Murach's Android Programming (2nd Ed.)

Updates for 2017

Since *Murach's Android Programming (2nd Edition)* was published in 2015, there have been some significant improvements to Android Studio that impact chapters 1 and 2. Most of this document attempts to show you how to take advantage of these improvements. I recommend reading it after you've read pages 1-15 of *Murach's Android Programming*.

Fortunately, almost all of the Java code presented in this book still works as described. However, there are a couple problems with the code for the Run Tracker app presented in chapter 18. That's why the last figure of this document describes these problems and shows how to fix them.

How to work with an existing app

To get you started, this document begins by showing how to open an existing app that uses the new *constraint layout* instead of the old relative layout. But first, it shows how to set up your system, so you can follow along with this document on your own computer.

How to set up your system

Figure 1 starts by presenting four easy steps to set up your system. To start, you should install a JDK (Java Development Kit), Android Studio, and the source code for this book. In addition, if you have a physical Android device such as a phone or tablet that you want to use for testing, you should configure it for development.

For more details on all four of these steps, you can refer to the appendixes of *Murach's Android Programming (2nd Edition)*. Once you complete these four steps, you can open an existing project. In addition, you can install the necessary Android SDKs and create an emulator as described in the figures that follow.

Four easy steps to set up your system

1. If JDK (Java Development Kit) 6 or later isn't already on your system, install the latest version of the JDK. To do that, follow the instructions here:
   www.oracle.com/technetwork/java/javase/downloads
2. Install Android Studio 2.3 or later. To do that, follow the instructions here:
   https://developer.android.com/studio
3. Install the source code for this book. To do that, go here:
   https://www.murach.com/shop/murach-s-android-programming-2nd-edition-detail

   Then, scroll down, click on the FREE Downloads tab, and follow the instructions to download the zip file. Finally, extract the zip file into one of the directories shown in the next figure. To do that, you'll need to create the murach directory.
4. If you have a physical Android device such as a phone or tablet, configure it for development as shown below.

How to configure a physical device for development

1. Enable developer options on your device. For Android 4.2 (API 17) and later, you can do it like this:
   - Settings→More→About Device→Build Number (tap on it 7 times).
2. Turn on USB Debugging. For Android 4.2 (API 17) and later, you can do it like this:
   - Settings→More→Developer Options→USB Debugging.
3. Connect your Android device to your computer's USB port.

Description

- This document works with Android Studio 2.3 or later and Android 7.1 (API 25) or later.
- For more details about installing these software products, see the appendixes of *Murach's Android Programming*.
- If you aren't able to run an app on a physical device as described later in this document, you may need to download a driver for the device. For more details, see the appendixes of *Murach's Android Programming*.

Figure 1 How to set up your system

How to open an existing project for an app

Figure 2 shows the Welcome page for Android Studio. This page is displayed the first time you start Android Studio. In addition, it's displayed if you close all open projects.

The right side of the Welcome page provides items that let you start new Android Studio projects from scratch and open existing Android Studio *projects*. In addition, it allows you to import projects.

The left side of the Welcome page provides a list of recent projects that have been opened. You can easily reopen any of these projects by clicking on them. In this figure, the list includes two new projects that were added to the download after the book was printed. These projects are named

```
ch03_TipCalculator_Constraint
```

and

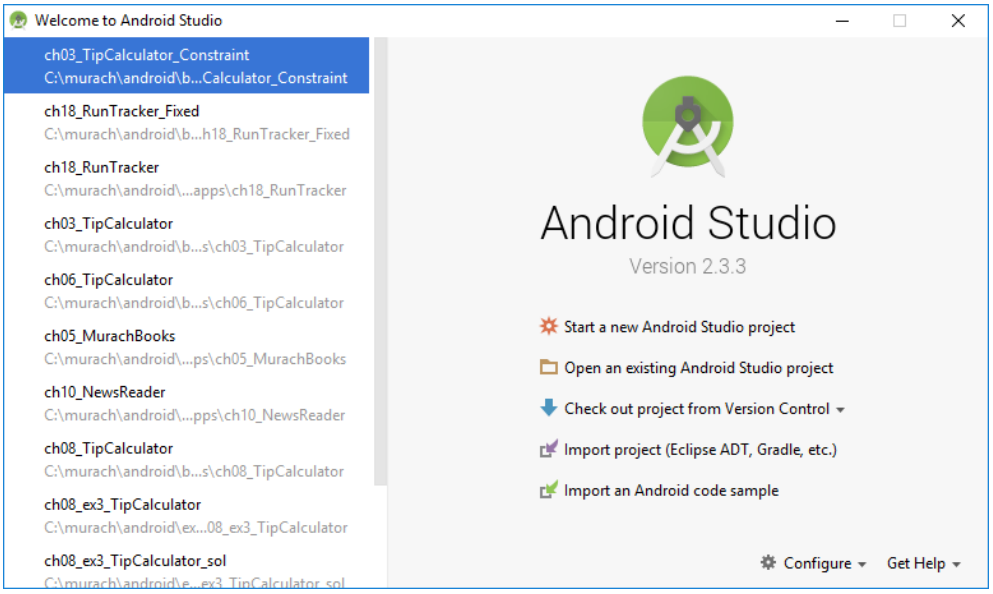
```
ch18_RunTracker_Fixed
```

The first project contains updated code for a Tip Calculator app that uses the constraint layout instead of the relative layout, and the second project contains code for a Run Tracker app that works with API 25 and later.

If you attempt to open an existing Android Studio project, you'll get a dialog box that lets you navigate to the directory that contains the project you want to open. For example, if you installed the source code for this book on a Windows system, all of the projects presented in this book are stored in subdirectories of this directory:

```
C:\murach\android\book_apps
```

The Welcome page



A typical directory for the applications presented in this book

For Windows

```
C:\murach\android\book_apps
```

For Mac OS X

```
User/name/murach/android/book_apps
```

For Linux

```
/home/username/murach/android/book_apps
```

Description

- An Android Studio *project* consists of a top-level directory that contains the directories and files for an app.
- When you start Android Studio for the first time, it displays a Welcome page. Android Studio also displays this page if you close all open projects.
- The left side of the Welcome page displays recently opened projects. You can reopen them by clicking on them.
- The right side of the Welcome page displays options that you can use to start a new project, open an existing project, or import a project.
- To open an existing project, click on the Open an existing Android Studio project and use the resulting dialog box to navigate to the directory for the project.

Figure 2 The Welcome page

How to view a project

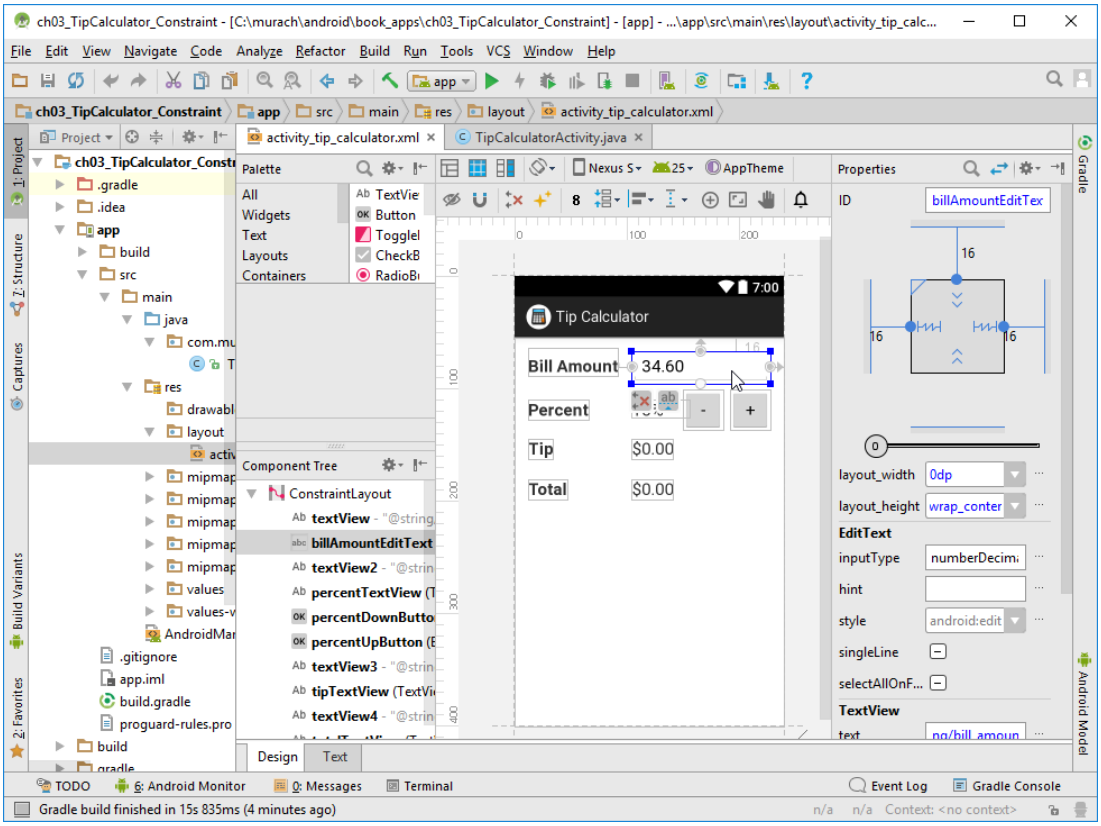
Figure 3 shows the main Android Studio window after the Tip Calculator app that uses the new constraint layout has been opened. This window works much like the layout that's shown in chapter 1 of *Murach's Android Programming*. However, the techniques for setting properties and aligning widgets work a little differently since you're using the constraint layout instead of the relative layout.

In this figure, I opened the *layout* for the user interface by using the Project window to expand the res/layout directory for the project. Then, I double-clicked on the XML file for the layout. This displayed the user interface in the graphical editor.

When you open a constraint layout in the graphical editor, you can typically use the graphical editor to get the layout to look the way you want it to work. However, you sometimes need to click the Text tab below the layout to access the XML that's generated by the graphical editor. Later in this document, you'll learn how to use the graphical editor to work with a constraint layout. In addition, you'll learn how to edit the XML for a layout to fix warnings, a common reason to use the Text view instead of the Design view.

If necessary, you can open the Java source code for an app by using the Project window to expand the java directory and the package that contains the Java file. Then, you can double-click on that file to open it in the code editor. This works as described in *Murach's Android Programming*.

The layout for the Tip Calculator activity in the graphical editor



Description

- In Android development, an *activity* defines one screen of an app.
- In Android development, a *layout* stores the XML that defines the user interface for an activity.
- In Android Studio, the different parts of the main window are known as *tool windows*.
- The Project tool window displays the directories and files that make up a project. If this window isn't visible, you can display it by clicking the Project tab on the left side of the main window.
- To work with the user interface, expand the res\layout directory for the project, and double-click on a layout file to open it. Then, you can click on the Design tab to use the graphical editor to work with the user interface. Or, you can click on the Text tab to work directly with the XML that defines the user interface.
- When using the graphical editor, the Palette, Component Tree, and Properties tool windows typically open to the left and the right of the user interface. These windows can be pinned to either side of the main window if you need more visual space.

Figure 3 How to view a project

How to use the Android SDK Manager

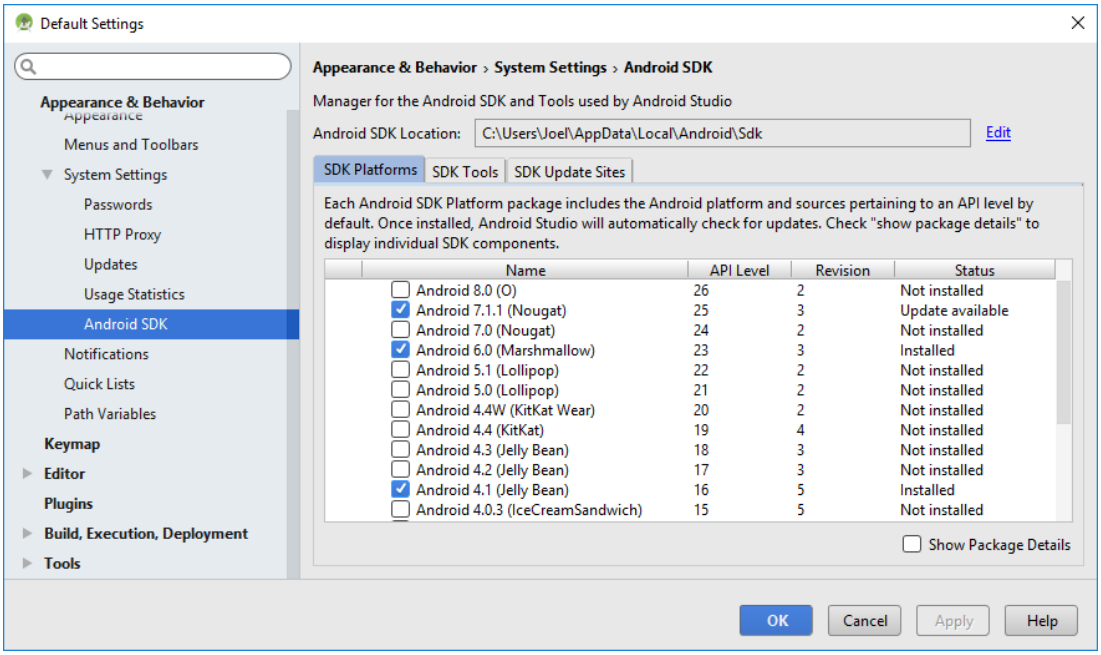
Android Studio includes a starting set of Android tools and platforms. However, I recommend following the procedure presented in figure 4 to make sure that you have the platforms that you'll need as you work through this book. For this book, we recommend that you install API 16, 23, and 25.

Throughout this book, we have used Android 6.0 (API 23) for testing new devices and Android 4.1 (API 16) for testing old devices. As a result, we recommend installing these platforms to get started. In addition, you may want to install Android 7.1 (API 25) or later so you can test the latest Android devices on your system.

If you prefer to use other versions of Android, you can do that too. For example, you may want to use newer or older versions than the ones we use in this book. However, it's usually easier to follow along if you use the same versions we use in this document and in *Murach's Android Programming*.

If you open an Android project that requires other libraries that aren't installed on your system, Android Studio prompts you to install those libraries. Typically, you can do that by clicking on a link and responding to the resulting dialog boxes.

The Android SDK Manager



Procedure

1. Start Android Studio and open an existing project.
2. Click the toolbar button for the SDK Manager.
3. To view the SDK platforms or SDK tools, click on the appropriate tab.
4. To install more platforms or tools, select the platforms or tools, click the OK button, and respond to the resulting dialog boxes. For *Murach's Android Programming*, we recommend installing all of the defaults including the following items:
   - SDK Platform→Android 7.1 (Nougat) API 25
   - SDK Platform→Android 6.0 (Marshmallow) API 23
   - SDK Platform→Android 4.1 (Jelly Bean) API 16

   On some systems, this may take an hour or longer.

Figure 4 How to use the Android SDK Manager

How to create an emulator

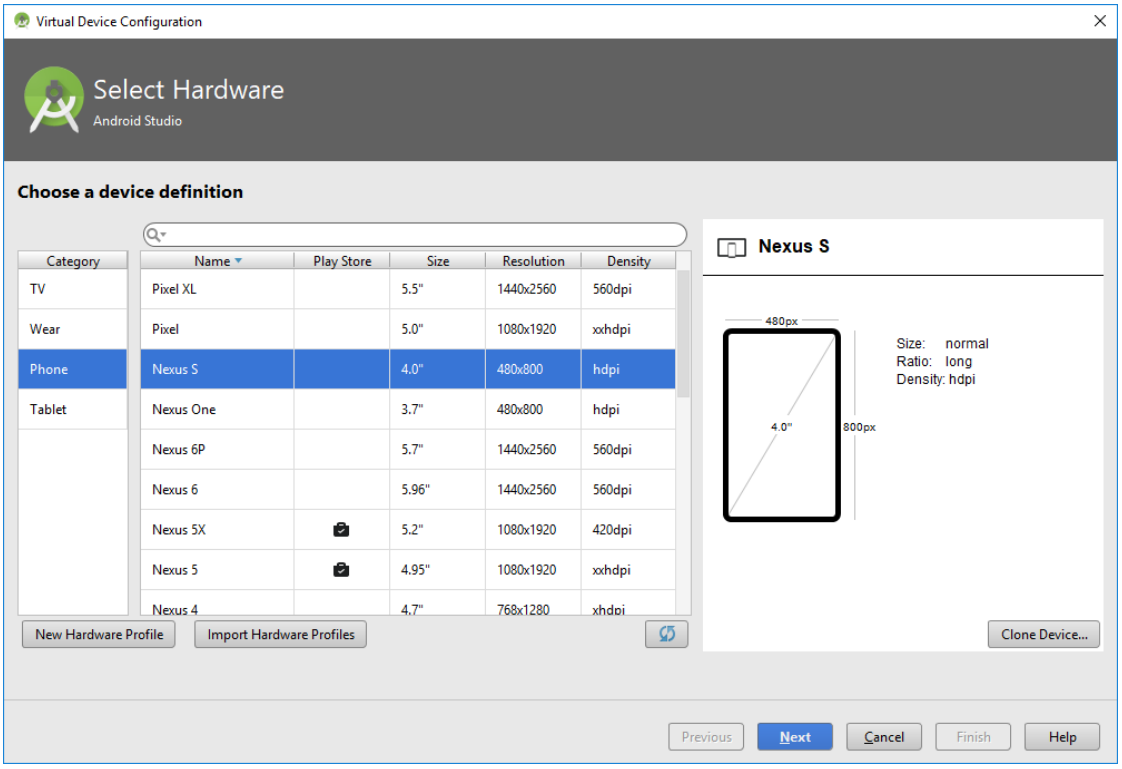
Figure 5 shows how to create an *Android Virtual Device* (*AVD*) that you can use to test your apps. Since an AVD emulates a physical device, an AVD can also be called an *emulator*. If necessary, you can create an emulator for each platform that you want to test. For this book, we recommend you create an emulator like the one shown in this figure.

When you create an emulator, you must provide a name for the emulator and the target platform. In this figure, I created an emulator named "Nexus S API 16".

This emulator is based upon a Nexus S device that has a 4-inch screen that's 480 by 800 pixels. As a result, this screen uses a high number of dots per inch (hdpi), which runs more quickly than an extra high number of dots per inch (xhdpi).

This emulator uses a system image for the Android 4.1 (API 16) platform. Since this API doesn't have as many features as later APIs, this also helps the emulator to run more quickly. In addition, when you create an emulator, you should download a system image for the API if you're prompted to do that. This helps the emulator run faster on your system.

The dialog box for creating an emulator



Procedure

1. Start Android Studio and open an existing project.
2. Click the toolbar button for the AVD Manager. This should start the Android Virtual Device Manager and show all existing virtual devices.
3. Click the Create Virtual Device button. This should display the dialog box shown above.
4. Select the phone named Nexus S. Then, click the Next button.
5. Select the Android API for the device. If you haven't already downloaded a system image for that API, you should click the Download link and respond to the resulting dialog boxes to do that. Then, click the Next button.
6. Respond to the resulting of the dialog boxes. Typically, you can accept the default options.

Description

- To test your Android apps, you can create an *Android Virtual Device* (*AVD*) for each platform that you wish to test. An Android Virtual Device can also be referred to as an *emulator*.

Figure 5 How to create an emulator

How to run a project

Now that you've set up an emulator for your system, you're ready to learn how to run an app on that emulator. Similarly, if you've configured a physical device for development and connected it to your system as described in the appendix of *Murach's Android Programming*, you're ready to learn how to run an app on that device.

Figure 6 begins by showing how to run an app. To do that, select the Run button in the toolbar. This typically displays a Select Deployment Target dialog box. If it doesn't, you can display this dialog by clicking on the Run→Edit Configurations item from the menus. Then, you can use the resulting dialog box to select the "Show chooser dialog" option. In addition, if the app is already running, you may need to click on the Stop button in the toolbar as described in the next figure.

From the Select Deployment Target dialog, you can select a connected device or an available emulator and click the OK button. In this figure, for example, you could select the connected Samsung device. Or, you could select one of the three Nexus S emulators.

If the Select Deployment Target dialog doesn't display the physical device, make sure a compatible physical device is connected to your computer. Typically, you use a USB cable to connect your device to your computer.

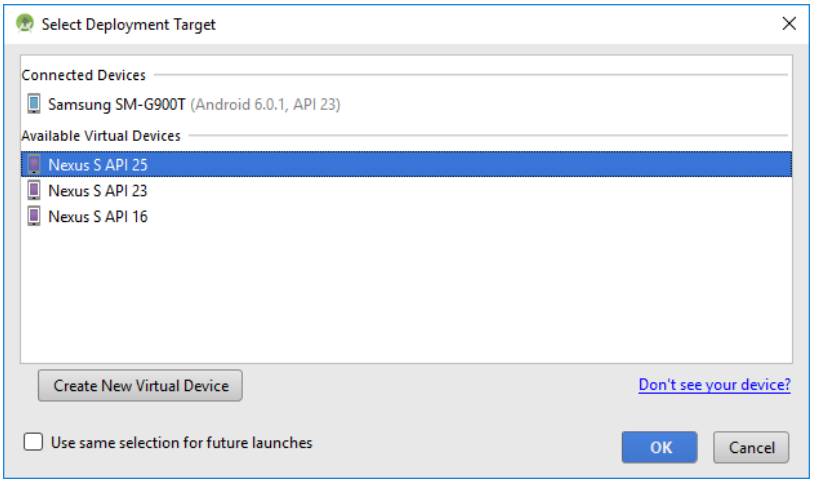
Once your app is running on the physical device, you can test that app using any of the hardware that's available on that device. For example, you can rotate the device to test how it handles screen orientation changes.

Once your app is running in an emulator, you can use the emulator to test your app. To do that, you can use your mouse to interact with the touchscreen to test the app.

When you're done testing an app in an emulator, you can leave the emulator open. That way, the next time you run the app, Android Studio won't need to start the emulator, which can take a frustratingly long time on most systems. As a result, Android Studio will be able to run your app much more quickly the next time.

By default, Android Studio automatically compiles an app before running it. Since this saves a step in the development process, this is usually what you want. Sometimes, though, Android Studio gets confused and doesn't compile a class that's needed by your project. For example, Android Studio sometimes doesn't compile the R class that contains the compiled resources needed by your project. In that case, you can usually fix this issue by selecting the Build→Clean Project item from the menu bar. This cleans the project and rebuilds it from scratch.

The dialog for choosing the device



Description

- To run any app, click the Run button in the toolbar. The first time you run an app, Android Studio typically displays a dialog to let you pick the device.
- To run on a physical device or emulator that's already running, select the device or emulator from the list of connected devices and click the OK button.
- To run on an emulator that's not yet running, select the emulator from the list of available virtual devices and click the OK button.
- On a device or emulator, you need to unlock the screen to allow the app to run.
- By default, a project is compiled automatically before it is run, which is usually what you want.
- To clean and rebuild a project, select the Build→Clean Project item from the menu bar and respond to the resulting dialog.

Notes

- If the dialog of choosing the device isn't displayed when you run an app, the run configuration might not be configured correctly or the app may already be running on a device.
- To edit the run configuration, select the Run→Edit Configurations item and use the resulting dialog to select the "Select Target Deployment Dialog" option.
- To stop the app from running, you can click the Stop button in the toolbar. Then, when you run the app again, the dialog for choosing devices should be displayed.
- If your physical device isn't displayed in the dialog, you can disconnect it and reconnect it. This should display the device.
- If you haven't authorized a device to work with the current computer, the device displays a dialog when you unlock the screen. You can use this dialog to authorize the device.

Figure 6 How to run an app

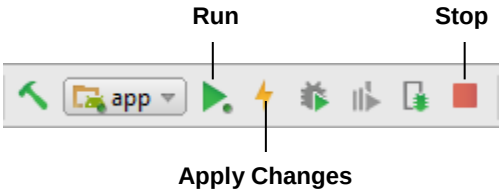
How to instantly apply changes to an app

Figure 7 shows how to use the *Instant Run feature* to apply changes to an app without recompiling it or restarting it. This feature significantly speeds up the development process, so you should be sure to use it whenever appropriate. The first time you run an app on a device or emulator, you must use the Run button as described in the previous figure.

Once an app is running on a device or emulator, you can click the Apply Changes button to apply your changes to that app. This uses the Instant Run feature to swap in the changes without recompiling or redeploying the app. For example, let's say you just ran the code in the Tip Calculator app and used it to calculate a tip. Then, you decided that you would like the Increase (+) button to increase the tip percent by 5% instead of 1%. In that case, you can change the code and click the Apply Changes button. When you do, Android Studio uses a hot swap to apply these changes almost instantly. This allows you to test your code changes much more quickly than you would be able to if you clicked the Run button. If you did that, Android Studio would recompile and redeploy the entire app, which could take a minute or longer.

If you want to run the app on a different device, you need to click the Stop button to disconnect Android Studio from the current device. Then, you can click the Run button to display the Select Target Device dialog box that allows you to select another device as described in the previous figure.

The toolbar for an app that's running on a device



Three swap types that Android Studio uses to apply your changes

Swap Type	Description
Hot swap	Occurs when you change the code within an existing method. Hot swapping happens instantly, does not cause the current activity to restart, and typically displays a message on the device that says, "Applied code changes without activity restart."
Warm swap	Occurs when you change a resource file such as a layout. Warm swapping causes the activity to restart and typically displays a message on the device that says, "Applied changes, restarted activity."
Cold swap	Occurs when you make a structural change to your code such as adding a new method. Cold swapping causes the entire app to restart. However, it doesn't re-install the app on the device.

Description

- The *Instant Run feature* can apply most of your code changes almost instantly. This allows you to test your code changes much more quickly than you could without Instant Run.
- The first time you run an app on a device, you must use the Run button as described in the previous figures. After that, you can click the Apply Changes button to apply your changes to that app. This uses the Instant Run feature to swap in the changes.
- To run the app on a different device, you can click the Stop button to disconnect Android Studio from the current device. Then, you can use the Run button to select another device as described in the previous figures.

Notes

- If you don't use Instant Run, Android Studio must compile the Java code into .class files and then into .dex files, it must create the .apk file, install the .apk file on the device or emulator, and launch the app. Even on fast systems, this takes a significant amount of time.
- If you use Instant Run, Android Studio can swap in most changes without having to recompile and reinstall the entire app. However, if you make changes to the Android manifest file, Android Studio must recompile and reinstall the app.

Figure 7 How to instantly apply changes to an app

Before you do the following exercise

Before you do any of the exercises in this document, you need to set up your system as described in figure 1. This may take several hours, so plan accordingly. See the appendixes for details.

Exercise 1 Open an existing project and run it

Open a project and review its code

1. Start Android Studio. Then, open the project that's stored in this directory:

   `\murach\android\book_apps\ch03_TipCalculator_Constraint`

2. Open the layout for the activity. This should open the layout in the graphical editor. If it doesn't, click the Design tab to display the graphical editor. Note the widgets displayed in this editor.
3. Click on the Text tab to view the XML for this layout. Note that the XML corresponds with the widgets displayed in the graphical editor.
4. In the toolbar, click the SDK Manager button. Then, use the SDK Manager to install each Android API that you want to use for testing.
5. In the toolbar, click the AVD Manager button. Then, create one emulator for each Android API that you want to use for testing.

Run the app on a physical Android device if you have one

6. In the toolbar, click the Run button. This should display the Select Deployment Target dialog. If it doesn't, edit your run configuration so it displays this dialog.
7. Connect the device you configured in the appendix. This should display the device in the Select Deployment Target dialog, though it may indicate that the device is unauthorized.
8. Unlock the screen on the device. If you get a dialog that indicates that the device is unauthorized for the computer, use the dialog to authorize the device.
9. Use the Select Deployment Target dialog to select the device.
10. Test the Tip Calculator app by using the soft keyboard to enter a bill amount and by clicking on the Increase (+) and Decrease (-) buttons to modify the tip percent.

Run the app on an emulator

11. In the toolbar, click the Stop button if it's enabled. Then, click the Run button. This should display the Select Deployment Target dialog. If it doesn't, edit your run configuration so it displays this dialog.

12. From the Select Deployment Target dialog, select the emulator and click the OK button. This should run the app on the emulator. Depending on your system, it may take a long time for the emulator to launch, so be patient! After loading, you need to unlock the emulator screen by dragging the lock icon to the right or up.
13. Test the Tip Calculator app by using the soft keyboard to enter a bill amount and by clicking on the Increase (+) and Decrease (-) buttons to modify the tip percent. Note that this increases or decreases the tip by 1%.
14. Make sure to leave the emulator running.

Test the Instant Run feature

15. In Android Studio, open the Java class for the activity. Review this code. Note how it works with the widgets displayed in the corresponding layout.
16. Modify the code so clicking on the Increase (+) button increases the tip percent by 5% instead of 1%.
17. Click the Apply Changes button in the toolbar to swap in the change without recompiling or redeploying the app.
18. Switch to the emulator and click on the Increase (+) button to make sure this change has been applied correctly.

How to start a new app

Now that you know how to use Android Studio 2.3 to work with an existing device, you're ready to learn how to use it to start a new project that uses the constraint layout to align its components.

How to create a new project for an app

If Android Studio displays the Welcome page when you start it, you can create a new project for an app by selecting the "Start a new Android Studio project" item. Otherwise, you can select the File→New→New Project item from the menu system.

Either way, Android Studio displays a Create New Project dialog box like the one shown in figure 8. This dialog box displays a series of steps that allow you to create a new project for an app like the Tip Calculator.

In the first step, you can enter a name for the application as well as a name for the project and package. For the Tip Calculator app, I entered a name of "Tip Calculator", and a company domain of "murach.com". This generated a package name of "com.murach.tipcalculator".

In the second step, you can select a minimum SDK for the application. For the Tip Calculator app, I accepted the default value for the minimum SDK. In this case, that happened to be Android 4.0.3 (API 15).

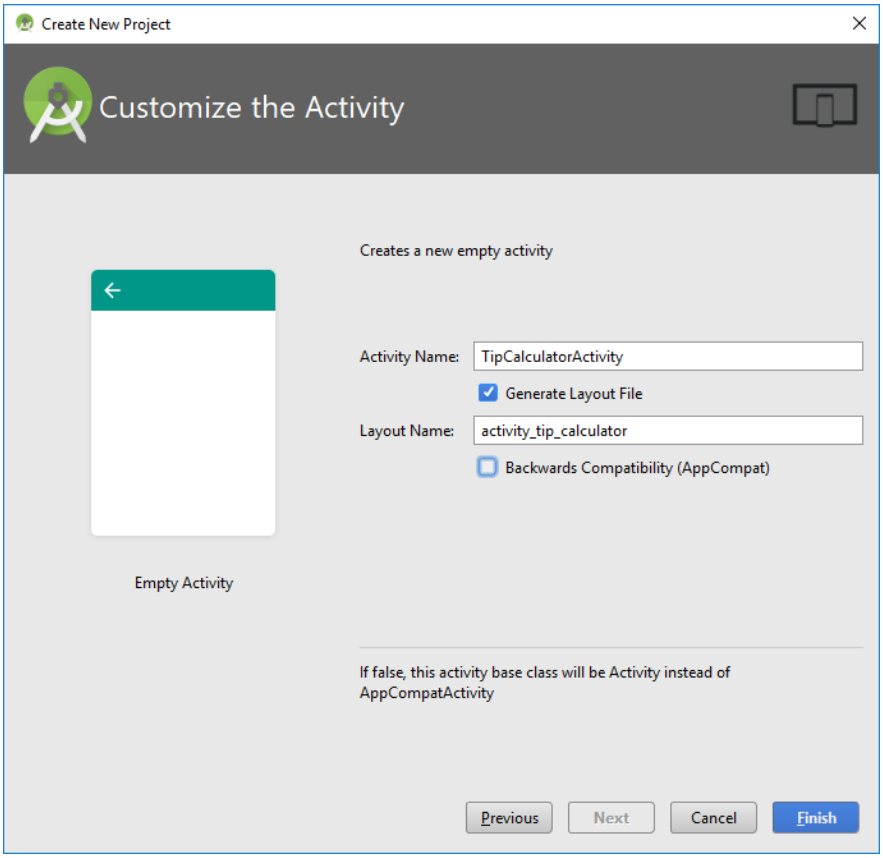
In the third step, you can select a template for the type of activity you want the project to start with. In Android development, an activity defines a screen. For the Tip Calculator app, I selected the Empty Activity template. This is the simplest template, and it's usually what you want when you're getting started.

In the fourth step, you can enter a name for the activity. For the Tip Calculator app, I entered a name of "TipCalculatorActivity" for the activity. This is the name of the Java class for the activity. When I did that, Android Studio automatically set the name of the corresponding layout to "activity_tip_calculator". This is the name of the file that stores the XML that defines the user interface for the activity. In Android development, a layout is a container that contains one or more child elements such as widgets and determines how they are displayed.

In the fourth step, I deselected the "Backwards Compatibility" option. As a result, the project doesn't include the v7 appcompat support library, and the TipCalculatorActivity class and doesn't inherit the AppCompatActivity class. Instead, the TipCalculatorActivity class inherits the Activity class as described in chapter 3 of *Murach's Android Programming*.

When you finish all four steps, you can click the Finish button. When you do, Android Studio creates a directory that corresponds with the project name, and it creates some additional directories and files that it uses to configure the project.

The Create New Project dialog box



Procedure

1. Start Android Studio.
2. If the Welcome page is displayed, select the "Start a new Android Studio project" item. Otherwise, select the File→New→New Project item from the menu system.
3. Enter a name for the application, project, and package and click the Next button.
4. Select the Phone and Tablet option, the minimum SDK for your project, and click the Next button.
5. Select the Empty Activity template and click the Next button.
6. Enter a name for the activity, deselect the "Backwards Compatibility" option, and click the Finish button.

Figure 8 How to create a new project for an app

How to work with a layout

The first step in developing an Android app is to develop the user interface. The easiest way to do this is to use the graphical layout editor that's provided by Android Studio to develop most of the user interface. This generates the XML for the user interface. Then, if necessary, you can review and modify this code.

Figure 9 shows the default layout of the activity that Android Studio generates when you create a new Android project that's based on the Empty Activity template as described in the previous figure.

Android provides several different types of layouts that you can learn about in *Murach's Android Programming*. For now, you'll learn how to use the constraint layout to create the interface for the Tip Calculator app. In this figure, the constraint layout contains a single TextView widget that displays a message of "Hello world!"

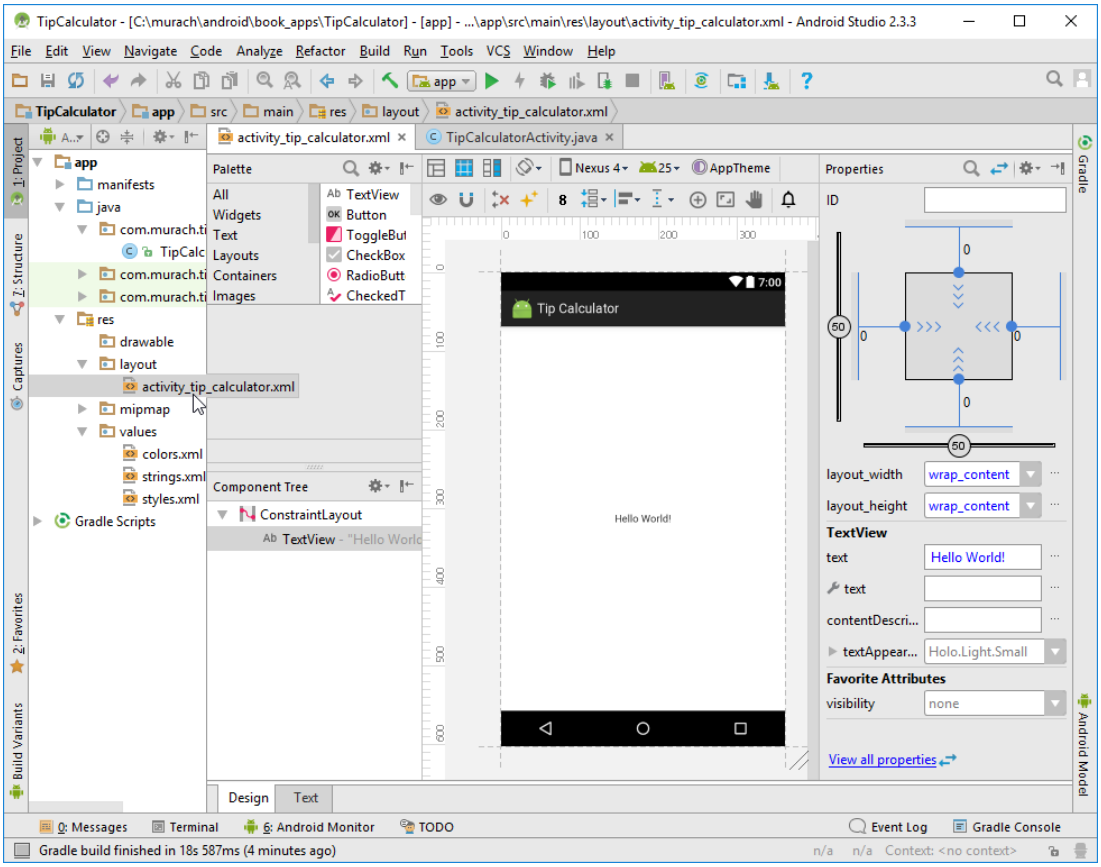
Since a new project typically generates at least one layout file for you, you can usually begin by opening an existing layout. To do that, display the Project window, expand the res (resources) directory, expand the layout directory, and open the XML file for the layout.

Once you've opened a layout, you can view its appearance in the graphical layout editor. Then, you can use the techniques described in the next few figures to add widgets to the layout, to set their properties, and to size them. Then, if you want to review or edit the XML code that's generated for the layout, you can click on the Text tab. Whenever you want, you can click the Design tab to return to the graphical layout editor.

If you want to add layouts to a project, you can do that by right-clicking on the layout directory in the Project window and selecting the New→Layout Resource File item. Then, you can enter the name of the file and the type of layout. For now, you can accept the default layout.

Conversely, if you want to delete a layout from a project, you can do that by right-clicking on the layout in the Project window and selecting the Delete item.

The default layout for an activity in a new project



Description

- To open an existing layout, open the Project window, expand the res (resources) directory, expand the layout directory, and double-click the name of the XML file.
- To view the layout in the graphical editor, click the Design tab.
- To view the XML for the layout, click on the Text tab.
- To create a new layout, right-click on the layout directory in the Project window and select the New→Layout Resource File item. Then, enter the name of the file and the type of layout. For now, you can accept the default layout.
- To delete a layout, right-click on the layout in the Project window and select the Delete item.

Figure 9 How to work with a layout

How to add and delete widgets

Figure 10 shows how to add widgets to a layout and how to delete widgets from a layout. To delete a widget, you can click on the widget in the graphical editor window or in the Component Tree window and press the Delete key. In this figure, for example, I deleted the "Hello World!" TextView widget shown in the previous figure.

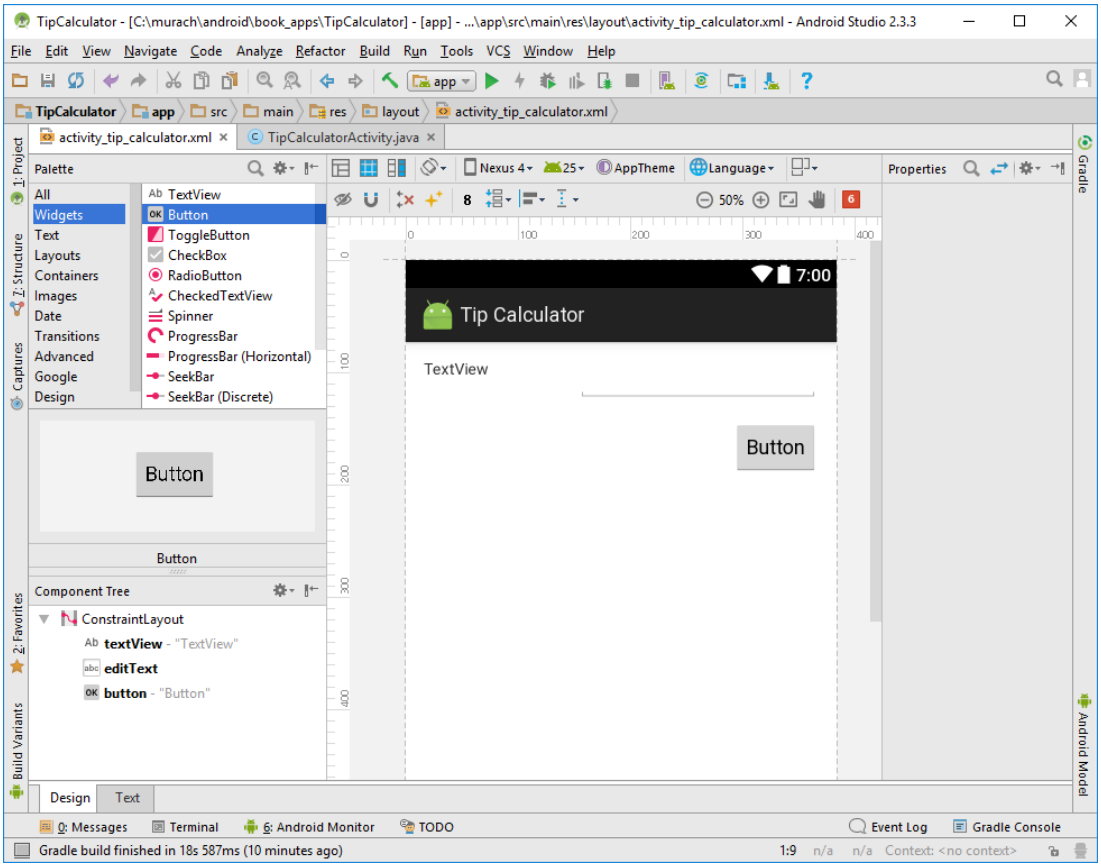
To add a widget, you just drag a widget from the Palette onto the layout. This figure, for example, shows three widgets after they have been added to the layout but before all of their properties have been set correctly.

As you add widgets to a layout, you often need to scroll through the categories in the Palette to help you find the widget you want to add. If necessary, you can click on the category for the widget to collapse or expand it. The Widgets category contains a variety of useful widgets including the TextView and Button widgets. The Text category, on the other hand, only contains variations of the EditText widget. For this figure, I used the EditText widget named Number (Decimal).

In most cases, you can align a widget by dragging it to where you want it. This sets properties of the widget such as its margins and constraints. You'll learn more about how to work with these properties later in this document.

As you add widgets to a layout, you may need to set its theme, which is a group of styles that define the look of the layout. To set the theme, select it from the toolbar that's displayed above the layout when it's in the graphical editor. In this figure, the AppTheme has been selected, which is usually what you want.

A layout after some widgets have been added to it



Description

- To add a widget to a layout, drag the widget from the Palette onto the layout. TextView and Button widgets are in the Widgets category, and EditText widgets are in the Text category.
- To delete a widget, click the widget in the graphical editor or the Component Tree window and press the Delete key.
- To align a widget, you can drag it to where you want it to be aligned. You'll learn more about how to align widgets later in this document.

Figure 10 How to add and delete widgets

How to set the display text and ID for a widget

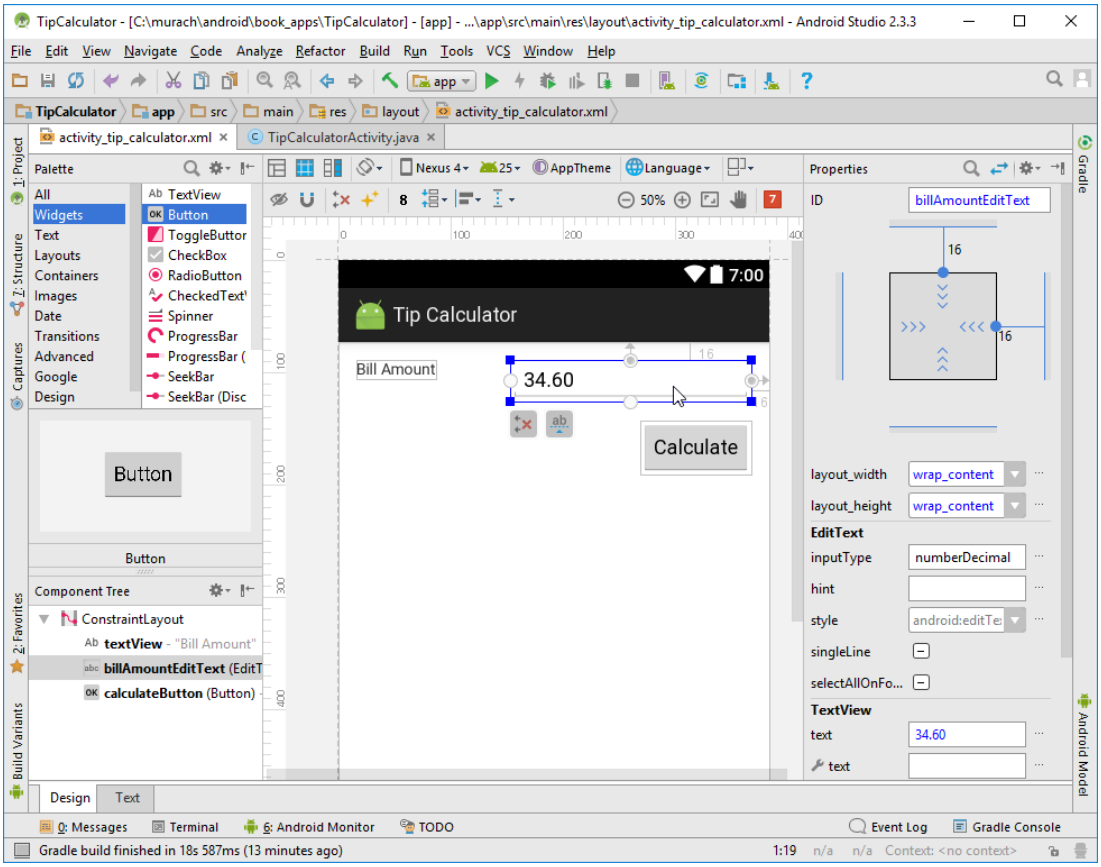
Figure 11 shows how to set the display text and ID for a widget. To do that, you can click on a widget and enter appropriate values into the Properties window. By default, Android Studio displays this window on the right side of the main window. If it isn't displayed there, you can display it by clicking on the Properties tab that's on the right side of the window.

By default, the Properties window displays the most common properties, including the text and ID properties. In this figure, I selected the EditText widget. Then, I set the text property to "34.60". Next, since the Java code for this app needs to access this widget, I set the ID for this widget to billAmountEditText. This ID clearly indicates that this widget is an EditText widget that contains the amount of the bill.

However, the Java code doesn't need to access the TextView widget that displays "Bill Amount". As a result, I didn't bother to set the ID property for this widget. Instead, I left it at its default value of textView. To view the ID for each widget, you can look at the Component Tree window.

As you review this figure, note that the inputType property sets the type of input that the EditText widget should accept. In this figure, the inputType property specifies a value of "numberDecimal". This indicates that the EditText widget should only accept numeric characters and the decimal point character. Of course, an EditText widget can accept many other types of input too such as plain text, email addresses, telephone numbers, passwords, and so on.

A layout after the display text and ID have been set for the widgets



Description

- To set the display text for a widget, select the widget in the main window and enter the display text in the text property of the Properties window.
- To set the ID for a widget, select the widget in the main window and specify an ID in the ID property of the Properties window.
- If you need to access a widget with Java code, it's a good practice to set its ID to a value that clearly identifies the widget.
- If you don't need to access a widget with Java code, you can leave the ID that was automatically generated by Android Studio.
- By default, the Properties window is displayed on the right side of the main window. If it isn't displayed there, you can click on the Properties tab on the right side of the main window to display it.

Figure 11 How to set the display text and ID for a widget

How to set other properties

Figure 12 shows how to use the Properties window to set other properties of a widget. To do that, select the widget and use the Properties window to change the property.

To search for a property, you can enter the first few letters of the property in the search box at the top of the Properties window. In this figure, for example, I entered "text" at the top of the Properties window. As a result, this Properties window only shows the properties that contain "text" in them. To display all properties again, you can click on the Cancel button that's displayed to the right of the search text.

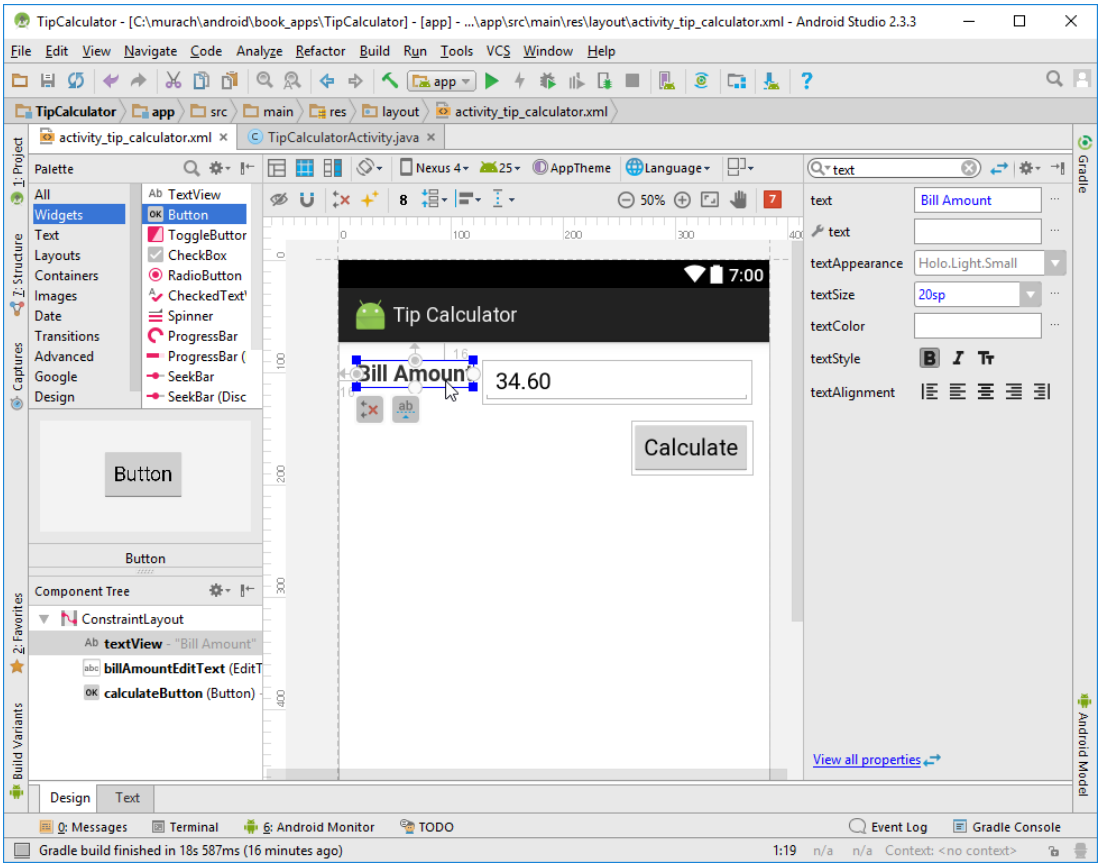
After you change a property from its default value, the Properties window displays the property in blue. In this figure, for example, the Properties window shows that the textStyle property has been set to bold. Similarly the textSize property has been set to 20sp.

For the textSize property, it's generally considered a best practice to use a unit of measurement known as a scale-independent pixel (sp). In this figure, the textSize property specifies a value of "20sp". A scale-independent pixel (sp) is a virtual pixel that you should use when specifying text size. Then, at runtime, Android scales these virtual pixels so the size of the text scales correctly for different screen densities.

For other properties such as margins, it's considered a best practice to use density-independent pixels (dp). A density-independent pixel is a virtual pixel that you should use when defining a layout. Then, at runtime, Android scales these virtual pixels to make sure it displays your layout properly for the density of the screen of the device that your app is running on.

To specify a density-independent pixel, it's possible to use an abbreviation of dip instead of dp. For example, you could use "16dip" instead of "16dp". In this book, I use dp for two reasons. First, it's shorter. Second, it's more consistent with the sp abbreviation that's used for scale-independent pixels.

A layout after some other properties of the widgets have been set



Common units of measurement for properties

Name	Abbreviation	Typical uses
Scale-independent pixels	sp	font sizes
Density-independent pixels	dp or dip	height, width, margins, etc.

Description

- To set a property for a widget, select the widget and use the Properties window to change the property.
- To switch between viewing all properties and the most common ones, you can click the View All Properties button at the top of the toolbar.
- To search for properties, you can click the Search button and type the first few letters of the property. To display all properties again, you can click the Cancel button that's displayed to the right of the letters that you typed.
- The Properties window displays the properties that have been changed from their default in blue.

Figure 12 How to set other properties

How to size widgets

Figure 13 shows how to size widgets. To do that, you often use the pre-defined values of match_parent, wrap_content, and 0dp described in this figure. In addition, you can specify a fixed size such as 150dp.

For a layout, the width and height properties are typically set to match_parent. This expands the layout so it takes up the entire parent. In most cases, this causes the layout to take up the entire screen.

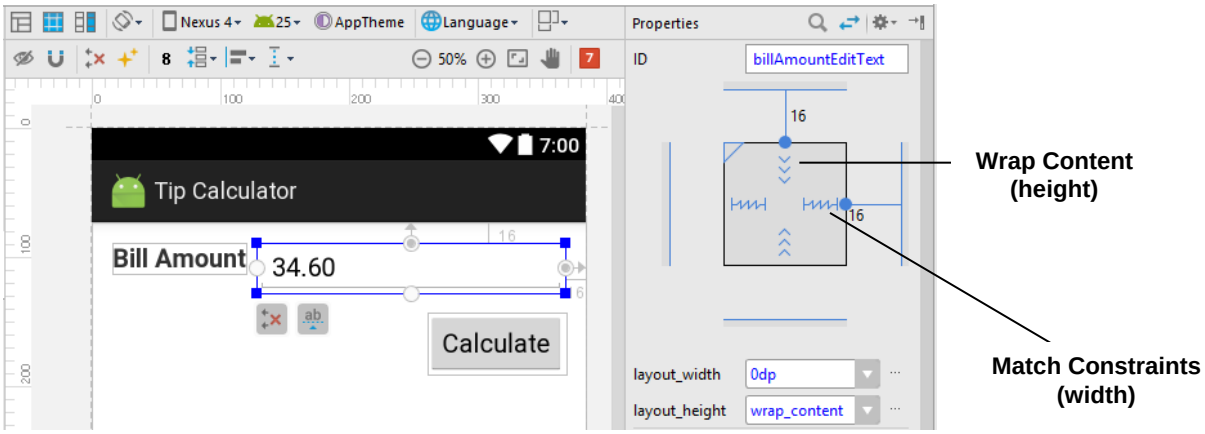
For a widget, the width and height properties are typically set to wrap_content. This forces the width and height to be just big enough to contain the widget and its content. In this figure, for example, the height for the EditText widget is set to wrap_content, and the Properties window displays this setting as three arrows (>>>).

If you specify a value of 0dp, the widget stretches so that it matches the widget's constraints. As a result, this setting is sometimes referred to as the *match constraints* setting. In this figure, for example, the first screen shows an EditText widget that uses the "match constraints" setting for its width, and the Properties window displays this setting as a squiggly line.

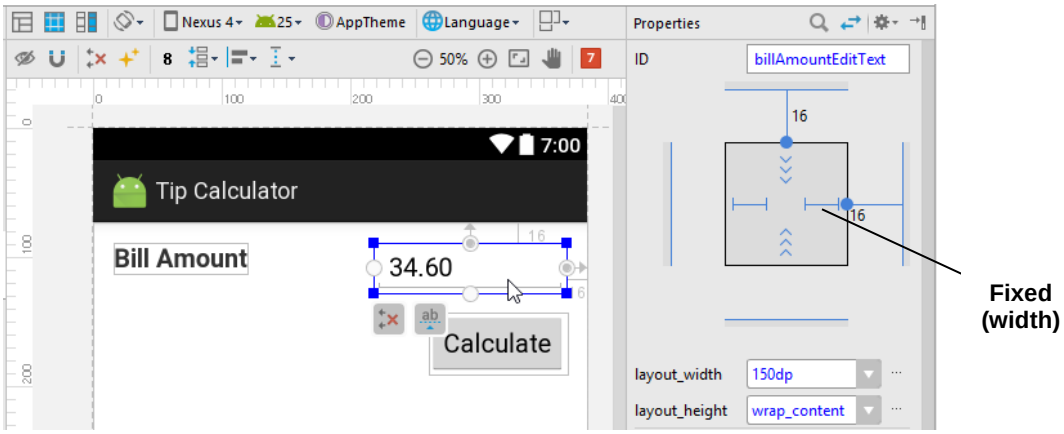
If you use density-independent pixels (dp) to specify the height or width of a widget, the widget has a fixed height or width. In this figure, for example, the second screen shows an EditText widget with a fixed width of 150dp, and the Properties window displays this setting with a straight line.

Although you can set the size of a widget by using its layout_width and layout_height properties, you can also set them by clicking on the icon for the setting in the Properties window. This cycles through the possible values for the height or width. For example, if you click repeatedly on the icon for a widget's width, it cycles through the wrap content, match constraints, and fixed width values.

A widget that determines its width by matching its constraints



A widget that uses a fixed amount for its width



Pre-defined values for setting height and width

Value	Description
`match_parent`	Stretches the height or width to match the parent container.
`wrap_content`	Wraps the height or width so it is large enough to display the content of the widget.
`0dp`	Stretches the height or width to match the widget's constraints. As a result, it's also known as the *match constraints* value.

Description

- To set the size of a widget, you can set its layout_width and layout_height properties to a fixed amount such as 150dp or to any of the predefined values shown above.
- You can also set the layout_width and layout_height properties by clicking on the size icons inside the Properties window.

Figure 13 How to size widgets

How to work with constraints

Now that you know how to add some widgets to a layout and set their properties, you're ready to learn how to use constraints to align them. To do that, you typically use Android Studio's graphical editor to work with the layout.

How to align a widget by a fixed amount

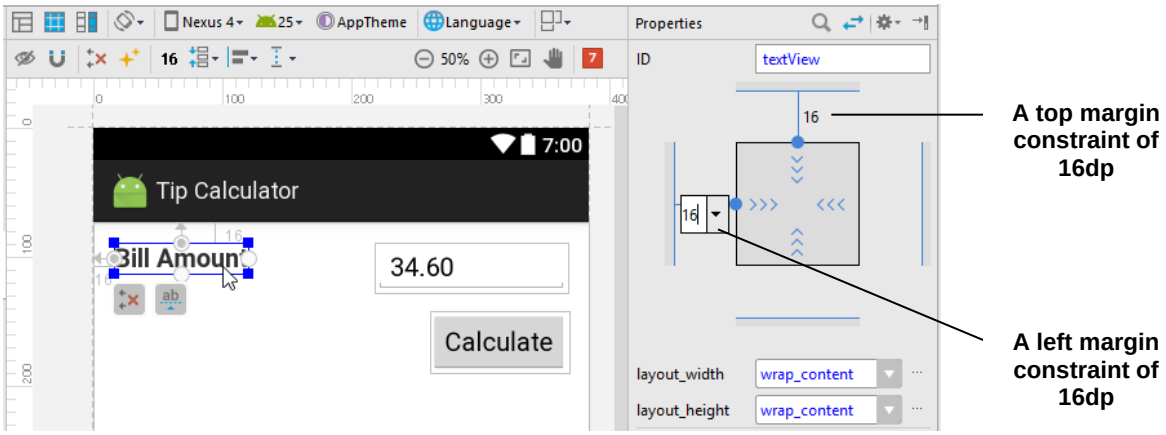
By default, Android Studio uses AutoConnect mode to automatically connect widgets to each other or to the edge of the layout. Typically, that results in aligning a widget by a fixed amount. In figure14, for example, the TextView widget is aligned by 16dp below the top edge of the layout and 16dp from the left side of the layout.

When you align a widget by a fixed amount, you must provide one fixed amount for the vertical axis (top or bottom) and one fixed amount for the horizontal axis (left or right). To do that, you can remove a constraint by clicking on one of its handle. For example, you could remove the top constraint for the TextView widget shown in this figure by clicking on the Top handle.

Then, you can add a new constraint by dragging a handle to another location. For example, to align a widget with the bottom of a layout, you could drag the bottom handle to the bottom edge of the layout. Then, if necessary, you can set the margin between the widget and the layout by clicking in the Properties window as shown in this figure and using the drop-down list to specify a new margin amount.

If you don't want Android Studio to automatically connect your widgets as you add them to a layout, you can click on the AutoConnect button that's in the toolbar above the layout to turn off AutoConnect mode. Then, when you add a widget, Android Studio will temporarily position your widgets in the graphical editor for the layout, but it won't set its constraints. As a result, you'll have to set them later or your layout won't display correctly.

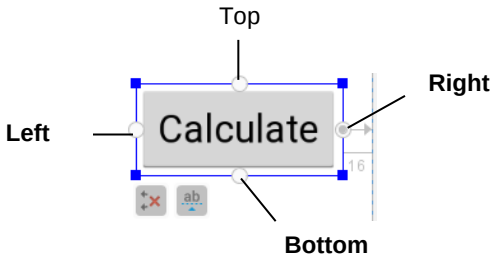
A widget that uses margin constraints to align it by a fixed amount



How Autoconnect mode works

- If Autoconnect mode is on, Android Studio automatically adds constraints when you drag a widget from the Palette window onto the layout. If it's off, you can manually add constraints after you add a widget to the layout.
- To turn Autoconnect mode on or off, you can click on the AutoConnect button in the toolbar immediately above the layout.

A close-up of a widget and its four constraint handles



How to add, modify, and clear a constraint

- In the graphical editor window and the Properties window, each widget has four constraint handles (top, right, bottom, and left). These handles are identified by circles.
- To add a constraint, you can drag the constraint handle to the edge of the layout or to another widget.
- To change the margins for a constraint, you can drag the widget or use the drop-down list in the Properties window.
- To clear a single constraint, you can click on the handle for the constraint in the graphical editor window or the Properties window.

Figure 14 How to align a widget by a fixed amount

How to align a widget by a percentage

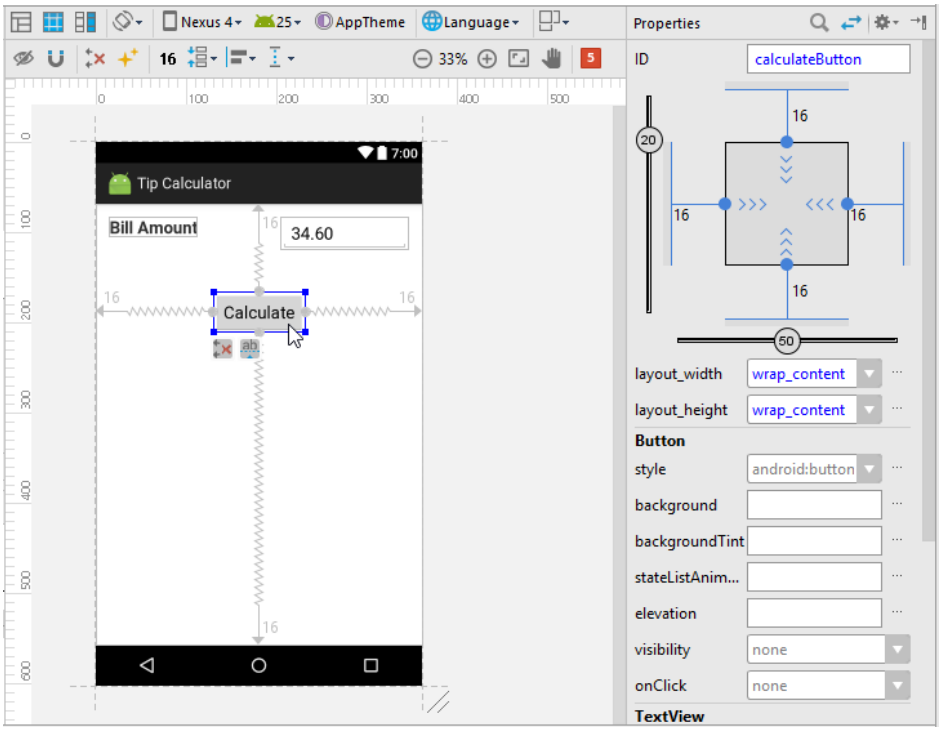
To align a widget by a fixed amount, the widget must have one constraint per axis. That way, Android can determine the horizontal and vertical location of the widget. However, if you have two constraints on the same axis, those constraints are known as *opposing constraints*. Then, you can use those constraints to align a widget by percentages as shown in figure 15.

In this figure, the Calculate button is using both constraints on the vertical axis (top and bottom) and both constraints on the horizontal axis (left and right). These opposing constraints are identified in the graphical editor by squiggly lines, not straight lines like the constraints for fixed widths. However, these opposing constraints also include margins that are of a fixed width. In this figure, the margins for these opposing constraints are all 16dp.

Within these margins, the Calculate button is aligned to be a certain percentage, or *bias*, for each axis. The sliders in the Properties window show that the Calculate button has a bias of 50% for its horizontal axis. As a result, it's centered horizontally. However, the vertical axis has a bias of 20%. As a result, it's displayed 20% of the way between the top margin and the bottom margin (closer to the top margin). To change the bias for either axis, you can drag the slider that's shown in the Propeties window. Or, you can just drag the widget within the graphical layout editor.

Opposing constraints are often useful if you need to center a widget. That way, the widget will be centered regardless of whether the app is displayed in portrait or lanscape orientation and regardless of whether the app is displayed on a device with a small or large screen.

A widget that uses opposing constraints to align it by percentages



Description

- When a widget has constraints on the same axis, those constraints are known as *opposing constraints*. For example, when a widget has left and right constraints, it has constraints on its horizontal axis.
- In the graphical editor, opposing constraints are identified by a squiggly line, not a straight line.
- When a widget has opposing constraints, you can align the widget on that axis using percentages such as 50% or 10%. This is known as setting the *bias* for that axis.
- To set the bias for an axis, you can display the Properties window and use the slider for the horizontal or vertical axis. Or, you can drag the widget within the graphical layout editor. This aligns the widget within its margin constraints.

Figure 15 How to align a widget by a percentage

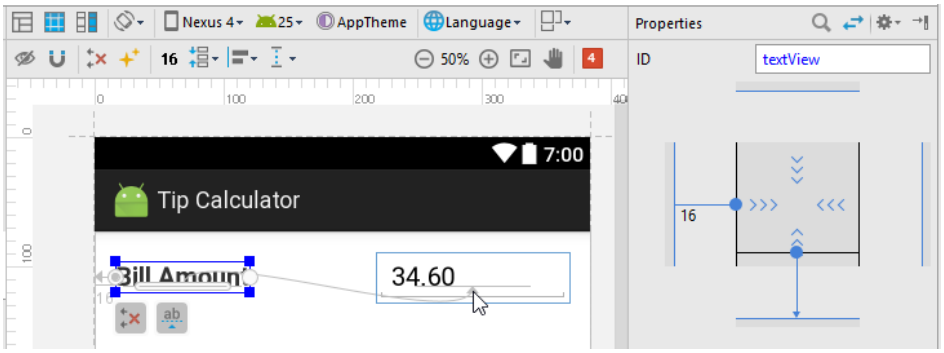
How to align widgets by their baselines

Figure 16 shows how to vertically align widgets by the text that the widgets contain. This is often a visually appealing way to align widgets. In this figure, for example, the baseline of the text that says "Bill Amount" is aligned with the baseline of the text for the amount of the bill, which is typically what you want.

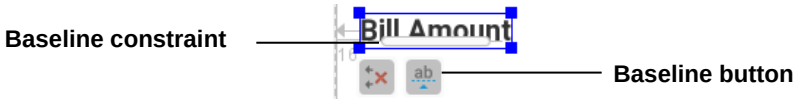
To align two widgets by their baselines, you start by positioning the cursor over the first widget to display some buttons for working with that widget. When these buttons are displayed, you can click the Baseline button to display a handle for the widget's Baseline constraint. This handle is an oval that appears immediately below the widget's text. Then, you drag this handle from the first widget and drop it on the second widget.

In the Properties window, a Baseline constraint is shown by a straight line that goes from the bottom of the widget to another straight line that's displayed below the widget. This shows that the widget has been aligned on its vertical axis. However, you still need to make sure to align the widget on its horizontal axis. In this figure, the widget uses a fixed width to align it on its horizontal axis by positioning it 16dp from the left side of the layout.

A widget that's aligned with another widget by its baseline



A close-up of a widget with its baseline constraint handle displayed



Procedure

1. Position the cursor over the widget to display its buttons.
2. If necessary, click the Baseline button to display the Baseline constraint.
3. Drag the Baseline constraint that's under the text to anther widget and drop it in that widget's Baseline constraint.

Description

- You can use a Baseline constraint to vertically align widgets by the baselines of the text that they contain.

Figure 16 How to align widgets by their baselines

How to finish the layout for a new app

At this point, you have all the skills you need to add widgets to a constraint layout and to get them to appear the way you want. At this point, you still have a little work to do to make sure the layout displays correctly and follows best practices.

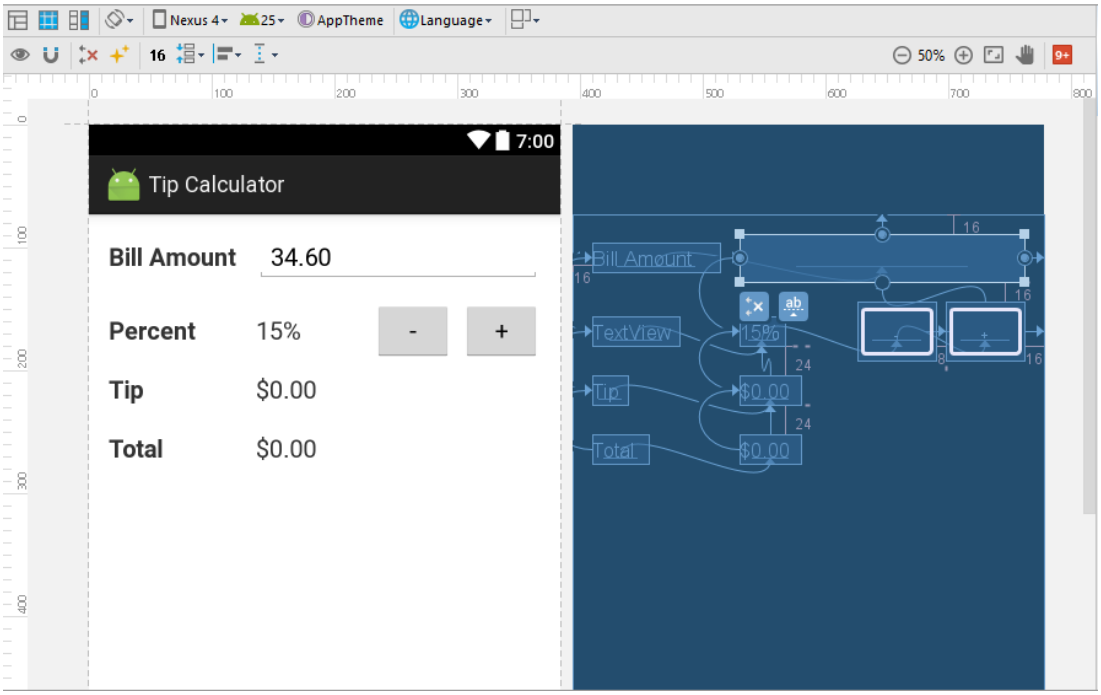
How to use the Design and Blueprint views

To start, you may want to take a quick look at your app the Blueprint view that's shown in figure 17 to quickly check to make sure that each widget has a constraint for both the vertical and horizontal axis. In addition, you may want to use this view to make sure that all of the constraints look correct to you.

You can use the buttons in the toolbar above the layout to switch between Design view, Blueprint view, or the Design + Blueprint view that displays both of these views side by side. Then, if the constraints aren't shown, you can display them by clicking on the Shows Constraints button.

The screen in this figure shows that the widgets of the Tip Calculator app are aligned in columns and rows. Here, all four widgets in the first column are left-aligned with the left side of the TextView widget that displays "Bill Amount". The four widgets in the second column are left-aligned with the left side of the EditText widget. The "+" button is right-aligned with the right edge of the EditText widget, and the "-" button is aligned just to the left of the "+" button. In addition, the widgets in each row are aligned vertically by their baselines. This shows how the Blueprint view provides a way for you to quickly get an idea of how the constraints for a layout work.

A layout displayed in the Design + Blueprint view



Description

- To switch views, click the Design button, the Blueprint button, or the Design + Blueprint button in the toolbar above the layout.
- To show or hide constraints in the Design and Blueprint views, click the Constraints button in the toolbar above the layout.

Figure 17 How to use the Design and Blueprint views

How to check a layout for different orientations and screen sizes

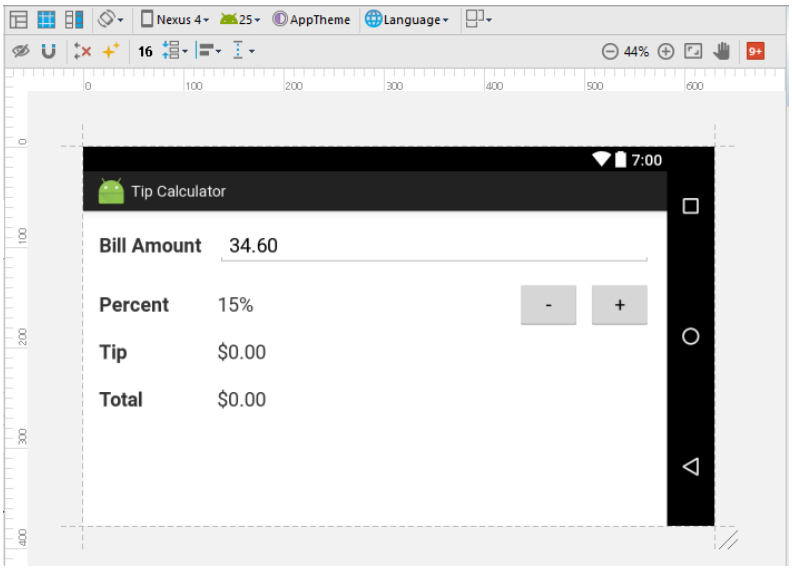
After you've checked your layout in Blueprint view, you should check it to make sure it displays correctly for different orientations and screen sizes. For example, the first screen capture in figure 18 shows the layout for the Tip Calculator app displayed in landscape orientation. And the second screen capture shows this layout displayed on a large screen size.

On both screens, the EditText widget stretches from the TextView that says "Bill Amount" to the right edge of the layout, which is what you want. In addition, the "+" button is aligned with the right side of the TextView widget, which is what you want.

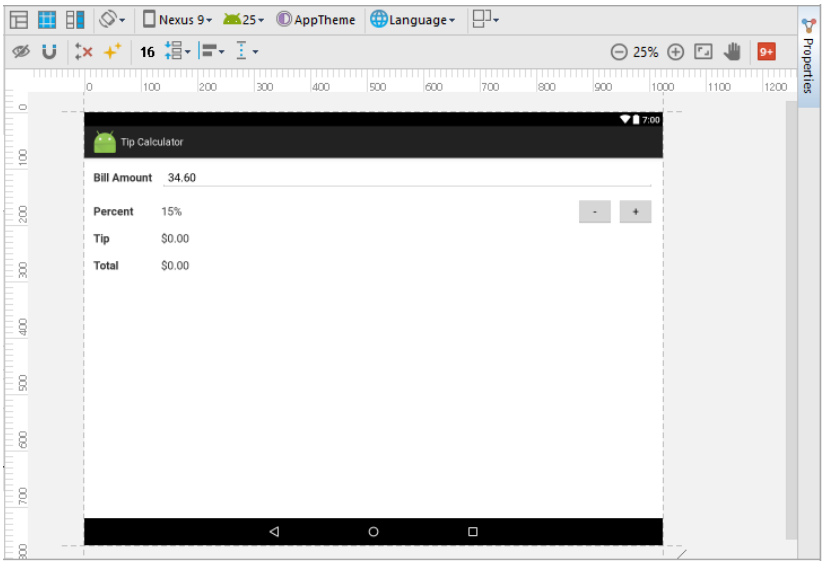
Fortunately, Android Studio makes it easy to check a layout for different orientations and screen sizes. To do that, you can use the toolbar that's displayed above the layout. You can click the Orientation button to switch between portrait and landscape orientation. Or, you can select a device with a different screen size from the drop-down list of devices. In this figure, the first screen is displayed on a Nexus device with a 4" screen, and the second screen is displayed on a Nexus device with a 9" screen.

Keep in mind that checking a layout on different screen sizes isn't the same as actually running the app on different devices or emulators with different screen sizes. Still, using the graphical editor to do some preliminary checking is a good first step and can greatly reduce the number of layout problems that you encounter later in the development process.

A layout displayed in landscape orientation



A layout displayed on a large screen size



Description

- To switch orientation, click the Orientation button in the toolbar above the layout.
- To switch to a device with a different screen size, select the device from the drop-down list that's available from the toolbar above the layout.

Figure 18 How to check a layout for different orientations and screen sizes

How to fix warnings and errors

When you develop a layout, Android Studio typically detects warnings and errors and displays a count of them in the Warnings and Errors button that's on the right side of the toolbar that's above the layout. To display these warnings, you can click this button. When you do, Android Studio displays a Lint Warnings dialog box like the one shown in figure 19.

This dialog box includes one line for each warning and error. To get more information about a warning or an error, you can click on it.

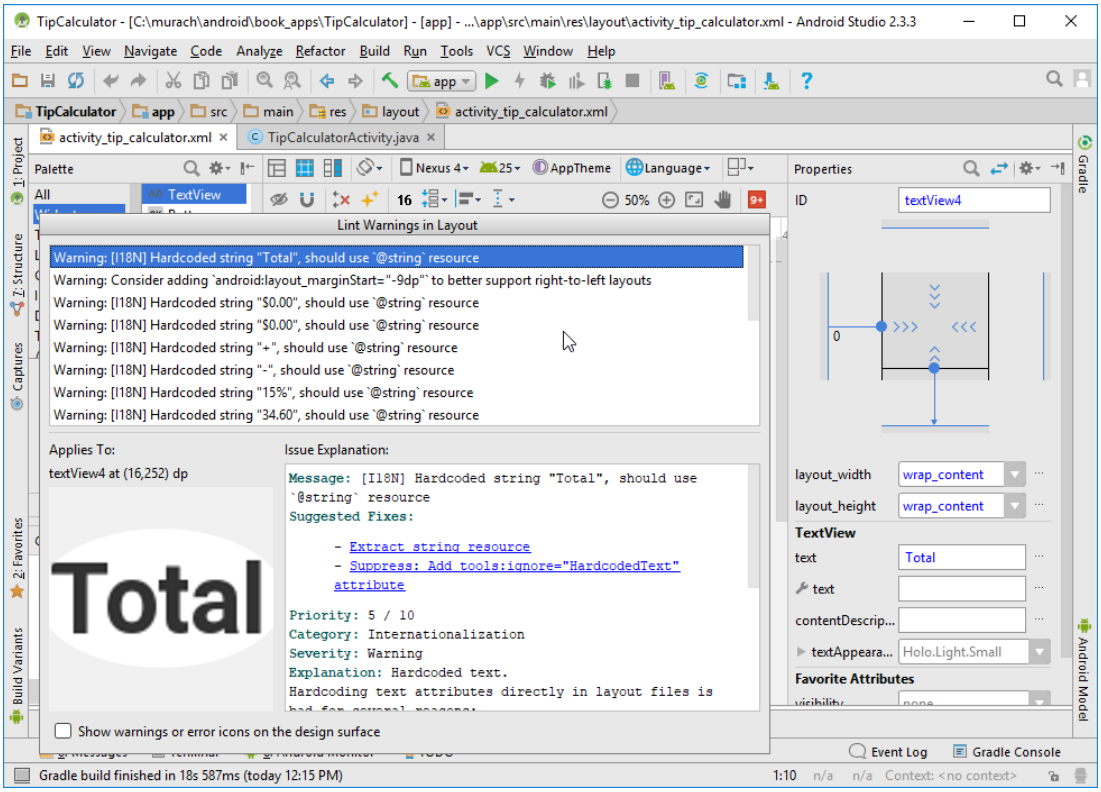
Some warnings include a link that you can use to fix the warning. In this figure, for example, the "Hardcoded string" warning includes a link that says "Extract string resource". As a result, you can click on this link to extract the string resource as shown in the next figure.

For other warnings, you may need to switch to Text view to modify the XML for the layout. In this figure, for example, the second warning says, "Consider adding android:layout_marginStart to better support right-to-left layouts". As a result, you can fix this warning by switching to Text view and modifying the XML for the layout by adding a layout_marginStart property directly below the layout_marginLeft property.

When you review warnings, remember that fixing them is optional. For example, extracting string resources is a good practice if you ever want to translate your app into multiple languages. However, if you aren't going to do that, this just creates some extra and unnecessary work. Similarly, if your app doesn't need to support right-to-left layouts, you probably don't need to add the layout_marginStart and layout_marginEnd properties to your layout.

However, if you encounter errors, you must fix them or your layout won't display properly. For example, if you have a widget that doesn't provide at least one constraint for its vertical axis, Android Studio can display it correctly in the graphical editor, but an Android device won't be able to determine where to display this widget. As a result, when you run the app, the layout won't display correctly, even though it might display correctly in the graphical editor.

A layout with the Lint Warnings dialog displayed



XML code that adds a layout_MarginStart property

```
android:layout_marginLeft="16dp"
android:layout_marginStart="16dp"
```

XML code that adds a layout_MarginEnd property

```
android:layout_marginRight="8dp"
android:layout_marginEnd="8dp"
```

Description

- To display the warnings for a layout, click the Warnings and Errors button that's in the toolbar above the layout. This should display the Lint Warnings dialog box.
- To get more information about an error or warning, click it. Some warnings include a link that makes it easy to fix the warning. For other warnings, you may need to switch to Text view and modify the XML for the layout.
- You must fix errors or your layout won't display correctly. Fixing warnings is optional, but recommended.

Figure 19 How to fix warnings and errors

How to extract the display text into a resource file

The Android development environment encourages separating the display text from the rest of the user interface. The advantage of this approach is that it's easier to create international apps that work in multiple languages. The disadvantage is that it takes a little extra work to set the display text for the user interface.

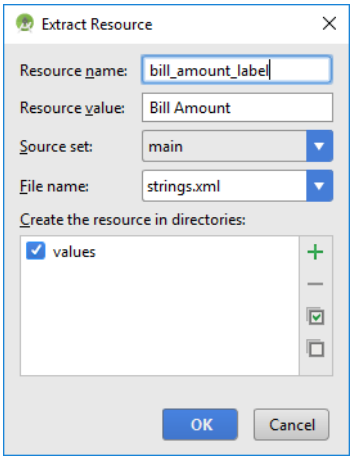
However, figure 20 shows a technique that makes it easy to set the display text, and this extra step shouldn't slow down your development much once you get used to it. To start, you display the Lint Warnings dialog shown in the previous figure. Then, you click on warning, click on the link that says "Extract string resource", and specify a name for the resource. In this figure, for example, I entered a resource name of bill_amount_label for a value of "Bill Amount". By default, this stores the string in the resource file named strings.xml that's shown in the next figure.

If you use the Properties window to check the value of the text property for the TextView widget that says "Bill Amount", you'll see that it is set to:

```
@string/bill_amount_label
```

This indicates that this text property has been set to the string named bill_amount_label that's stored in the strings.xml file.

The Resources dialog box



Procedure

1. In the graphical editor for the layout, click the Show Warnings and Errors button that's immediately above the layout.
2. If a warning says, "Hardcoded string should use '@string' resource", click on the warning.
3. Click on the link that says "Extract string resource".
4. Use the Extract Resource dialog to enter a name for the string and click the OK button.

Description

- It's generally considered a best practice to store the display text for your app in a separate XML file. This makes it easy to create international apps that work in multiple languages.
- The easiest way to store the display text in a separate XML file is to use extract it from the layout after you're sure it has been set correctly.

Figure 20 How to extract the display text into a resource file

The resource files for the user interface

When you use the technique shown in the previous figure to set the display text for your app, Android Studio stores the text in a strings.xml file like the one that's shown in figure 21. If you want, you can open the strings.xml file and edit it just as you would edit any other XML file.

This figure shows the strings.xml file for the Tip Calculator app. Here, the first string is set to "Tip Calculator". This string is displayed in the taskbar for the main activity of the app. In addition, it's used in the manifest file to identify the name of the app.

The next ten strings are used for the widgets. The strings that provide labels for widgets have names that end with "_label". Other strings provide default values for the bill amount, tip percent, tip amount, and total. These default values make it easier to use the graphical editor to set the properties for the widgets that display these values. In addition, they can make it easier to test an app. For example, when the app starts, it displays a value of 34.60 in the editable text view for the bill amount. As a result, you don't have to enter a bill amount to test the app.

However, before you put the app into production, you can modify the app so it sets these values to appropriate starting values for the app. For example, you probably want to supply an empty string for the bill amount. That way, when the app starts, the editable text view for the bill amount will be blank.

If you want to supply a strings.xml file for another country, you can create a values-xx directory within the res directory where xx is the ISO two-letter code for the country. For example, fr is the two-letter country code for France. Then, you can supply a strings.xml file within that directory that uses French instead of English. To find a list of these two-letter codes, you can search the Internet for "ISO two-letter country code".

The location of the strings.xml file

```
res/values/strings.xml
```

The strings.xml file for the Tip Calculator app

```
<resources>
    <string name="app_name">Tip Calculator</string>

    <string name="bill_amount_label">Bill Amount</string>
    <string name="bill_amount">34.60</string>

    <string name="tip_percent_label">Percent</string>
    <string name="tip_percent">15%</string>
    <string name="increase">+</string>
    <string name="decrease">-</string>

    <string name="tip_amount_label">Tip</string>
    <string name="tip_amount">$0.00</string>

    <string name="total_amount_label">Total</string>
    <string name="total_amount">$0.00</string>
</resources>
```

The location of a strings.xml file for France

```
res/values-fr/strings.xml
```

Description

- When you use the technique shown in the previous figure to extract the display text for your app into a resource file, Android Studio stores the text in the strings.xml file.
- If you want, you can open the strings.xml file and edit it just as you would edit any XML file.
- To provide a strings.xml file for a country that uses another language, you can create a values-*xx* directory within the res directory to store the strings.xml file. Here, you can replace *xx* with the ISO two-letter country code.

Figure 21 The resource file for the display text

Figure 22 shows the XML for the user interface. This code is stored in the file named activity_tip_calculator.xml. Most of this code is generated when you use the graphical editor to work with the widgets of the user interface. However, you can edit the XML for the user interface whenever you want. For example, you may need to do this to fix warnings as described earlier in this document. In this figure, the code that was manually added to fix warnings is highlighted, but the rest of the code was generated by the graphical editor.

To start, the ConstraintLayout element that's available from the android.support.layout library defines a constraint layout that contains the widgets for the user interface. Here, the layout_width and layout_height attributes use a value of match_parent to specify that the layout should use the whole screen.

The XML for this user interface includes comments to identify each row of widgets. These comments use the same syntax as an HTML comment (start with <!-- and end with -->).

The first TextView element sets the layout_width and layout_height attributes to a value of wrap_content to specify that the widget should be just wide enough and tall enough to fit its content. The text attribute sets the display text for the widget to the string resource named bill_amount_label. The textSize attribute sets the size of the text to 20 scale-independent pixels (20sp). The textStyle attribute sets the style of the text to bold. And the constraint attributes align this widget with the baseline of the EditText widget and 16 density-independent pixels (16dp) from the left edge of the layout.

The EditText element displays the widget that allows the user to enter a bill amount. Here, the layout_width has been set to a value of 0dp, so that the width matches its constraints. Then, one constraint aligns the left side of this widget with the right side of the "Bill Amount" TextView widget, and a second constraint aligns the right side of the EditText widget with the right side of the container. As a result, the EditText widget stretches horizontally to fill all space between these two constraints. Also, the id attribute sets the ID for this widget to billAmountEditText, which makes it easy for the Java code of the app to work with this widget. Finally, the inputType element specifies that the soft keyboard should only allow the user to enter decimal numbers.

The next two TextView elements define the widgets that display the tip percent. The Button elements define the buttons that allow the user to increase or decrease the tip percent.

The last four TextView elements define the widgets that display the tip and total amounts.

The XML for the layout **Page 1**

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".TipCalculatorActivity">

    <!-- The bill amount -->

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/bill_amount_label"
        android:id="@+id/textView"
        android:textSize="20sp"
        android:textStyle="bold"
        android:paddingTop="10dp"
        tools:layout_constraintTop_creator="1"
        tools:layout_constraintLeft_creator="1"
        app:layout_constraintLeft_toLeftOf="parent"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"

app:layout_constraintBaseline_toBaselineOf="@+id/billAmountEditText"/>

    <EditText
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"
        android:ems="8"
        android:id="@+id/billAmountEditText"
        android:text="@string/bill_amount"
        android:textSize="20sp"
        android:layout_marginRight="16dp"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="16dp"
        app:layout_constraintLeft_toRightOf="@+id/textView"
        android:layout_marginLeft="16dp"
        app:layout_constraintHorizontal_bias="0.0" />
```

Figure 22 The resource file for the layout (part 1 of 3)

The XML for the layout — Page 2

```
<!-- The tip percent -->

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/tip_percent_label"
    android:id="@+id/textView2"
    android:textSize="20sp"
    android:textStyle="bold"
    app:layout_constraintBaseline_toBaselineOf="@+id/percentTextView"
    android:layout_marginLeft="0dp"
    android:layout_marginStart="0dp"
    app:layout_constraintLeft_toLeftOf="@+id/textView" />

<TextView
    android:layout_width="70dp"
    android:layout_height="22dp"
    android:text="@string/tip_percent"
    android:id="@+id/percentTextView"
    android:textSize="20sp"
    android:layout_marginLeft="0dp"
    android:layout_marginStart="0dp"
    app:layout_constraintLeft_toLeftOf="@+id/billAmountEditText"
    app:layout_constraintBaseline_toBaselineOf="@+id/percentDownButton" />

<Button
    android:id="@+id/percentDownButton"
    android:layout_width="48dp"
    android:layout_height="48dp"
    android:layout_marginRight="8dp"
    android:layout_marginEnd="8dp"
    android:text="@string/decrease"
    android:textSize="20sp"
    app:layout_constraintBaseline_toBaselineOf="@+id/percentUpButton"
    app:layout_constraintRight_toLeftOf="@+id/percentUpButton" />

<Button
    android:id="@+id/percentUpButton"
    android:layout_width="48dp"
    android:layout_height="48dp"
    android:layout_marginRight="0dp"
    android:layout_marginEnd="0dp"
    android:layout_marginTop="8dp"
    android:text="@string/increase"
    android:textSize="20sp"
    app:layout_constraintRight_toRightOf="@+id/billAmountEditText"
    app:layout_constraintTop_toBottomOf="@+id/billAmountEditText" />
```

Figure 22 The resource file for the layout (part 2 of 3)

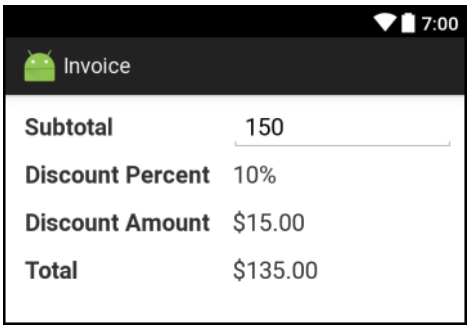
The XML for the layout **Page 3**

```
    <!-- The tip amount -->

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tip_amount_label"
        android:id="@+id/textView3"
        android:textSize="20sp"
        android:textStyle="bold"
        android:layout_marginLeft="0dp"
        android:layout_marginStart="0dp"
        app:layout_constraintLeft_toLeftOf="@+id/textView2"
        app:layout_constraintBaseline_toBaselineOf="@+id/tipTextView" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tip_amount"
        android:id="@+id/tipTextView"
        android:textSize="20sp"
        android:layout_marginTop="24dp"
        app:layout_constraintTop_toBottomOf="@+id/percentTextView"
        android:layout_marginLeft="0dp"
        android:layout_marginStart="0dp"
        app:layout_constraintLeft_toLeftOf="@+id/percentTextView" />

    <!-- The total amount -->

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/total_amount_label"
        android:id="@+id/textView4"
        android:textSize="20sp"
        android:textStyle="bold"
        android:layout_marginLeft="0dp"
        android:layout_marginStart="0dp"
        app:layout_constraintLeft_toLeftOf="@+id/textView3"
        app:layout_constraintBaseline_toBaselineOf="@+id/totalTextView" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/total_amount"
        android:id="@+id/totalTextView"
        android:textSize="20sp"
        android:layout_marginTop="24dp"
        app:layout_constraintTop_toBottomOf="@+id/tipTextView"
        android:layout_marginLeft="0dp"
        android:layout_marginStart="0dp"
        app:layout_constraintLeft_toLeftOf="@+id/tipTextView" />

</android.support.constraint.ConstraintLayout>
```

Figure 22 The resource file for the layout (part 3 of 3)

Exercise 2 Create the user interface for the Invoice Total app

In this exercise, you'll create the Invoice Total app. When you're done, a test run should look something like this:



Create the project

1. Start Android Studio.
2. Create a new project for an Android app and store it in a project named Invoice in this directory:

   \murach\android\ex_starts

   This project should be stored in a package named com.murach.invoice, it should be based on the Empty Activity template, and it should not include the Backwards Compatibility (AppCompat) library.
3. If you have an Android device, run the app on that device. Otherwise, run it in an emulator. This app should display a message that says "Hello world!" in the center of the screen.

Create the user interface

4. Open the layout for the activity that's stored in the res\layout directory. If necessary, click the Design tab to display the graphical editor.
5. Delete the TextView widget that displays the "Hello world!" message.
6. Add the seven TextView widgets and one EditText widget to the layout.
7. Set the text properties for each widget. When you're done, the user interface should have the widgets and text shown above. However, these widgets should look different since you haven't set their properties yet.
8. Set the id properties for the EditText widget to subtotalEditText. Set the id property for the three TextView widgets that display the calculations to the user.
9. Set the textSize property for all eight widgets to 20sp.
10. Set the textStyle property for the four widgets in the left column to bold.
11. Use the graphical editor to size and align the widgets so they are displayed correctly.

Check the user interface

12. Display the user interface in Blueprint view and use it to make sure that the constraints align the components correctly. Here, each constraint must have one constraint for its vertical axis and one for its horizontal axis.
13. In the graphical editor, check the user interface in landscape and portrait orientation to make sure it's displayed correctly for both orientations.
14. In the graphical editor, check the user interface on a small and large screens to make sure it's displayed correctly on different screen sizes.
15. Test the user interface by running the app on a physical device or an emulator. The app should allow you to enter a subtotal, but that shouldn't cause the discount percent, discount amount, or total values to change.

Fix all warnings and errors

16. In the graphical editor, click the Warnings and Errors button in the toolbar. Then, review the warnings and errors displayed in this dialog box.
17. Display all "Hardcoded string" warnings and use the "Extract" link to extract the string into the string resources file.
18. Open the strings.xml file that's in the res/values directory. Then, review the code. It should contain the display text for your app.
19. Click the Text tab to review the XML for this layout. Note how the XML attributes correspond with the properties that you have set with the graphical editor.
20. If you want to fix the "marginStart" and "marginEnd" warnings, add the layout_marginStart and layout_marginEnd attributes directly below the layout_marginLeft and layout_marginRight attributes.
21. In the graphical editor, check whether the layout has any warnings or errors. If it does, click the Warnings and Errors button in the toolbar to review the warnings and errors. At this point, you shouldn't have any significant warnings or errors.
22. Test the user interface by running the app on a physical device or emulator. The app should allow you to enter a subtotal, but that shouldn't cause the discount percent, discount amount, or total values to change.

Updates for chapter 3 and beyond

Now that you know how to use Android Studio to develop a Tip Calculator app that uses the constraint layout, you're ready to continue with chapter 3 of *Murach's Android Programming*. However, as you progress through this book, you may want to convert relative layouts to constraint layouts. In addition, when you get to chapter 18, you may want to fix the Run Tracker app that's presented so that it works with Android 6.0 (API 23) and later.

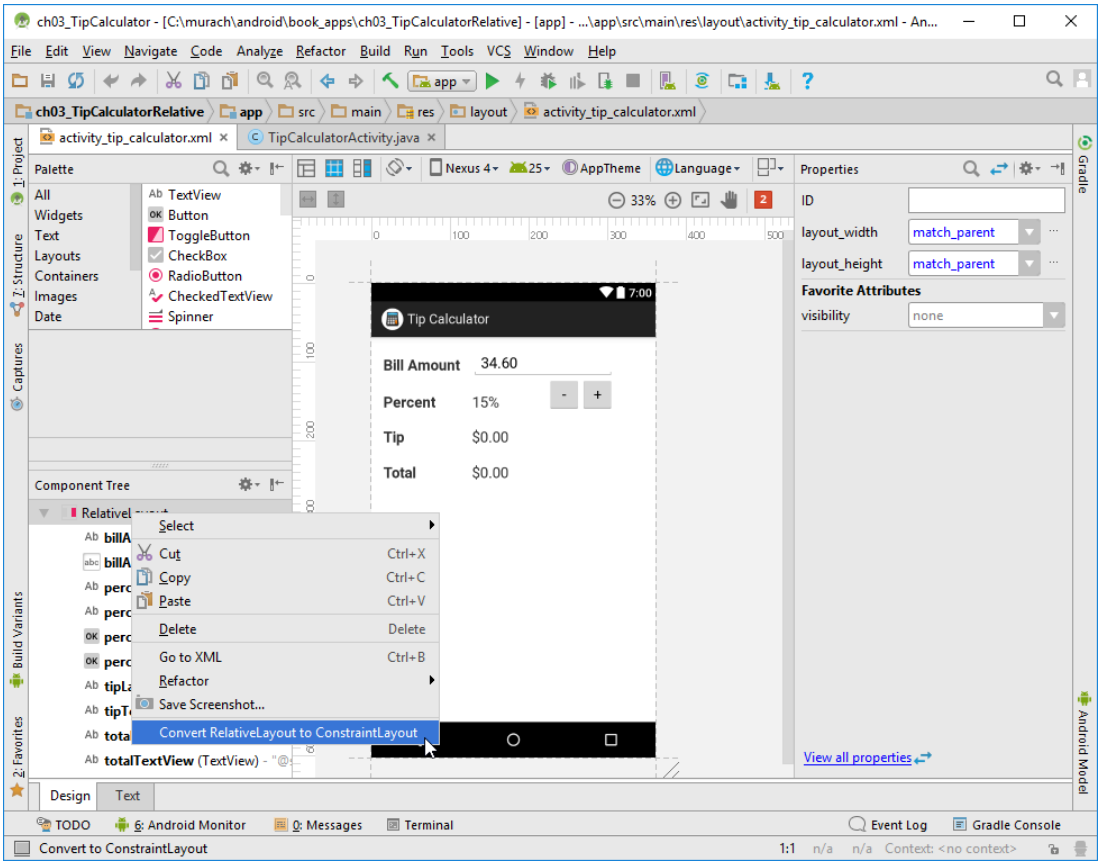
How to convert a relative layout to a constraint layout

In chapter 3, *Murach's Android Programming* shows how to write the Java code that provides the functionality for the user interface of the Tip Calculator application. This code is the same regardless of whether the user interface uses a relative layout or a constraint layout, and the code for both versions is included in the download for this book. More specifically, the ch03_TipCalculator project uses the relative layout, and the ch03_TipCalculator_Constraint project uses the constraint layout.

After chapter 3, the download for this book only provides the apps with the older relative layout. That way, the apps in the download match the code presented in *Murach's Android Programming*. Fortunately, if you want to convert these user interfaces to the constraint layout, Android Studio makes it easy to do that. All you need to do is follow the procedure presented in figure 23.

In most cases, you don't need to perform this conversion. However, converting to the constraint layout makes it easier to use Android Studio to modify a layout and improve its appearance. As a result, if you want to do that, you may want to convert to the constraint layout. Then, you can use the techniques shown in this document to modify the appearance of the layout.

A relative layout that's being converted to a constraint layout



Procedure

1. In the Component Tree window, right-click on the RelativeLayout component and select the Convert RelativeLayout to ConstraintLayout command.
2. At the Convert to ConstraintLayout dialog, click the OK button.
3. If you get a dialog that asks you if you want to add the constraint-layout library, click OK to add that library.
4. If the constraints don't display the widgets on your layout correctly, use the techniques presented earlier in this document to modify the constraints so they display the widgets correctly.

Description

- Android Studio makes it easy to convert old layouts such as the RelativeLayout to the new ConstraintLayout.
- If you want to remove all constraints and start from scratch, you can click the Clear All Constraints button in the toolbar above the layout.

Figure 23 How to convert a relative layout to a constraint layout

How to fix the Run Tracker app

In chapter 18, *Murach's Android Programming* shows how to work with locations and maps. To illustrate these concepts, this chapter presents the Run Tracker app. Unfortunately, this app doesn't request location permissions at runtime, which is required with Android 6.0 (API 23) and later. As a result, it crashes if you try to run it on devices that are running API 23 or later.

To fix this issue, you can add the code shown in part 1 of figure 24 to the StopwatchActivity class. This class defines the activity that's loaded when the app starts. As a result, it's a good point in the app to request the location permissions that are necessary for this app to work correctly.

To start, it's a good practice to code a constant that specifies a code for the permission that you are going to request. In this case, the first code example creates a constant that has a name that's appropriate for requesting the ACCESS_FINE_LOCATION permission.

The second code example begins with an if statement that checks whether the Android version of the device is greater than API 22. If so, it executes the code that's necessary to request permissions at runtime.

The nested if statement uses the checkSelfPermission method to check whether the device has already granted the ACCESS_FINE_LOCATION permission that's specified in the Manifest file to the app. If not, this code uses the requestPermissions method to display a dialog that requests the permissions from the user.

If the user grants the permission, the app will work correctly. However, if the user denies the permission, the app will crash when you use it. Of course, you could modify the nested if statement so it displays an appropriate message and prevents the app from crashing.

The RunTracker app that contains these fixes

```
book_apps\ch18_RunTracker_Fixed
```

Two methods for checking permissions with API 23 and later

Method	Description
checkSelfPermission(permission**)**	This method checks the specified permission and returns Package.PERMISSION_GRANTED constant if the user has granted the permission or Package.PERMISSION_DENIED if the user has not.
requestPermissions(permissions, request_code**)**	This method displays a dialog that asks the user to grant or deny the specified array of permissions and it sets the .

A constant of the StopwatchActivity class for a request code

```
private final int PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION = 1;
```

Code in the onStart method that checks the permission

```
if (Build.VERSION.SDK_INT > 22) {
    if (checkSelfPermission(
            Manifest.permission.ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED) {
        requestPermissions(
                new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                PERMISSIONS_REQUEST_ACCESS_FINE_LOCATION);
    }
}
```

Description

- Android 6.0 (API 23) and later and later request permissions at runtime when the user needs them instead of requesting them when they install the app.
- Since the Run Tracker app presented in *Murach's Android Programming* does not request these permissions at runtime, it crashes on API 23 and later when you click the Start button of the Stopwatch activity. To fix this issue, you can add the code shown above to the StopwatchActivity class.
- You can also manually grant the Run Tracker location permissions after it crashes. To do that, run the app on a device. Then, after it crashes, use the device's operating system to specify Settings→Application Manager→Run Tracker→Allow Location.

Figure 24 How to fix the Run Tracker app (part 1 of 2)

The Run Tracker app presented in chapter 18 also uses the deprecated getMap method in its RunMapActivity class. To fix this, you can use the getMapAsync method instead. However, for this to work, you must first implement the OnMapReadyCallback interface and its onMapReady method as shown in part 2 of figure 24.

With the deprecated getMap method, you had to check whether the GoogleMap object was null to prevent an excpetion. However, the onMapReady method is called when the GoogleMap object is ready. As a result, you don't need to check if this object is null. Instead, you can just use this object as shown in the second code example.

A method of the SupportMapFragment class

Method	Description
getMapAsync(callback**)**	Specifies the callback object that contains the onMapReady method that's triggered when the GoogleMap instance is ready.

A RunMapActivity class that implements the OnMapReadyCallback

```
public class RunMapActivity extends FragmentActivity
        implements OnClickListener, ConnectionCallbacks,
                   OnConnectionFailedListener, OnMapReadyCallback  {
```

The onMapReady method

```
    @Override
    public void onMapReady(GoogleMap googleMap) {
        map = googleMap;
        map.getUiSettings().setZoomControlsEnabled(true);
        googleApiClient.connect();
    }
```

An onStart method that calls the getMapAsync method

```
    @Override
    protected void onStart() {
        super.onStart();

        // if GoogleMap object is not already available, get it
        if (map == null) {
            FragmentManager manager = getSupportFragmentManager();
            SupportMapFragment fragment =
                (SupportMapFragment) manager.findFragmentById(R.id.map);
            fragment.getMapAsync(this);
        }
    }
```

Description

- The Run Tracker app presented in *Murach's Android Programming* uses the deprecated getMap method to get a map. To fix this, you can use the getMapAsync method instead. For this to work, you must implement the OnMapReadyCallback interface as shown above.

Figure 24 How to fix the Run Tracker app (part 2 of 2)

Perspective

In this chapter, you learned how to use Android Studio 2.3 or later with the constraint layout to create a user interface for the Tip Calculator app that's presented in chapters 1 through 3 of *Murach's Java Programming*. In addition, you learned how to fix the Run Tracker app that's presented in chapter 18 so it doesn't crash or use the deprecated getMap method. With those skills, you should be able to use *Murach's Java Programming* with the latest versions of Android Studio and the Android SDKs.

Summary

- Android Studio 2.3 and later includes an *Instant Run feature* that allows you to apply code changes to an app while it's running without having to recompile and redeploy the app to the device. This can significantly speed the time it takes to test code changes.
- Many of the apps presented in *Murach's Android Programming* use the relative layout, which was the default layout prior to Android Studio 2.2. However, it's now considered a better practice to use the *constraint layout*, which is integrated into Android 2.3 and later.
- This document shows how to use Android Studio 2.3 with the constraint layout, including how to convert a relative layout to a constraint layout.
- The Run Tracker app that's presented in *Murach's Android Programming* crashes when you attempt to run it on Android 6.0 (API 23) or later. To fix this issue, you need to request location permissions when the app starts.
- The download for *Murach's Android Programming* now includes a Tip Calculator app that uses the constraint layout as well as a Run Tracker app that's updated to work with Android 6.0 (API 23) and later.