# Canvas and Drawables

The Android framework APIs provides a set of 2D-drawing APIs that allow you to render your own custom graphics onto a canvas or to modify existing Views to customize their look and feel. When drawing 2D graphics, you'll typically do so in one of two ways:

a. Draw your graphics or animations into a View object from your layout. In this manner, the drawing of your graphics is handled by the system's normal View hierarchy drawing process — you simply define the graphics to go inside the View.
b. Draw your graphics directly to a Canvas. This way, you personally call the appropriate class's `onDraw()` method (passing it your Canvas), or one of the Canvas `draw...()` methods (like `drawPicture()`). In doing so, you are also in control of any animation.

**IN THIS DOCUMENT**

**SEE ALSO**

OpenGL with the Framework APIs
RenderScript

Option "a," drawing to a View, is your best choice when you want to draw simple graphics that do not need to change dynamically and are not part of a performance-intensive game. For example, you should draw your graphics into a View when you want to display a static graphic or predefined animation, within an otherwise static application. Read Drawables (#drawables) for more information.

Option "b," drawing to a Canvas, is better when your application needs to regularly re-draw itself. Applications such as video games should be drawing to the Canvas on its own. However, there's more than one way to do this:

- In the same thread as your UI Activity, wherein you create a custom View component in your layout, call `invalidate()` and then handle the `onDraw()` callback.
- Or, in a separate thread, wherein you manage a `SurfaceView` and perform draws to the Canvas as fast as your thread is capable (you do not need to request `invalidate()`).

## Draw with a Canvas

When you're writing an application in which you would like to perform specialized drawing and/or control the animation of graphics, you should do so by drawing through a `Canvas (/reference/android/graphics/Canvas.html)`. A Canvas works for you as a pretense, or interface, to the actual surface upon which your graphics will be drawn — it holds all of your "draw" calls. Via the Canvas, your drawing is actually performed upon an underlying `Bitmap (/reference/android/graphics/Bitmap.html)`, which is placed into the window.

In the event that you're drawing within the `onDraw() (/reference/android/view/View.html#onDraw(android.graphics.Canvas))` callback method, the Canvas is provided for you and you need only place your drawing calls upon it. You can also acquire a Canvas from `SurfaceHolder.lockCanvas() (/reference/android/view/SurfaceHolder.html#lockCanvas())`, when dealing with a SurfaceView object. (Both of these scenarios are discussed in the following sections.) However, if you need to create a new Canvas, then you must define the `Bitmap (/reference/android/graphics/Bitmap.html)` upon which drawing will actually be performed. The Bitmap is always required for a Canvas. You can set up a new Canvas like this:

```
Bitmap b = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);
Canvas c = new Canvas(b);
```

Now your Canvas will draw onto the defined Bitmap. After drawing upon it with the Canvas, you can then carry your Bitmap to another Canvas with one of the `Canvas.drawBitmap(Bitmap,...) (/reference/android/graphics/Canvas.html#drawBitmap(android.graphics.Bitmap, android.graphics.Matrix, android.graphics.Paint))` methods. It's recommended that you ultimately draw your final graphics through a

Canvas offered to you by `View.onDraw()` (/reference/android/view/View.html#onDraw(android.graphics.Canvas)) or `SurfaceHolder.lockCanvas()` (/reference/android/view/SurfaceHolder.html#lockCanvas()) (see the following sections).

The `Canvas` (/reference/android/graphics/Canvas.html) class has its own set of drawing methods that you can use, like `drawBitmap(...)`, `drawRect(...)`, `drawText(...)`, and many more. Other classes that you might use also have `draw()` methods. For example, you'll probably have some `Drawable` (/reference/android/graphics/drawable/Drawable.html) objects that you want to put on the Canvas. Drawable has its own `draw()` (/reference/android/graphics/drawable/Drawable.html#draw(android.graphics.Canvas)) method that takes your Canvas as an argument.

## On a View

If your application does not require a significant amount of processing or frame-rate speed (perhaps for a chess game, a snake game, or another slowly-animated application), then you should consider creating a custom View component and drawing with a Canvas in `View.onDraw()` (/reference/android/view/View.html#onDraw(android.graphics.Canvas)). The most convenient aspect of doing so is that the Android framework will provide you with a pre-defined Canvas to which you will place your drawing calls.

To start, extend the `View` (/reference/android/view/View.html) class (or descendant thereof) and define the `onDraw()` (/reference/android/view/View.html#onDraw(android.graphics.Canvas)) callback method. This method will be called by the Android framework to request that your View draw itself. This is where you will perform all your calls to draw through the `Canvas` (/reference/android/graphics/Canvas.html), which is passed to you through the `onDraw()` callback.

The Android framework will only call `onDraw()` as necessary. Each time that your application is prepared to be drawn, you must request your View be invalidated by calling `invalidate()` (/reference/android/view/View.html#invalidate()). This indicates that you'd like your View to be drawn and Android will then call your `onDraw()` method (though is not guaranteed that the callback will be instantaneous).

Inside your View component's `onDraw()`, use the Canvas given to you for all your drawing, using various `Canvas.draw...()` methods, or other class `draw()` methods that take your Canvas as an argument. Once your `onDraw()` is complete, the Android framework will use your Canvas to draw a Bitmap handled by the system.

> **Note:** In order to request an invalidate from a thread other than your main Activity's thread, you must call `postInvalidate()` (/reference/android/view/View.html#postInvalidate()).

For information about extending the `View` (/reference/android/view/View.html) class, read Building Custom Components (/guide/topics/ui/custom-components.html).

For a sample application, see the Snake game, in the SDK samples folder: `<your-sdk-directory>/samples/Snake/`.

## On a SurfaceView

The `SurfaceView` (/reference/android/view/SurfaceView.html) is a special subclass of View that offers a dedicated drawing surface within the View hierarchy. The aim is to offer this drawing surface to an application's secondary thread, so that the application isn't required to wait until the system's View hierarchy is ready to draw. Instead, a secondary thread that has reference to a SurfaceView can draw to its own Canvas at its own pace.

To begin, you need to create a new class that extends `SurfaceView` (/reference/android/view/SurfaceView.html). The class should also implement `SurfaceHolder.Callback` (/reference/android/view/SurfaceHolder.Callback.html). This subclass is an interface that will notify you with information about the underlying `Surface` (/reference/android/view/Surface.html), such as when it is created, changed, or destroyed. These events are important so that you know when you can start drawing, whether you need to make adjustments based on new surface properties, and when to stop drawing and potentially kill some tasks. Inside your SurfaceView class is also a good place to define your secondary Thread class, which will perform all the drawing procedures to your Canvas.

Instead of handling the Surface object directly, you should handle it via a `SurfaceHolder` (/reference/android/view/SurfaceHolder.html). So, when your SurfaceView is initialized, get the SurfaceHolder by calling `getHolder()` (/reference/android/view/SurfaceView.html#getHolder()). You should then notify the SurfaceHolder that you'd like to receive SurfaceHolder callbacks (from `SurfaceHolder.Callback` (/reference/android/view/SurfaceHolder.Callback.html)) by calling `addCallback()` (/reference/android/view/SurfaceHolder.html#addCallback(android.view.SurfaceHolder.Callback)) (pass it *this*). Then override each of the `SurfaceHolder.Callback` (/reference/android/view/SurfaceHolder.Callback.html) methods inside your SurfaceView class.

In order to draw to the Surface Canvas from within your second thread, you must pass the thread your SurfaceHandler and retrieve the Canvas with `lockCanvas()` (/reference/android/view/SurfaceHolder.html#lockCanvas()). You can now take the Canvas given to you by the SurfaceHolder and do your necessary drawing upon it. Once you're done drawing with the Canvas, call `unlockCanvasAndPost()` (/reference/android/view/SurfaceHolder.html#unlockCanvasAndPost(android.graphics.Canvas)), passing it your Canvas object. The Surface will now draw the Canvas as you left it. Perform this sequence of locking and unlocking the canvas each time you want to redraw.

> **Note:** On each pass you retrieve the Canvas from the SurfaceHolder, the previous state of the Canvas will be retained. In order to properly animate your graphics, you must re-paint the entire surface. For example, you can clear the previous state of the Canvas by filling in a color with `drawColor()` (/reference/android/graphics/Canvas.html#drawColor(int)) or setting a background image with `drawBitmap()` (/reference/android/graphics/Canvas.html#drawBitmap(android.graphics.Bitmap, android.graphics.Rect, android.graphics.RectF, android.graphics.Paint)). Otherwise, you will see traces of the drawings you previously performed.

For a sample application, see the Lunar Lander game, in the SDK samples folder: `<your-sdk-directory>/samples/LunarLander/`. Or, browse the source in the Sample Code (/guide/samples/index.html) section.

## Drawables

Android offers a custom 2D graphics library for drawing shapes and images. The `android.graphics.drawable` (/reference/android/graphics/drawable/package-summary.html) package is where you'll find the common classes used for drawing in two-dimensions.

This document discusses the basics of using Drawable objects to draw graphics and how to use a couple subclasses of the Drawable class. For information on using Drawables to do frame-by-frame animation, see Drawable Animation (/guide/topics/graphics/drawable-animation.html).

A `Drawable` (/reference/android/graphics/drawable/Drawable.html) is a general abstraction for "something that can be drawn." You'll discover that the Drawable class extends to define a variety of specific kinds of drawable graphics, including `BitmapDrawable` (/reference/android/graphics/drawable/BitmapDrawable.html), `ShapeDrawable` (/reference/android/graphics/drawable/ShapeDrawable.html), `PictureDrawable` (/reference/android/graphics/drawable/PictureDrawable.html), `LayerDrawable` (/reference/android/graphics/drawable/LayerDrawable.html), and several more. Of course, you can also extend these to define your own custom Drawable objects that behave in unique ways.

There are three ways to define and instantiate a Drawable: using an image saved in your project resources; using an XML file that defines the Drawable properties; or using the normal class constructors. Below, we'll discuss each the first two techniques (using constructors is nothing new for an experienced developer).

### Creating from resource images

A simple way to add graphics to your application is by referencing an image file from your project resources. Supported file types are PNG (preferred), JPG (acceptable) and GIF (discouraged). This technique would obviously be preferred for application icons, logos, or other graphics such as those used in a game.

To use an image resource, just add your file to the `res/drawable/` directory of your project. From there, you can reference it from your code or your XML layout. Either way, it is referred using a resource ID, which is the file name without the file type extension (E.g., `my_image.png` is referenced as *my_image*).

> **Note:** Image resources placed in `res/drawable/` may be automatically optimized with lossless image

compression by the `aapt` tool during the build process. For example, a true-color PNG that does not require more than 256 colors may be converted to an 8-bit PNG with a color palette. This will result in an image of equal quality but which requires less memory. So be aware that the image binaries placed in this directory can change during the build. If you plan on reading an image as a bit stream in order to convert it to a bitmap, put your images in the `res/raw/` folder instead, where they will not be optimized.

#### Example code

The following code snippet demonstrates how to build an `ImageView (/reference/android/widget/ImageView.html)` that uses an image from drawable resources and add it to the layout.

```
LinearLayout mLinearLayout;

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Create a LinearLayout in which to add the ImageView
  mLinearLayout = new LinearLayout(this);

  // Instantiate an ImageView and define its properties
  ImageView i = new ImageView(this);
  i.setImageResource(R.drawable.my_image);
  i.setAdjustViewBounds(true); // set the ImageView bounds to match the Drawable's
  i.setLayoutParams(new Gallery.LayoutParams(LayoutParams.WRAP_CONTENT,
      LayoutParams.WRAP_CONTENT));

  // Add the ImageView to the layout and set the layout as the content view
  mLinearLayout.addView(i);
  setContentView(mLinearLayout);
}
```

In other cases, you may want to handle your image resource as a `Drawable (/reference/android/graphics/drawable/Drawable.html)` object. To do so, create a Drawable from the resource like so:

```
Resources res = mContext.getResources();
Drawable myImage = res.getDrawable(R.drawable.my_image);
```

**Note:** Each unique resource in your project can maintain only one state, no matter how many different objects you may instantiate for it. For example, if you instantiate two Drawable objects from the same image resource, then change a property (such as the alpha) for one of the Drawables, then it will also affect the other. So when dealing with multiple instances of an image resource, instead of directly transforming the Drawable, you should perform a tween animation (/guide/topics/graphics/view-animation.html#tween-animation).

#### Example XML

The XML snippet below shows how to add a resource Drawable to an `ImageView (/reference/android/widget/ImageView.html)` in the XML layout (with some red tint just for fun).

```
<ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:tint="#55ff0000"
        android:src="@drawable/my_image"/>
```

For more information on using project resources, read about Resources and Assets (/guide/topics/resources/index.html).

### Creating from resource XML

By now, you should be familiar with Android's principles of developing a User Interface (/guide/topics/ui/index.html). Hence, you understand the power and flexibility inherent in defining objects in XML. This philosophy caries over from Views to Drawables. If there is a Drawable object that you'd like to create, which is not initially dependent on variables defined by your application code or user interaction, then defining the Drawable in XML is a good option. Even if you expect your Drawable to change its properties during the user's experience with your application, you should consider defining the object in XML, as you can always modify properties once it is instantiated.

Once you've defined your Drawable in XML, save the file in the `res/drawable/` directory of your project. Then, retrieve and instantiate the object by calling `Resources.getDrawable() (/reference/android/content/res/Resources.html#getDrawable(int))`, passing it the resource ID of your XML file. (See the example below (#drawable-xml-example).)

Any Drawable subclass that supports the `inflate()` method can be defined in XML and instantiated by your application. Each Drawable that supports XML inflation utilizes specific XML attributes that help define the object properties (see the class reference to see what these are). See the class documentation for each Drawable subclass for information on how to define it in XML.

**Example**

Here's some XML that defines a TransitionDrawable:

```
<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/image_expand">
    <item android:drawable="@drawable/image_collapse">
</transition>
```

With this XML saved in the file `res/drawable/expand_collapse.xml`, the following code will instantiate the TransitionDrawable and set it as the content of an ImageView:

```
Resources res = mContext.getResources();
TransitionDrawable transition = (TransitionDrawable)
    res.getDrawable(R.drawable.expand_collapse);
ImageView image = (ImageView) findViewById(R.id.toggle_image);
image.setImageDrawable(transition);
```

Then this transition can be run forward (for 1 second) with:

```
transition.startTransition(1000);
```

Refer to the Drawable classes listed above for more information on the XML attributes supported by each.

## Shape Drawable

When you want to dynamically draw some two-dimensional graphics, a `ShapeDrawable (/reference/android/graphics/drawable/ShapeDrawable.html)` object will probably suit your needs. With a ShapeDrawable, you can programmatically draw primitive shapes and style them in any way imaginable.

A ShapeDrawable is an extension of `Drawable (/reference/android/graphics/drawable/Drawable.html)`, so you can use one wherever a Drawable is expected — perhaps for the background of a View, set with `setBackgroundDrawable() (/reference/android/view/View.html#setBackgroundDrawable(android.graphics.drawable.Drawable))`. Of course, you can also draw your shape as its own custom `View (/reference/android/view/View.html)`, to be added to your layout however you please. Because the ShapeDrawable has its own `draw()` method, you can create a subclass of View that draws the ShapeDrawable during the `View.onDraw()` method. Here's a basic extension of the View class that does just this, to draw a ShapeDrawable as a View:

```
public class CustomDrawableView extends View {
  private ShapeDrawable mDrawable;

  public CustomDrawableView(Context context) {
    super(context);

    int x = 10;
    int y = 10;
    int width = 300;
    int height = 50;

    mDrawable = new ShapeDrawable(new OvalShape());
    mDrawable.getPaint().setColor(0xff74AC23);
    mDrawable.setBounds(x, y, x + width, y + height);
  }

  protected void onDraw(Canvas canvas) {
    mDrawable.draw(canvas);
  }
}
```

In the constructor, a ShapeDrawable is defines as an `OvalShape` (/reference/android/graphics/drawable/shapes/OvalShape.html). It's then given a color and the bounds of the shape are set. If you do not set the bounds, then the shape will not be drawn, whereas if you don't set the color, it will default to black.

With the custom View defined, it can be drawn any way you like. With the sample above, we can draw the shape programmatically in an Activity:

```
CustomDrawableView mCustomDrawableView;

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  mCustomDrawableView = new CustomDrawableView(this);

  setContentView(mCustomDrawableView);
}
```

If you'd like to draw this custom drawable from the XML layout instead of from the Activity, then the CustomDrawable class must override the `View(Context, AttributeSet)` (/reference/android/view/View.html#View(android.content.Context, android.util.AttributeSet)) constructor, which is called when instantiating a View via inflation from XML. Then add a CustomDrawable element to the XML, like so:

```
<com.example.shapedrawable.CustomDrawableView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        />
```

The ShapeDrawable class (like many other Drawable types in the `android.graphics.drawable` (/reference/android/graphics/drawable/package-summary.html) package) allows you to define various properties of the drawable with public methods. Some properties you might want to adjust include alpha transparency, color filter, dither, opacity and color.

You can also define primitive drawable shapes using XML. For more information, see the section about Shape Drawables in the Drawable Resources (/guide/topics/resources/drawable-resource.html#Shape) document.

## Nine-patch

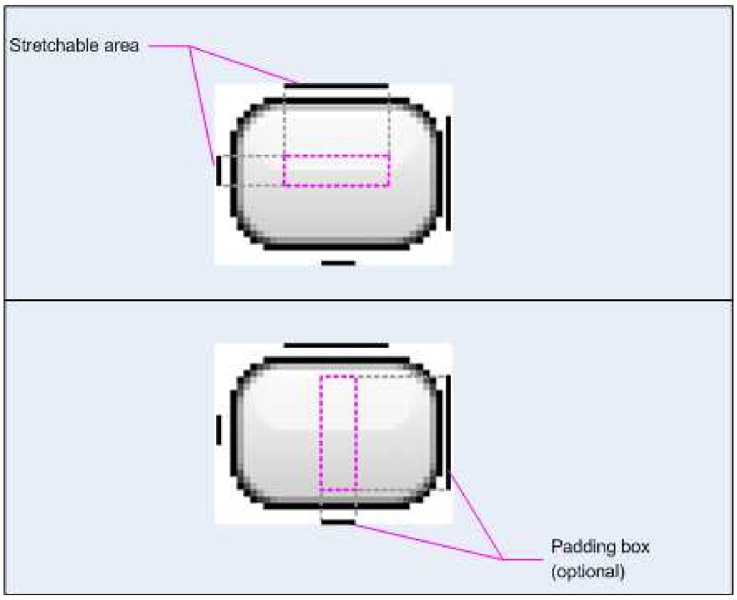
A `NinePatchDrawable` (/reference/android/graphics/drawable/NinePatchDrawable.html) graphic is a stretchable bitmap image, which Android will automatically resize to accommodate the contents of the View in which you have placed it as the background. An example use of a NinePatch is the backgrounds used by standard Android buttons — buttons must stretch to accommodate strings of various lengths. A NinePatch drawable is a standard PNG image that includes an extra 1-pixel-wide border. It must be saved with the extension `.9.png`, and saved into the `res/drawable/` directory of your project.

The border is used to define the stretchable and static areas of the image. You indicate a stretchable section by drawing one (or more) 1-pixel-wide black line(s) in the left and top part of the border (the other border pixels should be fully transparent or white). You can have as many stretchable sections as you want: their relative size stays the same, so the largest sections always remain the largest.

You can also define an optional drawable section of the image (effectively, the padding lines) by drawing a line on the right and bottom lines. If a View object sets the NinePatch as its background and then specifies the View's text, it will stretch itself so that all the text fits inside only the area designated by the right and bottom lines (if included). If the padding lines are not included, Android uses the left and top lines to define this drawable area.

To clarify the difference between the different lines, the left and top lines define which pixels of the image are allowed to be replicated in order to stretch the image. The bottom and right lines define the relative area within the image that the contents of the View are allowed to lie within.

Here is a sample NinePatch file used to define a button:



This NinePatch defines one stretchable area with the left and top lines and the drawable area with the bottom and right lines. In the top image, the dotted grey lines identify the regions of the image that will be replicated in order to stretch the image. The pink rectangle in the bottom image identifies the region in which the contents of the View are allowed. If the contents don't fit in this region, then the image will be stretched so that they do.

The Draw 9-patch (/tools/help/draw9patch.html) tool offers an extremely handy way to create your NinePatch images, using a WYSIWYG graphics editor. It even raises warnings if the region you've defined for the stretchable area is at risk of producing drawing artifacts as a result of the pixel replication.

## Example XML

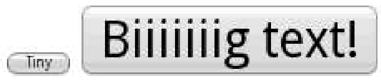
Here's some sample layout XML that demonstrates how to add a NinePatch image to a couple of buttons. (The NinePatch image is saved as `res/drawable/my_button_background.9.png`

```
<Button id="@+id/tiny"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerInParent="true"
        android:text="Tiny"
        android:textSize="8sp"
        android:background="@drawable/my_button_background"/>

<Button id="@+id/big"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerInParent="true"
        android:text="Biiiiiiig text!"
        android:textSize="30sp"
        android:background="@drawable/my_button_background"/>
```

Note that the width and height are set to "wrap_content" to make the button fit neatly around the text.

Below are the two buttons rendered from the XML and NinePatch image shown above. Notice how the width and height of the button varies with the text, and the background image stretches to accommodate it.


Tiny
Biiiiiiig text!