# 9

# Creating User Interfaces with Layouts and Widgets

In this chapter, you will learn more about layouts and widgets while adding some style to your list items in the **`RecyclerView`**. You will also learn about a new tool called **`ConstraintLayout`**. Figure 9.1 shows what **`CrimeListFragment`**'s view will look like once you chisel down your existing app to build up your masterpiece.

Figure 9.1 CriminalIntent, now with beautiful pictures


Before you dive in to **`ConstraintLayout`**, you must do a little legwork. You will need a copy of that fancy handcuff image from Figure 9.1 in your project. Navigate to the solutions file and open the `09_LayoutsAndWidgets/CriminalIntent/app/src/main/res` directory. Copy each density version of `ic_solved.png` into the appropriate drawable folder in your project. For information on how to access the solutions files, refer back to the section called *Adding an Icon* in Chapter 2.

## Using the Graphical Layout Tool

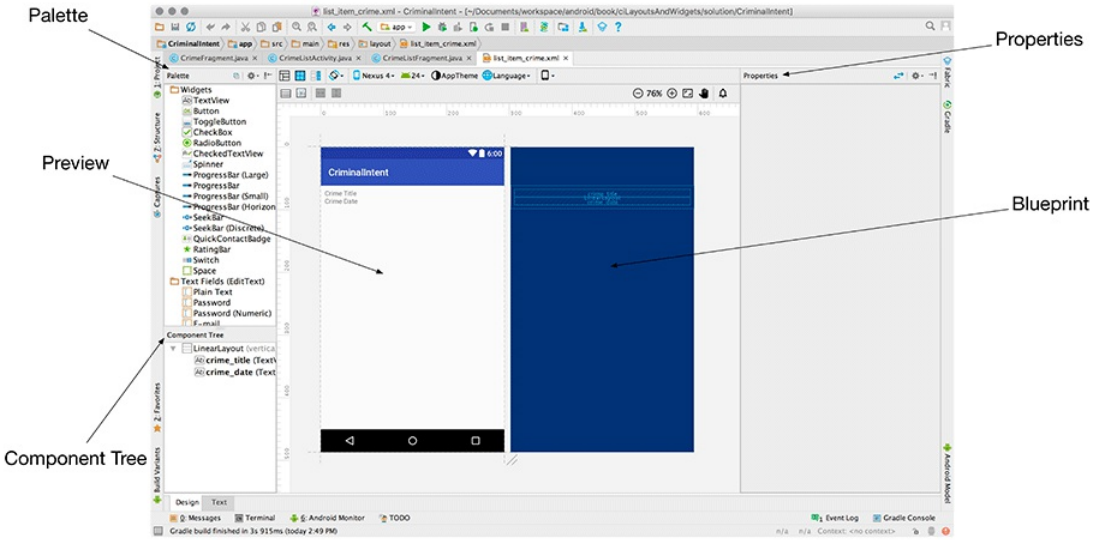
So far, you have created layouts by typing XML. In this section, you will use Android Studio's graphical layout tool.

Open `list_item_crime.xml` and select the Design tab at the bottom of the file.

In the middle of the graphical layout tool is the preview you have already seen. Just to the right of the preview is the *blueprint*. The blueprint view is like the preview but shows an outline of each of your views. This can be useful when you need to see how big each view is, not just what it is displaying.

On the lefthand side of the screen is the *palette*. This view contains all the widgets you could wish for, organized by category (Figure 9.2).

Figure 9.2 Views in the graphical layout tool




The *component tree* is in the bottom left. The tree shows how the widgets are organized in the layout.

On the right side of the screen is the *properties view*. In this view, you can view and edit the attributes of the widget selected in the component tree.

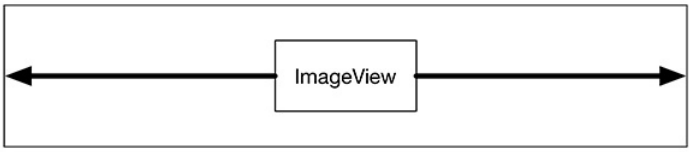
## Introducing ConstraintLayout

With **`ConstraintLayout`**, instead of using nested layouts you add a series of *constraints* to your layout. A constraint is like a rubber band. It pulls two things toward each other. So, for example, you can attach a constraint from the right edge of an **`ImageView`** to the right edge of its parent (the **`ConstraintLayout`** itself), as shown in Figure 9.3. The constraint will hold the **`ImageView`** to the right.

Figure 9.3 **`ImageView`** with a constraint on the right edge


You can create a constraint from all four edges of your **`ImageView`** (left, top, right, and bottom). If you have opposing constraints, they will equal out and your **`ImageView`** will be right in the center of the two constraints (Figure 9.4).

Figure 9.4 **`ImageView`** with opposing constraints




So that is the big picture: To place views where you want them to go in a **`ConstraintLayout`**, you give them constraints instead of dragging them around the screen.

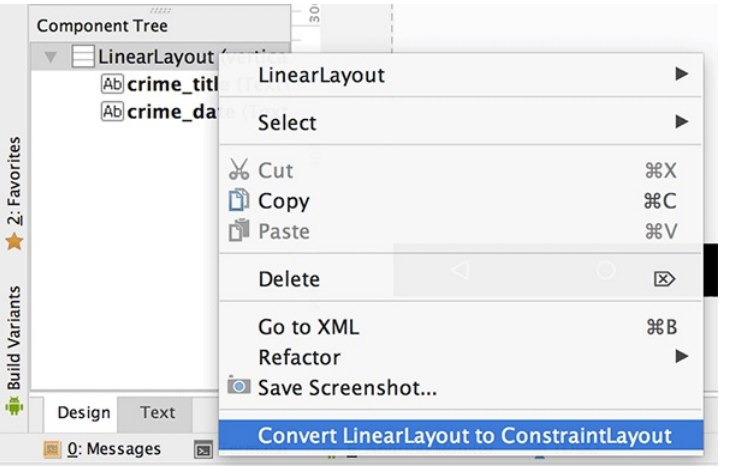
What about sizing widgets? For that, you have three options: Let the widget decide (your old friend `wrap_content`), decide for yourself, or let your widget expand to fit your constraints.

With all those tools, you can achieve a great many layouts with a single **`ConstraintLayout`**, no nesting required. As you go through this chapter, you will see how to use constraints with `list_item_crime`.

## Using ConstraintLayout

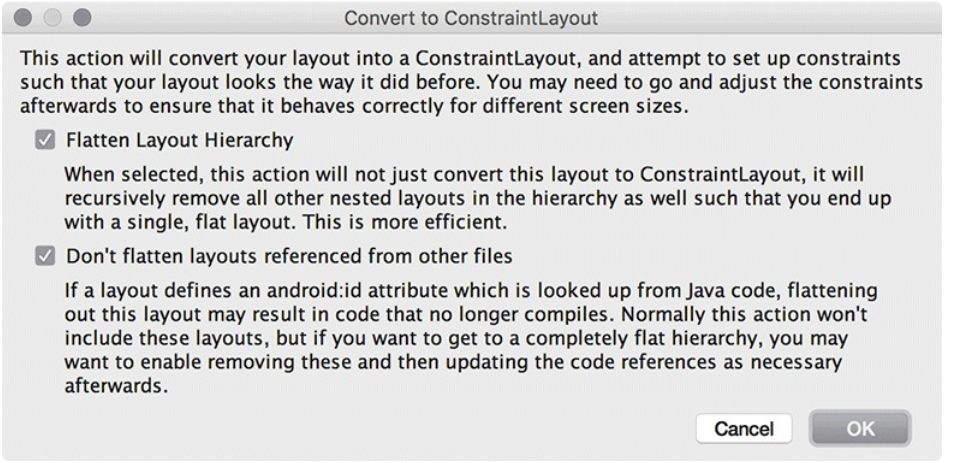
Now, convert `list_item_crime.xml` to use a **`ConstraintLayout`**. Right-click on your root **`LinearLayout`** in the component tree and select Convert LinearLayout to ConstraintLayout (Figure 9.5).

Figure 9.5 Converting the root view to a **`ConstraintLayout`**



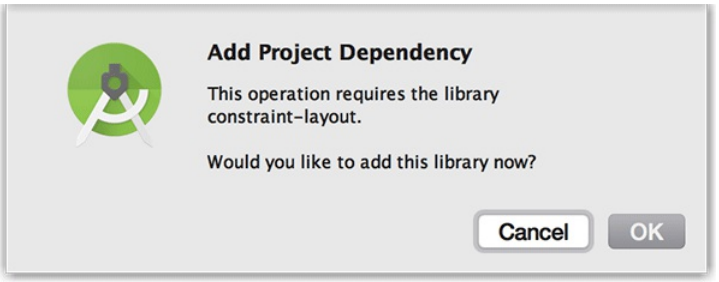

Android Studio will ask you in a pop-up how aggressive you would like this conversion process to be (Figure 9.6). Since `list_item_crime` is a simple layout file, there is not much that Android Studio can optimize. Leave the default values checked and select OK.

Figure 9.6 Converting with the default configuration



Finally, you will be asked to add the constraint layout dependency to your project (Figure 9.7). **`ConstraintLayout`** lives in a library, like **`RecyclerView`**. To use the tool, you must add a dependency to your Gradle file. Or, you can select OK on this dialog and Android Studio will do it for you.

Figure 9.7 Adding the **`ConstraintLayout`** dependency




If you peep your `app/build.gradle` file, you will see that the dependency has been added:

Listing 9.1 **`ConstraintLayout`** project dependency (`app/build.gradle`)

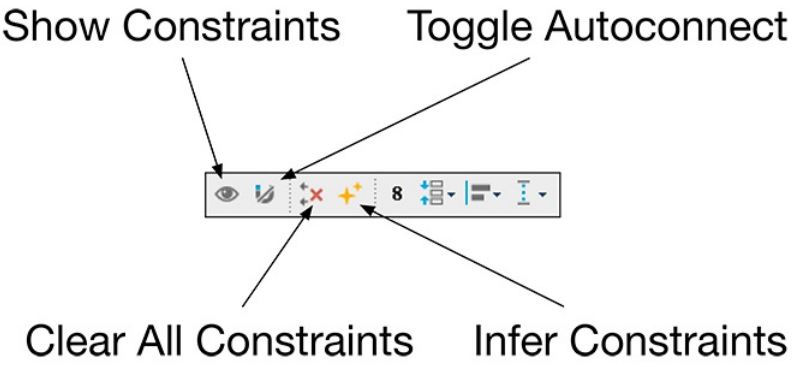
```
dependencies {
    ...
    compile 'com.android.support.constraint:constraint-layout:1.0.0-beta4'
}
```

Your **`LinearLayout`** has now been converted to a **`ConstraintLayout`**.

## The graphical editor

Look to the toolbar near the top of the preview and you will find a few editing controls (Figure 9.8).

Figure 9.8 Constraint controls

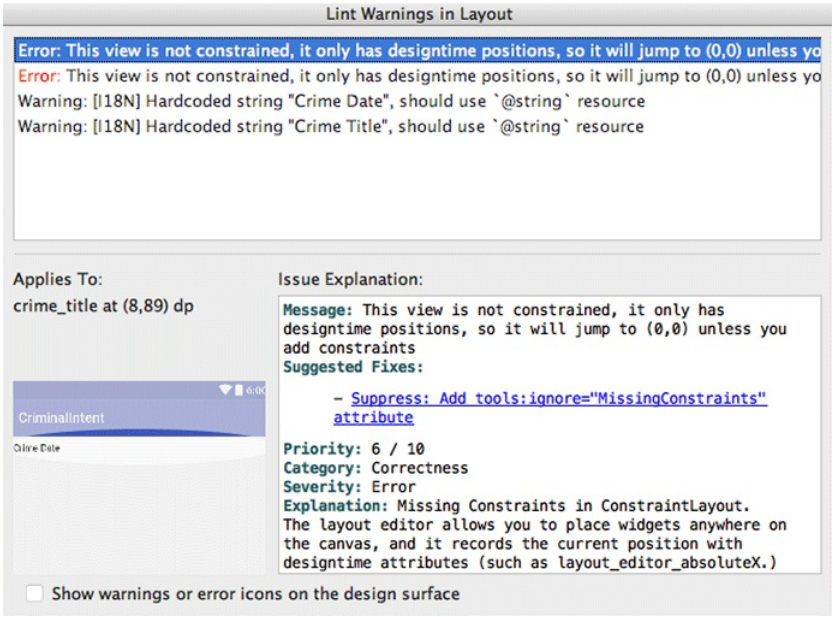



| | |
|---|---|
| Show Constraints | Show Constraints will reveal the constraints that are set up in the preview and blueprint views. You will find this option helpful at times and unhelpful at others. If you have many constraints, this button will trigger an overwhelming amount of information. |
| Toggle Autoconnect | When autoconnect is enabled, constraints will be automatically configured as you drag views into the preview. Android Studio will guess the constraints that you want a view to have and make those connections on demand. |
| Clear All Constraints | Clear All Constraints will remove all existing constraints in this layout file. You will use this option soon. |
| Infer Constraints | This option is similar to autoconnect in that Android Studio will automatically create constraints for you, but it is only triggered when you select this option. Autoconnect is active anytime you add a view to your layout file. |

When you converted `list_item_crime` to use **`ConstraintLayout`**, Android Studio automatically added the constraints it thinks will replicate the behavior of your old layout. However, to learn how constraints work you are going to start from scratch.

Select the ConstraintLayout view in the component tree, then choose the Clear All Constraints option from Figure 9.8. You will immediately see a red warning flag with the number 4 at the top right of the screen. Click on it to see what that is all about (Figure 9.9).

Figure 9.9 ConstraintLayout warnings




When views do not have enough constraints, **`ConstraintLayout`** cannot know exactly where to put them. Your **`TextView`**s have no constraints at all, so they each have a warning that says they will not appear in the right place at runtime.

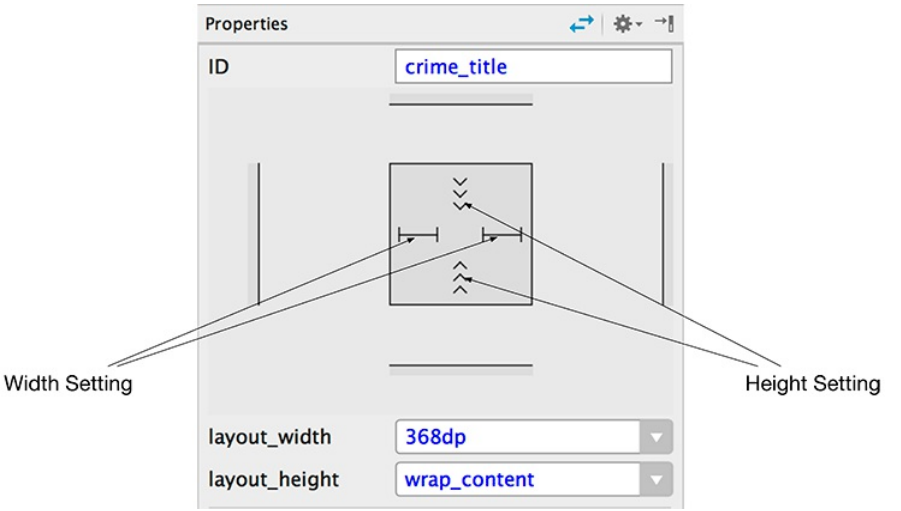
As you go through the chapter, you will add those constraints back to fix those warnings. In your own work, keep an eye on that warning indicator to avoid unexpected behavior at runtime.

## Making room

You need to make some room. Your two **`TextView`**s are taking up the entire area, which will make it hard to wire up anything else. Time to shrink those two widgets.

Select crime_title in the component tree and look at the properties view on the right (Figure 9.10).

Figure 9.10 Title **`TextView`**'s properties




The vertical and horizontal sizes of your **`TextView`** are governed by the height setting and width setting, respectively. These can be set to one of three view size settings (Figure 9.11), each of which corresponds to a value for `layout_width` or `layout_height`.

Figure 9.11 Three view size settings

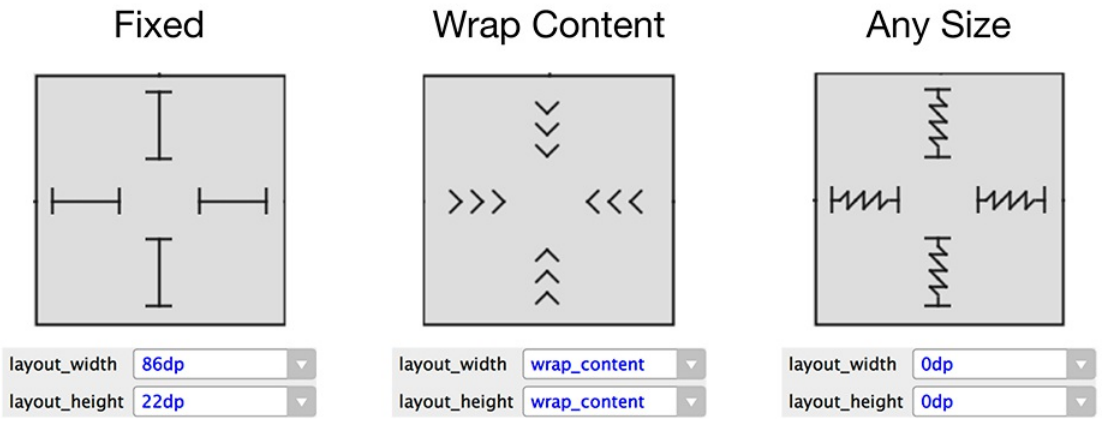
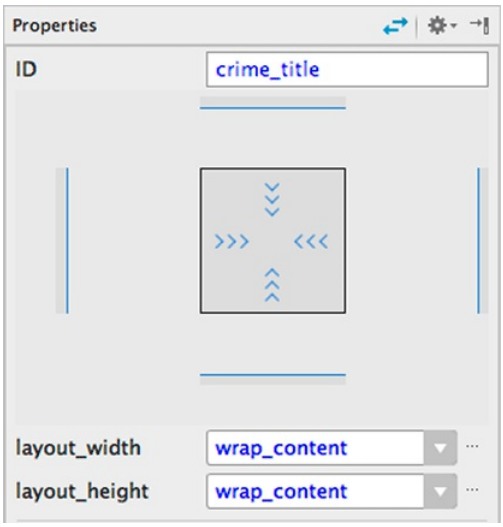


Table 9.1 View size setting types

| Setting type | Setting value | Usage |
|---|---|---|
| fixed | `Xdp` | Specifies an explicit size (that will not change) for the view. The size is specified in `dp` units. (More on `dp` units later in this chapter.) |
| wrap content | `wrap_content` | Assigns the view its "desired" size. For a **`TextView`**, this means that the size will be just big enough to show its contents. |
| any size | `0dp` | Allows the view to stretch to meet the specified constraints. |

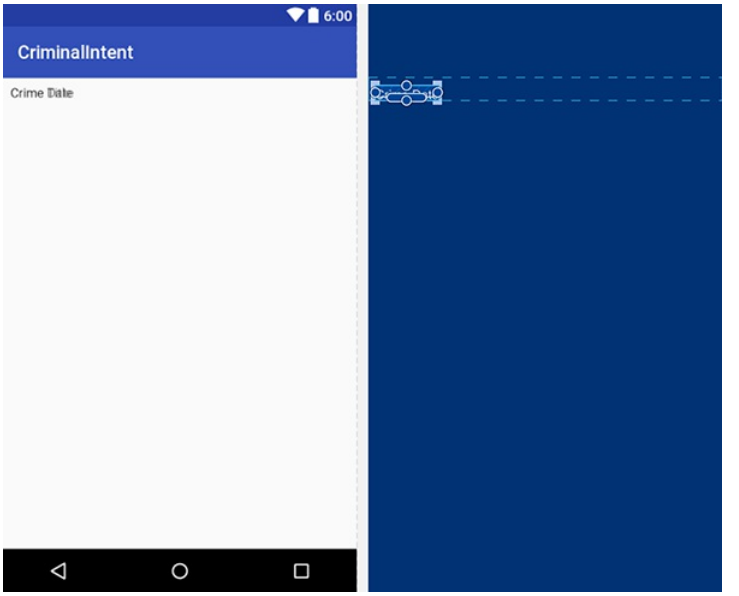
Both `crime_title` and `crime_date` are set to a large fixed width, which is why they are taking up the whole screen. Adjust the width and height of both of these widgets. With crime_title still selected in the component tree, click the width setting until it cycles around to the wrap content setting. If necessary, adjust the height setting until the height is also set to wrap content (Figure 9.12).

Figure 9.12 Adjusting the title width and height



Repeat the process with the crime_date widget to set its width and height. Now, the two widgets overlap but are much smaller (Figure 9.13).

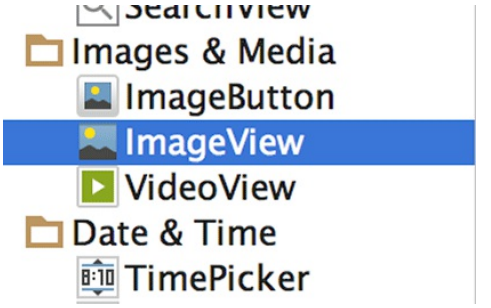
Figure 9.13 Overlapping **`TextView`**s




## Adding widgets

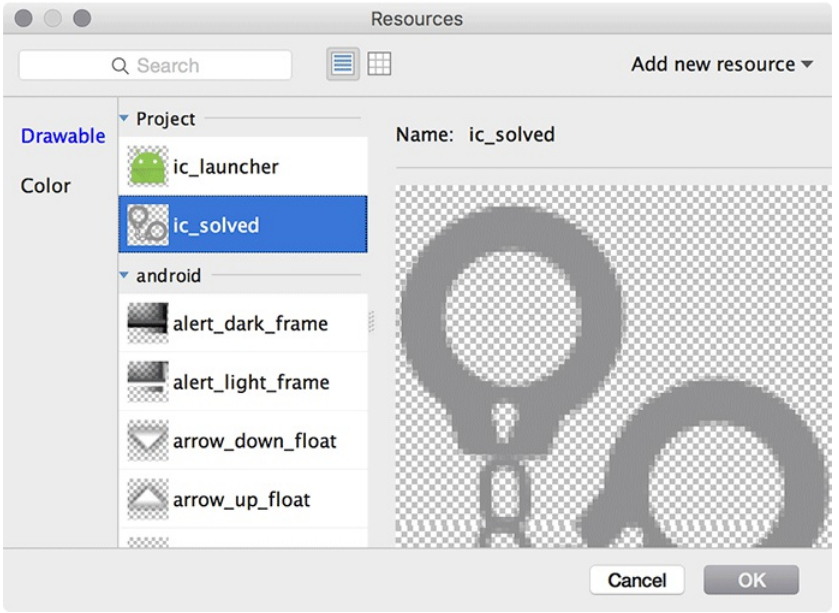
With your other widgets out of the way, you can add the handcuffs image to your layout. Add an **`ImageView`** to your layout file. In the palette, find **`ImageView`** (Figure 9.14). Drag it into your component tree as a child of **`ConstraintLayout`**, just underneath `crime_date`.

Figure 9.14 Finding the **`ImageView`**



In the pop-up, choose `ic_solved` as the resource for the **`ImageView`** (Figure 9.15). This image will be used to indicate which crimes have been solved.

Figure 9.15 Choosing the **`ImageView`**'s resource




The **`ImageView`** is now a part of your layout, but it has no constraints. So while the graphical editor gives it a position, that position does not really mean anything.

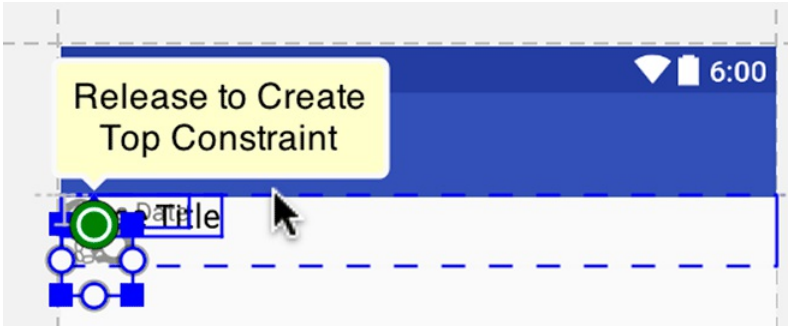
Time to add some constraints. Click on your **`ImageView`** in the preview or in the component tree. You will see dots on each side of the **`ImageView`** (Figure 9.16). Each of these dots represents a *constraint handle*.

Figure 9.16 **`ImageView`**'s constraint handles

You want the **ImageView** to be anchored in the right side of the view. To accomplish this, you need to create constraints from the top, right, and bottom edges of the **ImageView**.

First, you are going to set a constraint between the top of the **ImageView** and the top of the **ConstraintLayout**. The top of the **ConstraintLayout** is a little difficult to see, but it is just under the blue CriminalIntent toolbar. In the preview, drag the top constraint handle from the **ImageView** to the top of the **ConstraintLayout** – you will need to drag to the right somewhat, because the image is at the top of the constraint layout. Watch for the constraint handle to turn green and a pop-up reading Release to Create Top Constraint to appear (Figure 9.17), and release the mouse.

Figure 9.17 Creating a top constraint

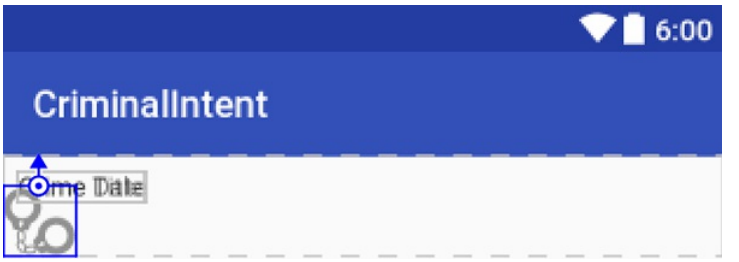



Be careful to avoid clicking when the mouse cursor is a corner shape – this will resize your **ImageView** instead. Also, make sure you do not inadvertently attach the constraint to one of your **TextView**s. If you do, click on the constraint handle to delete the bad constraint, then try again.

When you let go and set the constraint, the view will snap into position to account for the presence of the new constraint. This is how you move views around in a **ConstraintLayout** – by setting and removing constraints.

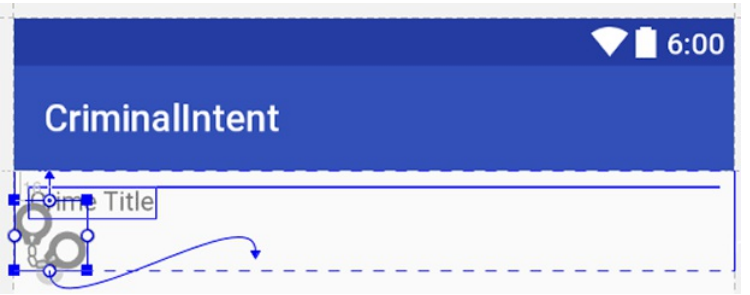
Verify that your **ImageView** has a top constraint connected to the top of the **ConstraintLayout** by hovering over the **ImageView** with your mouse. It should look like Figure 9.18.

Figure 9.18 **ImageView** with a top constraint



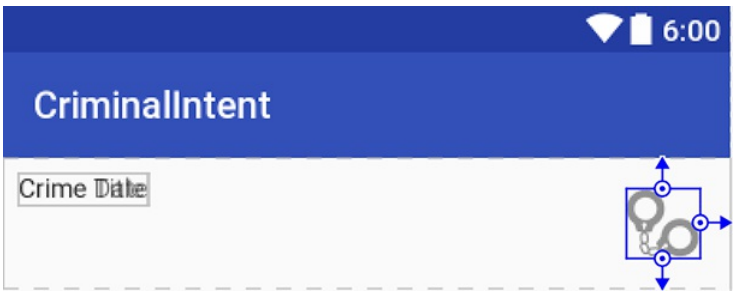
Do the same for the bottom constraint handle, dragging it from the **`ImageView`** to the bottom of the root view, also taking care to avoid attaching to the **`TextView`**s. Again, you will need to drag the connection toward the center of the root view and then slightly down, as shown in Figure 9.19.

Figure 9.19 **`ImageView`** connection in progress




Finally, drag the right constraint handle from the **`ImageView`** to the right side of the root view. That should set all of your constraints. Hovering over the **`ImageView`** will show all of them. Your constraints should look like Figure 9.20.

Figure 9.20 **`ImageView`**'s three constraints



## ConstraintLayout's inner workings

Any edits that you make with the graphical editor are reflected in the XML behind the scenes. You can still edit the raw **`ConstraintLayout`** XML, but the graphical editor will often be easier, because **`ConstraintLayout`** is much more verbose than other **`ViewGroup`**s.

Switch to the text view to see what happened to the XML when you created the three constraints on your **`ImageView`**.

### Listing 9.2 **`ImageView`**'s new XML constraints (`layout/list_item_crime.xml`)

```
<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="wrap_content">
    ...
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:srcCompat="@drawable/ic_solved"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        android:layout_marginBottom="16dp"
        android:layout_marginEnd="16dp"
        app:layout_constraintRight_toRightOf="parent"/>

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

Take a closer look at the top constraint:

```
app:layout_constraintTop_toTopOf="parent"
```

This attribute begins with `layout_`. All attributes that begin with `layout_` are known as *layout parameters*. Unlike other attributes, layout parameters are directions to that widget's *parent*, not the widget itself. They tell the parent layout how to arrange the child element within itself. You have seen a few layout parameters so far, like `layout_width` and `layout_height`.

The name of the constraint is `constraintTop`. This means that this is the top constraint on your **`ImageView`**.

Finally, the attribute ends with `toTopOf="parent"`. This means that this constraint is connected to the top edge of the parent. The parent here is the **`ConstraintLayout`**.

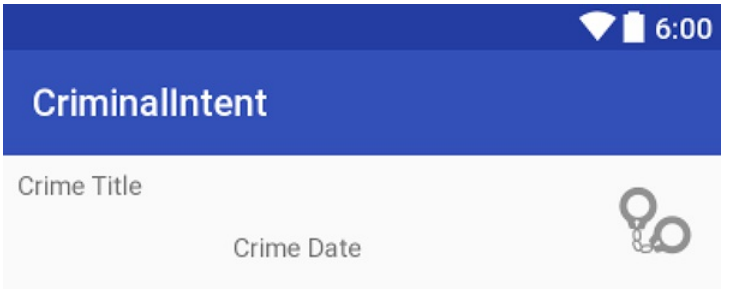
Whew, what a mouthful. Time to leave the raw XML behind and return to the graphical editor.

## Editing properties

Your **`ImageView`** is now positioned correctly. Next up: Position and size the title **`TextView`**.

First, select crime_date in the component tree and drag it out of the way (Figure 9.21). Remember that any changes you make to the position in the preview will not be represented when the app is running. At runtime, only constraints remain.

Figure 9.21 Get out of here, date




Now, select crime_title in the component tree. This will also highlight crime_title in the preview.

You want crime_title to be at the top left of your layout, positioned to the left of your new **`ImageView`**. That requires three constraints:

- from the left side of your view to the left side of the parent, with a `16dp` margin
- from the top of your view to the top of the parent, with a `16dp` margin
- from the right of your view to the left side of the new **`ImageView`**, with an `8dp` margin

Modify your layout so that all of these constraints are in place. (As of this writing, finding the right place to click can be tricky. Try to click inside of the **`TextView`**, and remember that you can always key Command+Z (Ctrl+Z) to undo and try again.)

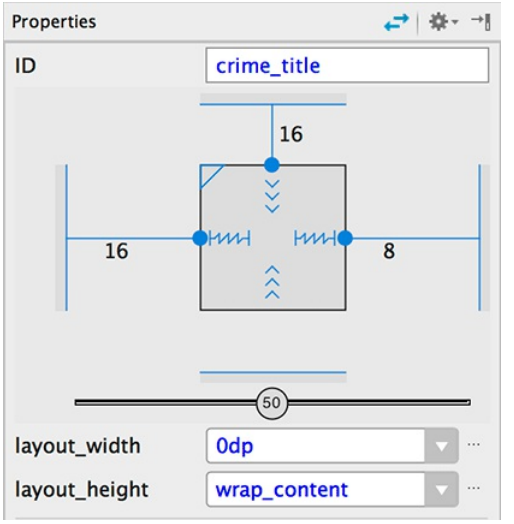
Verify that your constraints look like Figure 9.22. (The selected widget will show squiggly lines for any of its constraints that are stretching.)

Figure 9.22 Title **`TextView`**'s constraints


When you click on the **`TextView`**, you can see that it has an oval area that the **`ImageView`** did not have. **`TextView`**s have this additional constraint anchor that can be used to align text. You will not be using it in this chapter, but now you know what it is.

Now that the constraints are set up, you can restore the title **`TextView`** to its full glory. Adjust its horizontal view setting to any size (`0dp`) to allow the title **`TextView`** to fill all of the space available within its constraints. Adjust the vertical view size to `wrap_content`, if it is not already, so that the **`TextView`** will be just tall enough to show the title of the crime. Verify that your settings match those shown in Figure 9.23.

Figure 9.23 `crime_title` view settings

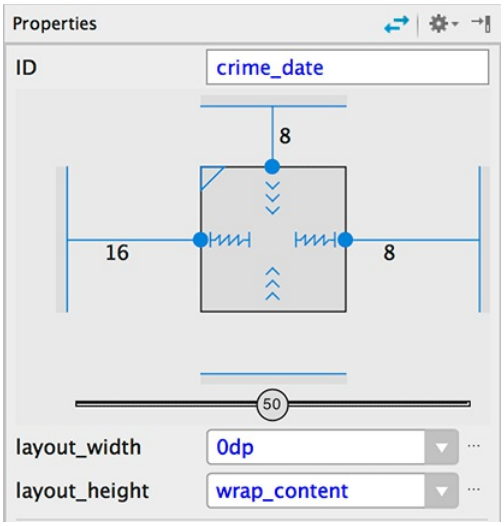



Now, add constraints to the date **`TextView`**. Select crime_date in the component tree. You are going to add three constraints:

- from the left side of your view to the left side of the parent, with a `16dp` margin
- from the top of your view to the bottom of the crime title, with an `8dp` margin
- from the right of your view to the left side of the new **`ImageView`**, with an `8dp` margin

After adding the constraints, adjust the properties of the **`TextView`**. You want the width of your date **`TextView`** to be Any Size and the height to be Wrap Content, just like the title **`TextView`**. Verify that your settings match those shown in Figure 9.24.

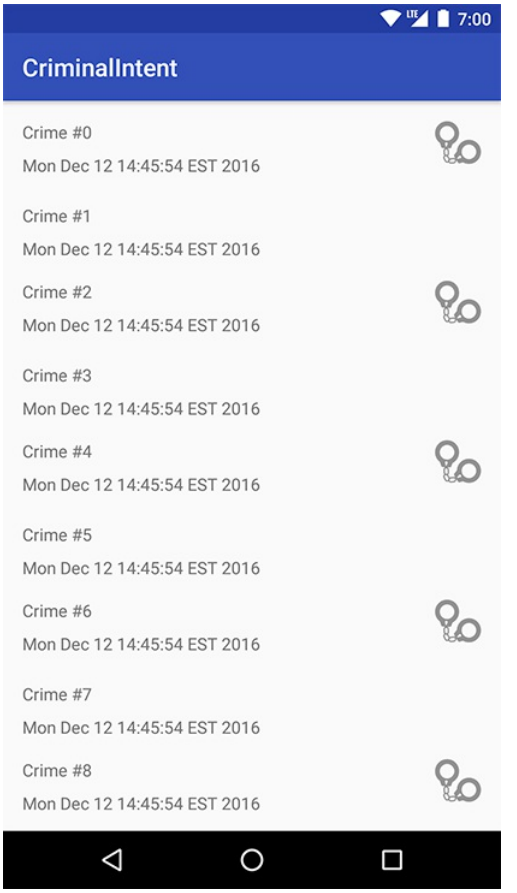
Figure 9.24 `crime_date` view settings




Your layout in the preview should look similar to Figure 9.1, at the beginning of the chapter.

Run CriminalIntent and verify that you see all three components lined up nicely in each row of your **`RecyclerView`** (Figure 9.25).

Figure 9.25 Now with three views per row

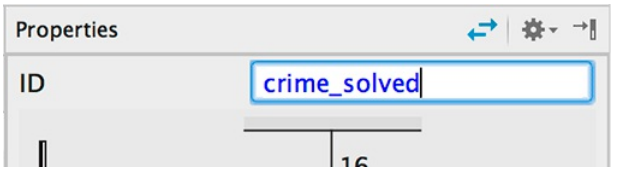


## Making list items dynamic

Now that the layout includes the right constraints, update the **ImageView** so that the handcuffs are only shown on crimes that have been solved.

First, update the ID of your **ImageView**. When you added the **ImageView** to your **ConstraintLayout**, it was given a default name. That name is not too descriptive. Select your **ImageView** in `list_item_crime.xml` and, in the properties view, update the ID attribute to `crime_solved` (Figure 9.26). You will be asked whether Android Studio should update all usages of the ID; select Yes.

Figure 9.26 Updating the image ID




With a proper ID in place, now you will update your code. Open `CrimeListFragment.java`. In **CrimeHolder**, add an **ImageView** instance variable and toggle its visibility based on the solved status of your crime.

Listing 9.3 Updating handcuff visibility (`CrimeListFragment.java`)

```
private class CrimeHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener {
    ...
    private TextView mDateTextView;
    private ImageView mSolvedImageView;

    public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
        super(inflater.inflate(R.layout.list_item_crime, parent, false));
        itemView.setOnClickListener(this);

        mTitleTextView = (TextView) itemView.findViewById(R.id.crime_title);
        mDateTextView = (TextView) itemView.findViewById(R.id.crime_date);
        mSolvedImageView = (ImageView) itemView.findViewById(R.id.crime_solved);
    }

    public void bind(Crime crime) {
        mCrime = crime;
        mTitleTextView.setText(mCrime.getTitle());
        mDateTextView.setText(mCrime.getDate().toString());
        mSolvedImageView.setVisibility(crime.isSolved() ? View.VISIBLE : View.GONE);
    }
    ...
}
```

Run CriminalIntent and verify that the handcuffs now appear on every other row.

# More on Layout Attributes

Let's add a few more tweaks to the design of `list_item_crime.xml` and, in the process, answer some lingering questions you might have about widgets and attributes.

Navigate back to the Design view of `list_item_crime.xml`. Select crime_title and adjust some of the attributes in the properties view.

Click the disclosure arrow next to textAppearance to reveal a set of text and font attributes. Update the textColor attribute to `@android:color/black` (Figure 9.27).

Figure 9.27 Updating the title color


Next, set the textSize attribute to `18sp`. Run CriminalIntent and be amazed at how much better everything looks with a fresh coat of paint.

## Screen pixel densities and dp and sp

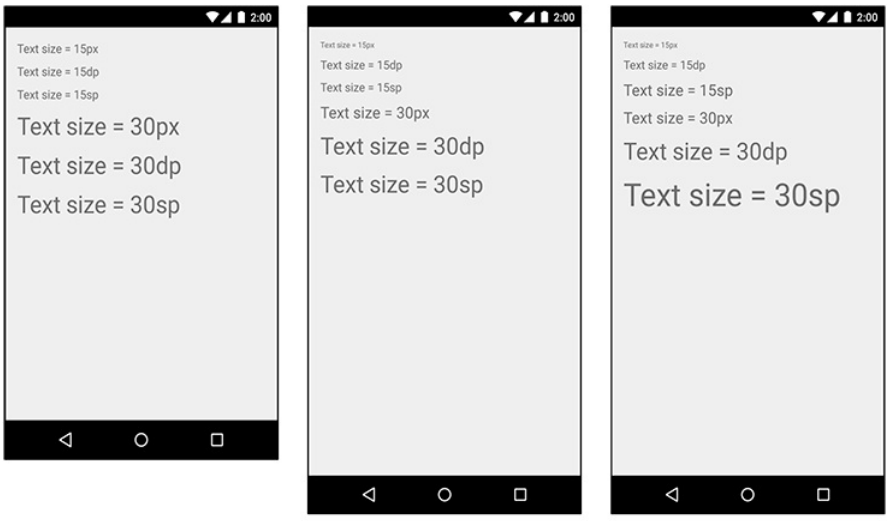
In `list_item_crime.xml`, you have specified attribute values in terms of `sp` and `dp` units. Now it is time to learn what they are.

Sometimes you need to specify values for view attributes in terms of specific sizes (usually in pixels, but sometimes points, millimeters, or inches). You see this most commonly with attributes for text size, margins, and padding. Text size is the pixel height of the text on the device's screen. Margins specify the distances between views, and padding specifies the distance between a view's outside edges and its content.

As you saw in the section called *Adding an Icon* in Chapter 2, Android automatically scales images to different screen pixel densities using density-qualified drawable folders (such as `drawable-xhdpi`). But what happens when your images scale, but your margins do not? Or when the user configures a larger-than-default text size?

To solve these problems, Android provides density-independent dimension units that you can use to get the same size on different screen densities. Android translates these units into pixels at runtime, so there is no tricky math for you to do (Figure 9.28).

Figure 9.28 Dimension units in action on **`TextView`**




`px` — Short for *pixel*. One pixel corresponds to one onscreen pixel, no matter what the display density is. Because pixels do not scale appropriately with device display density, their use is not recommended.

`dp` (or `dip`) — Short for *density-independent pixel* and usually pronounced "dip." You typically use this for margins, padding, or anything else for which you would otherwise specify size with a pixel value. One `dp` is always 1/160th of an inch on a device's screen. You get the same size regardless of screen density: When your display is a higher density, density-independent pixels will expand to fill a larger number of screen pixels.

`sp` — Short for *scale-independent pixel*. Scale-independent pixels are density-independent pixels that also take into account the user's font size preference. You will almost always use `sp` to set display text size.

`pt`, `mm`, `in` — These are scaled units, like `dp`, that allow you to specify interface sizes in points (1/72 of an inch), millimeters, or inches. However, we do not recommend using them: Not all devices are correctly configured for these units to scale correctly.

In practice and in this book, you will use `dp` and `sp` almost exclusively. Android will translate these values into pixels at runtime.

## Margins vs padding

In both GeoQuiz and CriminalIntent, you have given widgets margin and padding attributes. Beginning developers sometimes get confused about these two. Now that you understand what a layout parameter is, the difference is easier to explain.

Margin attributes are layout parameters. They determine the distance between widgets. Given that a widget can only know about itself, margins must be the responsibility of the widget's parent.

Padding, on the other hand, is not a layout parameter. The `android:padding` attribute tells the widget how much bigger than its contents it should draw itself. For example, say you wanted the date button to be spectacularly large without changing its text size (Figure 9.29).

Figure 9.29 I like big buttons and I cannot lie...


You could add the following attribute to the **`Button`**.

Listing 9.4 Padding in action (`fragment_crime.xml`)

```
<Button android:id="@+id/crime_date"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_marginLeft="16dp"
  android:layout_marginRight="16dp"
  android:padding="80dp" />
```

Alas, you should probably remove this attribute before continuing.

## Styles, themes, and theme attributes

A *style* is an XML resource that contains attributes that describe how a widget should look and behave. For example, the following is a style resource that configures a widget with a larger-than-normal text size:

```
<style name="BigTextStyle">
  <item name="android:textSize">20sp</item>
  <item name="android:padding">3dp</item>
</style>
```

You can create your own styles (and you will in Chapter 22). You add them to a styles file in `res/values/` and refer to them in layouts like this: `@style/my_own_style`.

Take another look at the **`TextView`** widgets in `fragment_crime.xml`; each has a `style` attribute that refers to a style created by Android. This particular style makes the **`TextView`**s look like list separators and comes from the app's *theme*. A theme is a collection of styles. Structurally, a theme is itself a style resource whose attributes point to other style resources.

Android provides platform themes that your apps can use. When you created CriminalIntent, the wizard set up a theme for the app that is referenced on the `application` tag in the manifest.

You can apply a style from the app's theme to a widget using a *theme attribute reference*. This is what you are doing in `fragment_crime.xml` when you use the value `?android:listSeparatorTextViewStyle`.

In a theme attribute reference, you tell Android's runtime resource manager, "Go to the app's theme and find the attribute named `listSeparatorTextViewStyle`. This attribute points to another style resource. Put the value of that resource here."

Every Android theme will include an attribute named `listSeparatorTextViewStyle`, but its definition will be different depending on the overall look and feel of the particular theme. Using a theme attribute reference ensures that the **`TextView`**s will have the correct look and feel for your app.

You will learn more about how styles and themes work in Chapter 22.

### Android's design guidelines

Notice that for your margins, Android Studio defaulted to either a `16dp` or a `8dp` value. This value follows Android's material design guidelines. You can find all of the Android design guidelines at `developer.android.com/design/index.html`.

Your Android apps should follow these guidelines as closely as possible. However, you should know that the guidelines rely heavily on newer Android SDK functionality that is not always available or easy to achieve on older devices. Many of the design recommendations can be followed using the AppCompat library, which you have seen and will read more about in Chapter 13.

## The Graphical Layout Tools and You

The graphical layout tools are useful, especially with **`ConstraintLayout`**. Not everyone is a fan, though. Many prefer the simplicity and clarity of working directly with XML, rather than relying on the IDE.

Do not feel that you have to choose sides. You can switch between the graphical editor and directly editing XML at any time. Feel free to use whichever tool you prefer to create the layouts in this book. From now on, we will show you a diagram rather than the XML when you need to create a layout. You can decide for yourself how to create it – XML, graphical editor, or some of each.

## Challenge: Formatting the Date

The **`Date`** object is more of a timestamp than a conventional date. A timestamp is what you see when you call **`toString()`** on a **`Date`**, so that is what you have on in each of your **`RecyclerView`** rows. While timestamps make for good documentation, it might be nicer if the rows just displayed the date as humans think of it – like "Jul 22, 2016." You can do this with an instance of the **`android.text.format.DateFormat`** class. The place to start is the reference page for this class in the Android documentation.

You can use methods in the **`DateFormat`** class to get a common format. Or you can prepare your own format string. For a more advanced challenge, create a format string that will display the day of the week as well – for example, "Friday, Jul 22, 2016."