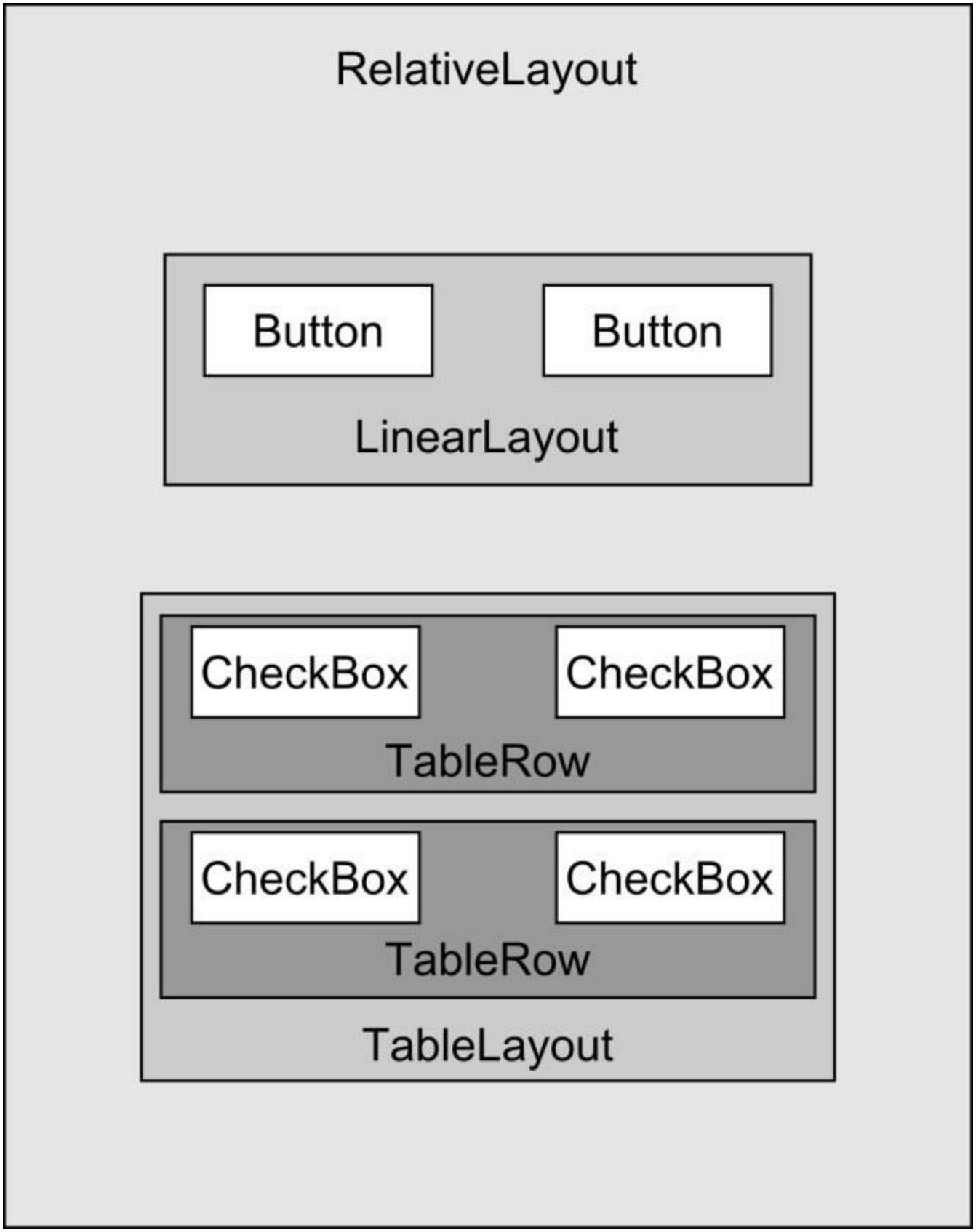


Figure 15-2

As 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 15-3:

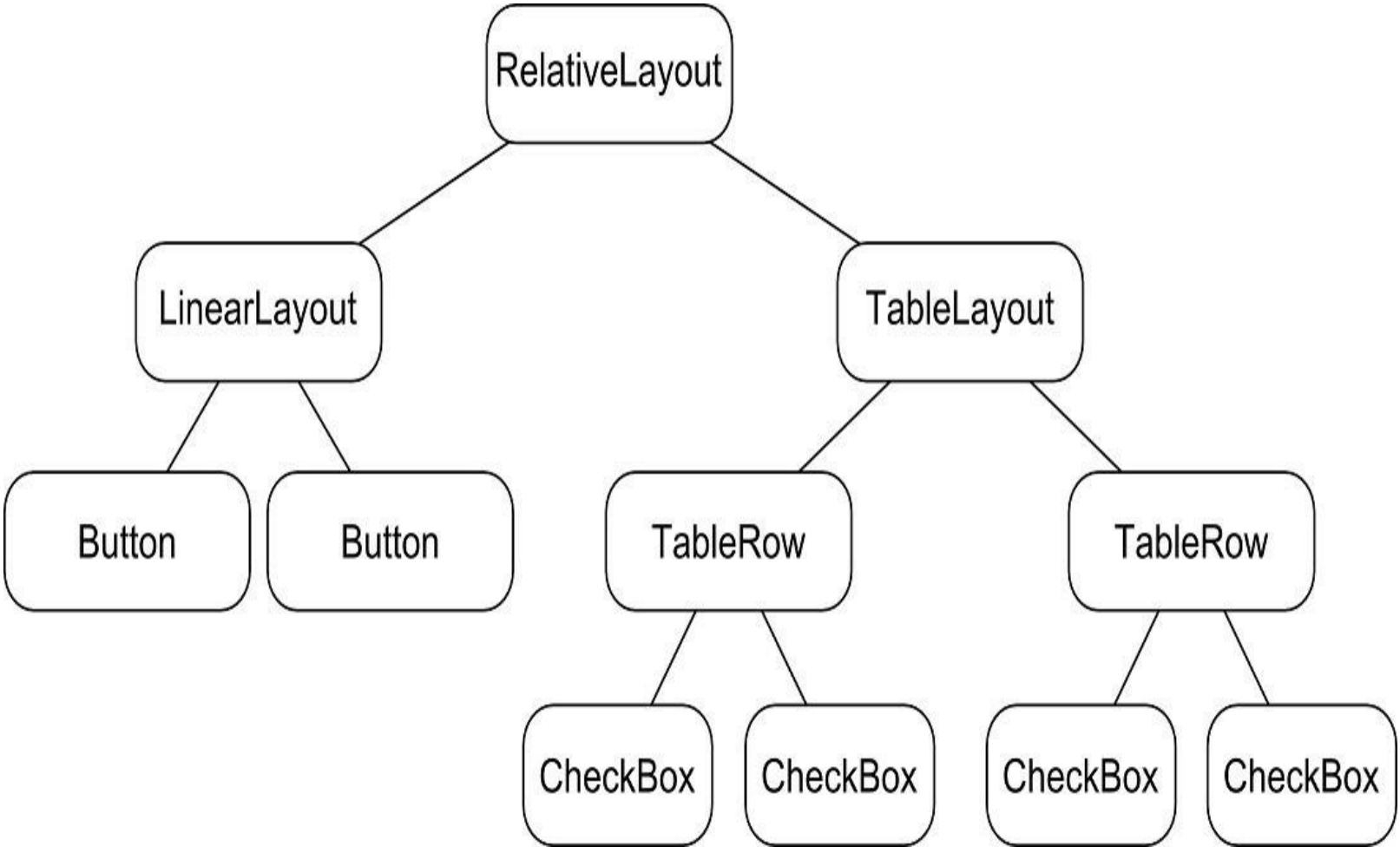


Figure 15-3

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 15-1. 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.

15.5 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.

15.6 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.

16. 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.

16.1 Basic vs. Empty Activity Templates

As outlined in the chapter entitled *The Anatomy of an Android Application*, 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.

The Empty Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object as shown in Figure 16-1:

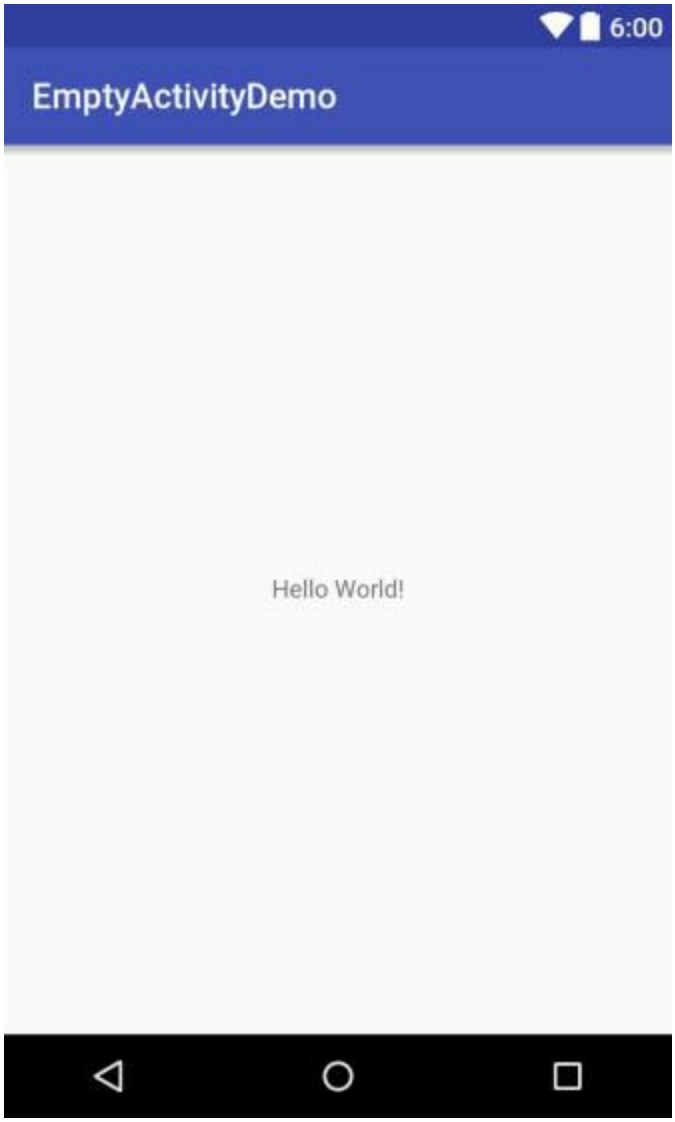


Figure 16-1

The Basic Activity, on the other hand, consists of two layout files. The top level layout file has a CoordinatorLayout as the root view, a configurable app bar, a menu preconfigured with a single menu item (A in Figure 16-2), a floating action button (B) and a reference to the second layout file in which the layout for the content area of the activity user interface is declared:

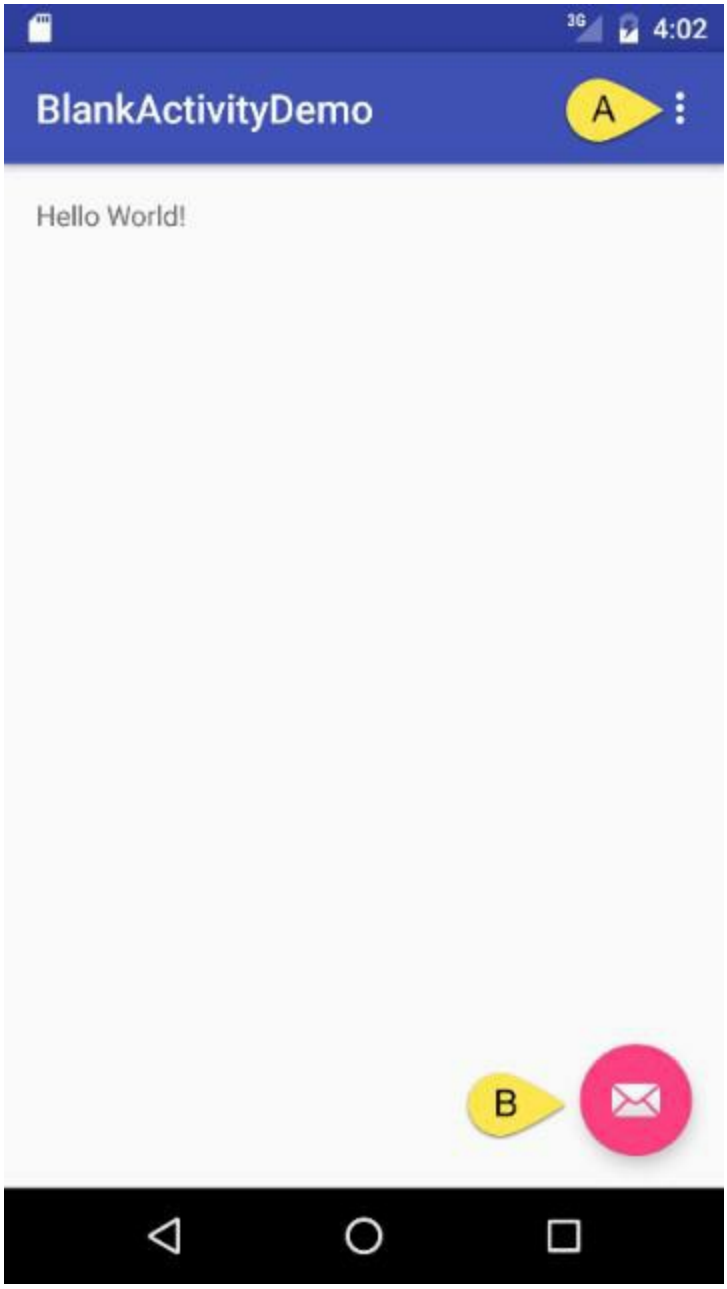


Figure 16-2

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 in the chapter entitled *Working with the AppBar and Collapsing Toolbar Layouts*). 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:

1. Double-click on the main *activity* layout file located in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. This will be the layout file prefixed with *activity_* and

not the content file prefixed with *content_*.

2. 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.
3. 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 = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    FloatingActionButton fab =
            (FloatingActionButton) 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:

1. Edit the activity class file and delete the *onCreateOptionsMenu* and *onOptionsItemSelected* methods.
2. 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.

16.2 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 Properties panel.

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 two distinct modes referred to as *Design mode* and *Text mode*.

16.3 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 16-3 highlights the key areas of the Android Studio Layout Editor tool in design mode:

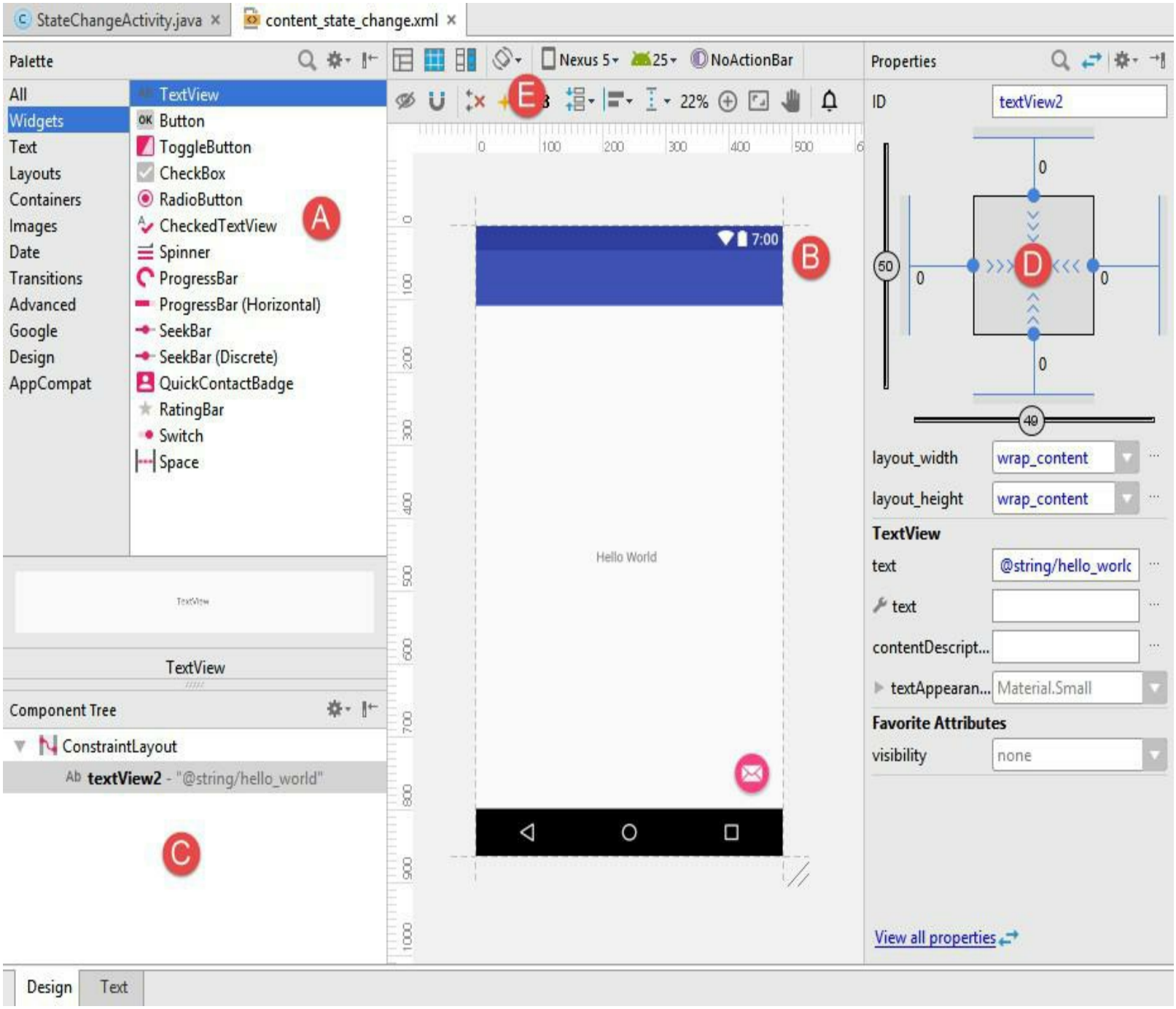


Figure 16-3

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 or preview panel 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 in the previous chapter (*Understanding Android Views, View Groups and Layouts*) 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 – Properties – All of the component views listed in the palette have associated with them a set of properties that can be used to adjust the behavior and appearance of that view. The Layout Editor's properties panel provides access to the properties 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 Tabs – The tabs located along the lower edge of the Layout Editor provide a way to switch back and forth between the Layout Editor tool's text and design modes.

16.4 The Palette

The Layout Editor palette is organized into three 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 16-4) 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. Finally, selecting a component from the list causes a rendering of that component to appear within the preview panel (C):

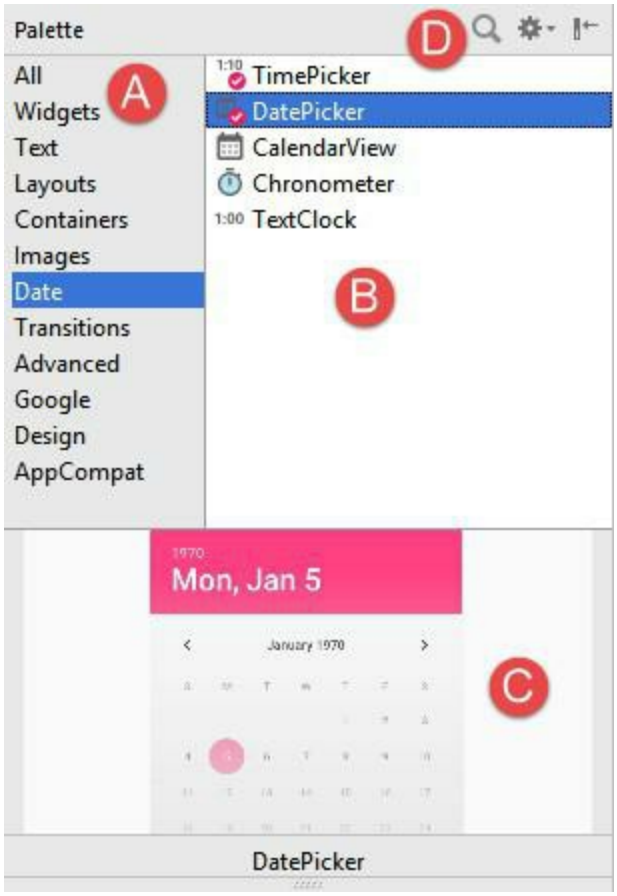


Figure 16-4

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 D in Figure 16-4 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.

16.5 Pan and Zoom

When first opened, the Layout Editor will size the layout canvas so that it fits within the available space. Zooming in and out of the layout can be achieved using the plus and minus buttons located in the editor toolbar. When the view is zoomed in, it can be useful to pan around the layout to locate a particular area of the design. Although this can be achieved using the scrollbars, another option is to use the pan menu bar button highlighted in Figure 16-5. Once selected, a pan and zoom panel will appear containing a lens which can be moved to alter the currently visible area of the layout.

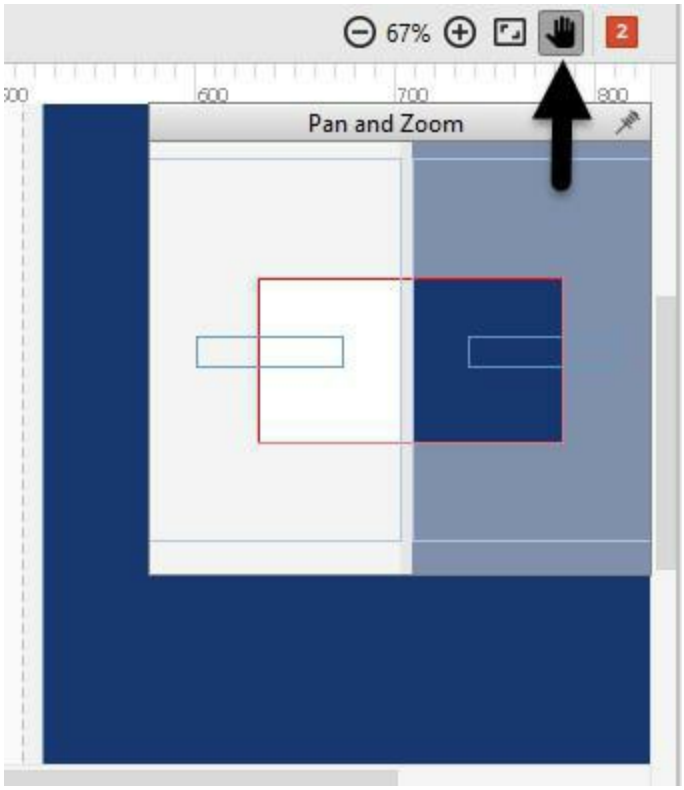


Figure 16-5

16.6 Design and Layout Views

When the Layout Editor tool is in Design mode, the layout can be viewed in two different ways. The view shown in Figure 16-3 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 Layout or Blueprint view can be shown either instead of, or concurrently with the Design view. Three toolbar buttons (highlighted in Figure 16-6) provide options to display the Design, Blueprint, or both views:

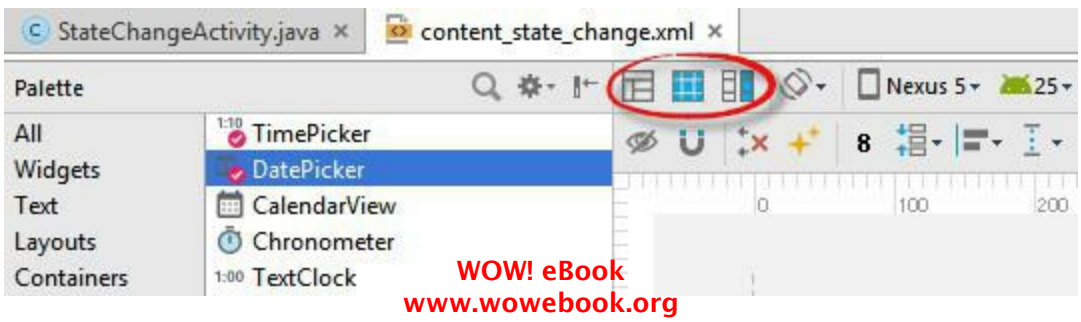


Figure 16-6

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 16-7:

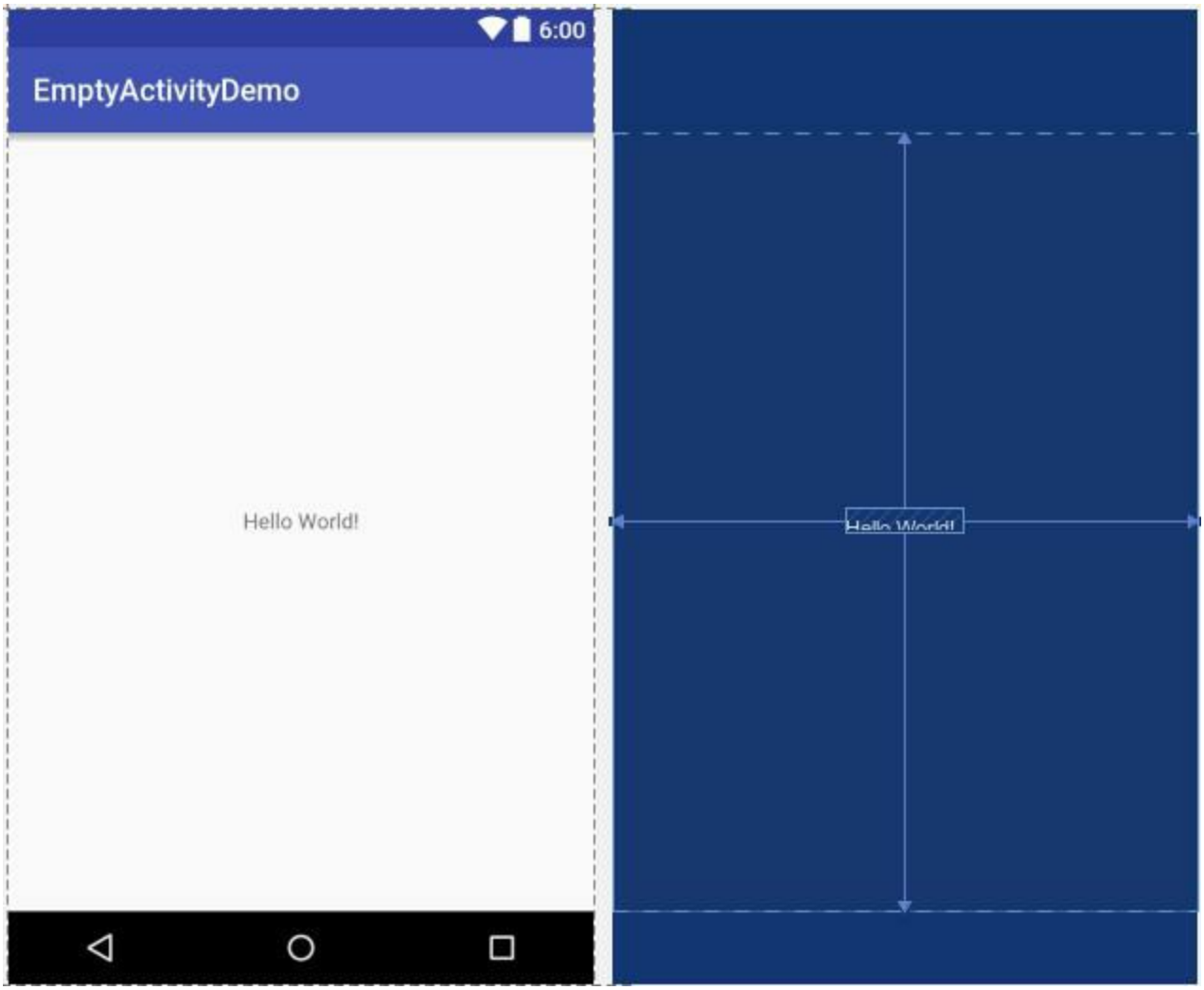


Figure 16-7

16.7 **Text 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 *Text* tab located at the bottom of the Layout Editor tool panel. To return to design mode, simply click on the *Design* tab.

Figure 16-8 highlights the key areas of the Android Studio Layout Editor tool in text mode:

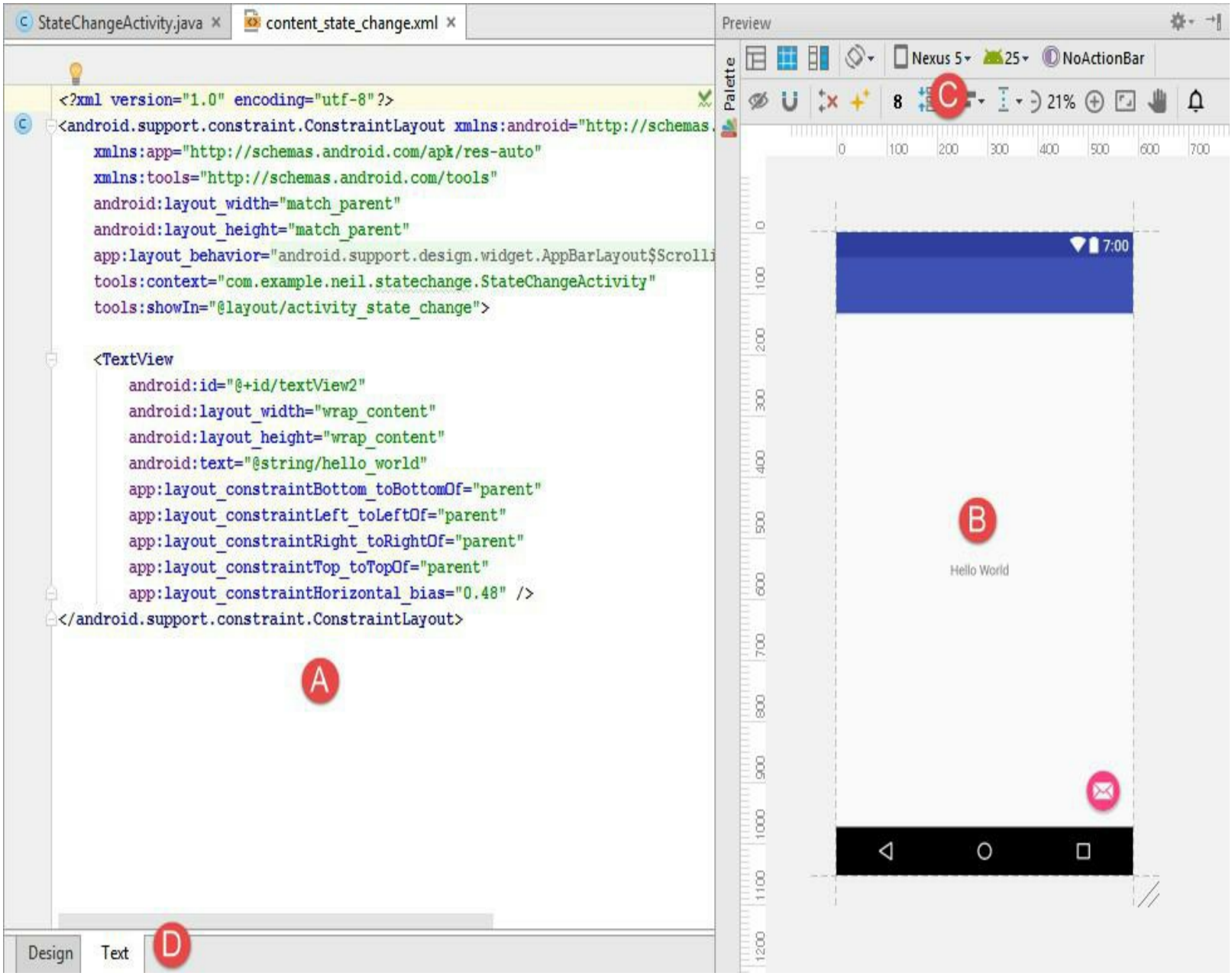


Figure 16-8

A – Editor – The editor panel displays the XML that makes up the current user interface layout design. This is the full Android Studio editor environment containing all of the features previously outlined in the *The Basics of the Android Studio Code Editor* chapter of this book.

B – Preview – As changes are made to the XML in the editor, these changes are visually reflected in the preview window. This provides instant visual feedback on the XML changes as they are made in the editor, thereby avoiding the need to switch back and forth between text and design mode to see changes. The preview also allows direct manipulation and design of the layout just as if the layout were in Design mode, with visual changes being reflected in the editor panel in real-time. As with Design mode, both the Design and Layout views may be displayed using the toolbar buttons highlighted in Figure 16-6 above.

C – Toolbar – The toolbar in text mode provides access to the same functions available in design mode.

D - Mode Switching Tabs – The tabs located along the lower edge of the Layout Editor provide a way to switch back and forth between the Layout Editor tool's Text and Design modes.

16.8 **Setting Properties**

The Properties panel provides access to all of the available settings for the currently selected component. By default, the properties panel shows the most commonly changed properties for the currently selected component in the layout. Figure 16-9, for example, shows this subset of properties for the TextView widget:

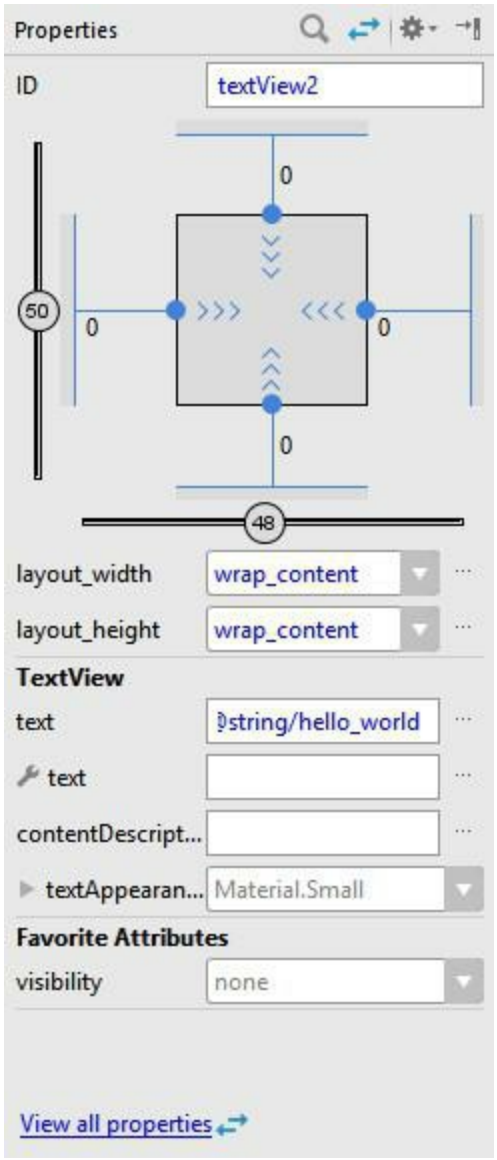


Figure 16-9

To access all of the properties for the currently selected widget, click on the button highlighted in Figure 16-10, or using the *View all properties* link at the bottom of the properties panel:

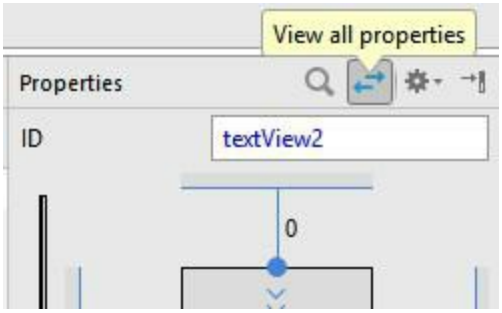


Figure 16-10

A search for a specific property may also be performed by selecting the search button in the toolbar of the properties tool window and typing in the property name. Select the *View all properties* button

or link either before or during a search to ensure that all of the properties for the currently selected component are included in the results.

Some properties contain a button displaying three dots. This indicates that a settings dialog is available to assist in selecting a suitable property value. To display the dialog, simply click on the button. Properties for which a finite number of valid options are available will present a drop down menu (Figure 16-11) from which a selection may be made.

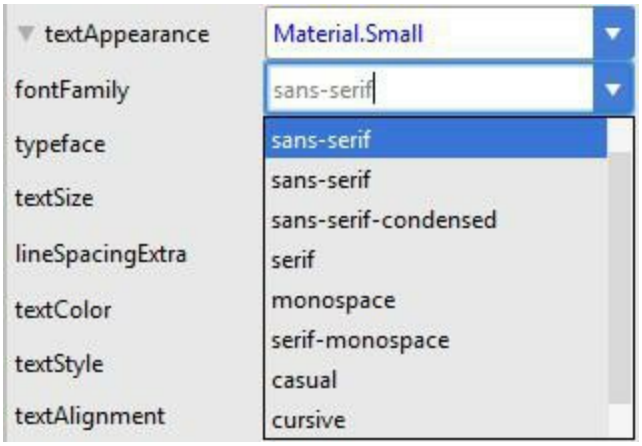


Figure 16-11

16.9 Configuring Favorite Attributes

The properties included on the initial subset property list may be extended by configuring *favorite attributes*. To add a property to the favorites list, display all the properties for the currently selected component and hover the mouse pointer so that it is positioned to the far left of the property entry within the property tool window. A star icon will appear to the left of the property name which, when clicked, will add the property to the favorites list. Figure 16-12, for example, shows the alpha, background and autoText properties for a TextView widget configured as favorite attributes:

Figure 16-12

Once added as favorites, the properties will be listed beneath the *Favorite Attributes* section in the subject properties list:

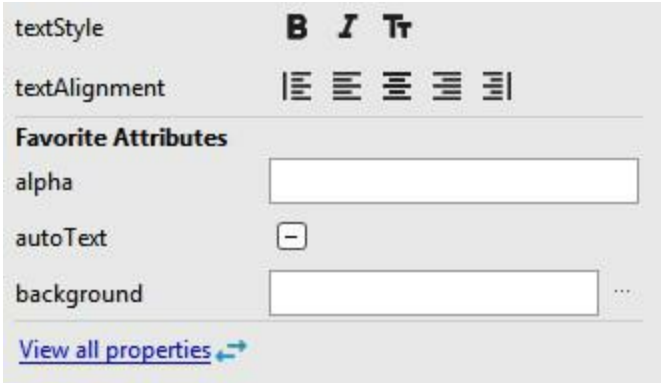


Figure 16-13

16.10 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 16-14) 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 the chapter entitled *Creating an Android Virtual Device (AVD) in Android Studio*.

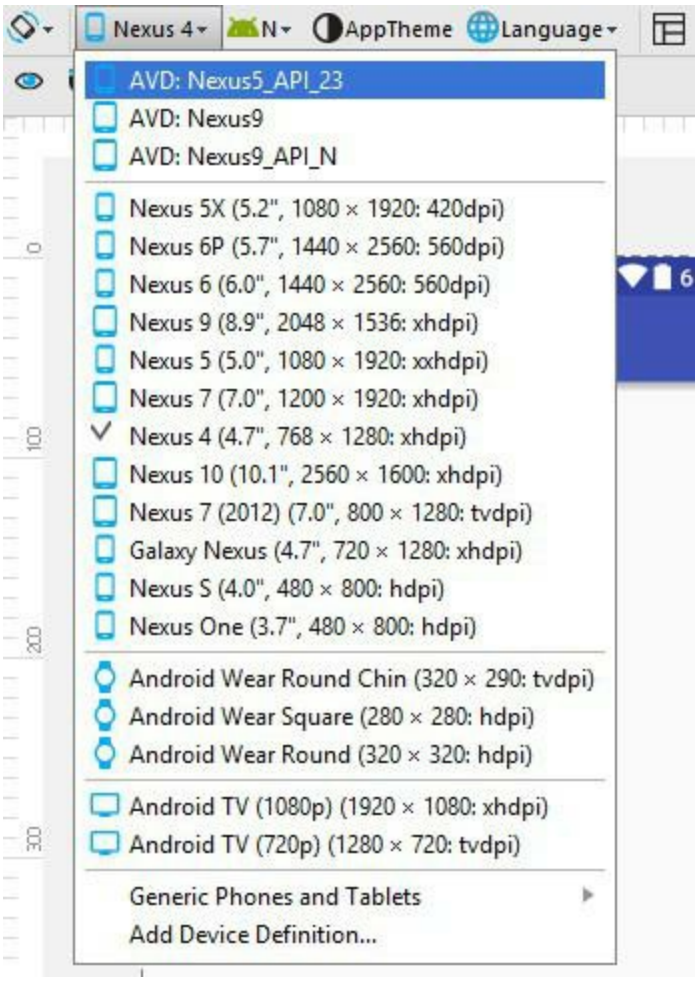


Figure 16-14

16.11 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by clicking on the resize handle located next to the bottom right-hand corner of the device screen (indicated in Figure 16-15) 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:

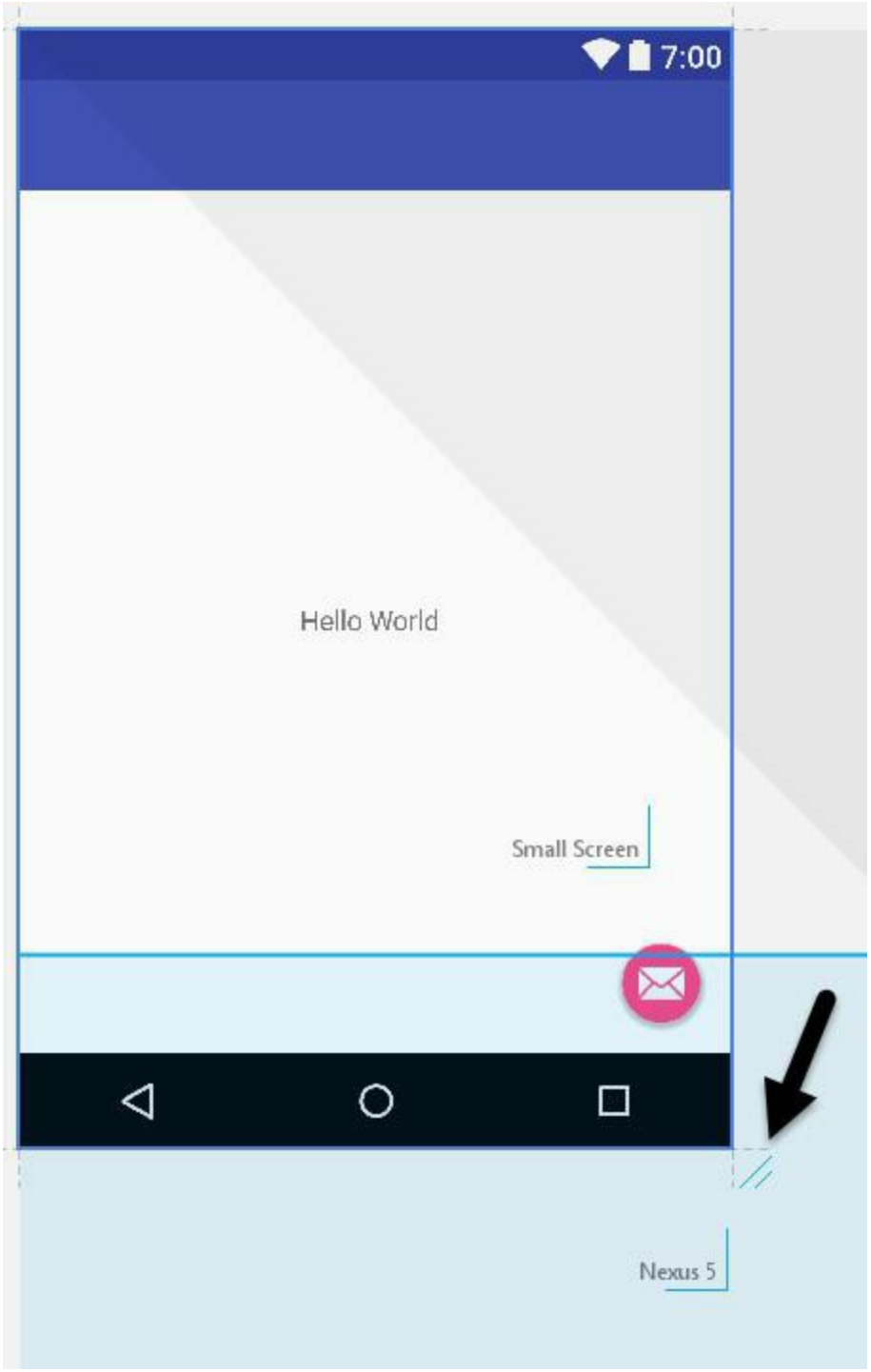


Figure 16-15

16.12 Summary

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 two 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 properties. In text mode, the underlying XML that represents the user interface layout can be directly edited, with changes reflected in a preview screen. These modes combine to provide an extensive and intuitive user interface design environment.

17. A Guide to the Android ConstraintLayout

As discussed in the chapter entitled *Understanding Android Views, View Groups and Layouts*, Android provides a number of layout managers for the purpose of designing user interfaces. With Android 7, Google has introduced a new layout that is intended to address many of the shortcomings of the older layout managers. This new layout, called ConstraintLayout, combines a simple, expressive and flexible layout system with powerful features built into the Android Studio Layout Editor tool to ease the creation of responsive user interface layouts that adapt automatically to different screen sizes and changes in device orientation.

This chapter will outline the basic concepts of ConstraintLayout while the next chapter will provide a detailed overview of how constraint-based layouts can be created using ConstraintLayout within the Android Studio Layout Editor tool.

17.1 How ConstraintLayout Works

In common with all other layouts, ConstraintLayout is responsible for managing the positioning and sizing behavior of the visual components (also referred to as widgets) it contains. It does this based on the constraint connections that are set on each child widget.

In order to fully understand and use ConstraintLayout, it is important to gain an appreciation of the following key concepts:

- Constraints
- Margins
- Opposing Constraints
- Constraint Bias
- Chains
- Chain Styles

17.1.1 Constraints

Constraints are essentially sets of rules that dictate the way in which a widget is aligned and distanced in relation to other widgets, the sides of the containing ConstraintLayout and special elements called *guidelines*. Constraints also dictate how the user interface layout of an activity will respond to changes in device orientation, or when displayed on devices of differing screen sizes. In order to be adequately configured, a widget must have sufficient constraint connections such that it's position can be resolved by the ConstraintLayout layout engine in both the horizontal and vertical planes.

17.1.2 Margins

A margin is a form of constraint that specifies a fixed distance. Consider a Button object that needs to be positioned near the top right-hand corner of the device screen. This might be achieved by implementing margin constraints from the top and right-hand edges of the Button connected to the corresponding sides of the parent ConstraintLayout as illustrated in Figure 7-1:

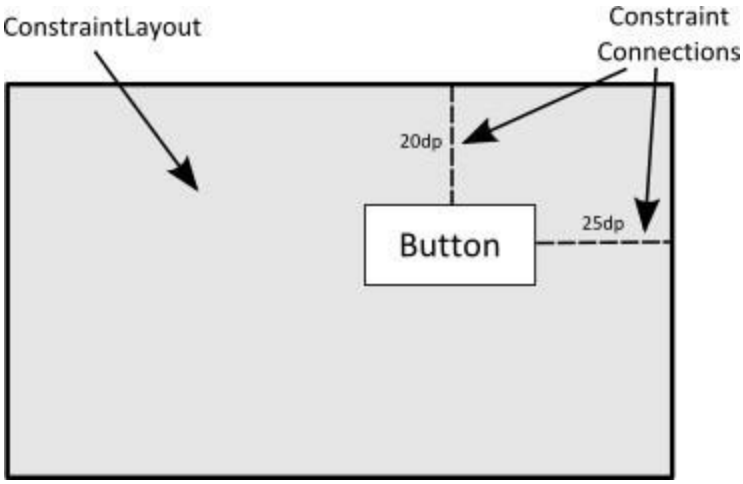


Figure 17-1

As indicated in the above diagram, each of these constraint connections has associated with it a margin value dictating the fixed distances of the widget from two sides of the parent layout. Under this configuration, regardless of screen size or the device orientation, the Button object will always be positioned 20 and 25 device-independent pixels (dp) from the top and right-hand edges of the parent ConstraintLayout respectively as specified by the two constraint connections.

While the above configuration will be acceptable for some situations, it does not provide any flexibility in terms of allowing the ConstraintLayout layout engine to adapt the position of the widget in order to respond to device rotation and to support screens of different sizes. To add this responsiveness to the layout it is necessary to implement opposing constraints.

17.1.3 Opposing Constraints

Two constraints operating along the same axis on a single widget are referred to as *opposing constraints*. In other words, a widget with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints. Figure 17-2, for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

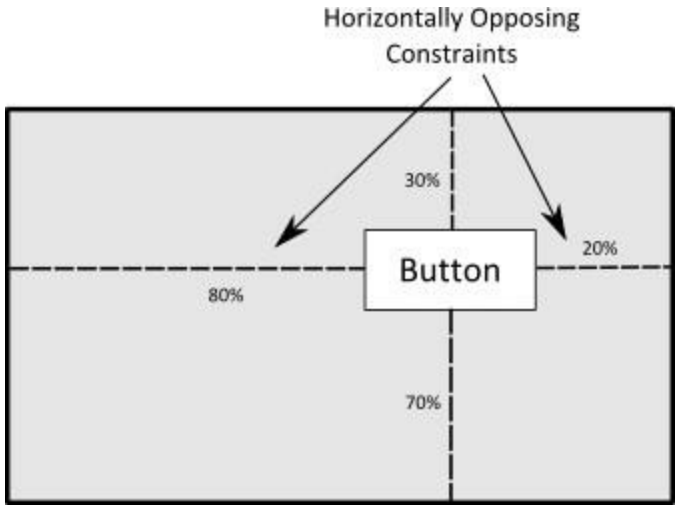


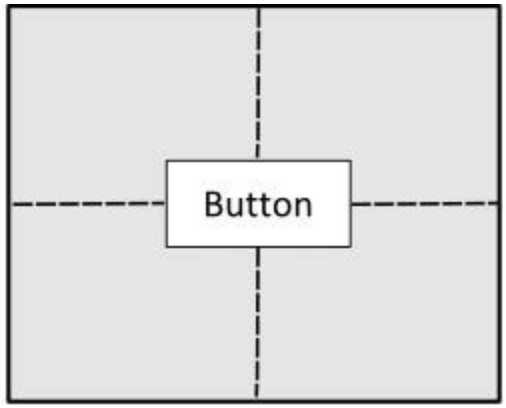
Figure 17-2

The key point to understand here is that once opposing constraints are implemented on a particular axis, the positioning of the widget becomes percentage rather than coordinate based. Instead of being fixed at 20dp from the top of the layout, for example, the widget is now positioned at a point 30% from the top of the layout. In different orientations and when running on larger or smaller screens, the Button will always be in the same location relative to the dimensions of the parent layout.

It is now important to understand that the layout outlined in Figure 17-2 has been implemented using not only opposing constraints, but also by applying *constraint bias*.

17.1.4 Constraint Bias

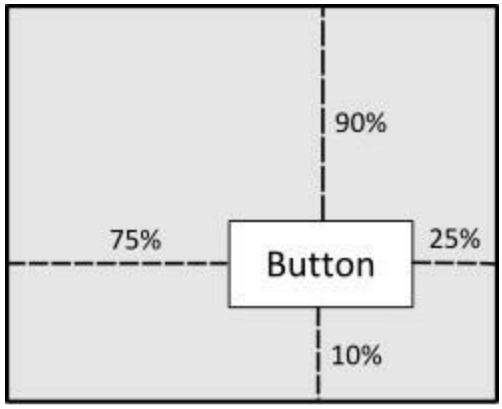
It has now been established that a widget in a ConstraintLayout can potentially be subject to opposing constraint connections. By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition. Figure 17-3, for example, shows a widget centered within the containing ConstraintLayout using opposing horizontal and vertical constraints:



Widget Centered by Opposing Constraints

Figure 17-3

To allow for the adjustment of widget position in the case of opposing constraints, the ConstraintLayout implements a feature known as *constraint bias*. Constraint bias allows the positioning of a widget along the axis of opposition to be biased by a specified percentage in favor of one constraint. Figure 17-4, for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:



Widget Offset using Constraint Bias

Figure 17-4

The next chapter, entitled *A Guide to using ConstraintLayout in Android Studio*, will cover these concepts in greater detail and explain how these features have been integrated into the Android Studio Layout Editor tool. In the meantime, however, a few more areas of the ConstraintLayout class need to be covered.

17.1.5 Chains

ConstraintLayout chains provide a way for the layout behavior of two or more widgets to be defined as a group. Chains can be declared in either the vertical or horizontal axis and configured to define how the widgets in the chain are spaced and sized.

Widgets are chained when connected together by bi-directional constraints. Figure 17-5, for example, illustrates three widgets chained in this way:

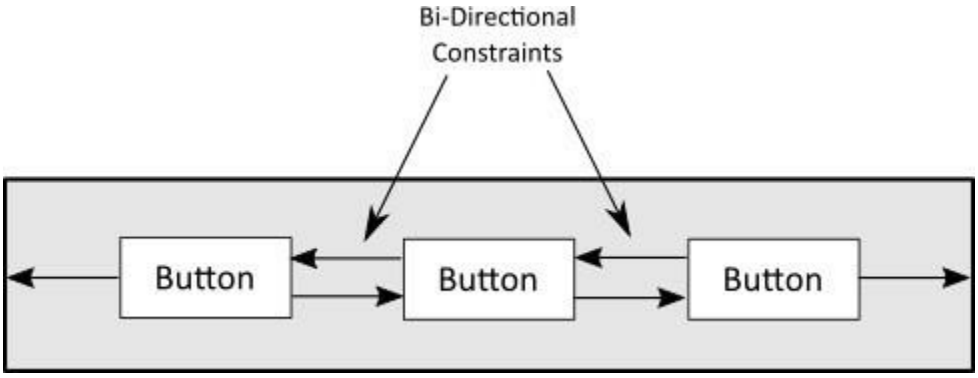


Figure 17-5

The first element in the chain is the *chain head* which translates to the top widget in a vertical chain or, in the case of a horizontal chain, the left-most widget. The layout behavior of the entire chain is primarily configured by setting attributes on the chain head widget.

17.1.6 Chain Styles

The layout behavior of a ConstraintLayout chain is dictated by the *chain style* setting applied to the chain head widget. The ConstraintLayout class currently supports the following chain layout styles:

- **Spread Chain** – The widgets contained within the chain are distributed evenly across the available space. This is the default behavior for chains.

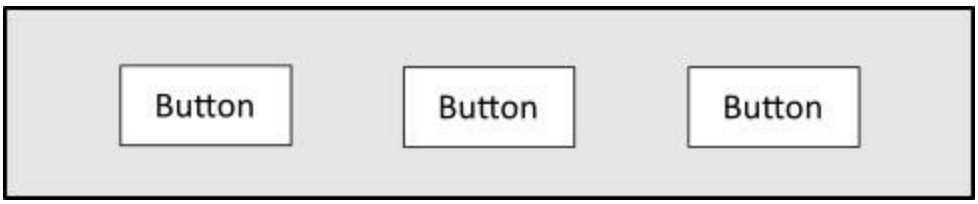


Figure 17-6

- **Spread Inside Chain** – The widgets contained within the chain are spread evenly between the chain head and the last widget in the chain. The head and last widgets are not included in the distribution of spacing.

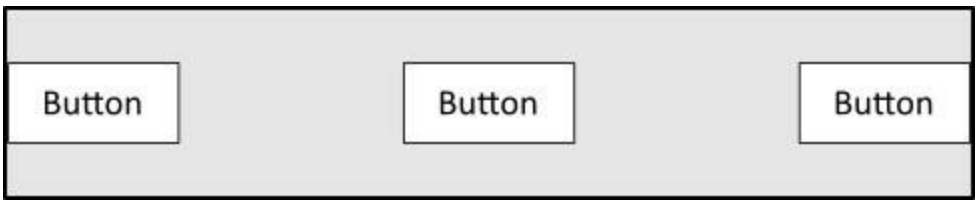


Figure 17-7

- **Weighted Chain** – Allows the space taken up by each widget in the chain to be defined via weighting properties.

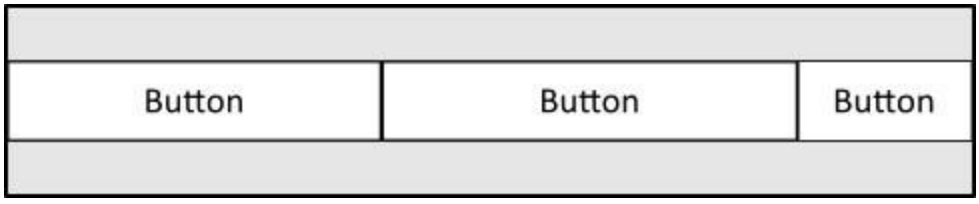


Figure 17-8

- **Packed Chain** – The widgets that make up the chain are packed together without any spacing. A

bias may be applied to control the horizontal or vertical positioning of the chain in relation to the parent container.

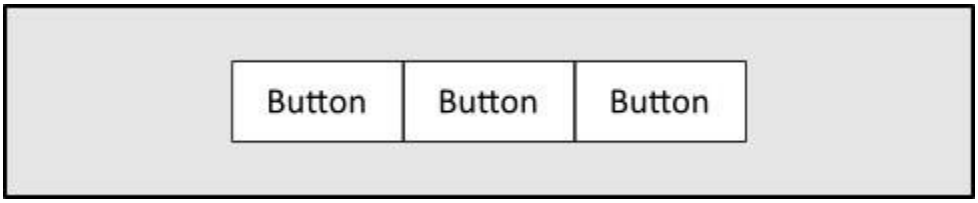


Figure 17-9

17.2 Baseline Alignment

So far, this chapter has only referred to constraints that dictate alignment relative to the sides of a widget (typically referred to as side constraints). A common requirement, however, is for a widget to be aligned relative to the content that it displays rather than the boundaries of the widget itself. To address this need, ConstraintLayout provides *baseline alignment* support.

As an example, assume that the previous theoretical layout from Figure 17-1 requires a TextView widget to be positioned 40dp to the left of the Button. In this case, the TextView needs to be *baseline aligned* with the Button view. This means that the text within the Button needs to be vertically aligned with the text within the TextView. The additional constraints for this layout would need to be connected as illustrated in Figure 17-10:

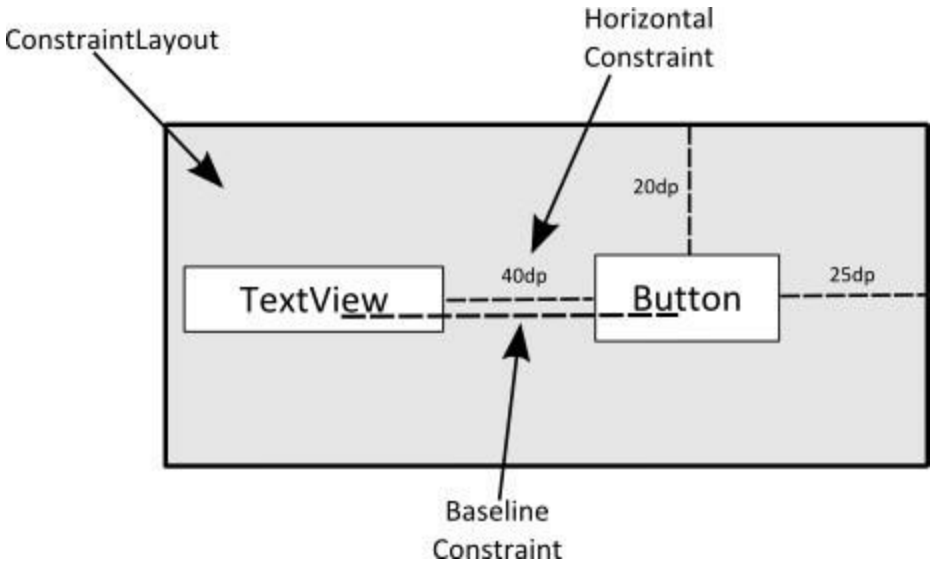


Figure 17-10

The TextView is now aligned vertically along the baseline of the Button and positioned 40dp horizontally from the Button object's left-hand edge.

17.3 Working with Guidelines

Guidelines are special elements available within the ConstraintLayout that provide an additional target to which constraints may be connected. Multiple guidelines may be added to a ConstraintLayout instance which may, in turn, be configured in horizontal or vertical orientations. Once added, constraint connections may be established from widgets in the layout to the guidelines. This is particularly useful when multiple widgets need to be aligned along an axis. In Figure 17-11, for example, three Button objects contained within a ConstraintLayout are constrained along a vertical guideline:

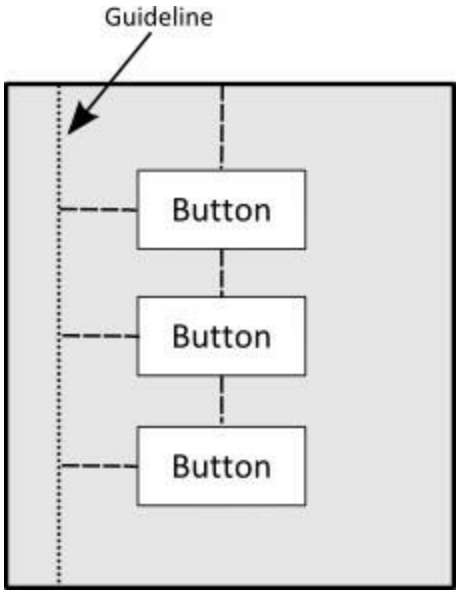


Figure 17-11

17.4 Configuring Widget Dimensions

Controlling the dimensions of a widget is a key element of the user interface design process. The ConstraintLayout provides three options which can be set on individual widgets to manage sizing behavior. These settings are configured individually for height and width dimensions:

- **Fixed** – The widget is fixed to specified dimensions.
- **Match Constraints** –Allows the widget to be resized by the layout engine to satisfy the prevailing constraints. Also referred to as the *AnySize* or MATCH_CONSTRAINT option.
- **Wrap Content** – The size of the widget is dictated by the content it contains (i.e. text or graphics).

17.5 Ratios

The dimensions of a widget may be defined using ratio settings. A widget could, for example, be constrained using a ratio setting such that, regardless of any resizing behavior, the width is always twice the height dimension.

17.6 ConstraintLayout Advantages

ConstraintLayout provides a level of flexibility that allows many of the features of older layouts to be achieved with a single layout instance where it would previously have been necessary to nest multiple layouts. This has the benefit of avoiding the problems inherent in layout nesting by allowing so called “flat” or “shallow” layout hierarchies to be designed leading both to less complex layouts and improved user interface rendering performance at runtime.

ConstraintLayout was also implemented with a view to addressing the wide range of Android device screen sizes available on the market today. The flexibility of ConstraintLayout makes it easier for user interfaces to be designed that respond and adapt to the device on which the app is running.

Finally, as will be demonstrated in the chapter entitled *A Guide to using ConstraintLayout in Android Studio*, the Android Studio Layout Editor tool was entirely re-written for Android Studio 2.2 specifically to add features for ConstraintLayout-based user interface design.

17.7 ConstraintLayout Availability

Although introduced with Android 7, ConstraintLayout is provided as a separate support library from the main Android SDK and is compatible with older Android versions as far back as API Level 9 (Gingerbread). This allows apps that make use of this new layout to run on devices running much older versions of Android.

17.8 Summary

ConstraintLayout is a layout manager introduced with Android 7. It is designed to ease the creation of flexible layouts that adapt to the size and orientation of the many Android devices now on the market. ConstraintLayout uses constraints to control the alignment and positioning of widgets in relation to the parent ConstraintLayout instance, guidelines and the other widgets in the layout. ConstraintLayout is the default layout for newly created Android Studio projects and is the recommended choice when designing user interface layouts. With this simple yet flexible approach to layout management, complex and responsive user interfaces can be implemented with surprising ease.

18. A Guide to using ConstraintLayout in Android Studio

As mentioned more than once in previous chapters, Google has made significant changes to the Android Studio Layout Editor tool, many of which were made solely to support user interface layout design using ConstraintLayout. Now that the basic concepts of ConstraintLayout have been outlined in the previous chapter, this chapter will explore these concepts in more detail while also outlining the ways in which the Layout Editor tool allows ConstraintLayout-based user interfaces to be designed and implemented.

18.1 Design and Layout Views

The chapter entitled *A Guide to the Android Studio Layout Editor Tool* explained that the Android Studio Layout Editor tool provides two ways to view the user interface layout of an activity in the form of Design and Layout (also known as blueprint) views. These views of the layout may be displayed individually or, as in Figure 18-1, side by side:

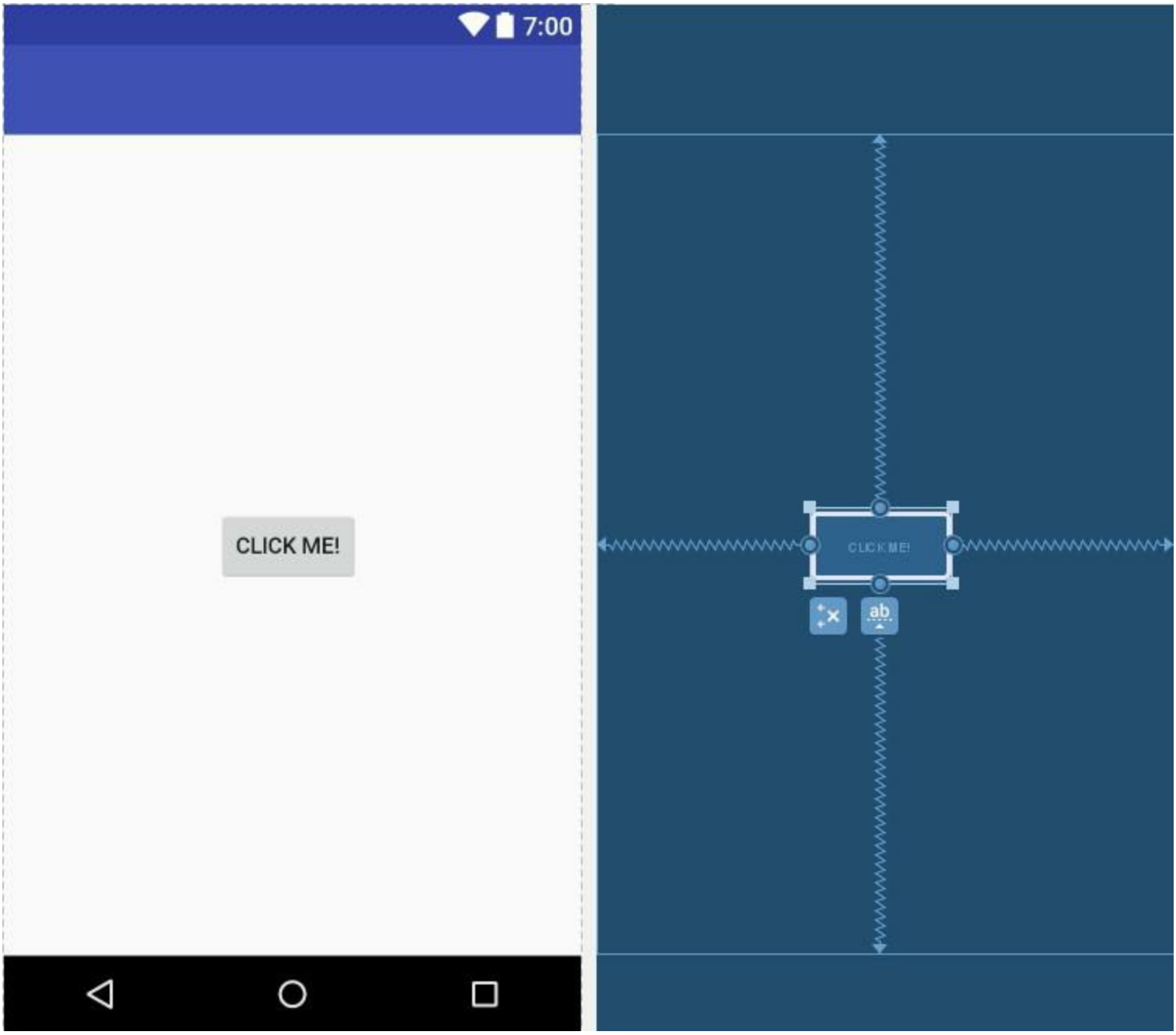


Figure 18-1

The Design view (positioned on the left in the above figure) presents a "what you see is what you get" representation of the layout, wherein the layout appears as it will within the running app. The Layout view, on the other hand, displays a blueprint style of view where the widgets are represented by shaded outlines. As can be seen in Figure 18-1 above, Layout view also displays the constraint connections (in this case opposing constraints used to center a button within the layout). These constraints are also overlaid onto the Design view when a specific widget in the layout is selected as illustrated in Figure 18-2:

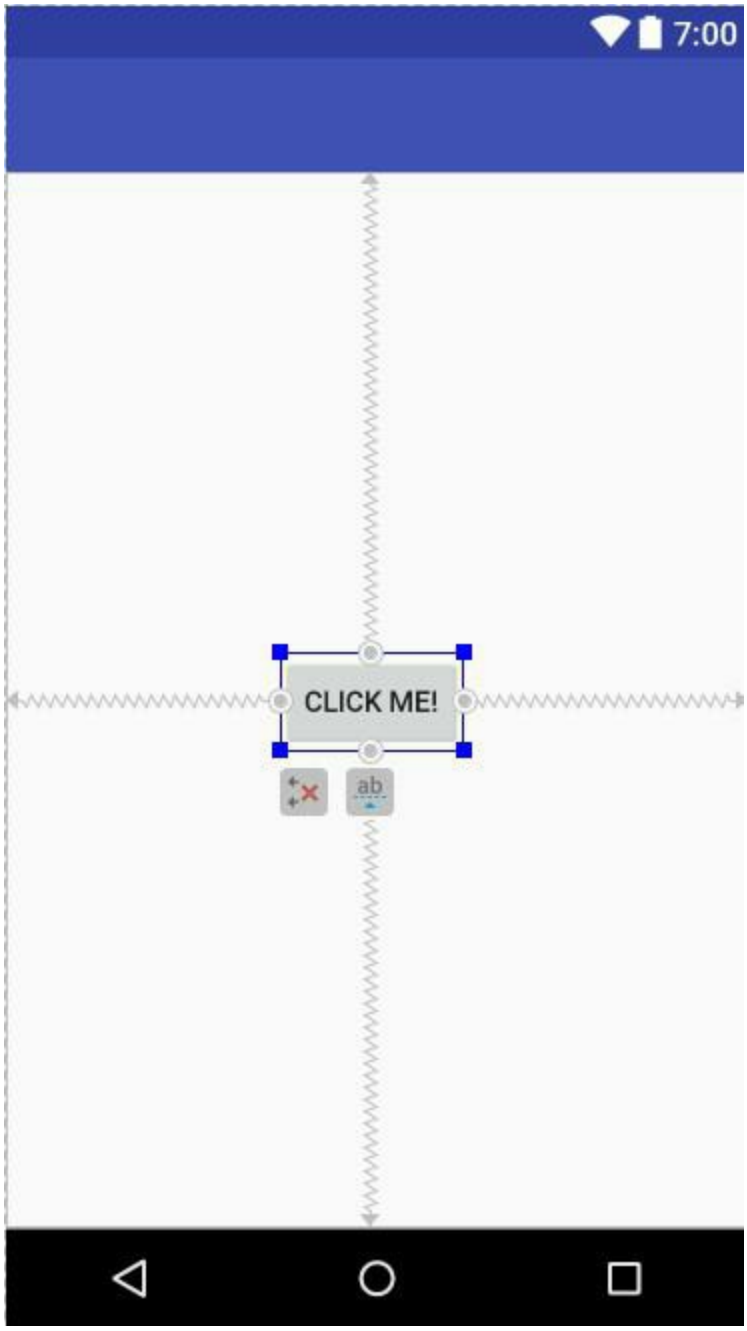


Figure 18-2

The appearance of constraint connections in both views can be toggled using the toolbar button highlighted in Figure 18-3:

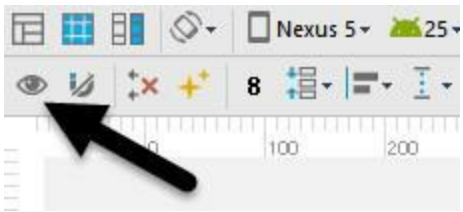


Figure 18-3

In addition to the two modes of displaying the user interface layout, the Layout Editor tool also provides three different ways of establishing the constraints required for a specific layout design.

18.2 Autoconnect Mode

Autoconnect, as the name suggests, automatically establishes constraint connections as items are added to the layout. Autoconnect mode may be enabled and disabled using the toolbar button indicated in Figure 18-4:

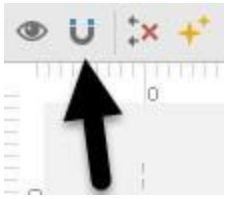


Figure 18-4

Autoconnect mode uses algorithms to decide the best constraints to establish based on the position of the widget and the widget's proximity to both the sides of the parent layout and other elements in the layout. In the event that any of the automatic constraint connections fail to provide the desired behavior, these may be changed manually as outlined later in this chapter.

18.3 Inference Mode

Inference mode uses a heuristic approach involving algorithms and probabilities to automatically implement constraint connections after widgets have already been added to the layout. This mode is usually used when the Autoconnect feature has been turned off and objects have been added to the layout without any constraint connections. This allows the layout to be designed simply by dragging and dropping objects from the palette onto the layout canvas and making size and positioning changes until the layout appears as required. In essence this involves "painting" the layout without worrying about constraints. Inference mode may also be used at any time during the design process to fill in missing constraints within a layout.

Constraints are automatically added to a layout when the *Infer constraints* button (Figure 18-5) is clicked:

Figure 18-5

As with Autoconnect mode, there is always the possibility that the Layout Editor tool will infer incorrect constraints, though these may be modified and corrected manually.

18.4 Manipulating Constraints Manually

The third option for implementing constraint connections is to do so manually. When doing so, it will be helpful to understand the various handles that appear around a widget within the Layout Editor tool. Consider, for example, the widget shown in Figure 18-6:

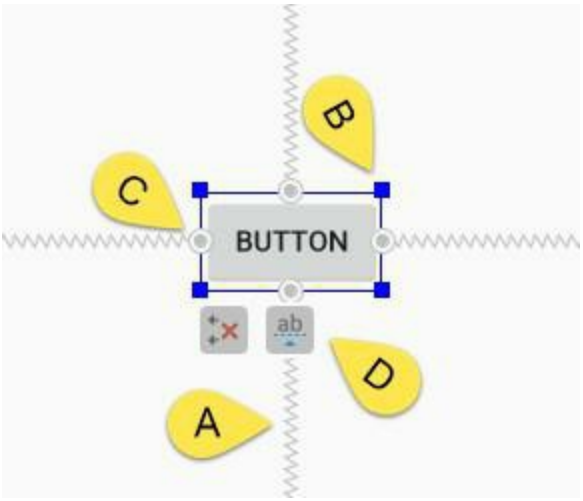


Figure 18-6

Clearly the spring-like lines (A) represent established constraint connections leading from the sides of the widget to the targets. The small square markers (B) in each corner of the object are resize handles which, when clicked and dragged, serve to resize the widget. The small circle handles (C) located on each side of the widget are the side constraint anchors. To create a constraint connection, click on the handle and drag the resulting line to the element to which the constraint is to be connected (such as a guideline or the side of either the parent layout or another widget) as outlined in Figure 18-7. When connecting to the side of another widget, simply drag the line to the side constraint handle of that widget and, when it turns green, release the line.

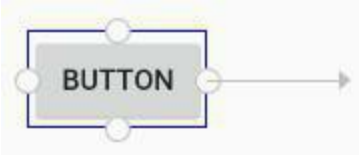


Figure 18-7

An additional marker indicates the anchor point for baseline constraints whereby the content within the widget (as opposed to outside edges) is used as the alignment point. To display this marker, simply click on the button displaying the letters ‘ab’ (referenced by marker D in Figure 18-6). To establish a constraint connection from a baseline constraint handle, simply hover the mouse pointer over the handle until it begins to flash before clicking and dragging to the target (such as the baseline anchor of another widget as shown in Figure 18-8). When the destination anchor begins to flash green, release the mouse button to make the constraint connection:

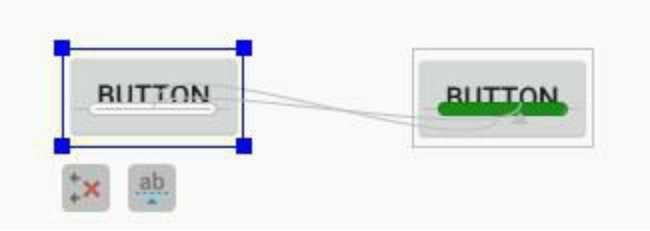


Figure 18-8

To hide the baseline anchors, simply click on the baseline button a second time.

18.5 **Deleting Constraints**

To delete an individual constraint, simply click within the anchor to which it is connected. The constraint will then be deleted from the layout (when hovering over the anchor it will glow red to indicate that clicking will perform a deletion).

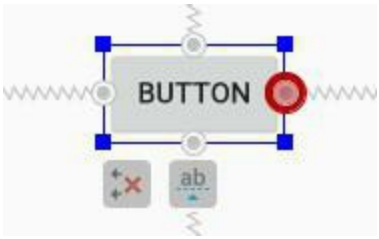


Figure 18-9

Alternatively, remove all of the constraints on a widget by selecting it and clicking on the *Delete All Constraints* button which appears next to the widget when it is selected in the layout as highlighted in Figure 18-10:

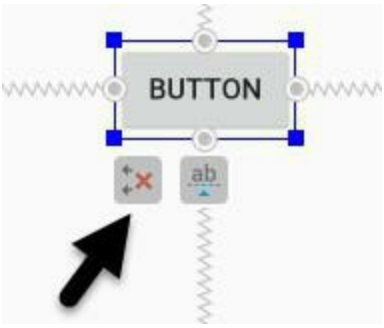


Figure 18-10

To remove all of the constraints from every widget in a layout, right-click on the layout and select the *Clear all constraints* option from the resulting menu, or use the toolbar button highlighted in Figure 18-11:

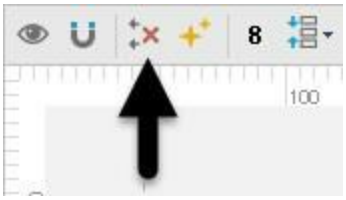


Figure 18-11

18.6 **Adjusting Constraint Bias**

In the previous chapter, the concept of using bias settings to favor one opposing constraint over another was outlined. Bias within the Android Studio Layout Editor tool is adjusted using the *Inspector* located in the Properties tool window and shown in Figure 18-12. The two sliders indicated by the arrows in the figure are used to control the bias of the vertical and horizontal opposing constraints of the currently selected widget.

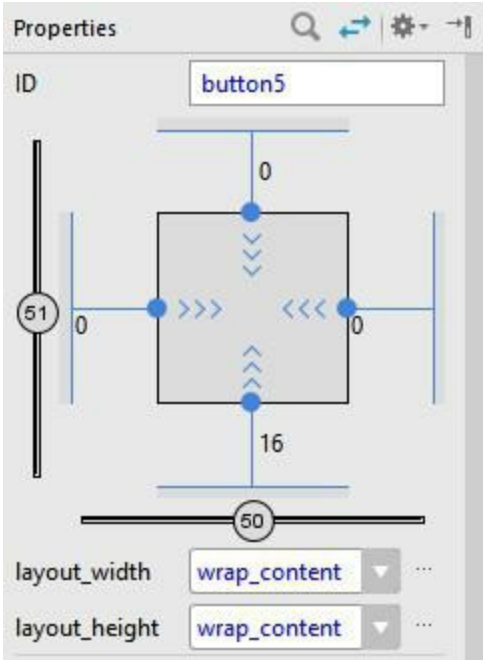


Figure 18-12

18.7 Understanding ConstraintLayout Margins

Constraints can be used in conjunction with margins to implement fixed gaps between a widget and another element (such as another widget, a guideline or the side of the parent layout). Consider, for example, the horizontal constraints applied to the Button object in Figure 18-13:

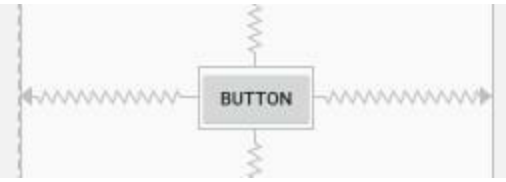


Figure 18-13

As currently configured, horizontal constraints run to the left and right edges of the parent ConstraintLayout. As such, the widget has opposing horizontal constraints indicating that the ConstraintLayout layout engine has some discretion in terms of the actual positioning of the widget at runtime. This allows the layout some flexibility to accommodate different screen sizes and device orientation. The horizontal bias setting is also able to control the position of the widget right up to the right-hand side of the layout. Figure 18-14, for example, shows the same button with 100% horizontal bias applied:

Figure 18-14

ConstraintLayout margins can appear at the end of constraint connections and represent a fixed gap into which the widget cannot be moved even when adjusting bias or in response to layout changes elsewhere in the activity. In Figure 18-15, the right-hand constraint now includes a 50dp margin into which the widget cannot be moved even though the bias is still set at 100%.

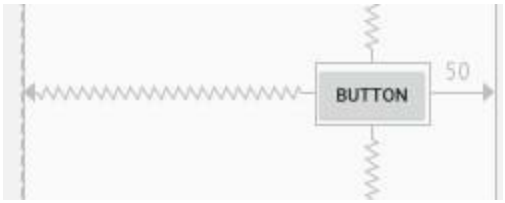


Figure 18-15

Existing margin values on a widget can be modified from within the Inspector. As can be seen in Figure 18-16, a dropdown menu is being used to change the right-hand margin on the currently selected widget to 16dp. Alternatively, clicking on the current value also allows a number to be typed into the field.

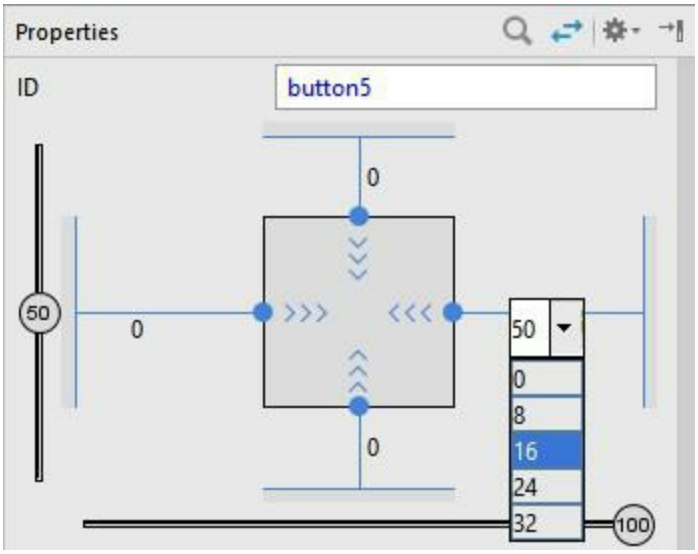


Figure 18-16

The default margin for new constraints can be changed at any time using the option in the toolbar highlighted in Figure 18-17:

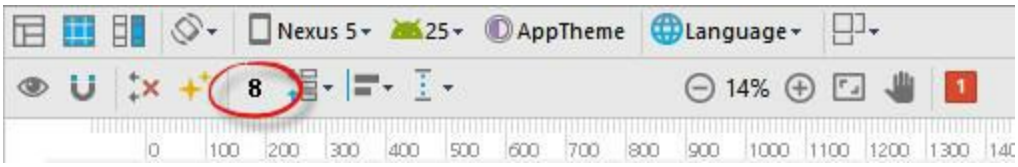


Figure 18-17

Margin constraints can also be created by clicking in the constraint handle of a widget, pressing and holding the Ctrl key and then dragging the line to the target. Once released, the margin constraint will be set at a fixed distance based on the current position of the widget:

Figure 18-18

18.8 The Importance of Opposing Constraints and Bias

As discussed in the previous chapter, opposing constraints, margins and bias form the cornerstone of responsive layout design in Android when using the ConstraintLayout. When a widget is constrained without opposing constraint connections, those constraints are essentially margin constraints. This is indicated visually within the Layout Editor tool by solid straight lines accompanied by margin

measurements as shown in Figure 18-19.

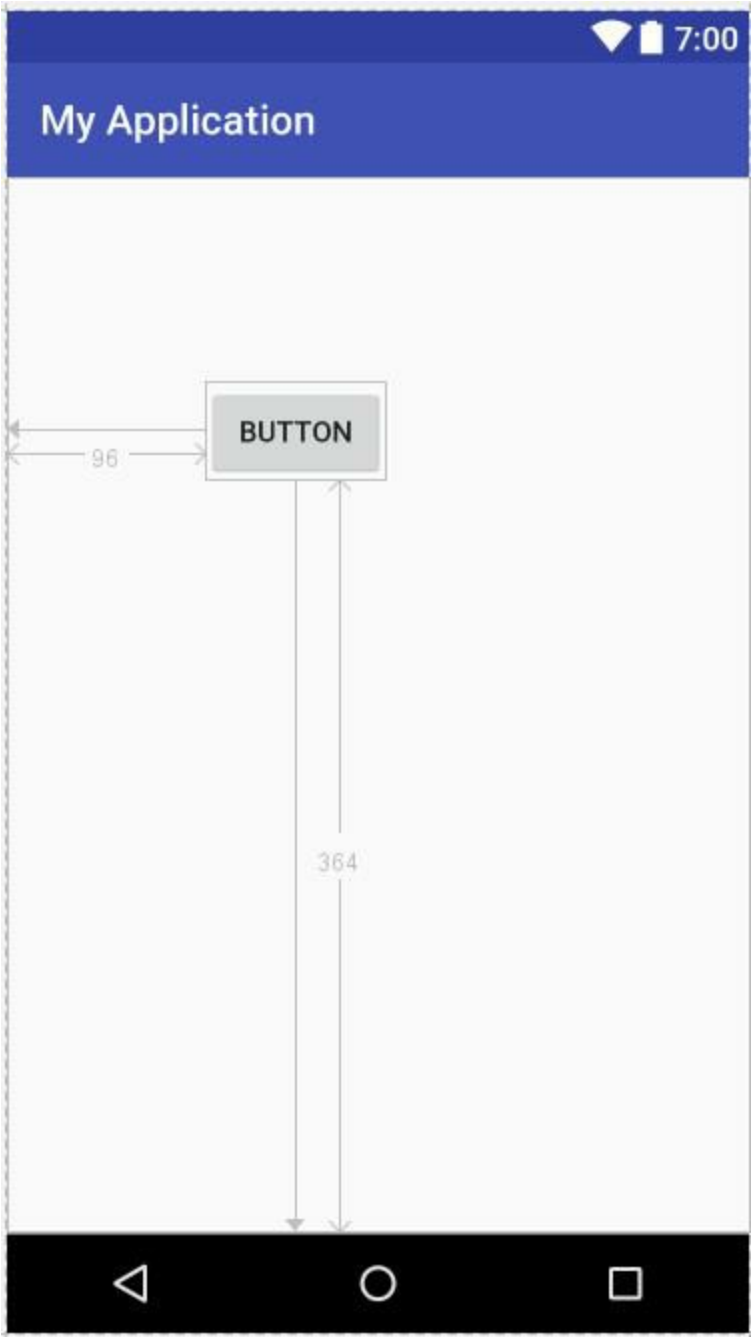


Figure 18-19

The above constraints essentially fix the widget at that position. The result of this is that if the device is rotated to landscape orientation, the widget will no longer be visible since the vertical constraint pushes it beyond the top edge of the device screen (as is the case in Figure 18-7). A similar problem will arise if the app is run on a device with a smaller screen than that used during the design process.

Figure 18-20

When opposing constraints are implemented, the constraint connection is represented by the spring-like jagged line (the spring metaphor is intended to indicate that the position of the widget is not fixed to absolute X and Y coordinates):

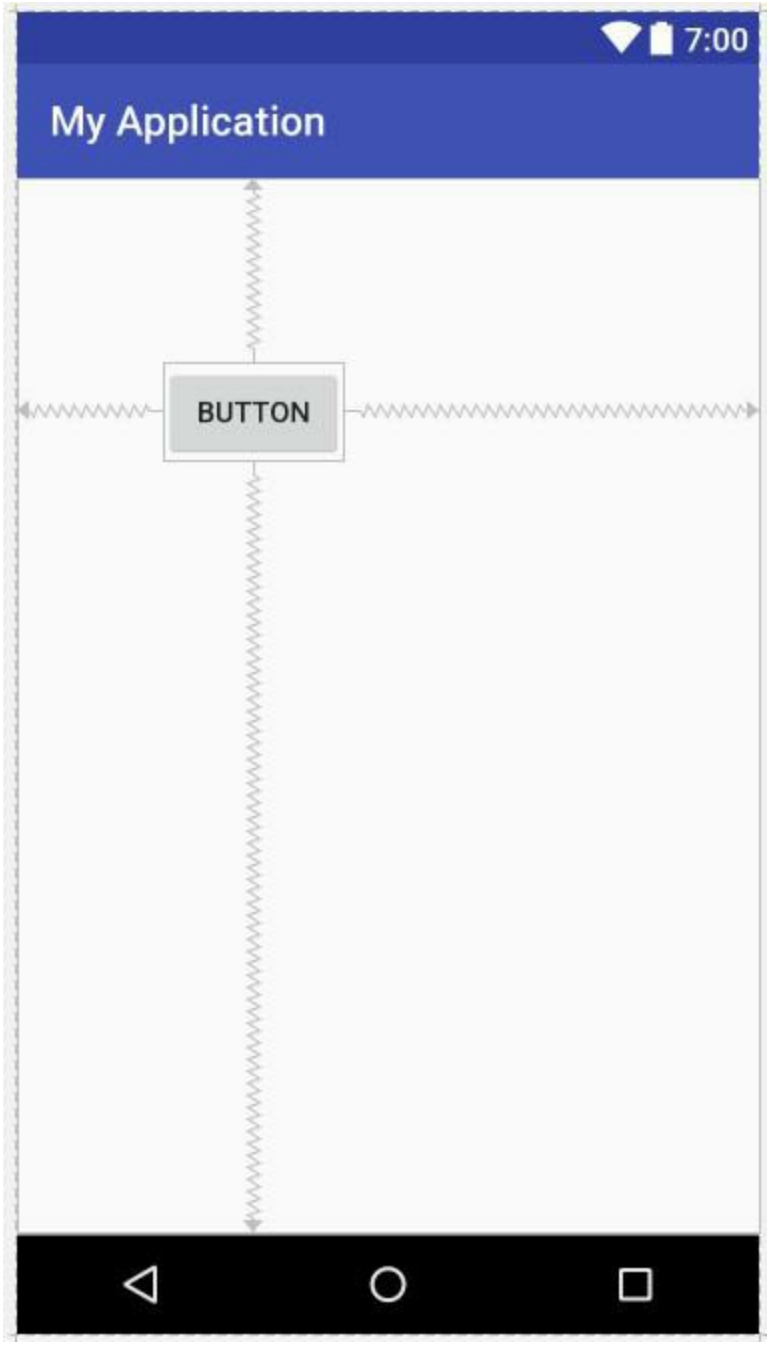


Figure 18-21

In the above layout, vertical and horizontal bias settings have been configured such that the widget will always be positioned 90% of the distance from the bottom and 35% from the left-hand edge of the parent layout. When rotated, therefore, the widget is still visible and positioned in the same location relative to the dimensions of the screen:

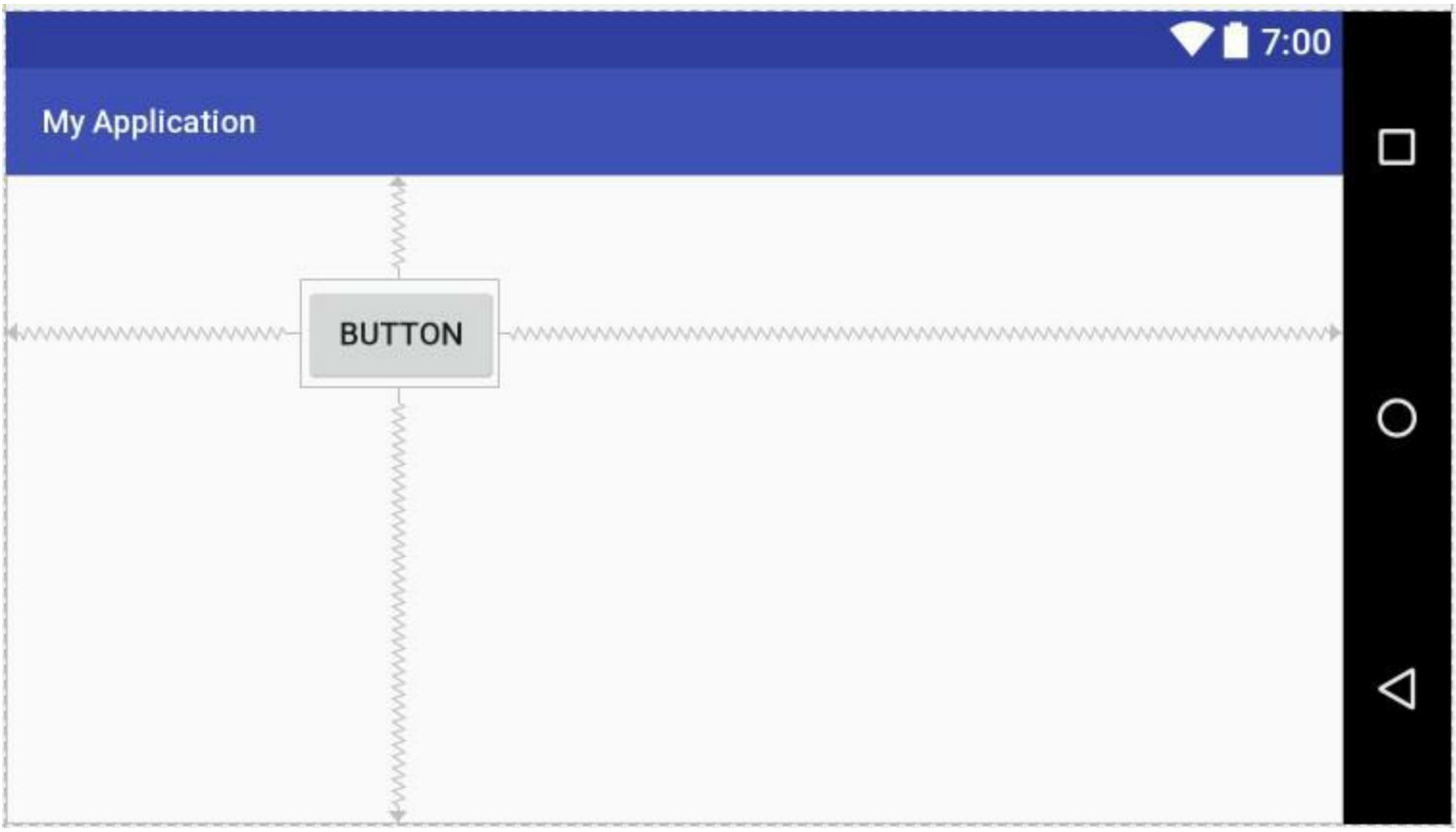


Figure 18-22

When designing a responsive and adaptable user interface layout, it is important to take into consideration both bias and opposing constraints when manually designing a user interface layout and making corrections to automatically created constraints.

18.9 Configuring Widget Dimensions

The inner dimensions of a widget within a ConstraintLayout can also be configured using the Inspector. As outlined in the previous chapter, widget dimensions can be set to wrap content, fixed or match constraints modes. The prevailing settings for each dimension on the currently selected widget are shown within the square representing the widget in the Inspector as illustrated in Figure 18-23:

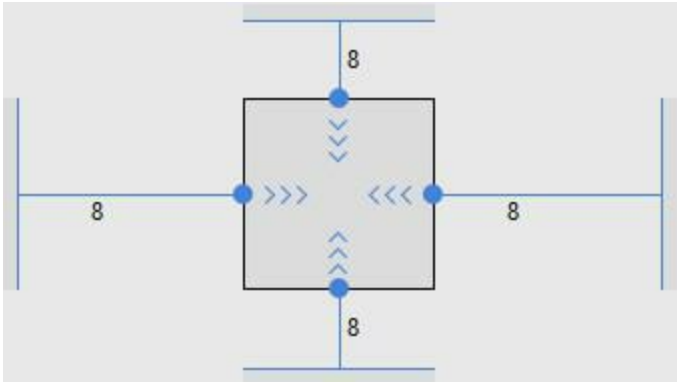


Figure 18-23

In the above figure, both the horizontal and vertical dimensions are set to wrap content mode (indicated by the inward pointing chevrons). The inspector uses the following visual indicators to represent the three dimension modes:

Fixed Size

Match Constraints

Wrap Content

To change the current setting, simply click on the indicator to cycle through the three different settings.

In addition, the size of a widget can be expanded either horizontally or vertically to the maximum amount allowed by the constraints and other widgets in the layout using the *Expand horizontally* and *Expand vertically* buttons. These are accessible by right clicking on a widget within the layout and selecting the option from the resulting menu (Figure 18-24). When used, the currently selected widget will increase in size horizontally or vertically to fill the available space around it.

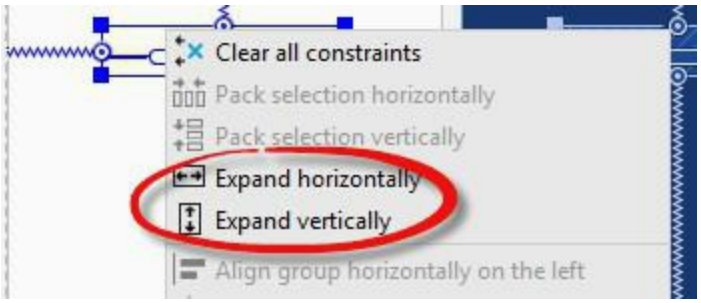


Figure 18-24

18.10 **Adding Guidelines**

Guidelines provide additional elements to which constraints may be anchored. Guidelines are added by right-clicking on the layout and selecting either the *Add Vertical Guideline* or *Add Horizontal Guideline* menu option or using the toolbar menu options as shown in Figure 18-25:

Figure 18-25

Once added, a guideline will appear as a dashed line in the layout and may be moved simply by clicking and dragging the line. To establish a constraint connection to a guideline, click in the constraint handler of a widget and drag to the guideline before releasing. In Figure 18-26, the left sides of two Buttons are connected by constraints to a vertical guideline.

The position of a vertical guideline can be specified as an absolute distance from either the left or the right of the parent layout (or the top or bottom for a horizontal guideline). The vertical guideline in the above figure, for example, is positioned 97dp from the left-hand edge of the parent.

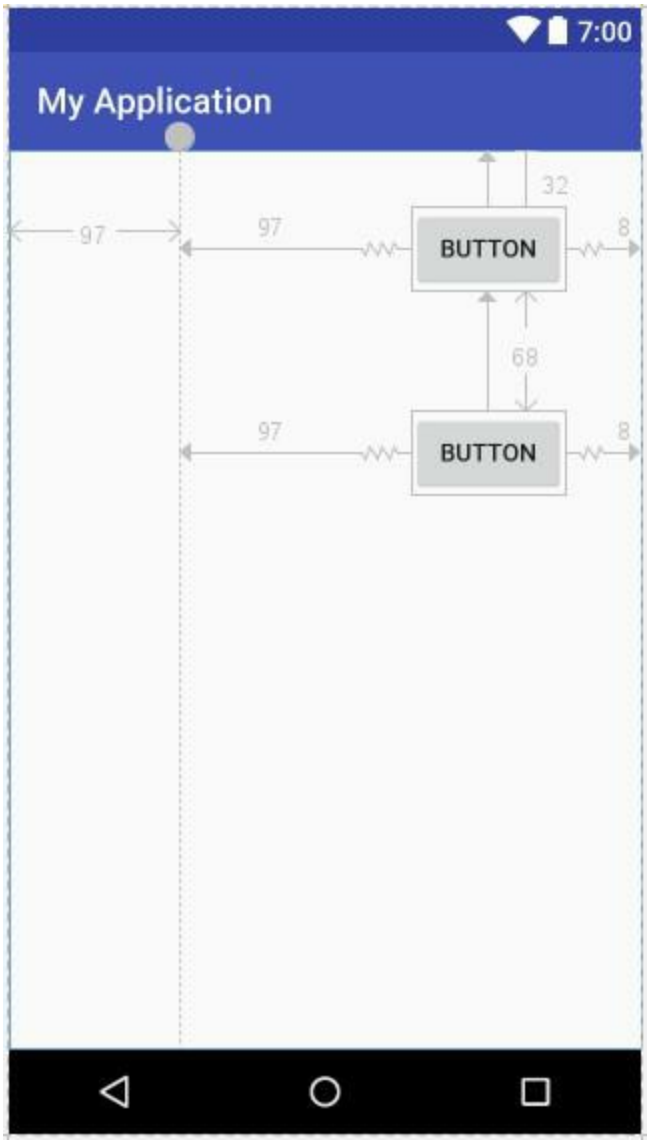


Figure 18-26

Alternatively, the guideline may be positioned as a percentage of overall width or height of the parent layout. To switch between these three modes, select the guideline and click on the circle at the top or start of the guideline (depending on whether the guideline is vertical or horizontal). Figure 18-27, for example, shows a guideline positioned based on percentage:

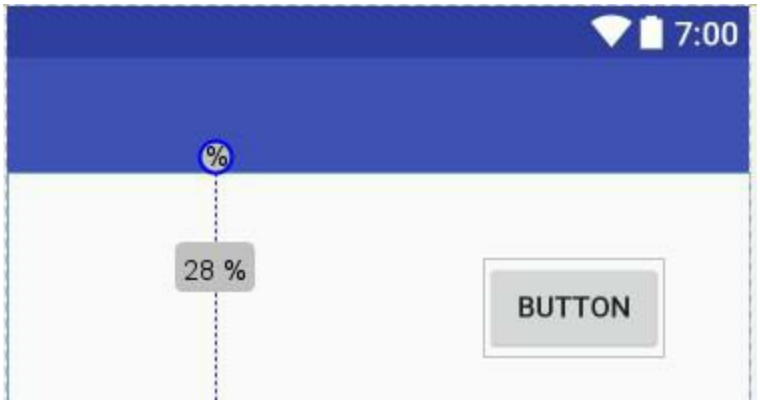


Figure 18-27

18.11 **Widget Group Alignment**

The Android Studio Layout Editor tool provides a range of alignment actions that can be performed when two or more widgets are selected in the layout. Simply shift-click on each of the widgets to be included in the action, right-click on the layout and make a selection from the many options displayed

in the menu:

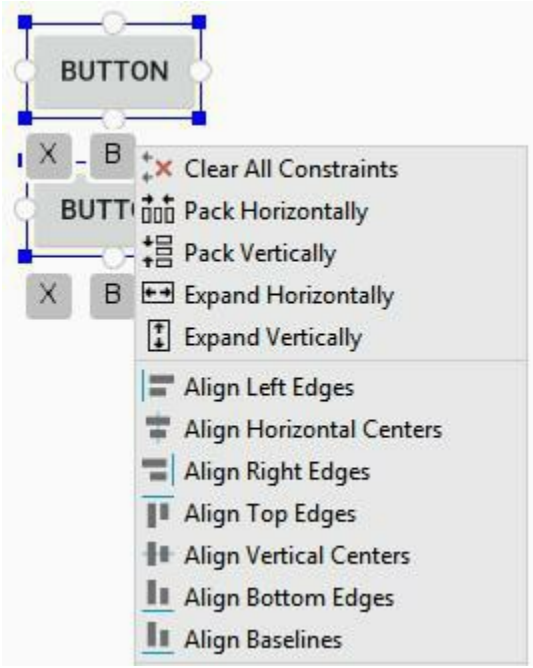


Figure 18-28

As shown in Figure 18-29 below, these options are also available as buttons in the Layout Editor toolbar:

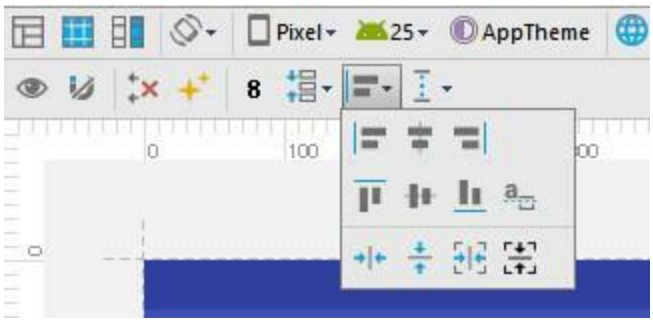


Figure 18-29

18.12 Converting other Layouts to ConstraintLayout

For existing user interface layouts that make use of one or more of the other Android layout classes (such as RelativeLayout or LinearLayout), the Layout Editor tool provides an option to convert the user interface to use the ConstraintLayout.

When the Layout Editor tool is open and in Design mode, the Component Tree panel is displayed beneath the Palette. To convert a layout to ConstraintLayout, locate it within the Component Tree, right-click on it and select the *Convert <current layout> to Constraint Layout* menu option:

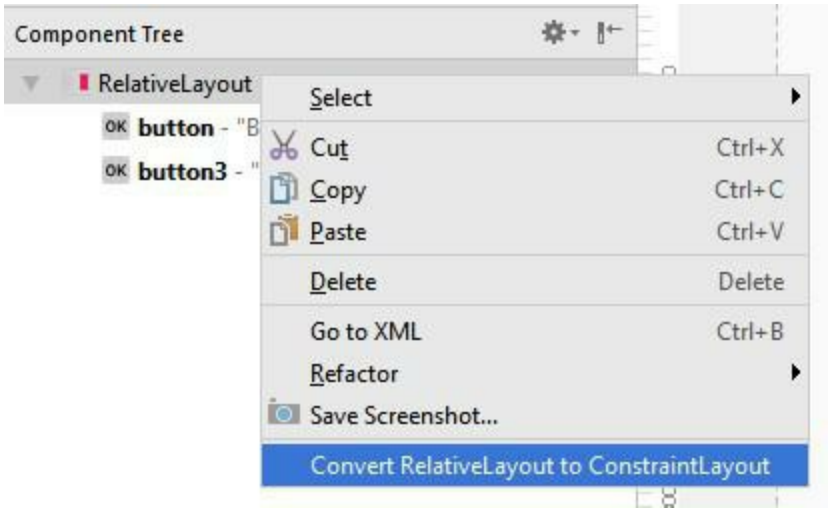


Figure 18-30

When this menu option is selected, Android Studio will convert the selected layout to a ConstraintLayout and use inference to establish constraints designed to match the layout behavior of the original layout type.

18.13 Summary

A redesigned Layout Editor tool combined with ConstraintLayout makes designing complex user interface layouts with Android Studio a relatively fast and intuitive process. This chapter has covered the concepts of constraints, margins and bias in more detail while also exploring the ways in which ConstraintLayout-based design has been integrated into the Layout Editor tool.

19. Working with ConstraintLayout Chains and Ratios in Android Studio

The previous chapters have introduced the key features of the ConstraintLayout class and outlined the best practices for ConstraintLayout-based user interface design within the Android Studio Layout Editor. Although the concepts of ConstraintLayout chains and ratios were outlined in the chapter entitled *A Guide to the Android ConstraintLayout*, we have not yet addressed how to make use of these features within the Layout Editor. The focus of this chapter, therefore, is to provide practical steps on how to create and manage chains and ratios when using the ConstraintLayout class.

19.1 Creating a Chain

Although improved support for chains is expected in future editions of Android Studio, the current Android Studio 2.3 release does not provide extensive support for visually creating ConstraintLayout chains within the Layout Editor tool. That being said, chains may be implemented by adding a few lines to the XML layout resource file of an activity or by using a shortcut involving use of the Layout Editor Design mode *Center Horizontally* or *Center Vertically* menu options.

Consider a layout consisting of three Button widgets constrained so as to be positioned in the top-left, top-center and top-right of the ConstraintLayout parent as illustrated in Figure 19-1:

Figure 19-1

To represent such a layout, the XML resource layout file might contain the following entries for the button widgets:

```
<Button
    android:text="Button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button1"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginLeft="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginStart="16dp"
    app:layout_constraintLeft_toLeftOf="parent" />

<Button
    android:text="Button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button2"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintRight_toLeftOf="@+id/button3"
```

```
        android:layout_marginRight="8dp"
        app:layout_constraintLeft_toRightOf="@+id/button1"
        android:layout_marginLeft="8dp" />

    <Button
        android:text="Button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button3"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="16dp"
        android:layout_marginRight="16dp"
        android:layout_marginEnd="16dp"
        app:layout_constraintRight_toRightOf="parent" />
```

As currently configured, there are no bi-directional constraints to group these widgets into a chain. To address this, additional constraints need to be added from the right-hand side of button1 to the left side of button2, and from the left side of button3 to the right side of button2 as follows:

```
    <Button
        android:text="Button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button1"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginLeft="8dp"
        android:layout_marginTop="16dp"
        android:layout_marginStart="16dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toLeftOf="@id/button2" />

    <Button
        android:text="Button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button2"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintRight_toLeftOf="@+id/button3"
        android:layout_marginRight="8dp"
        app:layout_constraintLeft_toRightOf="@+id/button1"
        android:layout_marginLeft="8dp" />

    <Button
        android:text="Button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button3"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="16dp"
        android:layout_marginRight="16dp"
        android:layout_marginEnd="16dp"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintLeft_toRightOf="@id/button2" />
```

With these changes, the widgets now have bi-directional horizontal constraints configured. This essentially constitutes a ConstraintLayout chain which is represented visually within the Layout

Editor by chain connections as shown in Figure 19-2 below. Note that in this configuration the chain has defaulted to the spread chain style.

Figure 19-2

Although the bi-directional constraints have been added manually via the XML code in this example, it is also useful to know that bi-directional constraints may also be added by selecting all of the widgets to be included in the chain, right-clicking on one of the widgets and selecting either the *Center Horizontally* or *Center Vertically* menu option (depending on whether a horizontal or vertical chain is being created).

19.2 Changing the Chain Style

If no chain style is configured, the ConstraintLayout will default to the spread chain style. The chain style can be altered by selecting any of the widgets in the chain and clicking on the chain button as highlighted in Figure 19-3:

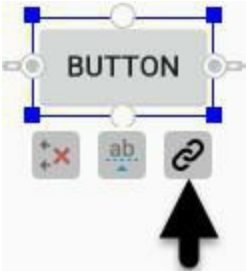


Figure 19-3

Each time the chain button is clicked the style will switch to another setting in the order of spread, spread inside and packed.

Alternatively, the style may be specified in the properties tool window by clicking on the *View all properties* link, unfolding the *Constraints* section and changing either the *layoutConstraintHorizontal_chainStyle* or *layoutConstraintVertical_chainStyle* property depending on the orientation of the chain:

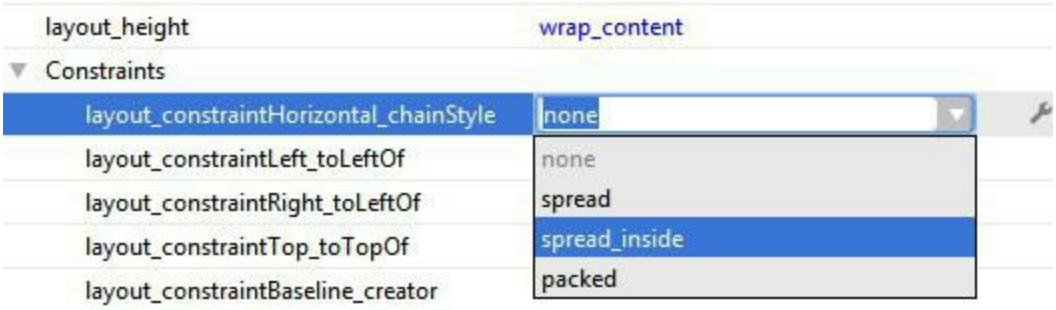


Figure 19-4

19.3 Spread Inside Chain Style

Figure 19-5 illustrates the effect of changing the chain style to spread inside chain style using the

above techniques:

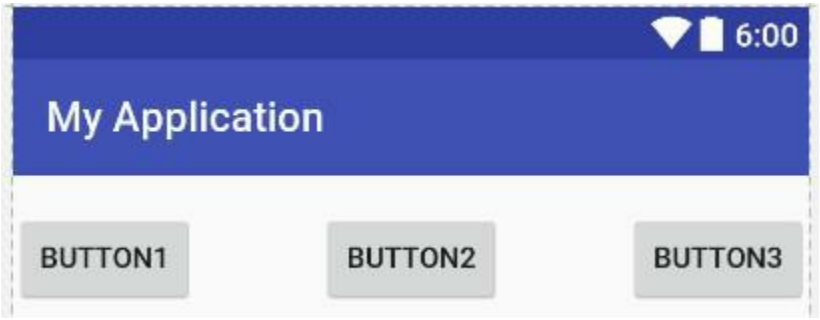


Figure 19-5

19.4 Packed Chain Style

Using the same technique, changing the chain style property to *packed* causes the layout to change as shown in Figure 19-6:

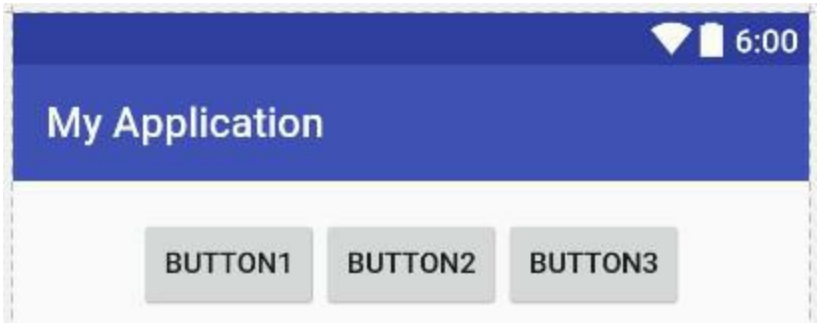


Figure 19-6

19.5 Packed Chain Style with Bias

The positioning of the packed chain may be influenced by applying a bias value. The bias can be any value between 0.0 and 1.0, with 0.5 representing the center of the parent. Bias is controlled by selecting the chain head widget and assigning a value to the *layout_constraintHorizontal_bias* or *layout_constraintVertical_bias* attribute in the Properties panel. Figure 19-7 shows a packed chain with a horizontal bias setting of 0.2:

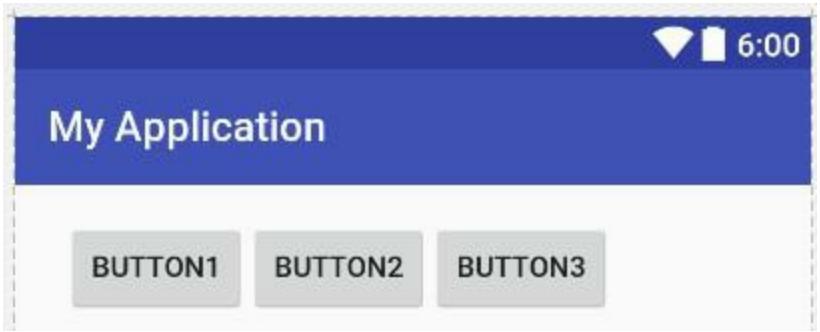


Figure 19-7

19.6 Weighted Chain

The final area of chains to explore involves weighting of the individual widgets to control how much space each widget in the chain occupies within the available space. A weighted chain may only be implemented using the *spread* chain style and any widget within the chain that is to respond to the weight property must have the corresponding dimension property (height for a vertical chain and width for a horizontal chain) configured for *match constraint* mode. Match constraint mode for a widget dimension may be configured by selecting the widget, displaying the Properties panel and

changing the dimension to 0dp. In Figure 19-8, for example, the layout_width constraint for button1 has been set to 0dp to indicate that the width of the widget is to be determined based on the prevailing constraint settings:

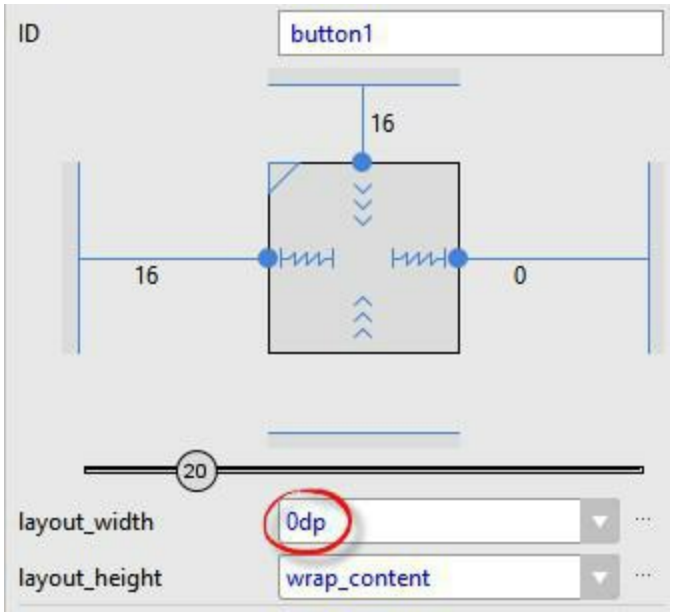


Figure 19-8

Assuming that the spread chain style has been selected, and all three buttons have been configured such that the width dimension is set to match the constraints, the widgets in the chain will expand equally to fill the available space:

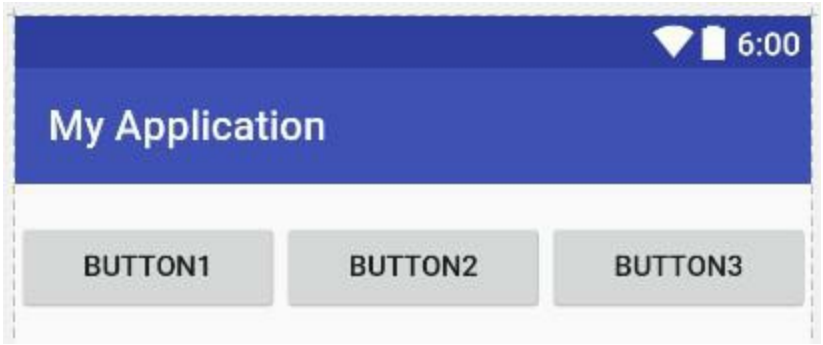


Figure 19-9

The amount of space occupied by each widget relative to the other widgets in the chain can be controlled by adding weight properties to the widgets. Figure 19-10 shows the effect of setting the *layout_constraintHorizontal_weight* property to 4 on button1, and to 2 on both button2 and button3:

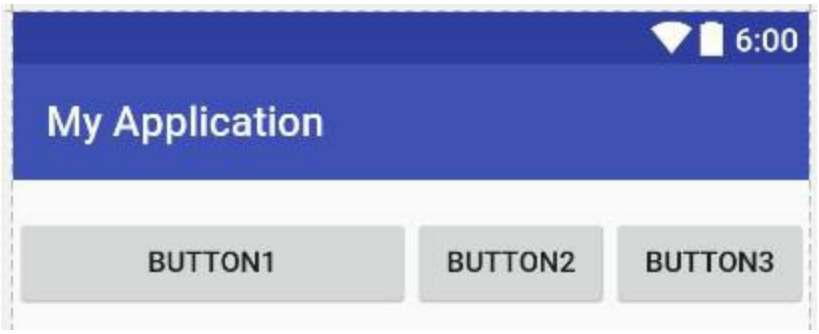


Figure 19-10

As a result of these weighting values, button1 occupies half of the space (4/8), while button2 and button3 each occupy one quarter (2/8) of the space.

19.7 Working with Ratios

ConstraintLayout ratios allow one dimension of a widget to be sized relative to the widget's other dimension (otherwise known as aspect ratio). An aspect ratio setting could, for example, be applied to an ImageView to ensure that its width is always twice its height.

A dimension ratio constraint is configured by setting the constrained dimension to match constraint mode (by setting the dimension to 0dp as outlined in the previous section) and configuring the *layout_constraintDimensionRatio* attribute on that widget to the required ratio. This ratio value may be specified either as a float value or a *width:height* ratio setting. The following XML excerpt, for example, configures a ratio of 2:1 on an ImageView widget:

```
<ImageView
        android:layout_width="0dp"
        android:layout_height="100dp"
        android:id="@+id/imageView"
        app:layout_constraintDimensionRatio="2:1" />
```

The above example demonstrates how to configure a ratio when only one dimension is set to match constraints. A ratio may also be applied when both dimensions are set to match constraint mode. This involves specifying the ratio preceded with either an H or a W to indicate which of the dimensions is constrained relative to the other.

Consider, for example, the following XML excerpt for an ImageView object:

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:id="@+id/imageView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintDimensionRatio="W,1:3" />
```

In the above example the height will be defined subject to the constraints applied to it. In this case constraints have been configured such that it is attached to the top and bottom of the parent view, essentially stretching the widget to fill the entire height of the parent. The width dimension, on the other hand, has been constrained to be one third of the ImageView's height dimension. Consequently, whatever size screen or orientation the layout appears on, the ImageView will always be the same height as the parent and the width one third of that height.

The same results may also be achieved without the need to manually edit the XML resource file. Whenever a widget dimension is set to match constraint mode, a ratio control toggle appears in the Inspector area of the property panel. Figure 19-11, for example, shows the layout width and height properties of a button widget set to match constraint mode and 100dp respectively, and highlights the ratio control toggle in the widget sizing preview:

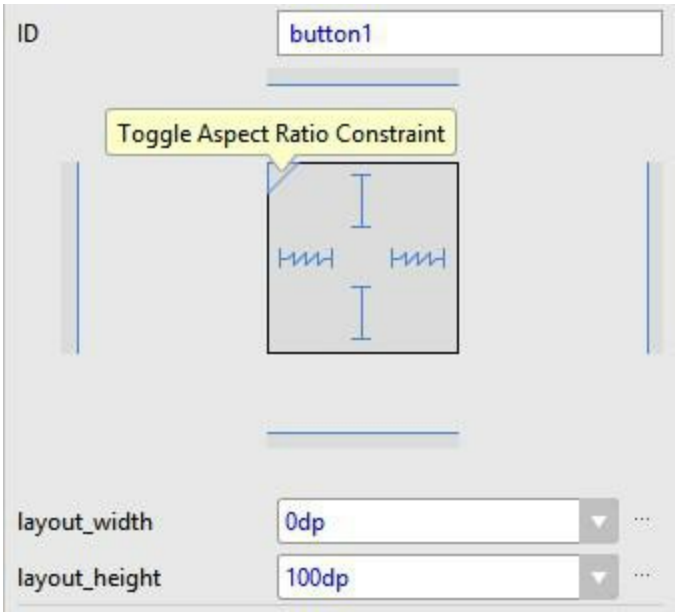


Figure 19-11

By default the ratio sizing control is toggled off. Clicking on the control enables the ratio constraint and displays an additional field where the ratio may be changed:

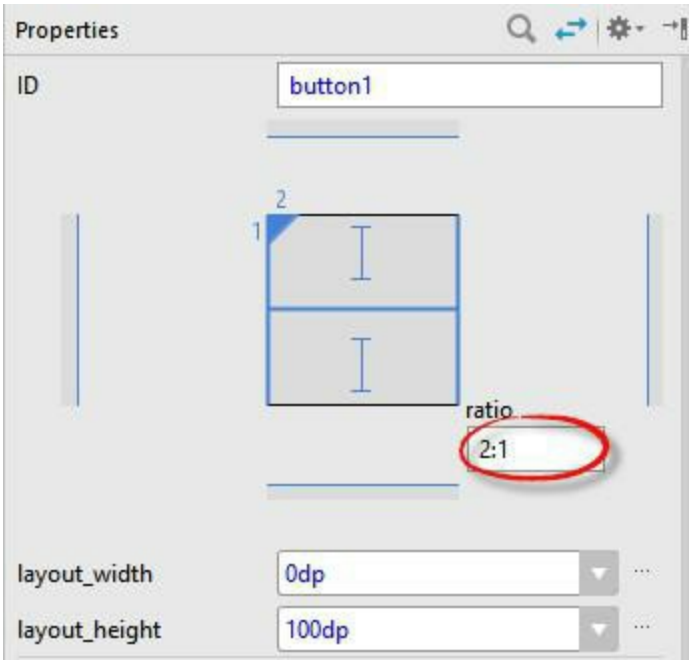


Figure 19-12

19.8 **Summary**

Both chains and ratios are powerful features of the ConstraintLayout class intended to provide additional options for designing flexible and responsive user interface layouts within Android applications. As outlined in this chapter, the Android Studio Layout Editor has been enhanced for Android Studio 2.3 to make it easier to use these features during the user interface design process.

20. An Android Studio Layout Editor ConstraintLayout Tutorial

By far the easiest and most productive way to design a user interface for an Android application is to make use of the Android Studio Layout Editor tool. The goal of this chapter is to provide an overview of how to create a ConstraintLayout-based user interface using this approach. The exercise included in this chapter will also be used as an opportunity to outline the creation of an activity starting with a "bare-bones" Android Studio project.

Having covered the use of the Android Studio Layout Editor, the chapter will also introduce the Layout Inspector and Hierarchy Viewer tools.

20.1 An Android Studio Layout Editor Tool Example

The first step in this phase of the example is to create a new Android Studio project. Begin, therefore, by launching Android Studio and closing any previously opened projects by selecting the *File -> Close Project* menu option. Within the Android Studio welcome screen click on the *Start a new Android Studio project* quick start option to display the first screen of the new project dialog.

Enter *LayoutSample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button and set the minimum SDK to API 14: Android 4.0 (IceCreamSandwich).

In previous examples, we have requested that Android Studio create a template activity for the project. We will, however, be using this tutorial to learn how to create an entirely new activity and corresponding layout resource file manually, so click *Next* once again and make sure that the *Add No Activity* option is selected before clicking on *Finish* to create the new project.

20.2 Creating a New Activity

Once the project creation process is complete, the Android Studio main window should appear with no tool windows open.

The next step in the project is to create a new activity. This will be a valuable learning exercise since there are many instances in the course of developing Android applications where new activities need to be created from the ground up.

Begin by displaying the Project tool window using the Alt-1 keyboard shortcut. Once displayed, unfold the hierarchy by clicking on the right facing arrows next to the entries in the Project window. The objective here is to gain access to the *app -> java -> com.ebookfrenzy.layoutsample* folder in the project hierarchy. Once the package name is visible, right-click on it and select the *New -> Activity -> Empty Activity* menu option as illustrated in Figure 20-1:

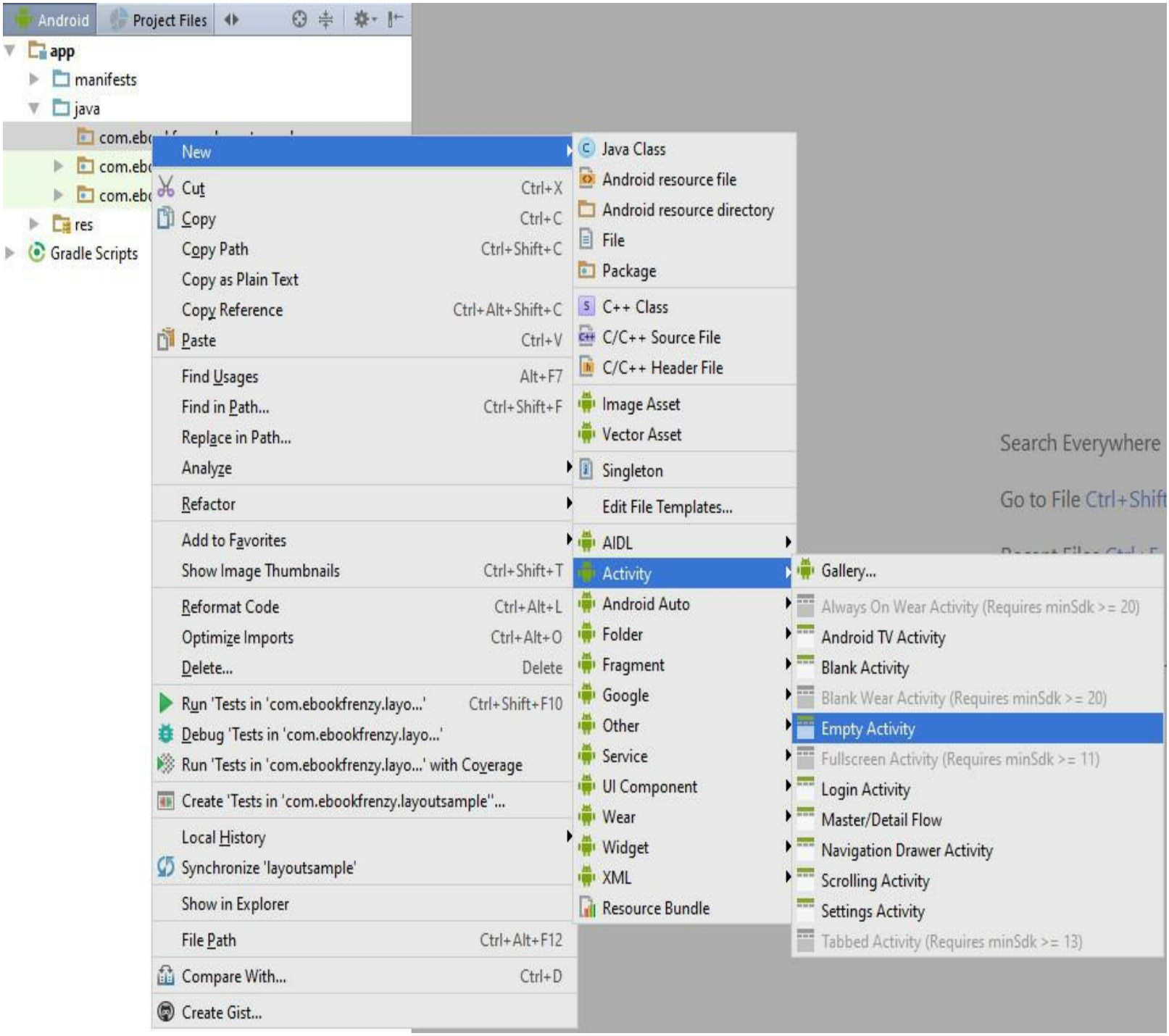


Figure 20-1

In the resulting *New Activity* dialog, name the new activity *LayoutSampleActivity* and the layout *activity_layout_sample*. The activity will, of course, need a layout resource file so make sure that the *Generate Layout File* option is enabled.

In order for an application to be able to run on a device it needs to have an activity designated as the *launcher activity*. Without a launcher activity, the operating system will not know which activity to start up when the application first launches and the application will fail to start. Since this example only has one activity, it needs to be designated as the launcher activity for the application so make sure that the *Launcher Activity* option is enabled before clicking on the *Finish* button.

At this point Android Studio should have added two files to the project. The Java source code file for the activity should be located in the *app -> java -> com.ebookfrenzy.layoutsample* folder.

In addition, the XML layout file for the user interface should have been created in the *app -> res -> layout* folder. Note that the Empty Activity template was chosen for this activity so the layout is

contained entirely within the *activity_layout_sample.xml* file and there is no separate content layout file.

Finally, the new activity should have been added to the *AndroidManifest.xml* file and designated as the launcher activity. The manifest file can be found in the project window under the *app -> manifests* folder and should contain the following XML:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".LayoutSampleActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
     android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

20.3 Preparing the Layout Editor Environment

Locate and double-click on the *activity_layout_sample.xml* layout file located in the *app -> res -> layout* folder to load it into the Layout Editor tool. Since the purpose of this tutorial is to gain experience with the use of constraints, turn off the Autoconnect feature using the button located in the Layout Editor toolbar. Once disabled, the button will appear with a line through as is the case in Figure 20-2:

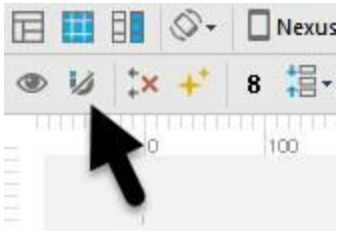


Figure 20-2

The user interface design will also make use of the ImageView object to display an image. Before proceeding, this image should be added to the project ready for use later in the chapter. This file is named *galaxys6.png* and can be found in the *project_icons* folder of the sample code download available from the following URL:

http://www.ebookfrenzy.com/retail/androidstudio23/index.php

Locate this image in the file system navigator for your operating system and copy the image file. Right-click on the *app -> res -> drawable* entry in the Project tool window and select Paste from the

menu to add the file to the folder. When the copy dialog appears, click on OK to accept the default settings.

Figure 20-3

20.4 Adding the Widgets to the User Interface

From within the *Images* palette category, drag an ImageView object into the center of the display view. Note that horizontal and vertical dashed lines appear indicating the center axes of the display. When centered, release the mouse button to drop the view into position. Once placed within the layout, the Resources dialog will appear seeking the image to be displayed within the view. In the search bar located at the top of the dialog, enter "galaxy" to locate the *galaxys6.png* resource as illustrated in Figure 20-4.

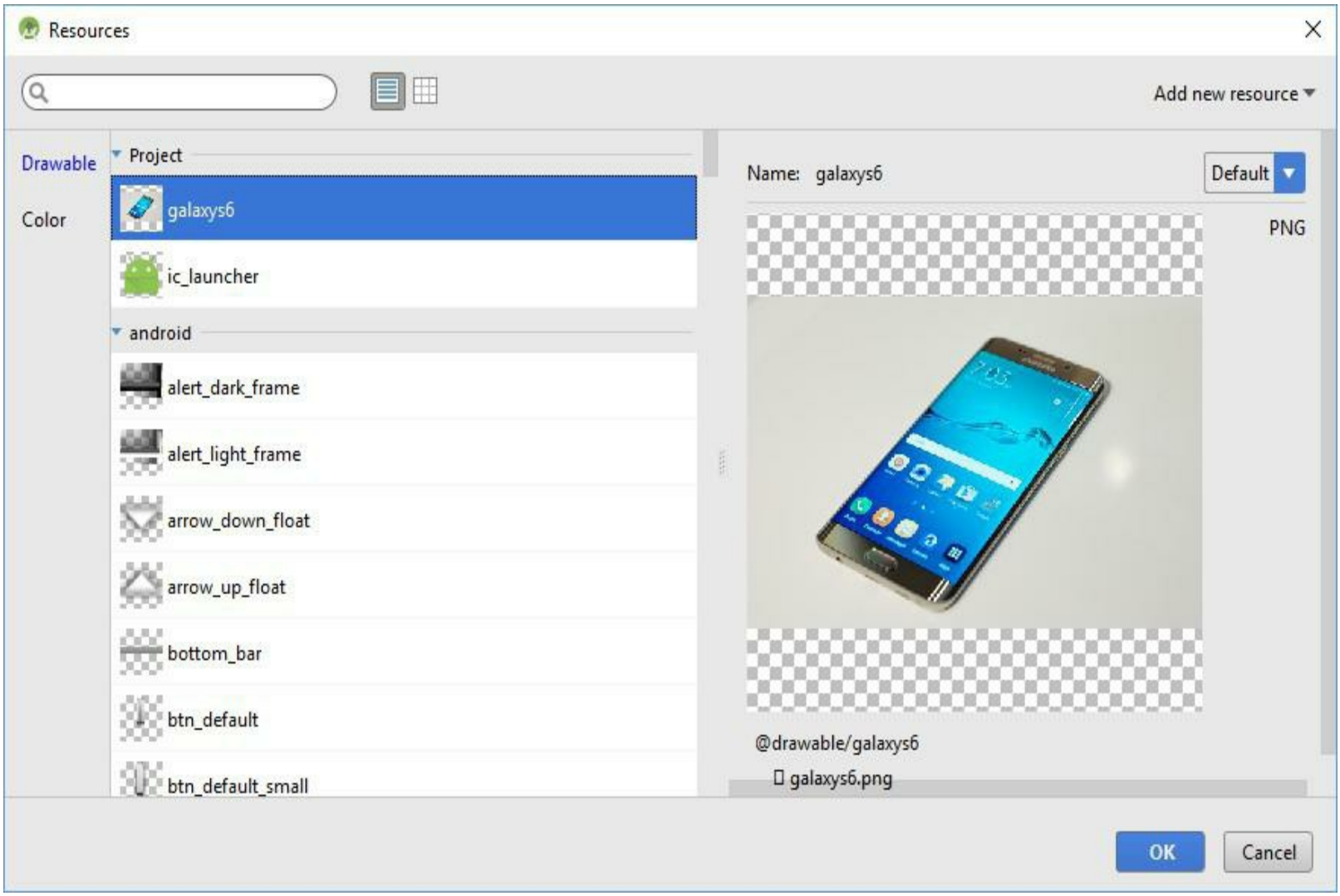


Figure 20-4

Select the image and click on OK to assign it to the ImageView object. If necessary, adjust the size of the ImageView using the resize handles and reposition it in the center of the layout. At this point the layout should match Figure 20-5:

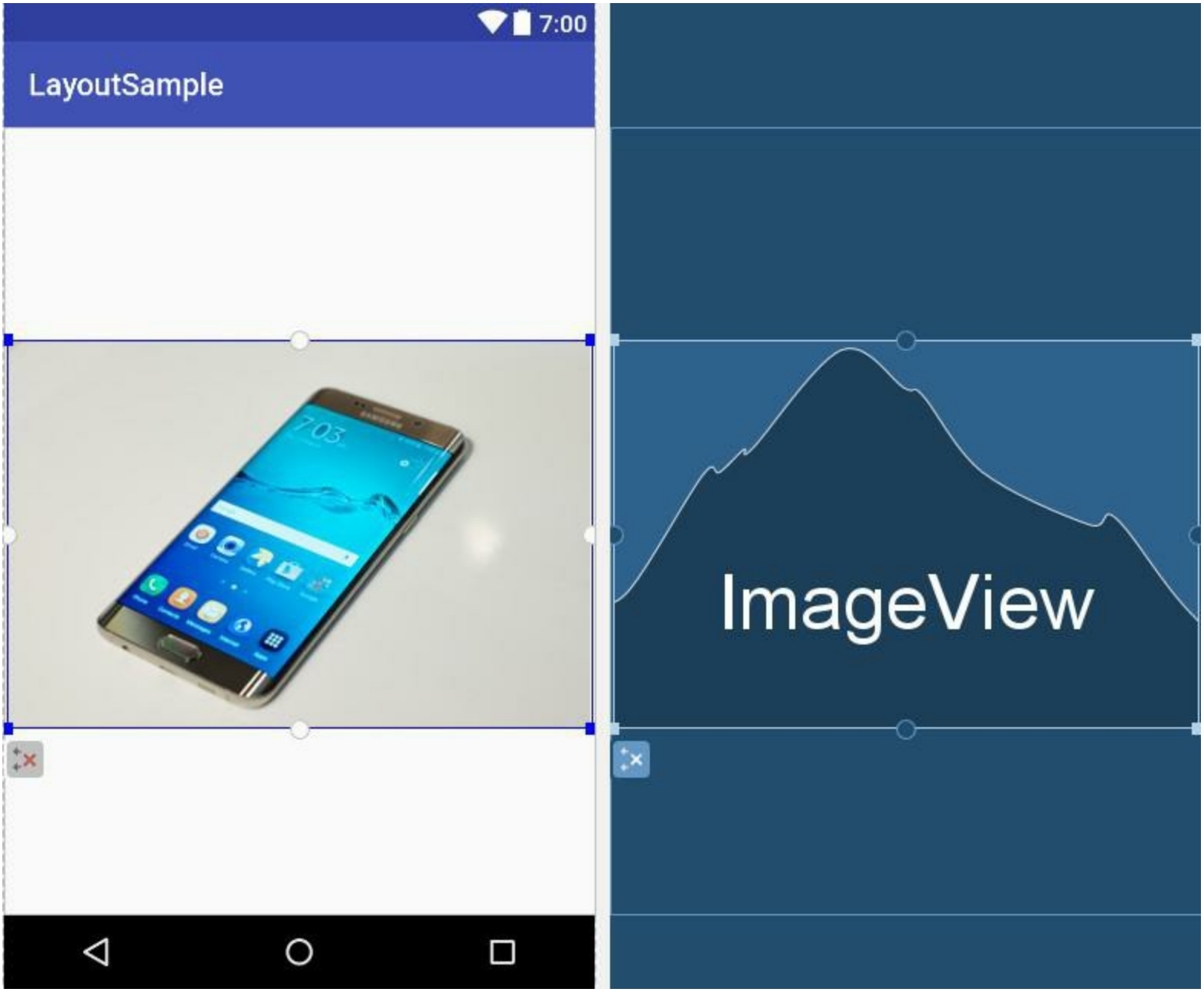


Figure 20-5

Click and drag a TextView object from the Widgets section of the palette and position it so that it appears above the ImageView as illustrated in Figure 20-6.

Using the Properties panel, change the *textSize* property to 24sp, the *textAlignment* setting to center and the text to "Samsung Galaxy S6".

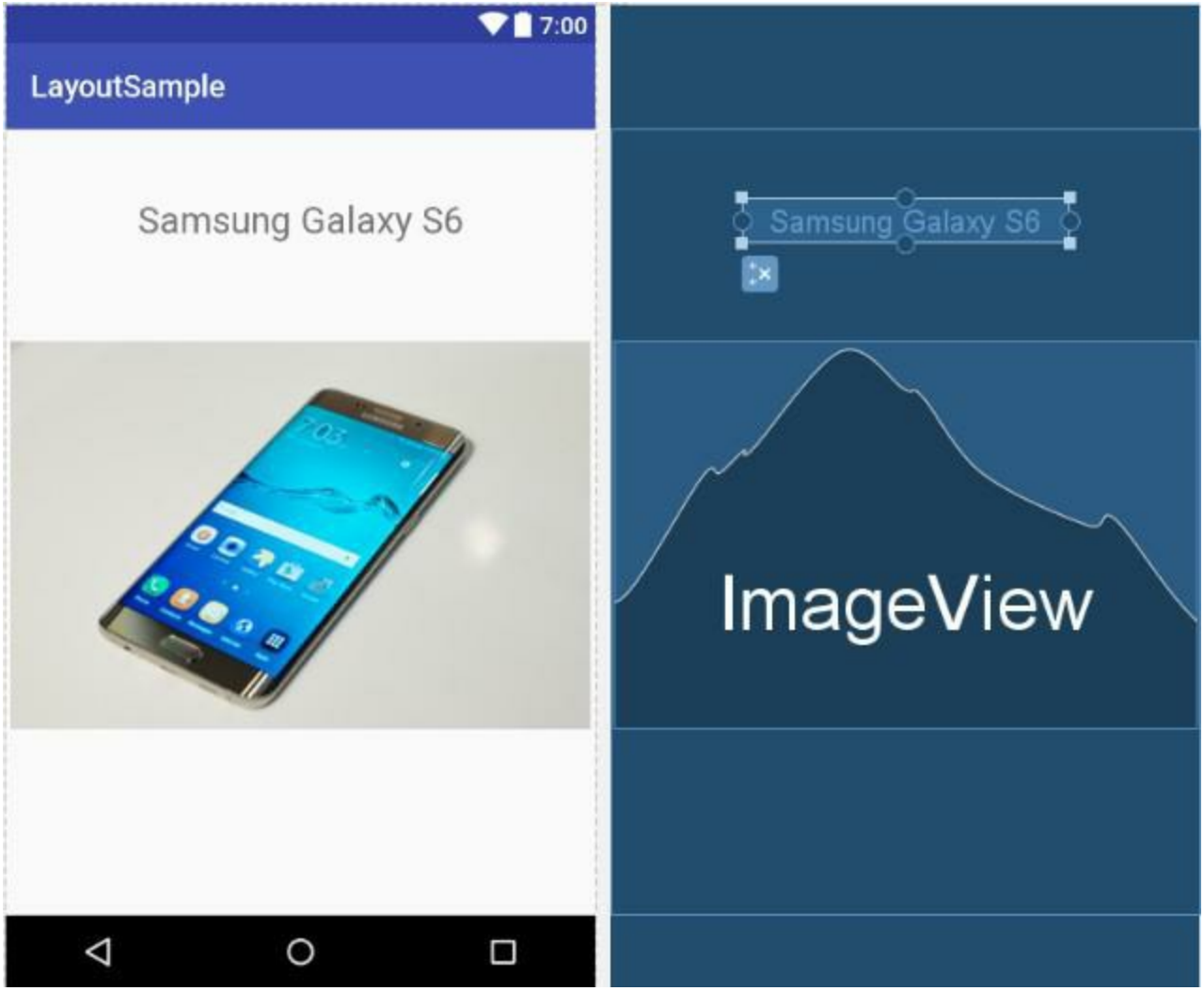


Figure 20-6

Next, add three Button widgets along the bottom of the layout and set the text properties of these views to “Buy Now”, “Pricing” and “Details”. The completed layout should now match Figure 20-7:

Figure 20-7

At this point, the widgets are not sufficiently constrained for the layout engine to be able to position and size the widgets at runtime. Were the app to run now, all of the widgets would be positioned in the top left-hand corner of the display.

With the widgets added to the layout, use the device rotation button located in the Layout Editor toolbar (indicated by the arrow in Figure 20-8) to view the user interface in landscape orientation:

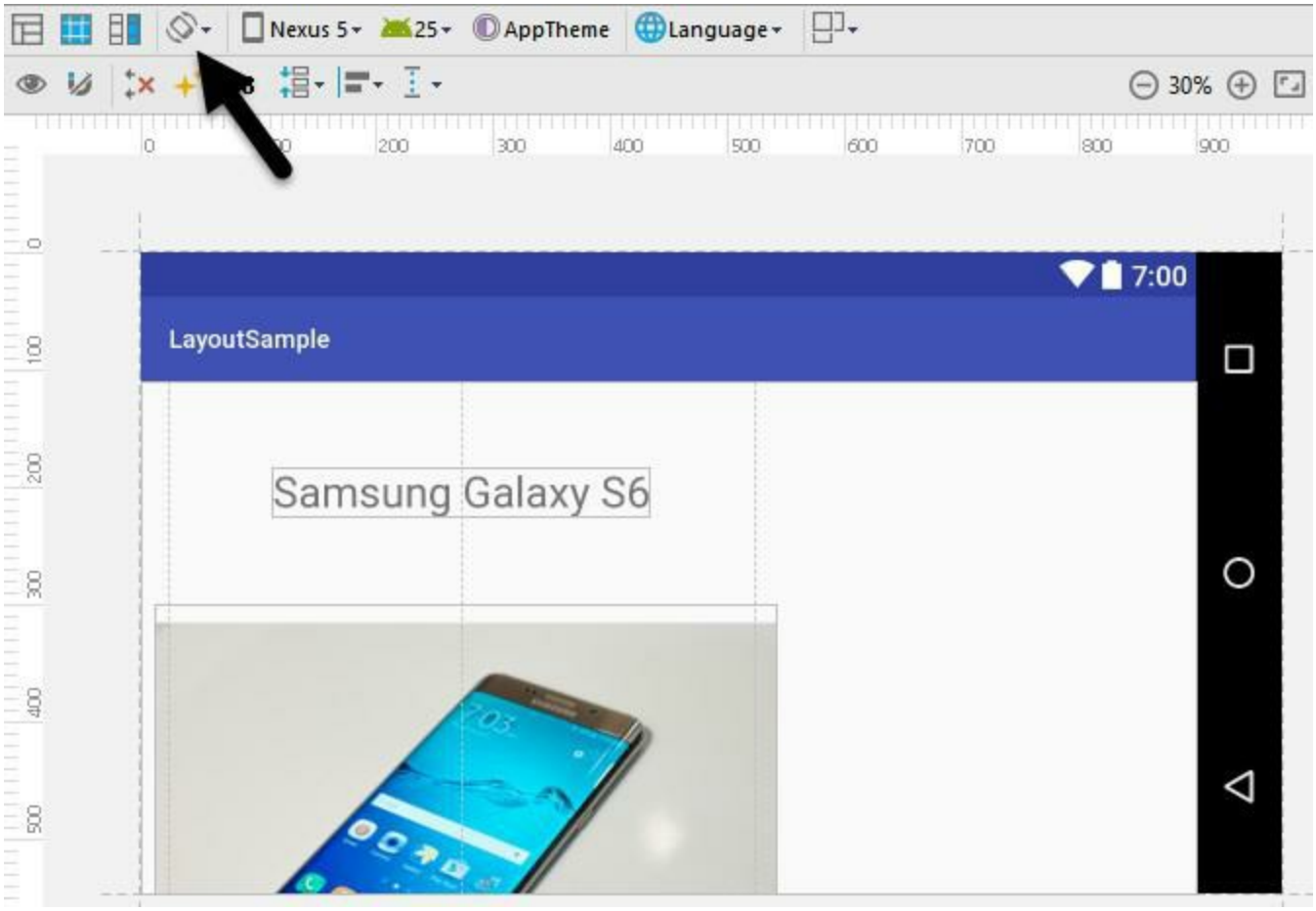


Figure 20-8

The absence of constraints results in a layout that fails to adapt to the change in device orientation, leaving the content off center and with part of the image and all three buttons positioned beyond the viewable area of the screen. Clearly some work still needs to be done to make this into a responsive user interface.

20.5 **Adding the Constraints**

Constraints are the key to creating layouts that can adapt to device orientation changes and different screen sizes. Begin by rotating the layout back to portrait orientation and selecting the TextView widget located above the ImageView. With the widget selected, establish constraints from the left, right and top sides of the TextView to the corresponding sides of the parent ConstraintLayout as shown in Figure 20-9:

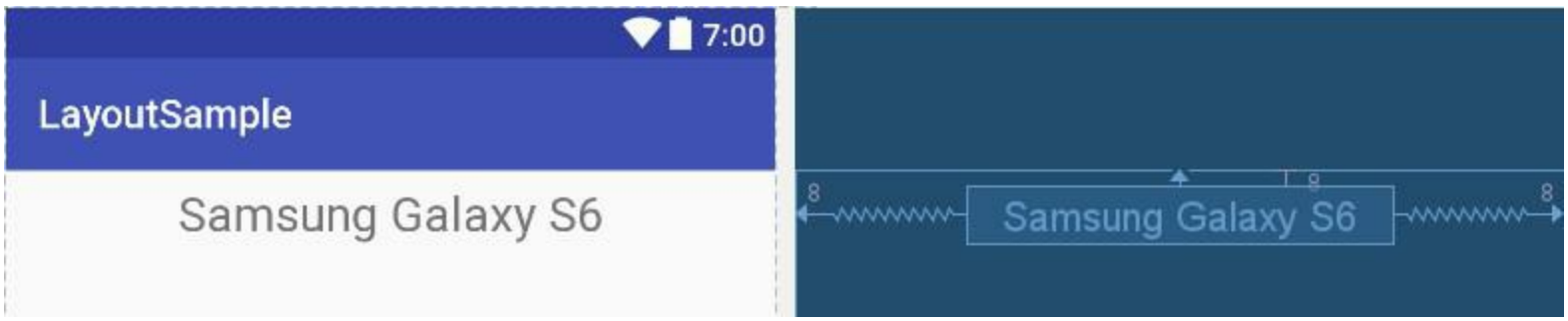


Figure 20-9

With the TextView widget constrained, select the ImageView instance and establish opposing constraints on the left and right-hand sides with each connected to the corresponding sides of the parent layout. Next, establish a constraint connection from the top of the ImageView to the bottom of the TextView and from the bottom of the ImageView to the top of the center Button widget. If

necessary, click and drag the ImageView so that it is still positioned in the vertical center of the layout.

With the ImageView still selected, use the Inspector in the properties panel to change the top and bottom margins on the ImageView to 24 and 8 respectively and to change both the widget height and width dimension properties to *0dp* so that the widget will resize to match the constraints. These settings will allow the layout engine to enlarge and reduce the size of the ImageView when necessary to accommodate layout changes:

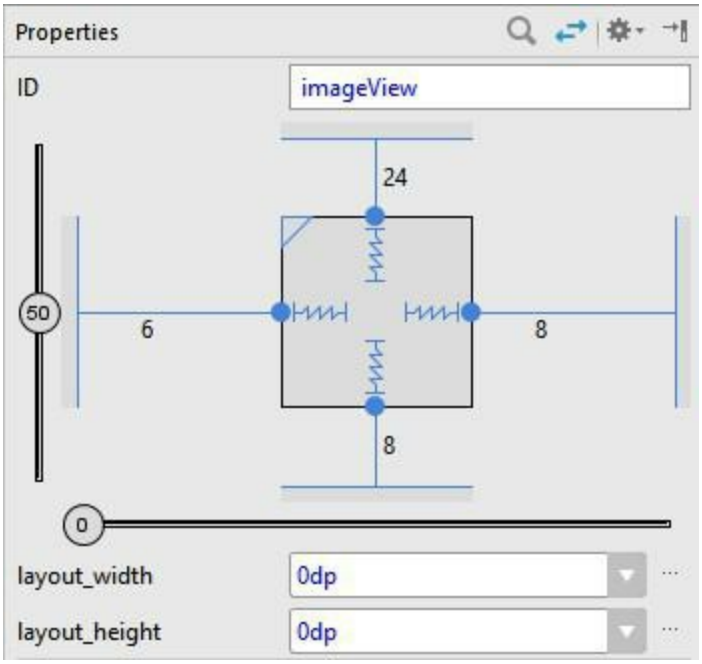


Figure 20-10

Figure 20-11, shows the currently implemented constraints for the ImageView in relation to the other elements in the layout:

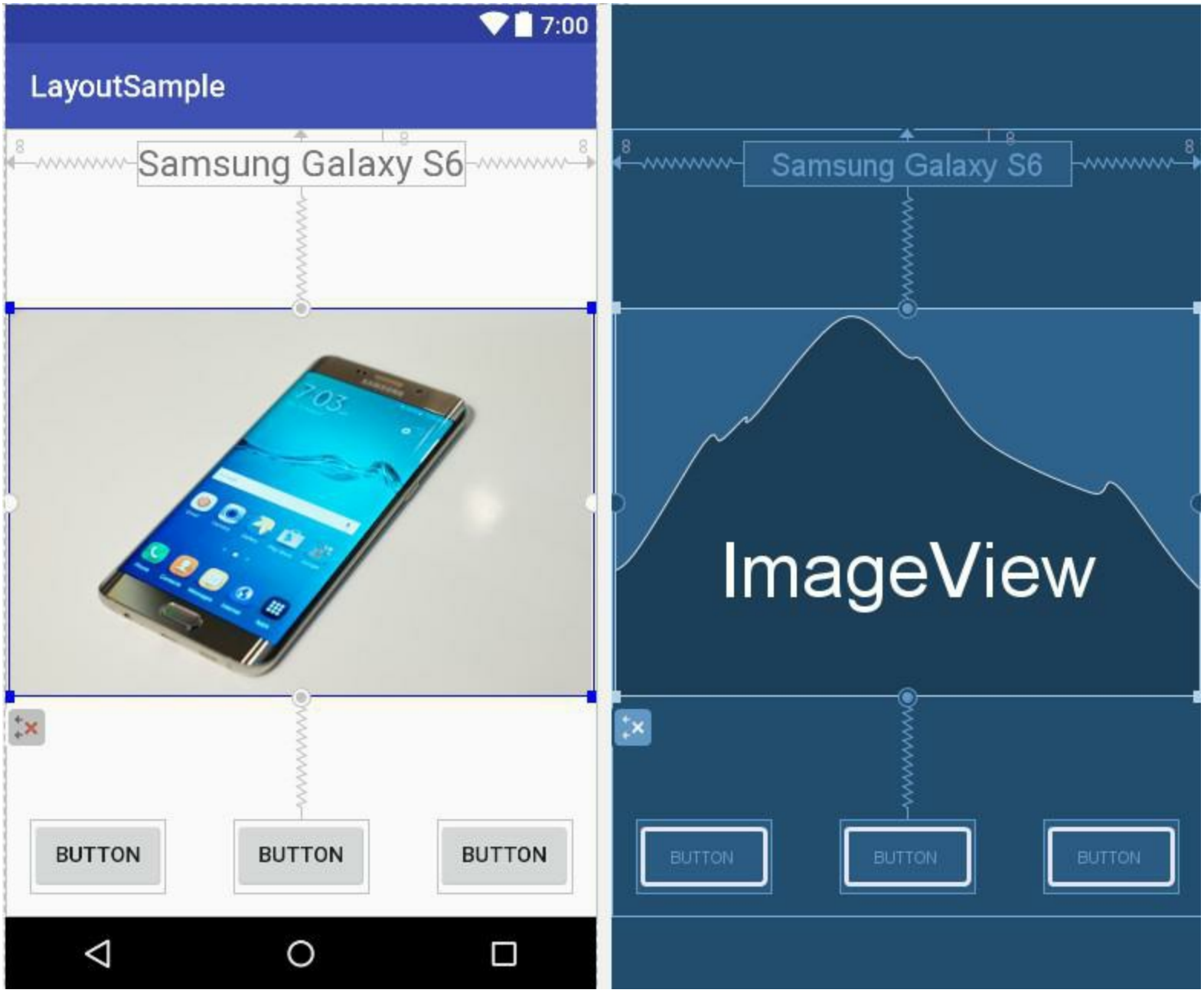


Figure 20-11

The final task is to add constraints to the three Button widgets. For this example, the buttons will be placed in a chain. Begin by turning on Autoconnect within the Layout Editor by clicking the toolbar button highlighted in Figure 20-2.

Next, click on the Buy Now button and then shift-click on the other two buttons so that all three are selected. Right-click on the Buy Now button and select the *Center Horizontally* menu option from the resulting menu. As discussed in the previous chapter, this is a useful shortcut for adding bi-directional constraints between the widgets when creating a chain. By default, the chain will be displayed using the spread style which is the correct behavior for this example.

Finally, establish a constraint between the bottom of the Buy Now button and the bottom of the layout. Repeat this step for the remaining buttons.

On completion of these steps the buttons should be constrained as outlined in Figure 20-12:

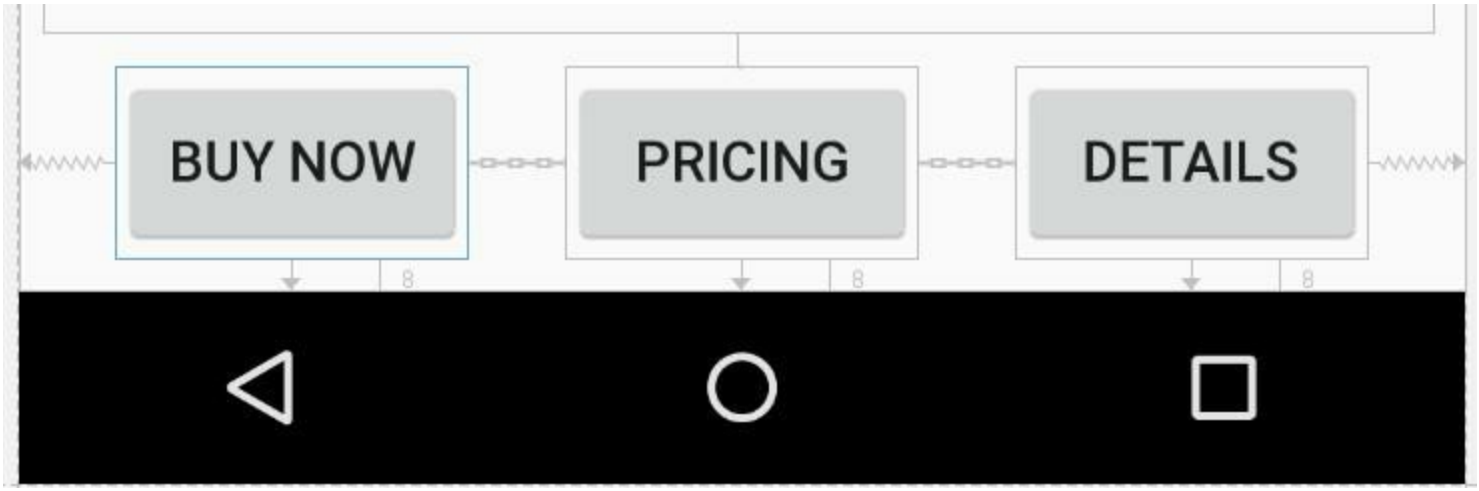


Figure 20-12

20.6 Testing the Layout

With the constraints added to the layout, rotate the screen into landscape orientation and verify that the layout adapts to accommodate the new screen dimensions.

While the Layout Editor tool provides a useful visual environment in which to design user interface layouts, when it comes to testing there is no substitute for testing the running app. Launch the app on a physical Android device or emulator session and verify that the user interface reflects the layout created in the Layout Editor. Figure 20-13, for example, shows the running app in landscape orientation:

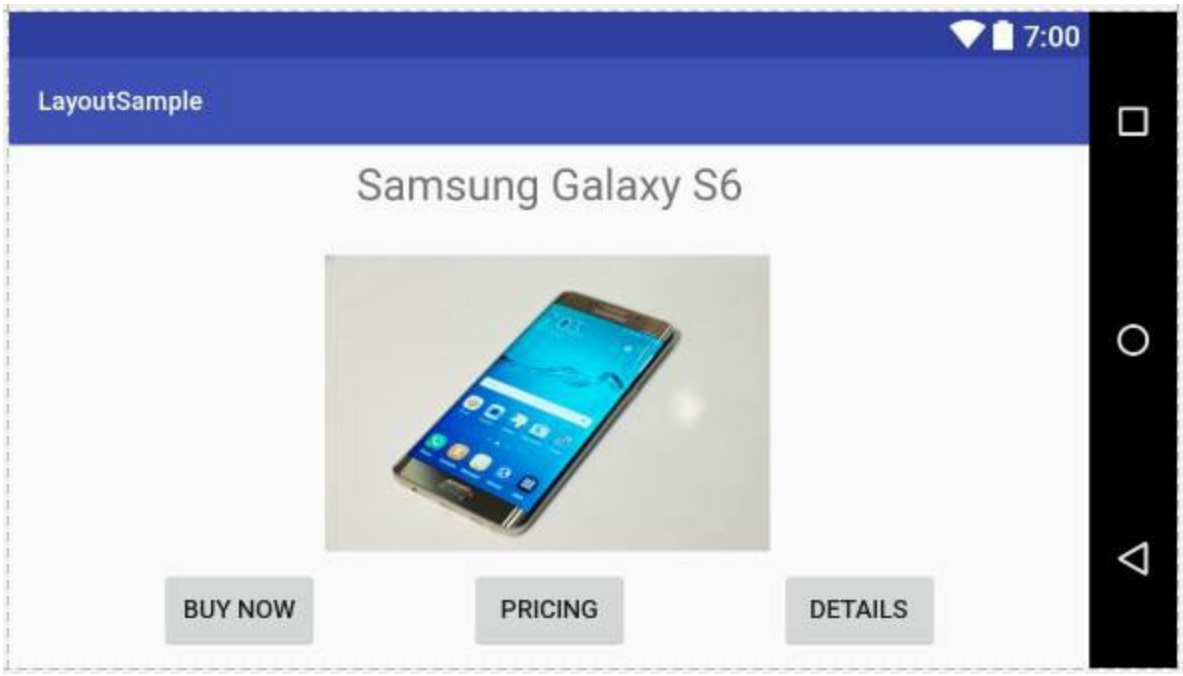


Figure 20-13

The very simple user interface design is now complete. Designing a more complex user interface layout is a continuation of the steps outlined above. Simply drag and drop views onto the display, position, constrain and set properties as needed.

20.7 Using the Layout Inspector

The hierarchy of components that make up a user interface layout may be viewed at any time using the Layout Inspector feature of the Android Monitor tool window. In order to access this information the app must be running on a device or emulator. Once the app is running, select the Android Monitor

tool window tab (marked A in Figure 20-14) and, once displayed, click on the Layout Inspector button (B) to display the inspector panels.

The left most panel (C) shows the hierarchy of components that make up the user interface layout. The center panel (D) shows a visual representation of the layout design. Clicking on a widget in the visual layout will cause that item to highlight in the hierarchy list making it easy to find where a visual component is situated relative to the overall layout hierarchy.

Finally, the rightmost panel (marked E in Figure 20-14) contains all of the property settings for the currently selected component, allowing for in-depth analysis of the component's internal configuration.

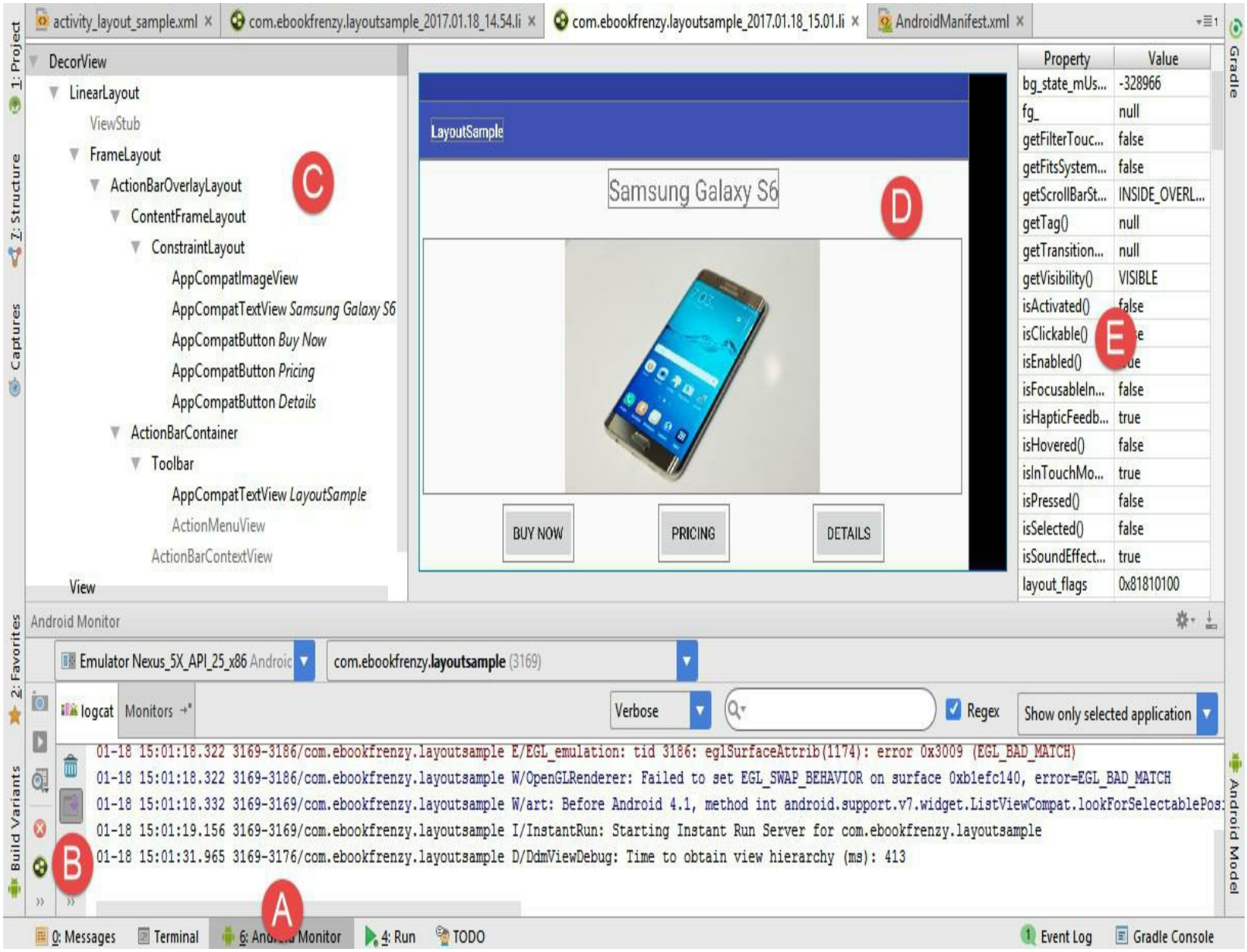


Figure 20-14

20.8 **Using the Hierarchy Viewer**

Another useful tool for closely inspecting the view hierarchy of an activity is the Hierarchy Viewer. The main purpose of the tool is to provide a detailed overview of the entire view tree for activities within currently running applications and to provide some insight into the layout rendering performance.

The hierarchy viewer can only be used to inspect applications that are either running within an

Android emulator, or on a device running a *development* version of Android. To run the tool on the *LayoutSample* application created in this chapter, launch the application on an Android Virtual Device emulator and wait until it has loaded and is visible on the emulator display. Once running, select the *Tools -> Android -> Android Device Monitor* menu option. In the DDMS window, select the *Window -> Open Perspective…* menu option and choose *Hierarchy View* from the resulting dialog before clicking on the *OK* button.

When the Hierarchy Viewer appears, it will consist of a number of different panels. The left-hand panel, illustrated in Figure 20-15, lists all the windows currently active on the device or emulator such as the navigation bar, status bar and launcher. The window listed in bold is the current foreground window, which should, in this case, be *LayoutSampleActivity*.

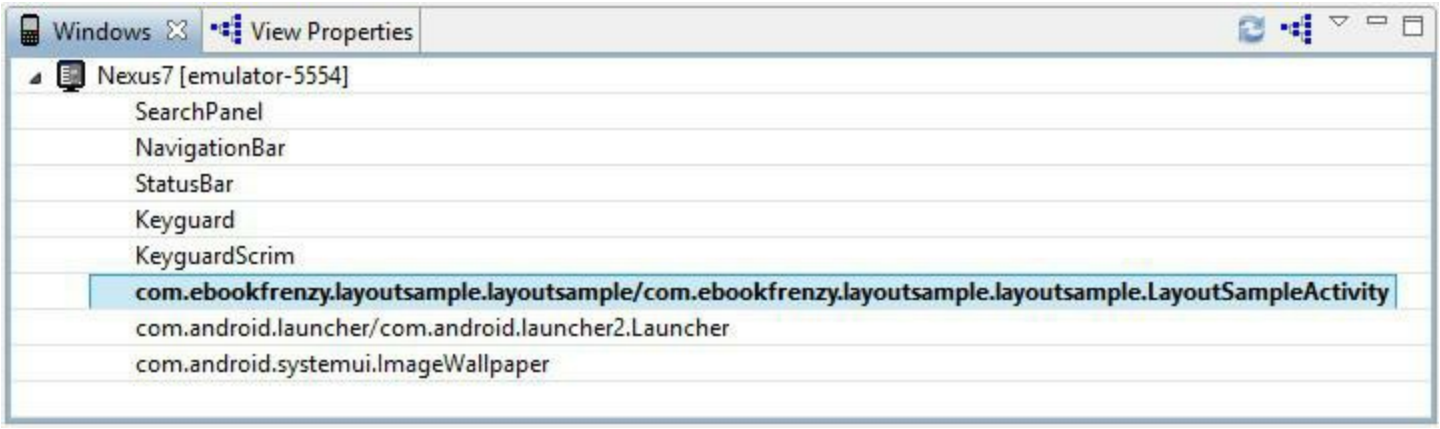


Figure 20-15

Double-clicking on the layout sample window will cause the hierarchy to load into the Tree View panel as shown in Figure 20-16 (note that there may be a short delay between selection of the window and the hierarchy diagram appearing):

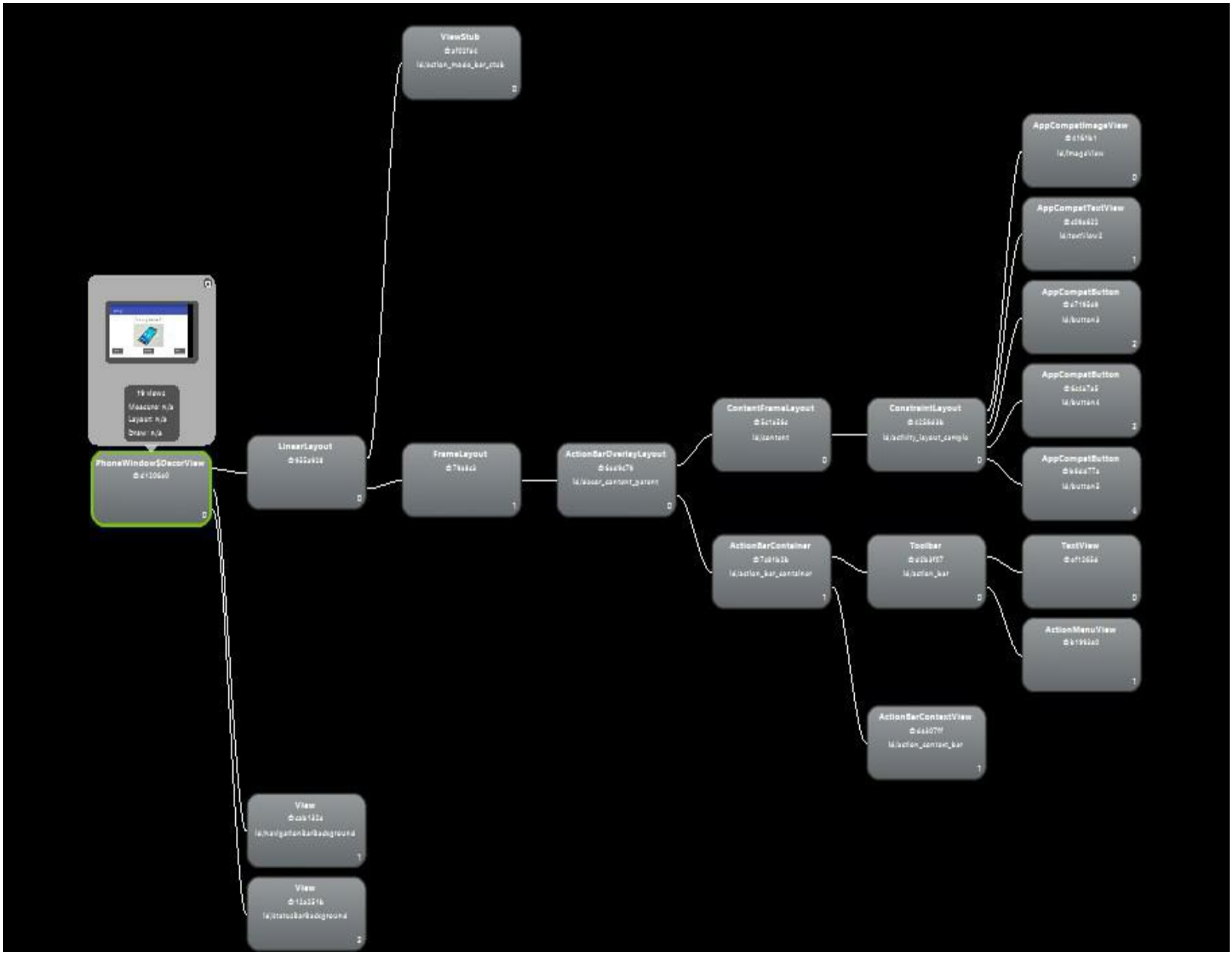


Figure 20-16

While it is possible to zoom in and out of the tree view using the scale at the bottom of the panel or by spinning the mouse wheel, in most cases the tree will be too large to view entirely within the Tree View panel. To move the view window around the tree simply click and drag in the Tree View panel, or move the lens within the Tree Overview panel (Figure 20-17):

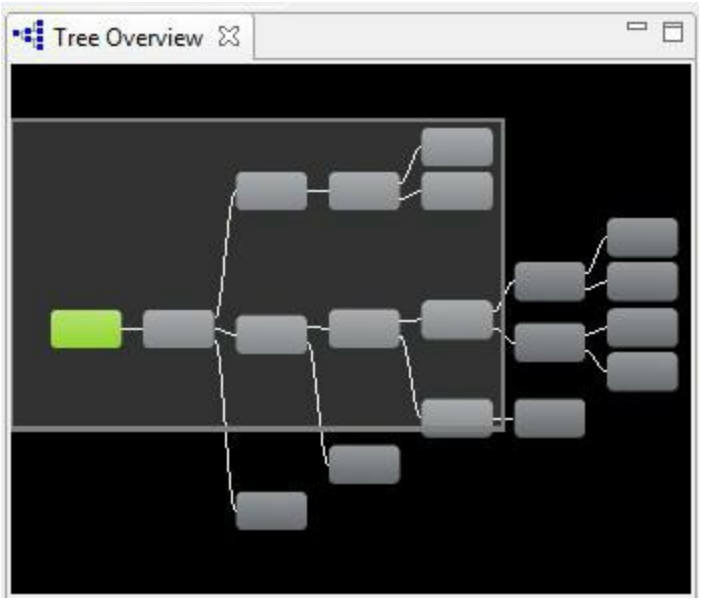


Figure 20-17

When reviewing the tree view, keep in mind that some views in addition to those included in the activity layout will be displayed. These are the views and layouts that, for example, display the action bar across the top of the screen and provide an area for the activity to be displayed.

Selecting a node in the Tree View will cause the corresponding element in the user interface representation to be highlighted in red in the Layout View. In Figure 20-18 the ConstraintLayout view is currently selected:

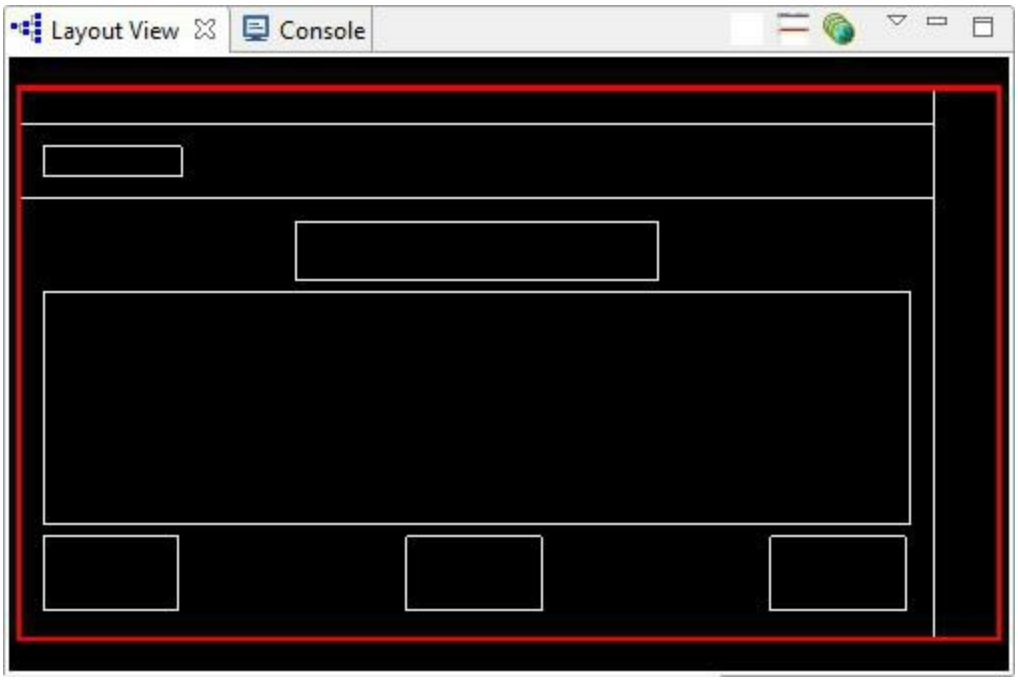


Figure 20-18

Similarly, selecting views from the Layout View will cause the corresponding node in the Tree View to highlight and move into view.

Additional information about a view can be obtained by selecting the node within the Tree View. A panel will then popup next to the node and can be dismissed by performing a right-click on the node. Double-clicking on a node will display a dialog containing a rendering of how the view appears within the application user interface. Figure 20-19, for example, shows a button from the LayoutSample application:

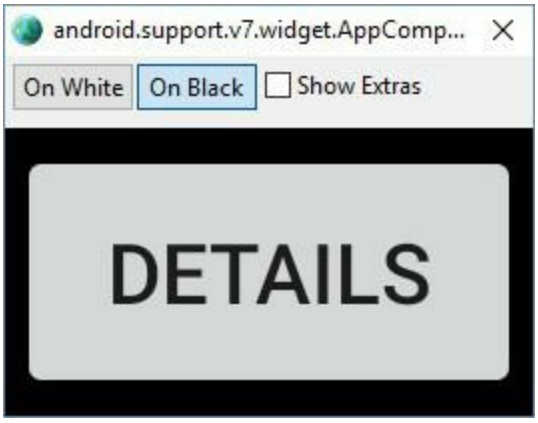


Figure 20-19

Options are also available within the tool to perform tasks such as invalidating a selected layout view (thereby forcing it to be redrawn) and to save the tree view as a PNG image file. The hierarchy viewer can also be used to display information about the speed with which the child views of a selected node are rendered when the user interface is created. To display this performance information, select the node to act as the root view and click on the toolbar button indicated in Figure 20-20.

Figure 20-20

When enabled, the colored dots within the nodes indicate the performance in each category (measure, layout and draw) with red indicating slower performance for the view relative to other views in the activity. Container views with larger numbers of child views may display red status simply because the view has to wait for each child to render. This is not necessarily an indication of a performance problem with that view.

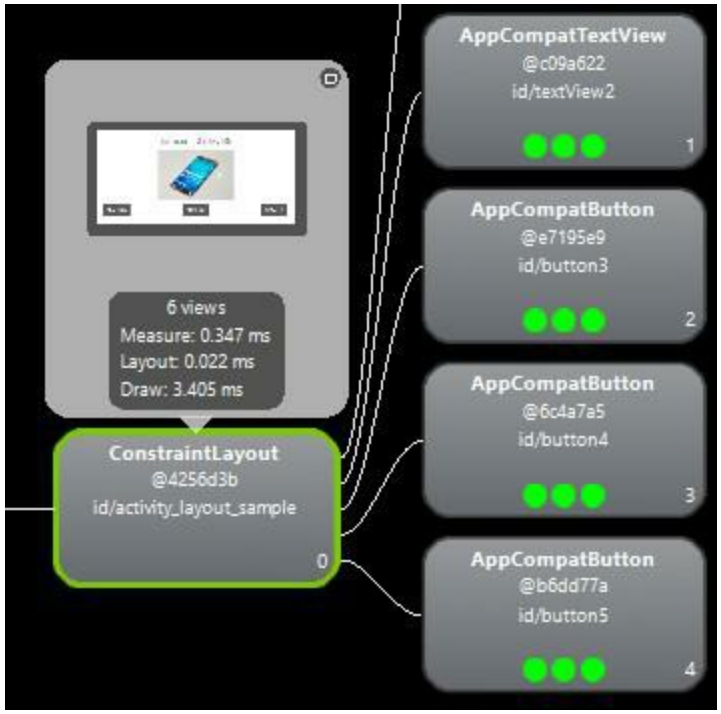


Figure 20-21

20.9 **Summary**

The Layout Editor tool in Android Studio has been tightly integrated with the ConstraintLayout class introduced in Android 7. This chapter has worked through the creation of an example user interface intended to outline the ways in which a ConstraintLayout-based user interface can be implemented using the Layout Editor tool in terms of adding widgets and setting constraints. This chapter also introduced the Layout Inspector and Hierarchy Viewer tools, both of which are useful for analyzing the structural composition of a user interface layout.