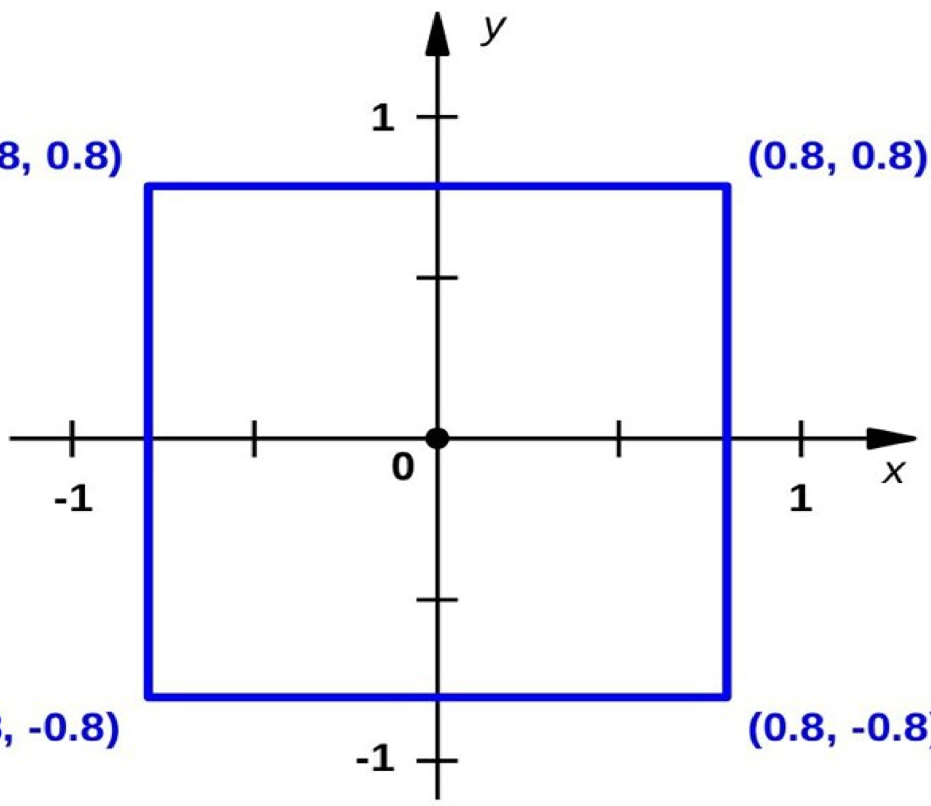
Android 2D Graphics

With Canvas API



Yevgen Karpenko

Table of Contents

Preface

Android provides two powerful graphics libraries: OpenGL ES and Canvas API.

OpenGL ES

OpenGL ES is a high-performance hardware accelerated API for 2D and 3D graphics on embedded systems. It is available on multiple platforms including Android and iOS. Android supports both Java and C/C++ versions of the API.

Canvas API

Google doesn't have any official name for this library. It is just a collection of Java classes in **android.graphics** package. The main class which provides methods for drawing graphics primitives is called Canvas. It is hard to write a book about a library or API which doesn't have a name, therefore some authors call this library Canvas or Canvas API. In this book we will also use this name.

Canvas API is an advanced two-dimensional graphics library. It provides methods for drawing text, lines, rectangles, circles and other graphics primitives. Since Android 3.0 the majority of the drawing done by the Canvas API can be hardware accelerated.

Intended Audience

This book is intended for programmers interested in learning how to use Canvas API in Android. We expect the reader to have some knowledge of Java and Android SDK. We also assume you understand basic Android concepts like Activities and Views and know how to create and run simple Android applications.

In this book we will explain basics of coordinate systems and transformations, show how to use different graphics primitives, and provide a lot of examples.

Contents Overview

- **Chapter 1. Introduction to Canvas API.** This chapter provides an overview of Canvas API and its main Java classes: Canvas, Paint, Path, Typeface and Matrix. Also, in this chapter we will write a simple application to display a view size and briefly talk about logical and device coordinates.
- **Chapter 2. Coordinate Systems.** In this chapter you will learn basics of different coordinate systems, such as Cartesian, device, logical, global and local coordinate systems.
- **Chapter 3. Introduction to Drawing Primitives.** In this chapter we will discuss lines and paths and some of its attributes, such as colors, paint styles, cap and join types, and dash path effects.

- **Chapter 4. Working with Text.** In this chapter we will take a closer look at Canvas API text rendering capabilities. You will learn how to use different fonts, how to measure and align text, and how to change other text attributes, such as size, color and shadow.
- **Chapter 5. Rectangles.** In this chapter you will learn how to use rectangles and create a simple bar chart.
- **Chapter 6. Circles, Ovals and Arcs.** In this chapter you will learn how to draw circles, ovals and arcs. Also, we will draw a circular chart which can be used to display statistical data or as a progress indicator.
- **Chapter 7. Gradients.** In this chapter you will learn how to use gradients.
- **Chapter 8. Introduction to Coordinate Transformations.** In this chapter we will discuss the math behind coordinate transformations and learn how to transform coordinates without matrices. Also, we will create two simple applications to draw a triangle and a stock chart.
- **Chapter 9. Transformation Matrices.** In this chapter you will learn how to use transformation matrices. We will modify two examples from the previous chapter to use transformation matrices instead of the Transform class we wrote ourselves. Also, we will discuss the current transformation matrix (CTM) and why you should not change it.
- **Chapter 10. Multithreading.** In this chapter we will discuss two common approaches to writing multithreaded graphics applications in Android. The first approach uses the Handler and Message classes to communicate between threads. The second approach utilizes the SurfaceView class. Also, we will create two versions of multithreaded stock chart applications and a multithreaded progress indicator.

Contacting the Author

If you have any comments of find an error in the text or in the code, we would be grateful for your feedback. You can contact the author at yevgen.books@gmail.com.

1. Introduction to Canvas API

As I already mentioned in the Preface, Google doesn't have any official name for the collection of classes in **android.graphics** package. Some authors call it Canvas library or Canvas API. In this book we will also use both those names.

Canvas API is a 2D drawing library that can be used to render your own custom graphics. It provides methods for drawing text, lines, rectangles, circles and other graphics primitives. All classes provided by this API are located in **android.graphics** package. Below is the short description of common classes from this library.

1.1. Canvas

To draw something, you need an instance of the Canvas class. It provides methods to draw graphics primitives, such as lines, rectangles, circles, ovals, arcs, text, etc.

The easiest way to get an instance of Canvas object is to override onDraw() method of a View as shown below.

```
public class MyView extends View
{
    @Override
    public void onDraw(Canvas canvas)
    {
    }
}
```

You get an instance of the Canvas class as a parameter of onDraw() method. Android calls this method to refresh the view when the view is initially drawn, when screen orientation changes, or on some system events, such as a touch event. Refresh of a view can also be requested by calling its invalidate() or postInvalidate() methods.

1.2. Paint

The Paint class holds the style and color information about graphics primitives. It also provides methods to measure text. All drawing methods take a Paint object as a parameter. It is common practice to create a Paint object in the constructor and then reuse it in onDraw() method as shown below.

```
public class MyView extends View
{
    private Paint paint;

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

        paint = new Paint();
        paint.setColor(Color.BLACK);
    }
    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawLine(10, 10, 100, 100, paint);
    }
}
```

1.3. Path

The Path class represents a series of straight line segments, curves or other drawing primitives. A path can be open or closed. Creation of a path and drawing a path are two separate operations.

A code snippet below shows how to draw a simple path.

```
@Override
public void onDraw(Canvas canvas)
{
    Path path = new Path();
    path.moveTo(10, 10);
    path.lineTo(100, 100);
    canvas.drawPath(path, paint);
}
```

1.4. Typeface

The Typeface class specifies the typeface and a style of a font. The following code snippet shows how to create a bold serif font.

```
Typeface font = Typeface.create("serif", Typeface.BOLD);
```

We will discuss fonts and other text APIs in Chapter 4.

1.5. Matrix

The Matrix class holds a 3x3 matrix for transforming coordinates.

This book provides a lot of useful information about coordinate systems and coordinate transformations. In Chapter 2 you will learn basics of different coordinate systems. In Chapter 8 we will discuss the math behind coordinate transformations and learn how to transform coordinates without matrices. Finally, in Chapter 9 you will learn how to use transformation matrices.

1.6. Example: View Size

Let's create a simple application that will display the width and height of a view in pixels.

If you don't have any Android programming experience, please learn some basics first. At least learn how to setup Android SDK and create a simple Hello World application. It is beyond the scope of this book to explain how to create a new Android project or activity.

Activity

We will start with creating new Android application project and activity. You can choose any package or activity name. It doesn't have to be exactly the same as in this example.

```
package com.example.graphics2d;

import android.app.Activity;
import android.os.Bundle;

public class GraphicsActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        MyView view = new MyView(this);
        this.setContentView(view);
    }
}
```

The code is very simple. We created an instance of a custom view, called MyView, and made it the main view (also known as the content view) of GraphicsActivity. There is no MyView class yet, so this code will not compile. We will fix it in a minute.

View

Now, let's create MyView class which extends the View and override a couple of its methods. This should fix compilation errors in GraphicsActivity class.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.view.View;


public class MyView extends View
{
    private int width;
    private int height;

    private Paint paint;

    public MyView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint();
        paint.setColor(Color.BLACK);
        paint.setTextAlign(Align.CENTER);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        float textSize = Math.min(width, height) * 0.12f;
        paint.setTextSize(textSize);

        float x = width * 0.5f;
        float y = height * 0.5f;

        String str = "w=" + width + ", h=" + height;
        canvas.drawText(str, x, y, paint);
    }
}
```

Code Review: onSizeChanged()

Let's review the code. We will start with the width and height class variables and onSizeChanged() method.

```
private int width;
private int height;

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh)
{
    width = w;
    height = h;
}
```

The onSizeChanged() method is called by Android framework when MyView size changes, either after the view was created or when Android device orientation changes (from portrait to landscape or vice versa). Old and new width and height are passed as parameters. We are only interested in new size of the view and use class variables width and height to store it.

The onSizeChanged() method is a convenient place to initialize some variables which depend on the view size and orientation, such as a transformation matrix to convert logical coordinates to device coordinates. We will discuss coordinate systems and transformations in chapters 2, 8, and 9.

Code Review: Paint

The Paint class is used to describe colors and styles for the drawing. When you create a new instance of this class it is initialized with default values which can be changed later. In our example, we only set color, text size and text alignment, but there are more attributes which default values can be changed.

```
paint = new Paint();
paint.setColor(Color.BLACK);
paint.setTextAlign(Align.CENTER);
```

For performance reasons, we declared a class variable called paint and initialized it in the constructor instead of creating it in onDraw() method. Remember that every object created in onDraw() method will be created and destroyed every time the screen refreshes.

Code Review: onDraw()

The onDraw() method is where we draw on the Canvas. Android calls this method to refresh the view when the view is initially drawn, when screen orientation changes, or on some system events like a touch event. The initialized Canvas object is passed to onDraw() method as a parameter by Android framework.

```
@Override
public void onDraw(Canvas canvas)
{
    float textSize = width * 0.12f;
    paint.setTextSize(textSize);

    float x = width * 0.5f;
    float y = height * 0.5f;

    String str = "w=" + width + ", h=" + height;
    canvas.drawText(str, x, y, paint);
}
```

We used the drawText() method of the Canvas class to display the size of the view in pixels.

```
canvas.drawText(str, x, y, paint);
```

Note how we passed a Paint object with text color and style to drawText() method in addition to a text and coordinates.

To make sure that our application looks the same on different Android devices we calculated several values dynamically. We made the text size 12% of the view width.

```
float textSize = width * 0.12f;
```

We also made the x and y coordinates 50% of the view width and height.

```
float x = width * 0.5f;
float y = height * 0.5f;
```

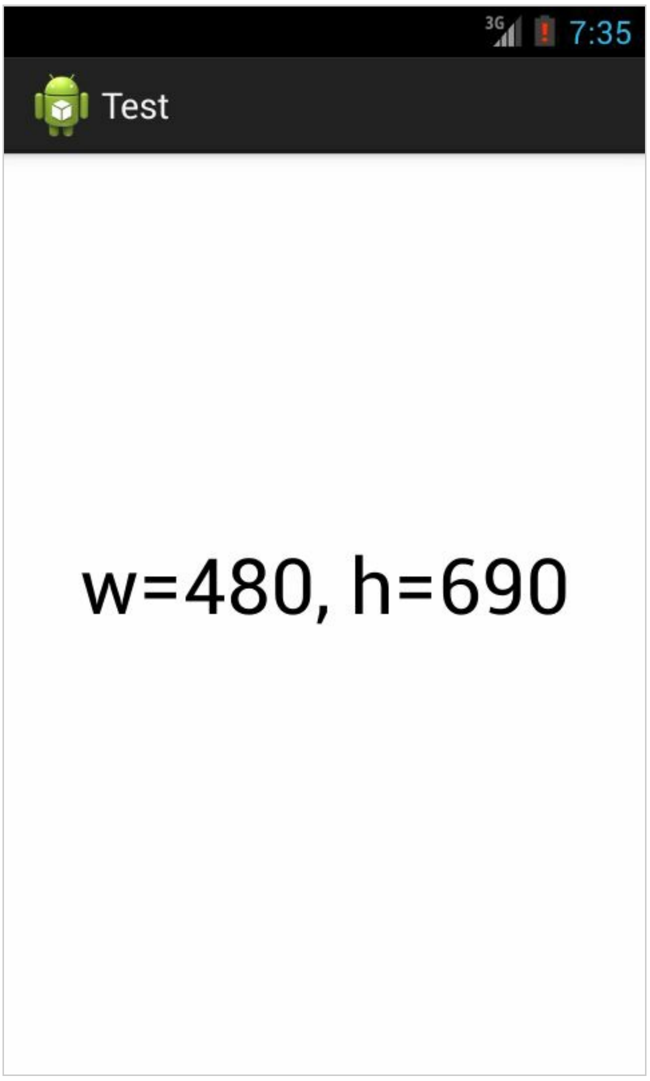
What we did is called coordinate transformation. We used logical coordinate system in which both view width and height range from 0 to 100% (or 0 to 1). The text size in our logical coordinate system is 12% (0.12) and x and y coordinates are 50% (0.5). We transformed logical coordinates (percentages) to device coordinates (pixels).

If you didn't understand the last paragraph, wait until the next chapter, where we will take a closer look at logical and physical coordinate systems.

Screenshots

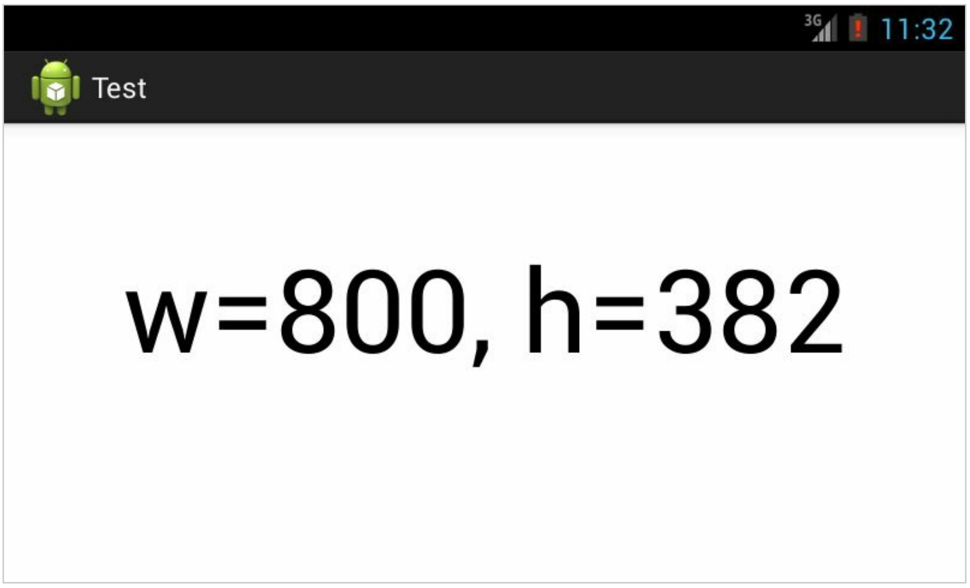
If you run the application, you should see MyView width and height in pixels displayed in the middle of a screen. Let's examine the width and height values on several devices in portrait and landscape modes.

Figures 1-1 and 1-2 show screenshots of the application on a phone with 480x800 screen resolution.



w=480, h=690

Figure 1-1.

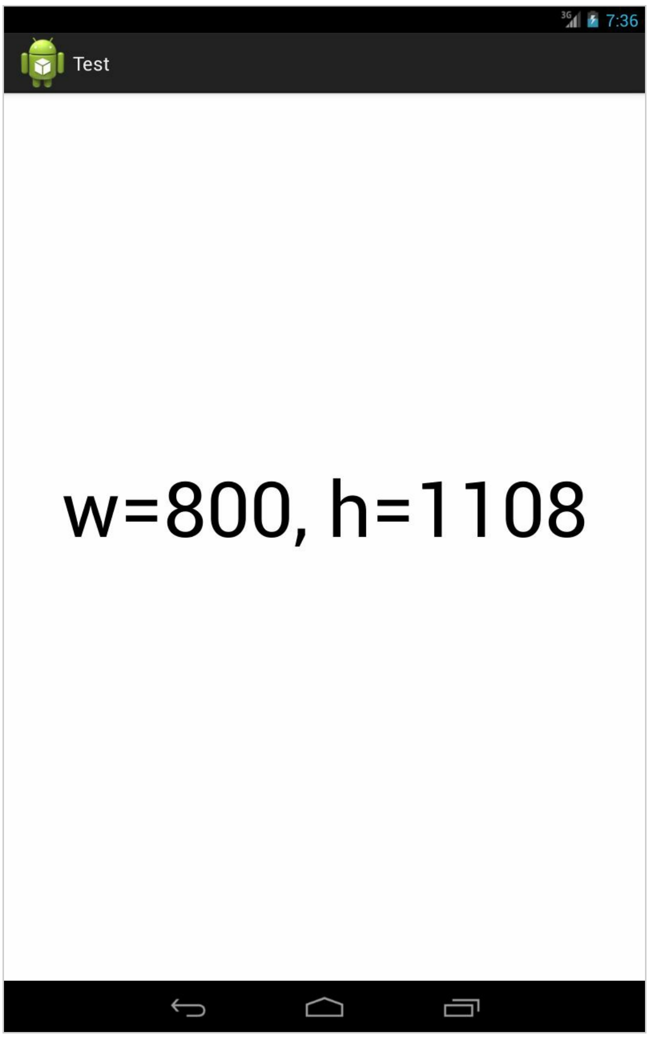


w=800, h=382

Figure 1-2.

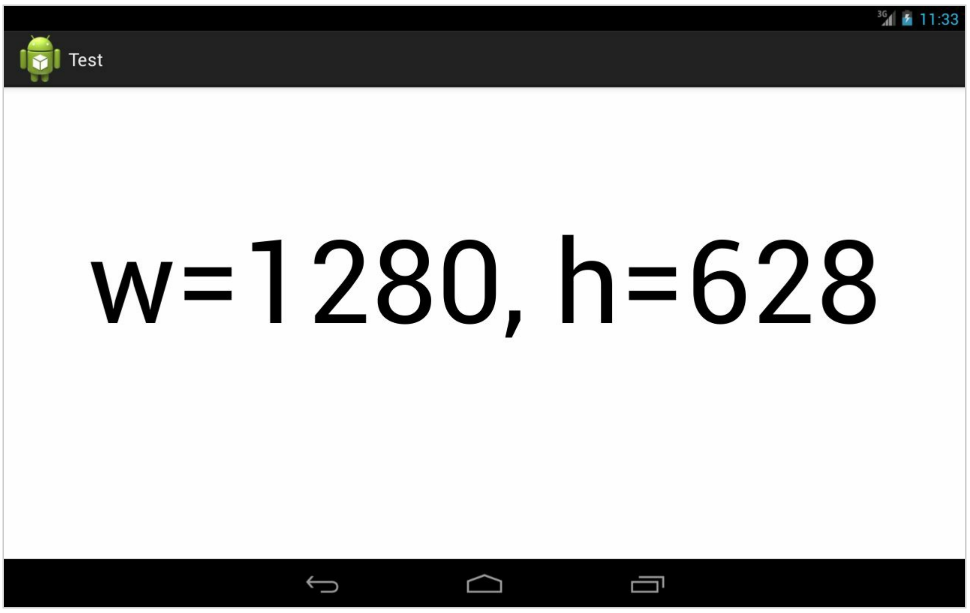
The phone runs Android 4.2 operating system and has physical buttons. Therefore there is no System Bar with virtual buttons at the bottom of the screen. There is the Action Bar or a window title at the top of the screen. It takes significant amount of space, especially in the landscape mode, where the Action Bar height is about 20% of the screen height.

Figures 1-3 and 1-4 show screenshots of the application on a tablet with 800x1280 screen resolution.



w=800, h=1108

Figure 1-3.



w=1280, h=628

Figure 1-4.

The tablet runs Android 4.2 operating system and has both the Action Bar at the top and the

System Bar at the bottom of the screen. Similar to the phone, they reduce the size of the application content view (MyView class).

Many applications hide the Action Bar, especially in the landscape mode to get extra 20% of the screen height.

You probably already knew it and this example just confirmed that in Android, it is very difficult to predict exact size of a view even if you know the screen resolution of a device. Therefore you should never use device coordinates or pixels for your graphics. You should define all your objects, including fonts in logical coordinates and convert them to device coordinates based on the view size and width-to-height ratio.

2. Coordinate Systems

Multiple coordinate systems and the conversion form one coordinate system to another are the most important and difficult to understand concepts in computer graphics. In this chapter we will introduce some common vocabulary and discuss different coordinate systems used in computers graphics. We will talk about coordinate system transformations in chapters 8 and 9. Let's start with some definitions.

Coordinate system is a method of representing the position of an object or a point in two-dimensional or three-dimensional space. Let's also clarify what the point is. In math, a **point** is an abstract concept representing an exact location. It has no size, only a position.

There are many different coordinate systems used in math, engineering and computer graphics. Let's discuss some of them.

2.1. Cartesian Coordinate System

Cartesian coordinate system, also known as rectangular coordinate system, uses two perpendicular reference lines, or coordinate axes that cross at a point called the origin. The horizontal axis is usually called X and the vertical axis is called Y. The location of a point is indicated by a pair of numerical coordinates x and y, which are the signed distances to the origin along X and Y axes. Figure 2-1 shows an example of a Cartesian coordinate system with two points A and B.

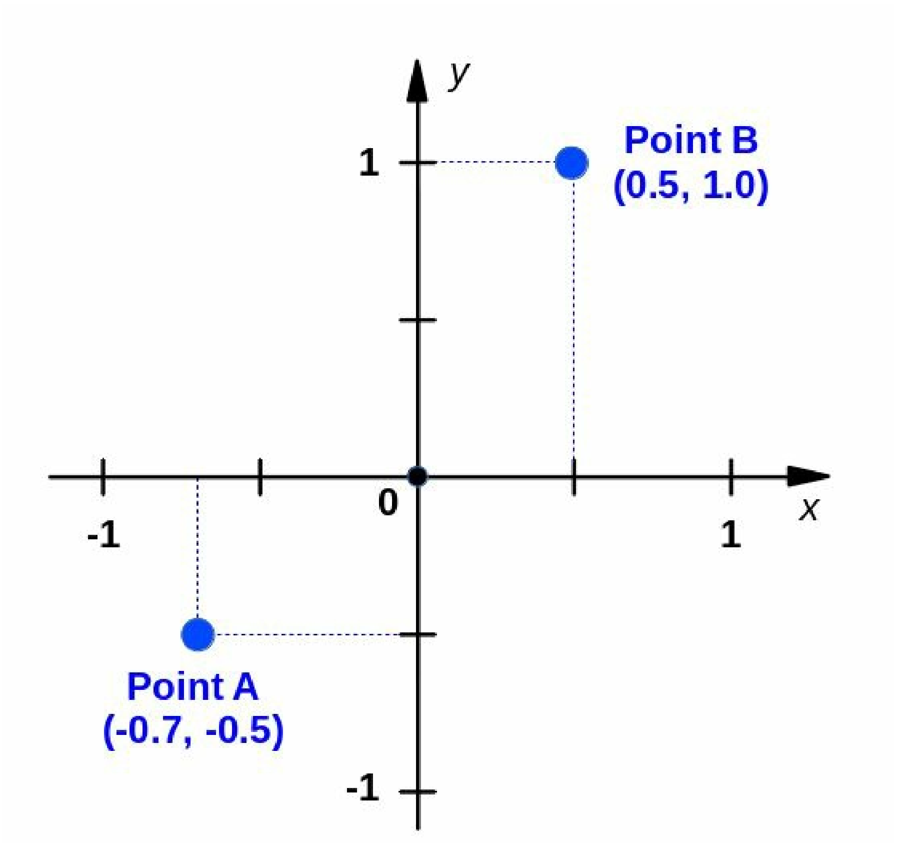


Figure 2-1.

Cartesian coordinate system is the most common system in geometry and computer graphics.

2.2. Device or Display Coordinate System

Displays of computers, cell phones and tablets are raster devices. The term raster device means a device that is composed of discrete pixels. **Pixel** or picture element is the smallest controllable element of a screen. Pixels are often represented by dots or squares and computer displays are represented by a two-dimensional grid of pixels as shown in Figure 2-2.

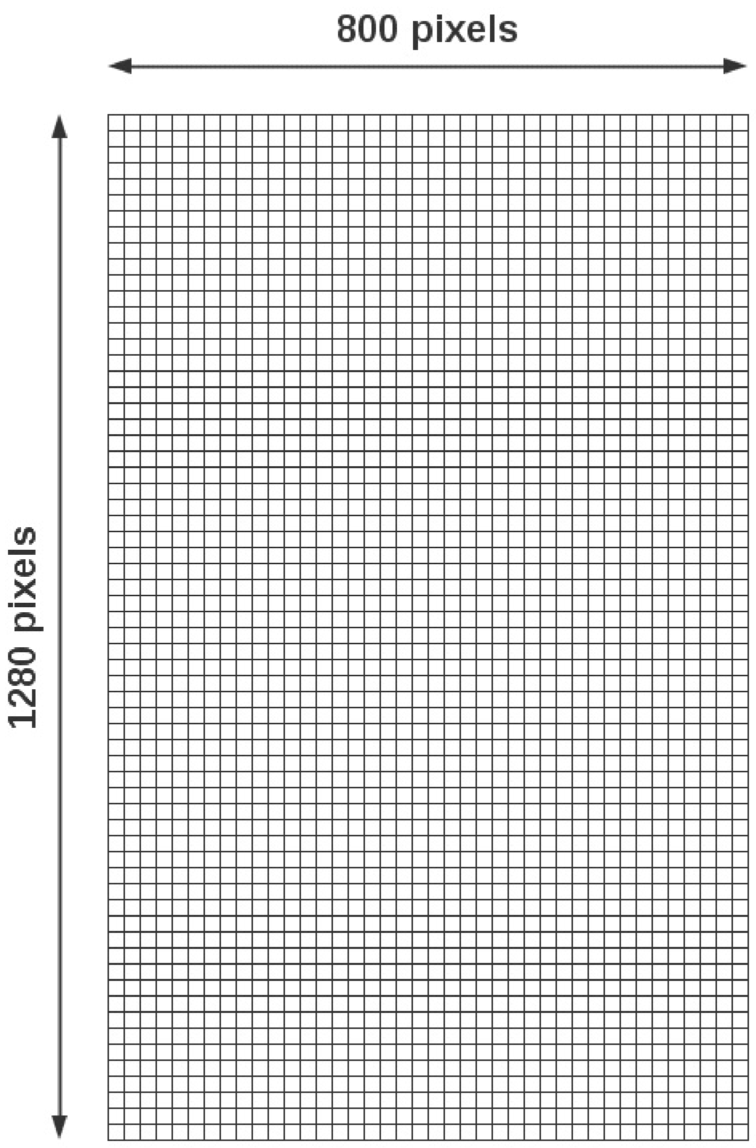


Figure 2-2.

Cartesian coordinate system with pixels as its units of distance is called **device coordinate system**. Many graphics libraries including Android Canvas API use the upper-left corner of the screen as the origin. The X coordinate increases to the right, and the Y coordinate increases downward. All coordinates are specified in pixels as positive integer values. An example of such coordinate system is shown in Figure 2-3.

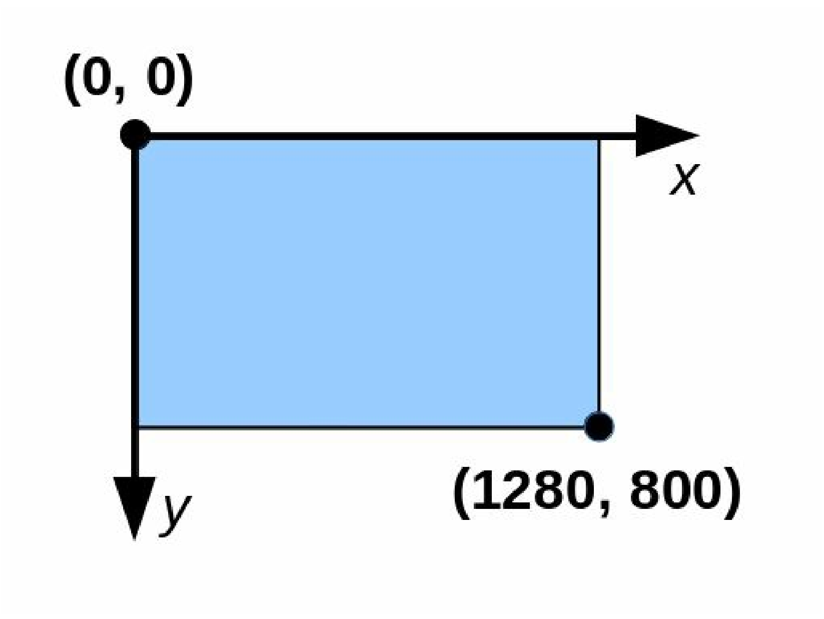


Figure 2-3.

The main disadvantage of this coordinate system is that maximum values of x and y coordinates are device dependent. For example, Google Nexus 7 screen resolution is 1280x800 pixels, but older devices may have resolution of 480x320 pixels.

2.3. Logical Coordinate System

Logical coordinate system is a device-independent coordinate system defined by a programmer for a particular task. Usually it is a Cartesian coordinate system that uses some logical units of distance. Horizontal and vertical axes can have either the same or different units. For example, if you want to draw a stock chart, you can use days as horizontal axis units and stock price as vertical axis units as shown in Figure 2-4.

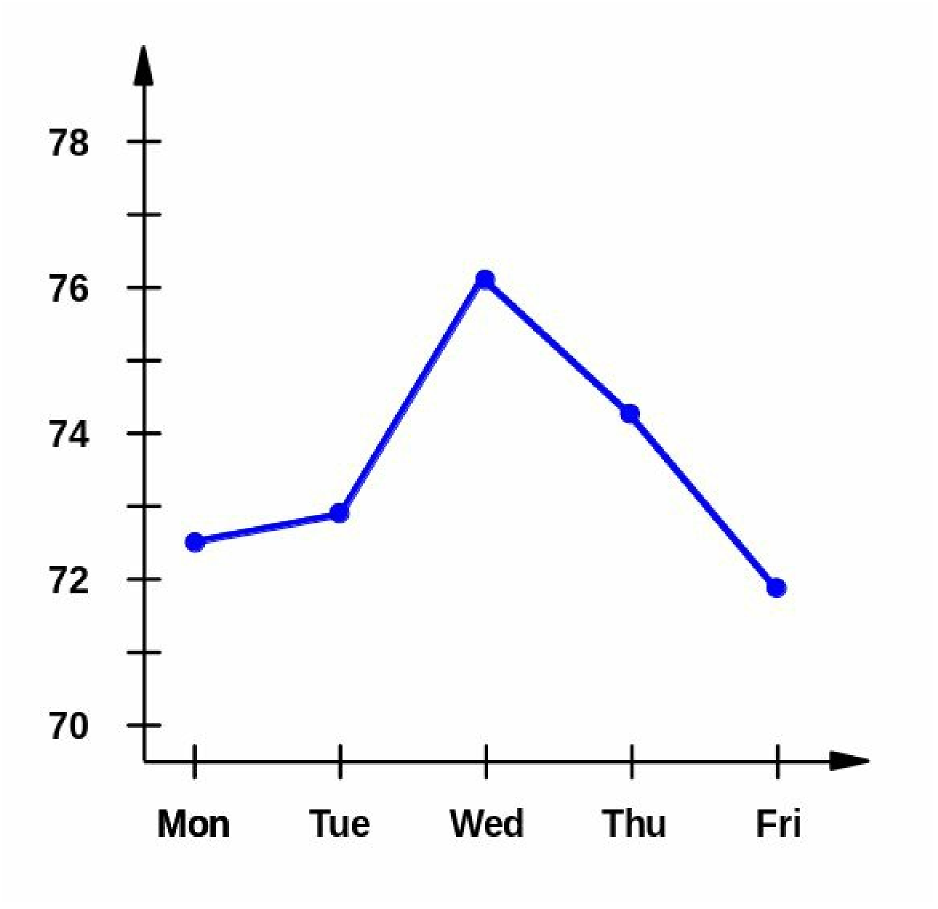


Figure 2-4.

2.4. Global and Local Coordinate Systems

I'm sure that many of you are already familiar with the concept of global and local coordinate systems. Many operating systems, like Microsoft Windows, Linux, or MacOS X can display multiple windows on a screen. Each window, such as a web browser window or a video player window, uses local coordinate system to display its content. An operating system converts local window coordinates to global screen coordinates.

In Android the View class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and uses its local coordinate system for drawing. Local view coordinates are converted to global screen coordinates by Android.

An example of a global and a local coordinate systems is shown in Figure 2-5.

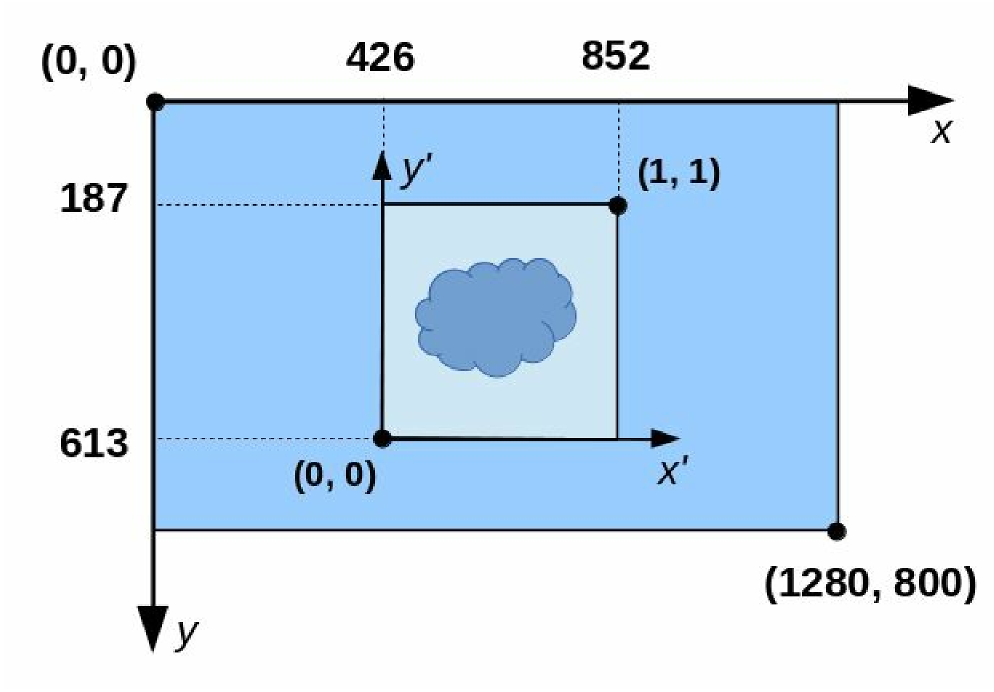


Figure 2-5.

In this example, the global coordinate system is a device coordinate system representing a screen with the resolution of 1280x800 pixels. The origin is in the upper-left corner of the screen. Also there is a local coordinate system. It's origin is offset from the global origin by 426 pixels horizontally and 613 pixels vertically. The orientation is also different. The local coordinate system has its origin in the lower-left corner and its horizontal and vertical axis values range from 0 to 1.

Note, that the global coordinate system doesn't have to be a device coordinate system. It could be a logical coordinate system as well.

3. Introduction to Drawing Primitives

Drawing primitives are simple geometric objects such as lines, rectangles, circles, ellipses, and paths, which can be used to create more complex objects. The Canvas class provides different methods to draw primitives. Each primitive has a set of attributes which affect its appearance. Two examples of such attributes are line color and line width. Attributes are stored in a Paint object which is passed to the Canvas draw methods as a parameter. When you want to draw something, you usually create a Paint object first, set some of its attributes and then call one of the Canvas draw methods with the Paint object as a parameter as shown below.

```
// Create Paint object
Paint paint = new Paint();
paint.setColor(Color.BLUE);
paint.setStrokeWidth(80);
...
// Draw a line
canvas.drawLine(100, 100, 700, 700, paint);
```

In this chapter we will discuss lines and paths and some of its attributes, such as colors, paint styles, cap and join types, and dash path effects.

3.1. Lines

The Canvas class provides several methods to draw lines. The most common method is shown below.

```
void drawLine(float startX, float startY, float endX, float endY, Paint pai
```

This method draws a line segment between two points (startX, startY) and (endX, endY) with the specified paint. The Paint object provides different line attributes or styles such as line color and width.

Example

In this example we will draw a thick blue line shown in Figure 3-1.

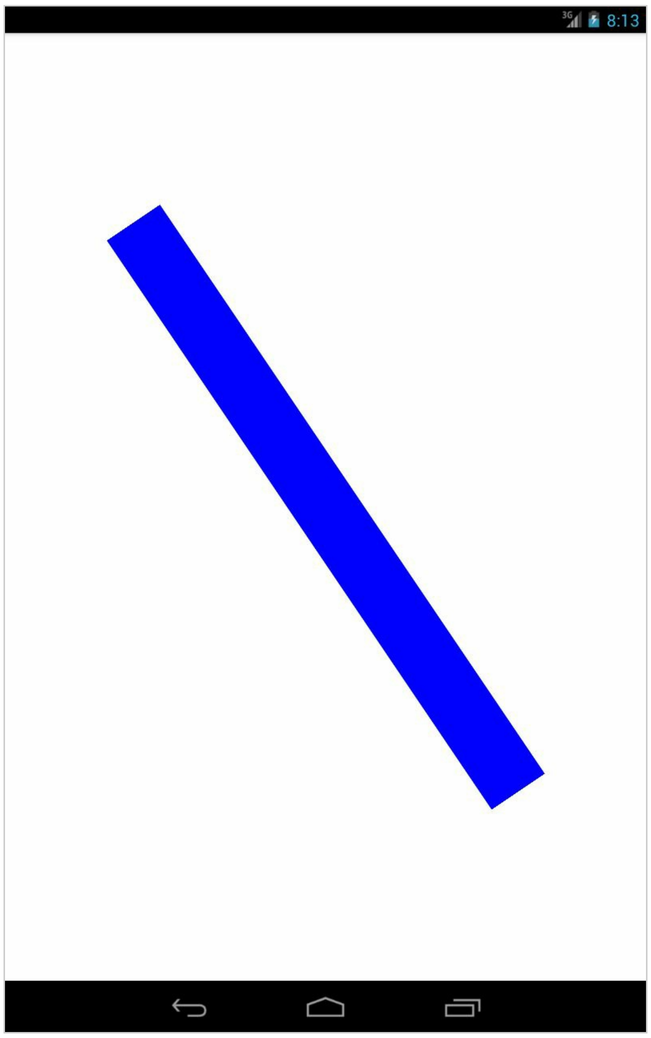


Figure 3-1.

In most examples in this book we will use the same GraphicsActivity class introduced in Chapter 1 and only change the name of the content view. The content view of this example is shown below.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.view.View;


public class LineView extends View
{
    private Paint paint;
    private int width;
    private int height;


    public LineView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);
        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLUE);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawLine(width*0.2f, height*0.2f, width*0.8f, height*0.8f, p
    }


    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
        paint.setStrokeWidth(Math.min(w, h) * 0.1f);
    }
}
```

Let's review the code.

Code Review: Constructor

First, we set the view background color.

```
setBackgroundColor(Color.WHITE);
```

Then we created a Paint object.

```
paint = new Paint(Paint.ANTI_ALIAS_FLAG);
```

The Paint.ANTI_ALIAS_FLAG attribute is used to make lines look smoother. Usually it is a good idea to enable this attribute.

Finally we set the line color.

```
paint.setColor(Color.BLUE);
```

Code Review: onSizeChanged()

First we stored the view width and height. These values are used in onDraw() method to calculate line coordinates.

```
width = w;
height = h;
```

Then we set the line width to 10% of the short side of the view. We used a percentage to make sure that this example looks the same on different Android devices.

```
paint.setStrokeWidth(Math.min(w, h) * 0.1f);
```

Code Review: onDraw()

In this method we drew the line with the paint created in the constructor.

```
canvas.drawLine(width*0.2f, height*0.2f, width*0.8f, height*0.8f, paint);
```

We used ratios to calculate start and end coordinates of the line.

3.2. Colors

Canvas library uses ARGB model to describe a color. In this model each color is represented by four components or channels: alpha, red, green and blue. Each component can have a value from 0 to 255.

Alpha Channel

The alpha channel describes opacity. The value of 0 corresponds to a fully transparent (invisible) pixel, whereas the value of 255 represents a fully opaque pixel.

Integers

Colors are represented by integers. As you may know, in Java, integers are 4 bytes long. Each byte is used to store one of four components of a color in the following order from the most significant byte to the least significant: alpha, red, green, blue.

An integer color value can be easily expressed using 8 hexadecimal digits with each pair of digits representing the values of the alpha, red, green and blue channel, respectively. For example a blue opaque color has the following hexadecimal value:

```
int blue = 0xFF0000FF;
```

Color Class

The Color class provides several helper methods for creating color integers, such as

```
paint.setColor(Color.rgb(0, 0, 255));
```

and

```
paint.setColor(Color.argb(255, 255, 255, 255));
```

It also defines several constants, such as WHITE and BLUE, we used in the previous example:

```
setBackgroundColor(Color.WHITE);
...
paint.setColor(Color.BLUE);
```

3.3. Cap Styles

As you can see from Figure 3-1, the line has flat edges and looks like a rectangle. Usually when you draw a line, you don't want it to look like that. Fortunately, default decoration of line ends can be changed. Lines can have one of three cap styles: butt, round, or square as shown in Figure 3-2.

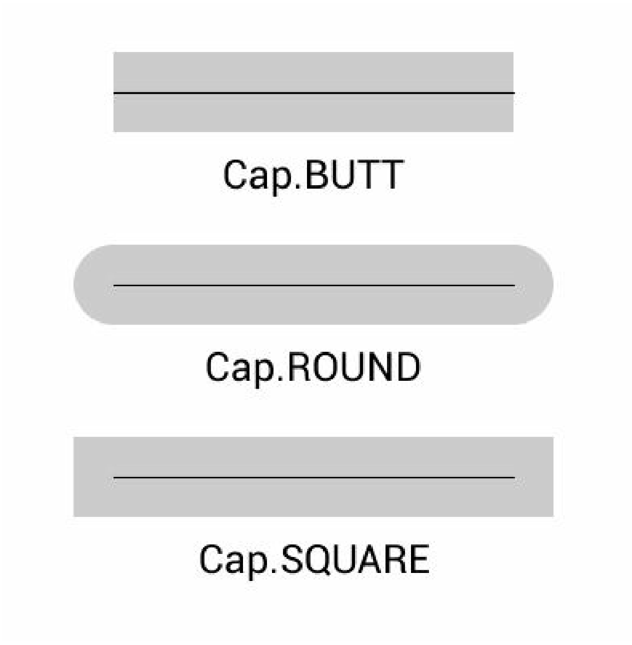


Figure 3-2.

Three cap styles are defined in Paint.Cap enum.

- **Cap.BUTT**. A flat edge is put perpendicular to each end of the line with no cap added. This is the default cap style.
- **Cap.ROUND**. A semicircle or rounded end cap is added to each end of the line. The cap diameter equals the line width value. This style adds 1/2 of the line width to each end of a line.
- **Cap.SQUARE**. A square end cap is added to each end of the line. The cap width is 1/2 of the line width.

The following method is used to set a cap style.

```
paint.setStrokeCap(Cap.ROUND);
```

A cap style is applied to both ends of a line segment. It is not possible to have different cap styles for each end of a line.

Unfortunately there is a serious bug in Android Canvas API. In accelerated drawing mode, which is usually the default mode, drawLine() method always uses Paint.Cap.BUTT style. Even if we modify the previous example to set round line cap in the constructor as shown below

```
...
paint.setStrokeCap(Cap.ROUND);
...
```

on most modern Android devices, the line will still have flat edges as in Figure 3-1. Fortunately there is a workaround. Instead of drawLine() method we can use Path API.

3.4. Introduction to Paths

The Path class represents a series of straight line segments, curves or other drawing primitives. A path can be open or closed. Creation of a path and drawing a path are two separate operations.

First, you create a path.

```
Path path = new Path();
```

Then, you add some drawing primitives to it, like a line segment.

```
path.moveTo(100, 100);
path.lineTo(700, 700);
```

And finally, you draw a path.

```
canvas.drawPath(path, paint);
```

Example

In this example we will create and draw a simple path consisting of a single line, similar to the line from the previous example, but with round caps. The code of the application content view is shown below.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Cap;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.view.View;


public class PathView extends View
{
    private Paint paint;
    private Path path;


    public PathView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLUE);
        paint.setStyle(Style.STROKE);
        paint.setStrokeCap(Cap.ROUND);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        paint.setStrokeWidth(Math.min(width, height) * 0.1f);

        path = new Path();
        path.moveTo(width*0.2f, height*0.2f);
        path.lineTo(width*0.8f, height*0.8f);
    }

}
```

If you run this example you should see a line with round caps as shown in Figure 3-3.

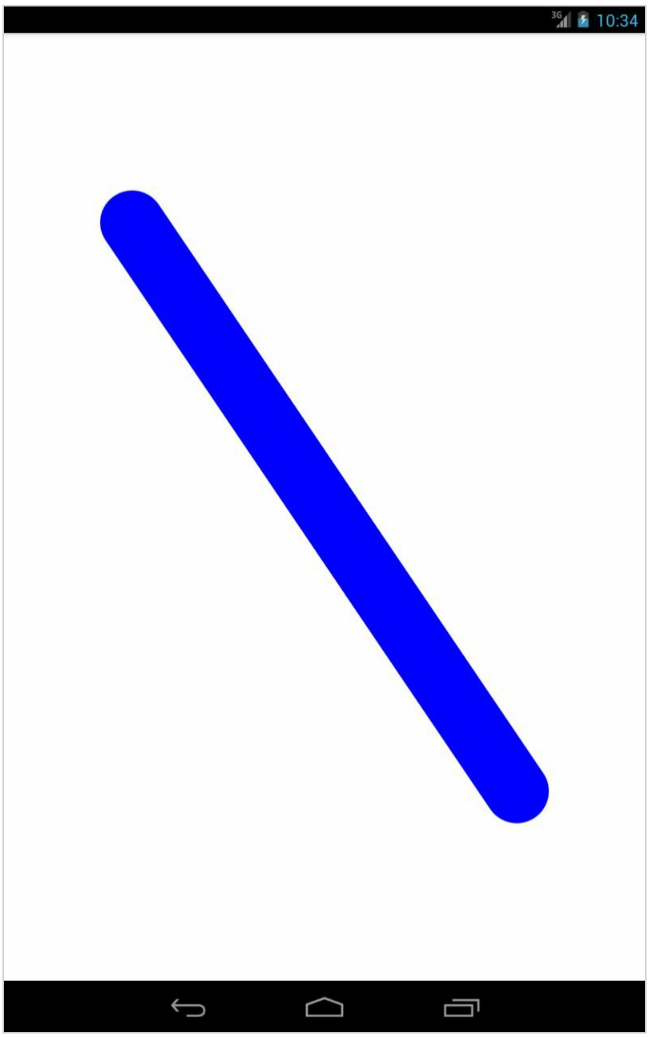


Figure 3-3.

Let's review the code.

Code Review: Constructor

In the constructor we set the view background color and created a Paint object. We also set the line color.

Then we set the paint style.

```
paint.setStyle(Style.STROKE);
```

The STROKE paint style tells Canvas API to stroke the path or draw a line. There are two more paint styles: FILL and FILL_AND_STROKE. We will discuss different paint styles later in this chapter.

Finally we set the line cap style.

```
paint.setStrokeCap(Cap.ROUND);
```

Code Review: onSizeChanged()

First we set the line width to 10% of the short side of the view.

```
paint.setStrokeWidth(Math.min(width, height) * 0.1f);
```

Then we created a Path object.

```
path = new Path();
path.moveTo(width*0.2f, height*0.2f);
path.lineTo(width*0.8f, height*0.8f);
```

The moveTo() and lineTo() methods use the concept of a current point or a location of the pen. The moveTo() method sets a new current point (moves a pen to this point without drawing). The lineTo() method adds a line segment to the path from the current point. After that the current point is set to the end point of the line segment.

Note that creation of a path and drawing a path are two separate operations. To optimize performance of your application you should create a path in a constructor or onSizeChanged() method and then draw it in onDraw() method.

Code Review: onDraw()

In this method we drew the path created in onSizeChanged() method.

```
canvas.drawPath(path, paint);
```

3.5. Join Styles

Another attribute which affects line appearance is a join style. Line join styles control the type of corner that is created when two lines meet. Three styles provided by Canvas API are shown in Figure 3-4.

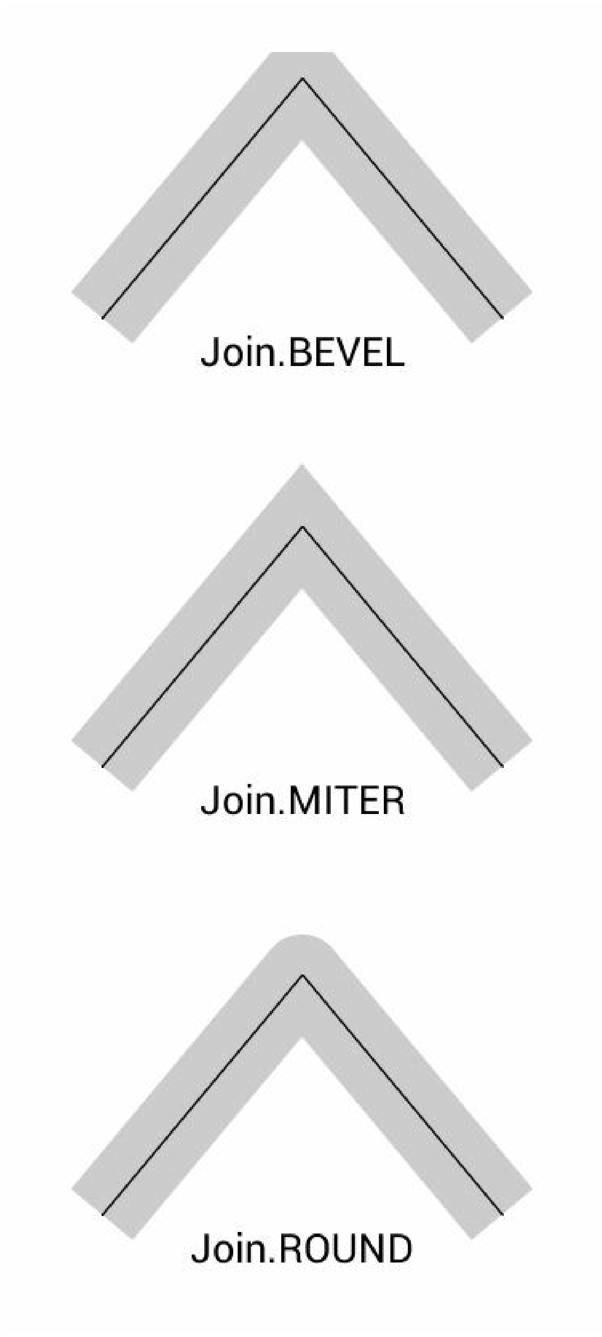


Figure 3-4.

Three join styles are defined in Paint.Join enum.

- **Join.BEVEL**. A filled triangle connects two lines, creating a beveled corner.
- **Join.MITER**. The outside edges of the lines are continued until they intersect. The resulting triangle is filled, creating a sharp or pointed corner. This is the default style.
- **Join.ROUND**. A filled arc connects two lines, creating a rounded corner.

Example

In this example we will draw a triangle and try different join styles. Let's look at the code first and then review it.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Join;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.view.View;


public class TriangleView extends View
{
    private Paint paint;
    private Path path;


    public TriangleView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLUE);
        paint.setStyle(Style.STROKE);
        paint.setStrokeJoin(Join.MITER);
    }


    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        paint.setStrokeWidth(Math.min(width, height) * 0.1f);

        path = new Path();
        path.moveTo(width*0.5f, height*0.2f);
        path.lineTo(width*0.8f, height*0.8f);
        path.lineTo(width*0.2f, height*0.8f);
        path.close();
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }
}
```

Code Review: Constructor

In the constructor we set the view background color, created a Paint object and set its color and style. We also set the join style.

```
paint.setStrokeJoin(Join.MITER);
```

Code Review: onSizeChanged()

First we set the line width to 10% of the short side of the view.

```
paint.setStrokeWidth(Math.min(width, height) * 0.1f);
```

Then we created a Path object.

```
path = new Path();
```

We used three points A(width*0.5f, height*0.2f), B(width*0.8f, height*0.8f), and C(width*0.2f, height*0.8f) to draw a triangle. The moveTo() method sets the start of the path at point A.

```
path.moveTo(width*0.5f, height*0.2f);
```

The lineTo() methods add two line segments AB and BC to the path.

```
path.lineTo(width*0.8f, height*0.8f);
path.lineTo(width*0.2f, height*0.8f);
```

Finally, the close() method closes the path. If the last point of a path is not equal to the first point, a line segment is automatically added. In our example, line segment CA is automatically added to the path.

```
path.close();
```

Code Review: onDraw()

In this method we drew the path created in onSizeChanged() method.

```
canvas.drawPath(path, paint);
```

Screenshots

If you run this example you should see a triangle shown in Figure 3-5.

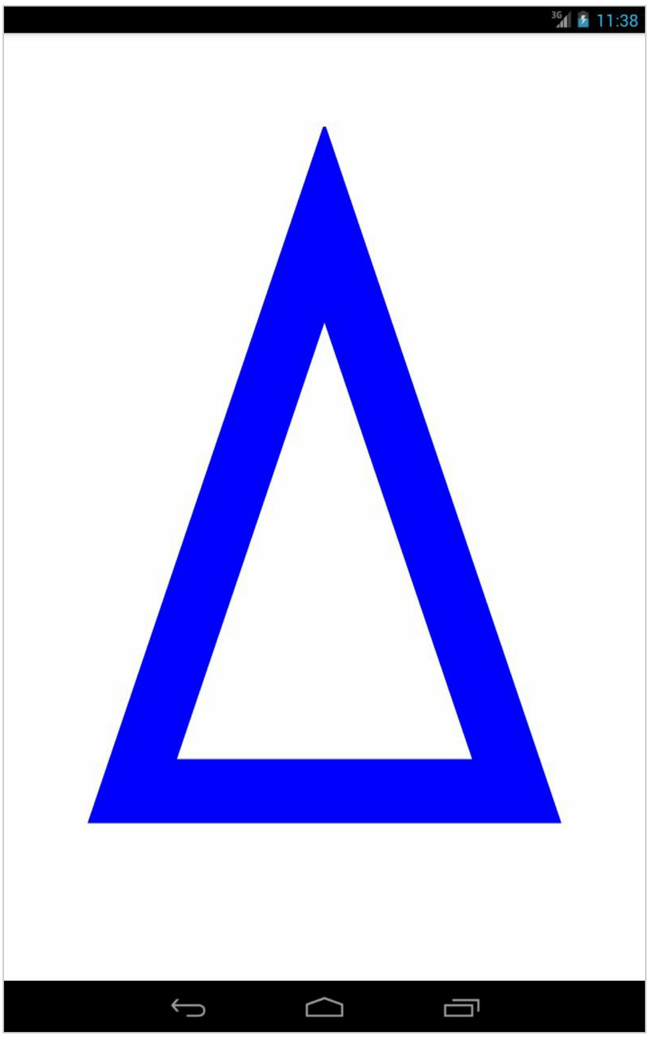


Figure 3-5.

If you change the join style to ROUND

```
paint.setStrokeJoin(Join.ROUND);
```

you should see a triangle shown in Figure 3-6.

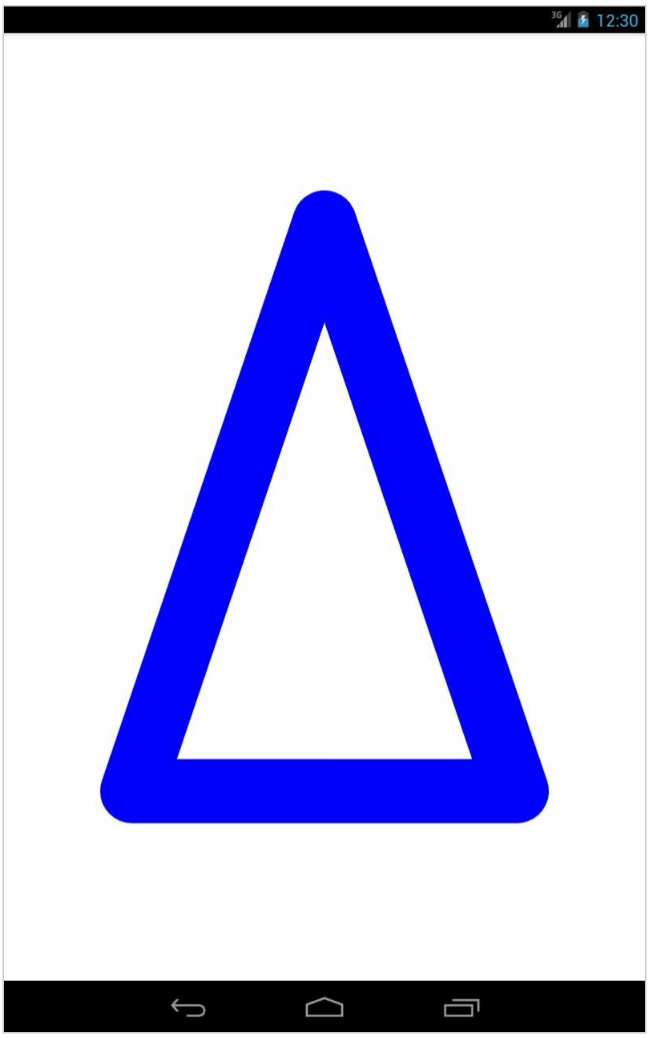


Figure 3-6.

Finally, let's change the join style to BEVEL

```
paint.setStrokeJoin(Join.BEVEL);
```

You should see a triangle shown in Figure 3-7.

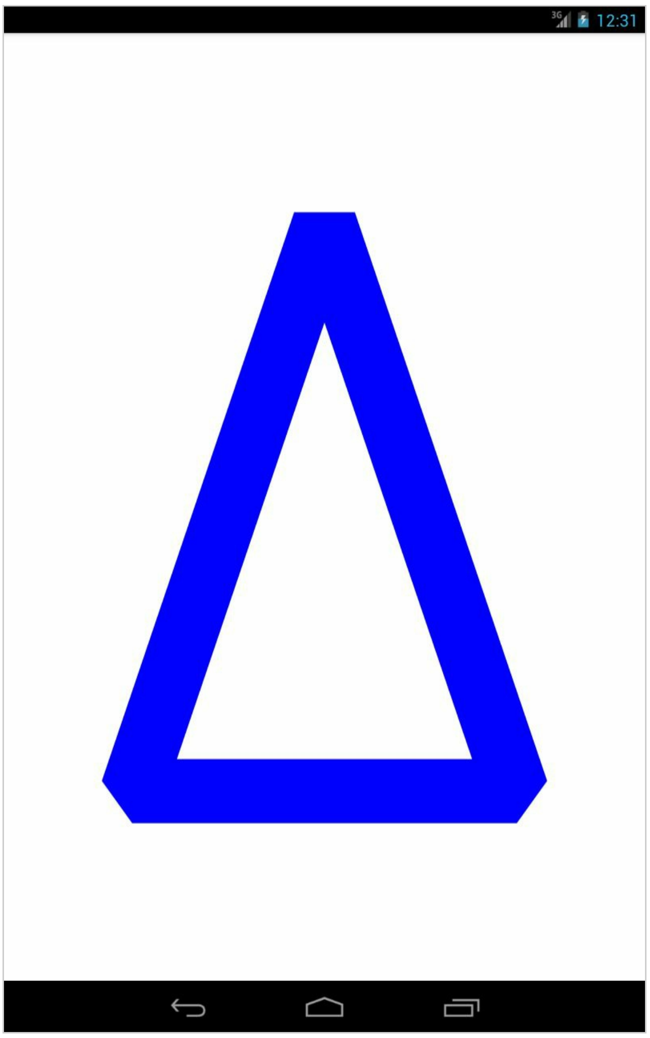


Figure 3-7.

Non-Closed Path

If you do not close a path, a join style will not be applied to the starting and end points of a path. Let's use the ROUND line cap and MITER join

```
paint.setStrokeCap(Cap.ROUND);
paint.setStrokeJoin(Join.MITER);
```

and draw a triangle without closing it.

```
path = new Path();
path.moveTo(width*0.5f, height*0.2f);
path.lineTo(width*0.8f, height*0.8f);
path.lineTo(width*0.2f, height*0.8f);
path.lineTo(width*0.5f, height*0.2f);
```

You should see a triangle shown in Figure 3-8.

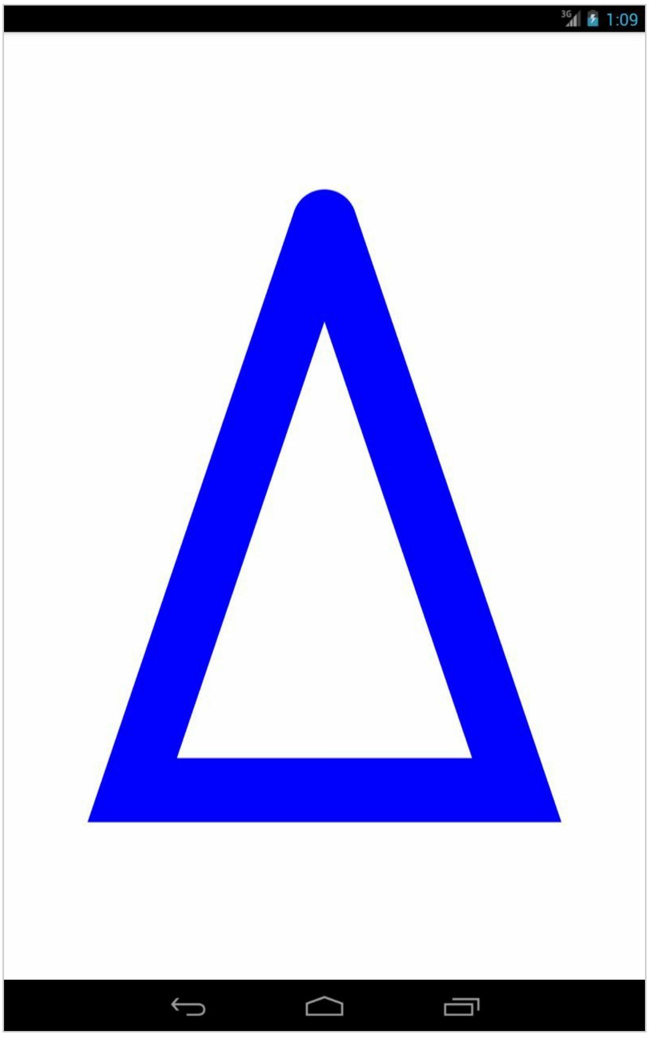


Figure 3-8.

3.6. Paint Styles: Stroke and Fill

There are three paint styles in Canvas API.

- **Style.STROKE** draws a line with specified stroke width, color and other attributes.
- **Style.FILL** fills the area contained within the closed path or another drawing primitive such as a circle or rectangle. This is the default style.
- **Style.FILL_AND_STROKE** strokes and fills with the same color. This style is almost never used, because usually you want to fill a path with one color and stroke it with another one.

The following method is used to set a paint style.

```
paint.setStyle(Style.STROKE);
```

Example

This example shows how to fill a triangle with one color and stroke it with another color. Let's look at the code first and then review it.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Join;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.view.View;


public class TriangleViewFill extends View
{
    private Paint paint;
    private Path path;


    public TriangleViewFill(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStrokeJoin(Join.ROUND);
    }


    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        paint.setStrokeWidth(Math.min(width, height) * 0.1f);

        path = new Path();
        path.moveTo(width*0.5f, height*0.2f);
        path.lineTo(width*0.8f, height*0.8f);
        path.lineTo(width*0.2f, height*0.8f);
        path.close();
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        // Fill
        paint.setStyle(Style.FILL);
        paint.setColor(Color.rgb(255, 153, 0));
        canvas.drawPath(path, paint);

        // Stroke
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.rgb(204, 0, 0));
        canvas.drawPath(path, paint);
    }
}
```

Code Review: Constructor

In the constructor we set the view background color, created a Paint object and set its join style.

Code Review: onSizeChanged()

In this method we set the stroke width

```
paint.setStrokeWidth(Math.min(width, height) * 0.1f);
```

and created the triangle path.

```
path = new Path();
path.moveTo(width*0.5f, height*0.2f);
path.lineTo(width*0.8f, height*0.8f);
path.lineTo(width*0.2f, height*0.8f);
path.close();
```

Code Review: onDraw()

In this method we drew the path twice.

First, the path was filled with yellow color.

```
paint.setStyle(Style.FILL);
paint.setColor(Color.rgb(255, 153, 0));
canvas.drawPath(path, paint);
```

Then it was stroked with red color.

```

paint.setStyle(Style.STROKE);
paint.setColor(Color.rgb(204, 0, 0));
canvas.drawPath(path, paint);
```

Screenshot

If you run this example you should see a triangle shown in Figure 3-9.

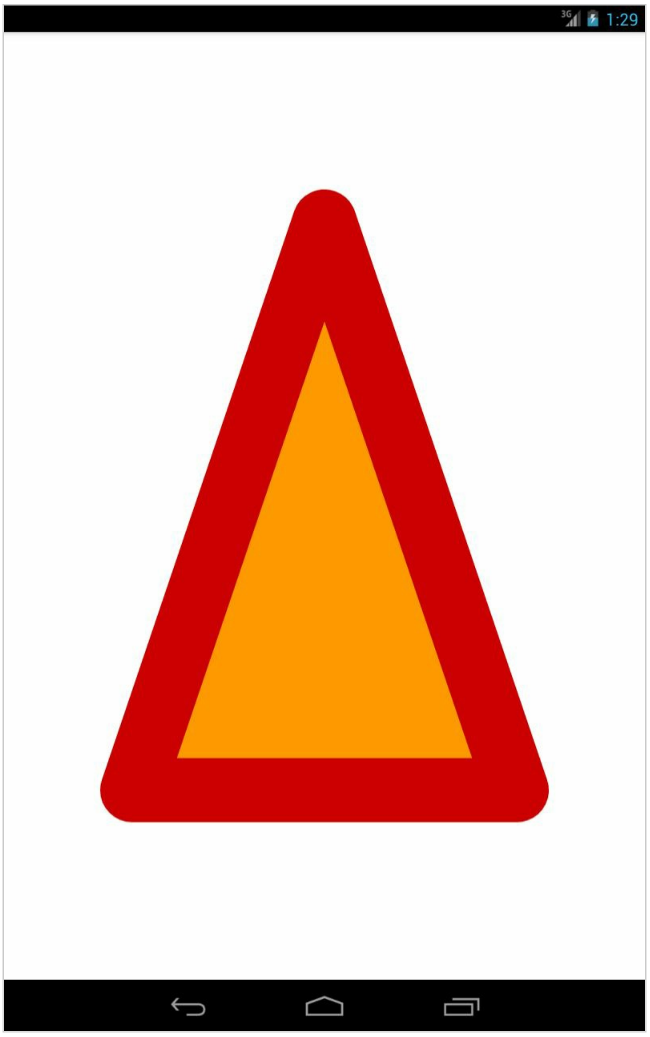


Figure 3-9.

3.7. Dashed Lines

So far we only drew solid lines, but Canvas library also supports dashed and dotted lines.

Do draw a dashed line you have to:

1. Create an instance of the DashPathEffect class.
2. Configure a Paint object to use this path effect.

You create a DashPathEffect object by calling its constructor.

```
DashPathEffect(float[] intervals, float phase)
```

It has the following parameters:

- **intervals** - elements of this array specify widths of dashes in pixels. Even elements specify "on" intervals, and odd elements specify "off" intervals. The array must contain an even number of entries.

- **phase** - horizontal offset in pixels. Usually this parameter is 0.

After a path effect is created you configure a Paint object to use this effect by calling the following method of the Paint class.

```
public PathEffect setPathEffect(PathEffect effect)
```

As a convenience, the parameter passed is also returned. You can pass null to clear any previous path effect.

Example

This example shows how to use the DashPathEffect class to draw dashed and dotted lines. Let's look at the code first and then review it.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.DashPathEffect;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.view.View;


public class DashedLineView extends View
{
    private Paint paint;
    private Path path;

    public DashedLineView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(30);
        paint.setStyle(Style.STROKE);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        path = new Path();
        path.moveTo(50, 100);
        path.lineTo(width - 50, 100);

        DashPathEffect effect = new DashPathEffect(new float[] {90, 30}, 0)
        paint.setPathEffect(effect);
    }
}
```

Code Review

In the constructor we set the view background color and created a Paint object.

In onSizeChanged() method we created a path consisting of a single horizontal line.

```
path = new Path();
path.moveTo(50, 100);
path.lineTo(width - 50, 100);
```

Then we created a DashPathEffect object.

```
DashPathEffect effect = new DashPathEffect(new float[] {90, 30}, 0);
```

The intervals array in DashPathEffect constructor has two values: 90 pixels for the painted segment of the line and 30 pixels for the unpainted segment. The phase parameter is 0.

Finally, we configured the Paint object to use this path effect to stroke the path.

```
paint.setPathEffect(effect);
```

In onDraw() method we drew the path.

```
canvas.drawPath(path, paint);
```

If you run this example you should see a dashed line shown in Figure 3-10.

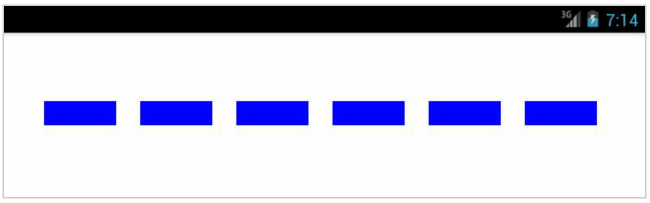


Figure 3-10.

Now, let's see how the phase parameter in the DashPathEffect constructor affects the dash pattern. If you set the phase parameter to 60 pixels as shown below

```
DashPathEffect effect = new DashPathEffect(new float[] {90, 30}, 60);
```

the dash pattern will shift 60 pixels to the left as shown in Figure 3-11.

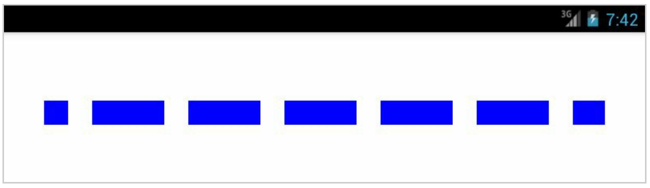


Figure 3-11.

You can also use round stroke cap style to create dotted lines as shown below.

```
paint.setStrokeCap(Cap.ROUND);
DashPathEffect effect = new DashPathEffect(new float[] {1, 60}, 0);
```

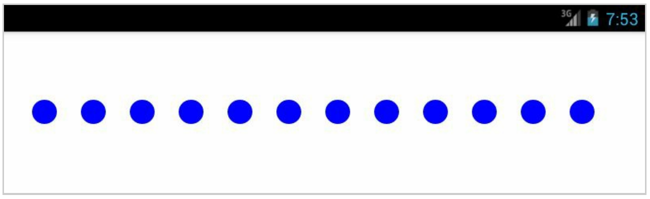


Figure 3-12.

We used 1 pixel width for the painted segment of a line in the DashPathEffect constructor. Round line caps were added by Canvas library.

4. Working with Text

As you may remember, we already used basic text API in our first program in Chapter 1 to display a view size. In this chapter we will take a closer look at Canvas API text rendering capabilities. You will learn how to use different fonts, how to measure and align text, and how to change other text attributes, such as size, color and shadow.

4.1. Standard Fonts

Every Android device comes with three standard fonts: one proportional sans-serif font, one proportional serif font and one monospaced font. The actual fonts installed on a device depend on the device model and the version of Android operating system. Many devices use Droid Sans, Droid Serif, and Droid Sans Mono fonts. The default sans-serif font on the Android Ice Cream Sandwich operating system is Roboto. There is no API to find names of fonts installed on a device.

Let's discuss some common typographic terms.

Typeface

A typeface or a font family is a set of characters that share common design features. After the advent of digital typography and desktop publishing, there is no clear distinction between terms font and typeface. Usually term typeface refers to a design, like Times New Roman or Palatino and fonts are implementations of a design, such as TrueType, OpenType, or PostScript fonts of Times New Roman typeface.

Serifs

In typography, a serif is a small decorative line added as embellishment to a character. Typefaces are often described as being serif or sans serif (without serifs).

- **Serif** typefaces use small decorative marks to embellish characters. Serif fonts include Times New Roman, Courier, New Century Schoolbook, and Palatino.
- **Sans serif** typefaces are composed of simple lines and do not use serifs or small lines at the ends of characters. Popular sans serif fonts are Helvetica, Avant Garde, Arial, and Geneva.

Proportion

Typefaces can be proportional and non-proportional.

- **Proportional** typefaces contain characters of varying widths. For example, the W and M letters are wider than most letters, and the i is narrower.
- **Non-proportional** or **monospaced** typefaces use the same width for all characters. The first monospaced typefaces were designed for typewriters, therefore monospaced fonts are sometimes called typewriter fonts. Popular monospaced typefaces are Courier and

Monaco.

Typeface Class

The Typeface class specifies the typeface and a style of a font. Every Android device supports three typefaces:

- sans-serif
- serif
- monospace

and four styles:

- NORMAL
- BOLD
- ITALIC
- BOLD_ITALIC

The following method can be used to create an instance of the Typeface class, given a font family name and a style.

```
Typeface font = Typeface.create("serif", Typeface.ITALIC);
```

If null is passed for the name, then the default font will be chosen. On most devices it is the sans-serif font.

The Typeface class also provides several predefined typefaces.

```
Typeface.DEFAULT = Typeface.create(null, Typeface.NORMAL);
Typeface.DEFAULT_BOLD = Typeface.create(null, Typeface.BOLD);
Typeface.SANS_SERIF = Typeface.create("sans-serif", Typeface.NORMAL);
Typeface.SERIF = Typeface.create("serif", Typeface.NORMAL);
Typeface.MONOSPACE = Typeface.create("monospace", Typeface.NORMAL);
```

The following method of the Paint class is used to set a typeface.

```
public Typeface setTypeface(Typeface typeface)
```

If null is passed, any previous typeface will be cleared. This method returns the parameter passed.

Example

Let's create a simple application that will display several standard fonts available on every Android device.

Activity

We start with the new project and activity.

```
package com.example.graphics2d;

import android.app.Activity;
import android.os.Bundle;
import android.view.Window;

public class GraphicsActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);
        TextView view = new TextView(this);
        this.setContentView(view);
    }
}
```

The code is similar to other examples from previous chapters. To save some space we will not show the title or the action bar which takes a lot of space in Android 4.x.

```
this.requestWindowFeature(Window.FEATURE_NO_TITLE);
```

View

Next, we will create the view.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Typeface;
import android.view.View;


public class TextView extends View
{
    private Paint paint;

    public TextView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLACK);
        paint.setTextSize(70);
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        int x = 20;
        int y = 90;

        paint.setTypeface(Typeface.DEFAULT);
        canvas.drawText("Default", x, y, paint);
        y+=90;

        paint.setTypeface(Typeface.DEFAULT_BOLD);
        canvas.drawText("Default Bold", x, y, paint);

        y+=90;

        paint.setTypeface(Typeface.SANS_SERIF);
        canvas.drawText("Sans Serif", x, y, paint);

        y+=90;

        paint.setTypeface(Typeface.SERIF);
        canvas.drawText("Serif", x, y, paint);

        y+=90;

        paint.setTypeface(Typeface.MONOSPACE);
        canvas.drawText("Monospace", x, y, paint);

        y+=90;

        Typeface font = Typeface.create("serif", Typeface.BOLD);
        paint.setTypeface(font);
        canvas.drawText("Serif Bold", x, y, paint);

        y+=90;

        font = Typeface.create("serif", Typeface.ITALIC);
        paint.setTypeface(font);
        canvas.drawText("Serif Italic", x, y, paint);
    }
}
```

Code Review

As you may have noticed, to make this example simpler, we hard-coded the font size and coordinates. Therefore this example will look different on different Android devices. In real applications you should always calculate the font size and device coordinates based on the view size and width-to-height ratio.

In the constructor we created a Paint object, set the background and paint colors. Also we set font size in pixels.

```
paint.setTextSize(70);
```

In onDraw() method we drew several standard fonts. We used both predefined fonts:

```
paint.setTypeface(Typeface.DEFAULT);
```

and created several new typeface objects:

```
Typeface font = Typeface.create("serif", Typeface.BOLD);
```

To select a font, we used the following method of the Paint object.

```
paint.setTypeface(font);
```

Screenshot

A screenshot of the application is shown in Figure 4-1.

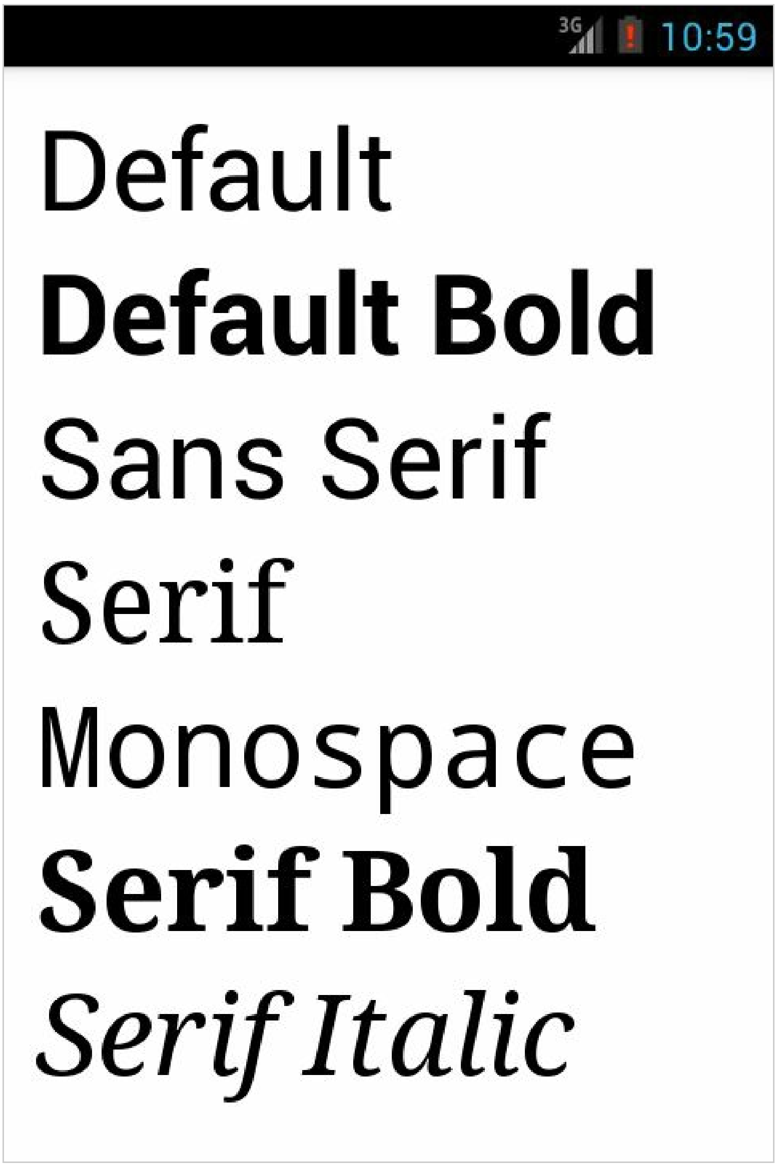


Figure 4-1.

Because we hard-coded the font size and coordinates, this example may look different on your device.

4.2. Custom Fonts

In addition to three standard fonts, Canvas API supports any TrueType or OpenType fonts. Any custom font you want to use has to be bundled with your application. The assets folder at the project root is a convenient location to store raw files. Any file you save in the assets folder is included in the application as-is and the original filename is preserved.

The following method of the Typeface class can be used to create a custom font.

```
public static Typeface createFromAsset(AssetManager mgr, String path)
```

This method has two parameters.

The first parameter is an instance of the AssetManager class which provides access to the application raw asset files. Usually you get it from the Context by calling

```
context.getAssets();
```

The second parameter is a file path of a font in the assets directory.

Example

The following example shows how to use custom fonts in your application.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Typeface;
import android.view.View;


public class TextView extends View
{
    private Paint paint;
    private Typeface font1, font2, font3;

    public TextView2(Context context)
    {
        super(ctx);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLACK);
        paint.setTextSize(90);

        font1 = Typeface.createFromAsset(ctx.getAssets(), "EuphoriaScript-R
        font2 = Typeface.createFromAsset(ctx.getAssets(), "Pacifico.ttf");
        font3 = Typeface.createFromAsset(ctx.getAssets(), "Chantelli_Antiqu
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        int x = 15;
        int y = 100;

        paint.setTypeface(font1);
        canvas.drawText("Euphoria Script", x, y, paint);
        y+= 150;

        paint.setTypeface(font2);
        canvas.drawText("Pacifico", x, y, paint);

        y+= 150;

        paint.setTypeface(font3);
        canvas.drawText("Chantelli", x, y, paint);
    }
}
```

Code Review

We downloaded three free fonts from www.fontsquirrel.com:

- EuphoriaScript-Regular.otf (OpenType)
- Pacifico.ttf (TrueType)
- Chantelli_Antiqua.ttf (TrueType)

and put them in the assets folder at the project root.

In the constructor we created three Typeface objects from font files stored in the assets folder.

```
font1 = Typeface.createFromAsset(ctx.getAssets(), "EuphoriaScript-Regular.o
font2 = Typeface.createFromAsset(ctx.getAssets(), "Pacifico.ttf");
font3 = Typeface.createFromAsset(ctx.getAssets(), "Chantelli_Antiqua.ttf");
```

To get a reference to the AssetManager we called

```
ctx.getAssets()
```

To select a font we used the same method as in the previous example.

```
paint.setTypeface(font);
```

Screenshot

A screenshot of the application is shown in Figure 4-2.

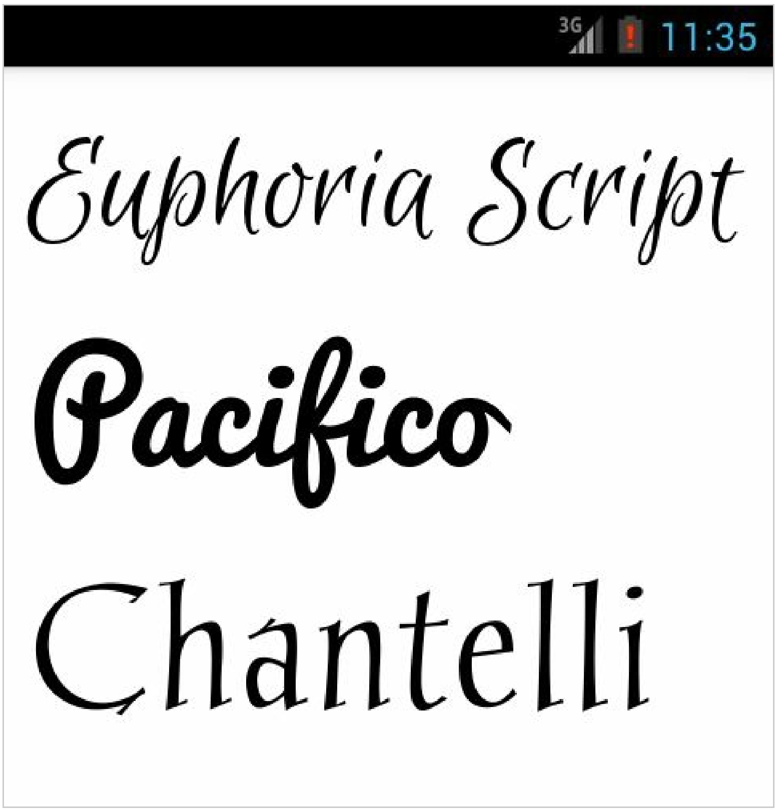


Figure 4-2.

Because we hard-coded the font size and coordinates, this example may look different on your device.

4.3. Horizontal Alignment

To align text horizontally you can either let Canvas library do it for you or if you want more control over exact location of a text you can calculate the x-coordinate yourself.

By default, text is left aligned. It means that text is drawn to the right of the (x,y) origin passed to drawText() method of the Canvas object.

Standard Alignment

The easiest way to change the text alignment is to call the following method of the Paint class.

```
void setTextAlign(Align align)
```

The align parameter of this method can have one of the following values:

- Align.LEFT
- Align.CENTER
- Align.RIGHT

Example

The example below shows how to use setTextAlign() method in your application.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.view.View;


public class TextView extends View
{
    private Paint paint;

    private int width;
    private int height;

    public TextViewAlign(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint();
        paint.setColor(Color.BLACK);
        paint.setTextSize(90);
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        int x = width/2;
        int y = 120;

        canvas.drawLine(x, 0, x, height, paint);

        paint.setTextAlign(Align.LEFT);
        canvas.drawText("Left", x, y, paint);

        y+= 120;

        paint.setTextAlign(Align.CENTER);
        canvas.drawText("Center", x, y, paint);

        y+= 120;

        paint.setTextAlign(Align.RIGHT);
        canvas.drawText("Right", x, y, paint);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
    }
}
```

A screenshot of the application is shown in Figure 4-3.

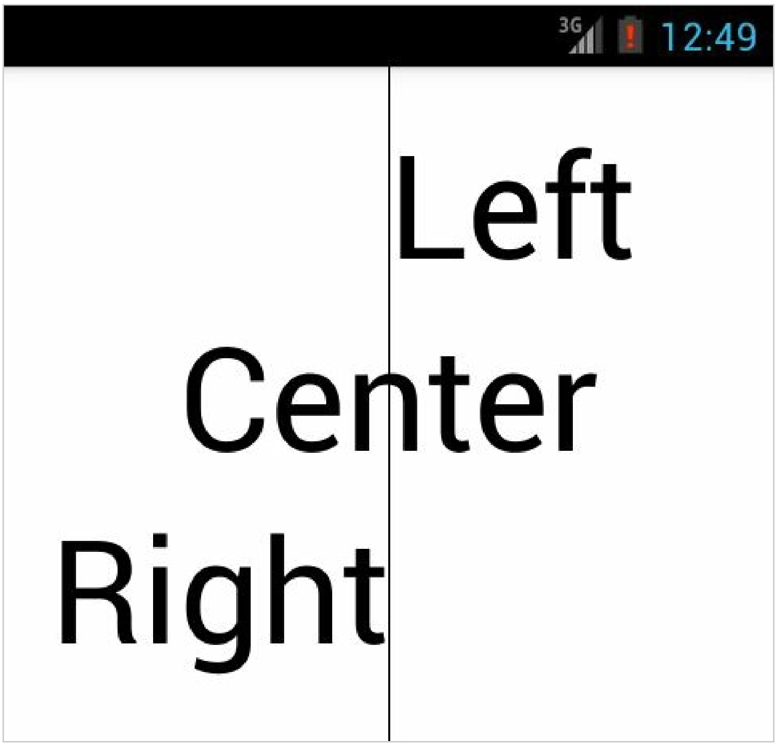


Figure 4-3.

This method of aligning a text should be enough for most applications, but if you want more control over exact text location, you can keep default text alignment and adjust the x-coordinate yourself.

Custom Alignment

Before calculating the x-coordinate, you have to measure the text. The Paint object provides a method for that.

```
void getTextBounds(String text, int start, int end, Rect bounds)
```

It has following parameters.

- text - a string to measure.
- start - index of the first character in the string. Usually you pass 0.
- end - index of the last character in the string plus one. Usually you pass string length here.
- bounds - returns the smallest rectangle that encloses all of the characters. Left, right, top and bottom values of the rectangle are relative to the origin or x- and y-coordinates passed to drawText() method.

Let's call getTextBounds() method and examine returned values in a debugger.

```
@Override
public void onDraw(Canvas canvas)
{
    int x = width/2;
    int y = 120;
    canvas.drawLine(x, 0, x, height, paint);

    Rect rect = new Rect();
    String str = "Left";
    paint.getTextBounds(str, 0, str.length(), rect);
    ......
}
```

The exact values depend on the screen resolution and fonts installed on a device. We got the following values.

```
rect.left = 7
rect.right = 150
rect.top = -71
rect.bottom = 1
```

To calculate the x-coordinate to adjust the horizontal text alignment we only need the left and right values. We will talk more about the top and bottom values in the next section.

As you can see from Figure 4-3, there is some space between the letter L and the vertical line. The space is 7 pixel wide in our case, but this value depends on a letter and a font. The rect.left tells you what the value is. If you don't want this extra space you can adjust your x-coordinate as shown below to shift the text left.

```
canvas.drawText(str, x-rect.left, y, paint);
```

To center the text, the following formula can be used.

```
float offset = (rect.left + rect.right) / 2.0f;
canvas.drawText(str, x-offset, y, paint);
```

Finally, if you want to right-align the text, you can do it like this.

```
canvas.drawText(str, x-rect.right, y, paint);
```

4.4. Vertical Alignment

Unfortunately Canvas API dosn't provide any methods to align text vertically. We have to measure text and calculate the y-coordinate ourselves before passing it to drawText() method. Before we learn how this can be done, let's discuss some font metrics vocabulary.

Baseline

The baseline is the imaginary horizontal line on which most letters sit. The y-coordinate in drawText() method is the y-coordinate of the baseline. Let's create a simple view to draw a baseline in addition to some text.

```
package com.example.graphics2d;
```

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.view.View;


public class TextViewBaseline extends View
{
    private Paint paint;

    private int width;
    private int height;

    public TextViewBaseline(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint();
        paint.setColor(Color.BLACK);
        paint.setTextSize(100);
        paint.setTypeface(Typeface.SERIF);
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        int x = width/2;
        int y = 150;

        canvas.drawLine(0, y, width, y, paint);
        paint.setTextAlign(Align.CENTER);
        canvas.drawText("Python", x, y, paint);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
    }
}
```

The screenshot of the example is shown in Figure 4-4.

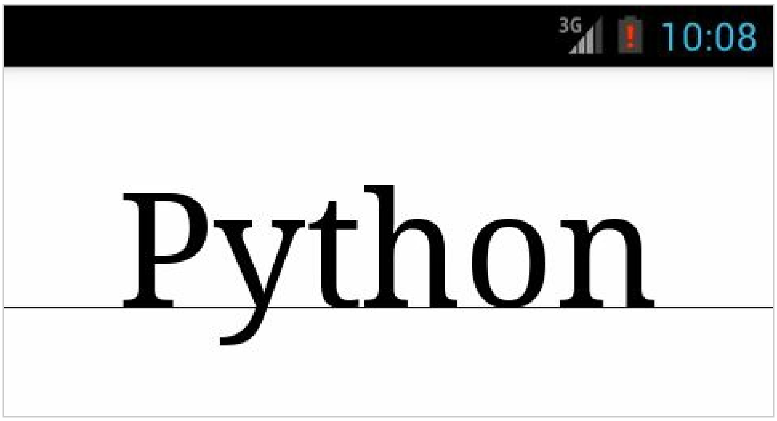


Figure 4-4.

As you can see most letters sit on the baseline and the letter y extends below the baseline.

Ascent and Desccent

The **ascent** is the distance between the baseline and the top of the tallest character. The ascent usually includes extra space required to display letters with accents, such as an umlaut in German language.

The **descent** is the distance between the baseline and the lowest descending letter in a typeface, such as the y letter.

Let's modify onDraw() method to draw both ascent and descent lines.

```
@Override
public void onDraw(Canvas canvas)
{
    int x = width/2;
    int y = 150;

    float ascent = paint.ascent();
    float descent = paint.descent();

    canvas.drawLine(0, y, width, y, paint);
    canvas.drawLine(0, y+ascent, width, y+ascent, paint);
    canvas.drawLine(0, y+descent, width, y+descent, paint);

    paint.setTextAlign(Align.CENTER);
    canvas.drawText("Python", x, y, paint);
}
```

The screenshot of the modified example is shown in Figure 4-5.

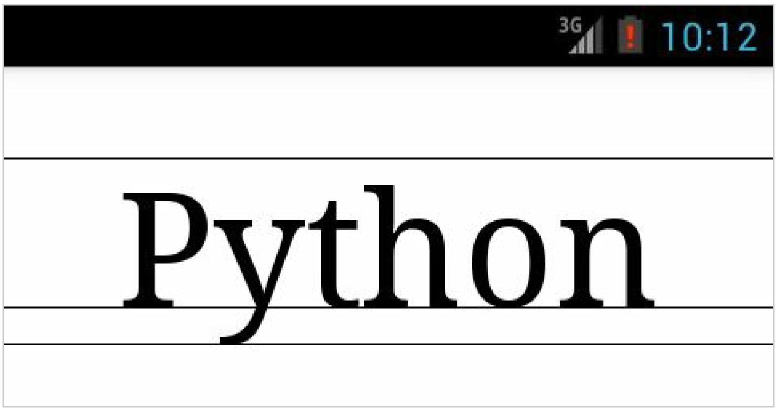


Figure 4-5.

As you can see there is some extra space between the ascent line and the tallest letters. That space is reserved for upper case letters with accents as shown in Figure 4-6.

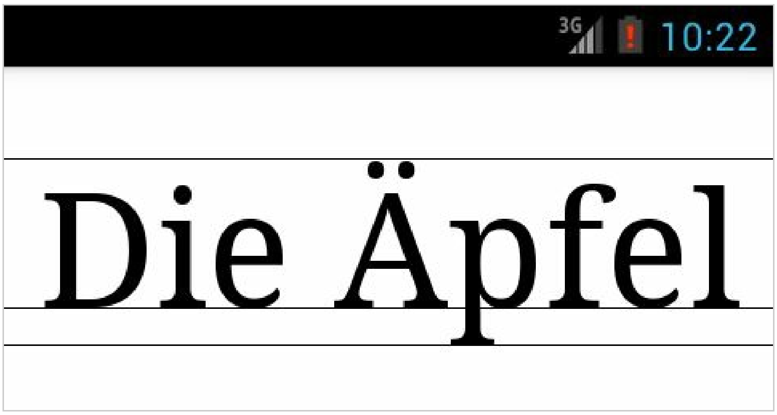


Figure 4-6.

Bottom Alignment

By default, text is bottom aligned at the font baseline as shown in Figure 4-4. You don't have to do any extra work to align it.

Center Alignment

There are two common ways to center align text. The first method is based on the font ascent and descent values and the second on exact text height.

The following example shows how to use both methods.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Typeface;
import android.graphics.Paint.Align;
import android.graphics.Rect;
import android.view.View;


public class TextView extends View
{
    private static final int FONT_SIZE = 100;

    private Paint paint;

    private int width;
    private int height;

    public TextViewAlignCenter(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint();
        paint.setColor(Color.BLACK);
        paint.setTextSize(FONT_SIZE);
        paint.setTypeface(Typeface.SERIF);
        paint.setTextAlign(Align.CENTER);
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        center1(canvas, "Python", 100);
        center2(canvas, "Python", 300);
    }

    private void center1(Canvas canvas, String str, int y)
    {
        int x = width/2;

        float offset = (paint.ascent() + paint.descent()) / 2;
        canvas.drawText(str, x, y-offset, paint);

        float topY = y - FONT_SIZE/2.0f;
        float bottomY = y + FONT_SIZE/2.0f;

        canvas.drawLine(0, y, width, y, paint);
        canvas.drawLine(0, topY, width, topY, paint);
        canvas.drawLine(0, bottomY, width, bottomY, paint);
    }


    private void center2(Canvas canvas, String str, int y)
    {
        int x = width/2;

        Rect bounds = new Rect();
        paint.getTextBounds(str, 0, str.length(), bounds);

        float offset = bounds.exactCenterY();
        canvas.drawText(str, x, y-offset, paint);

        float topY = y - FONT_SIZE/2.0f;
        float bottomY = y + FONT_SIZE/2.0f;

        canvas.drawLine(0, y, width, y, paint);
        canvas.drawLine(0, topY, width, topY, paint);
        canvas.drawLine(0, bottomY, width, bottomY, paint);
    }


    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
    }
}
```

A screenshot of the example is shown in Figure 4-7. There are two lines of text. The first line was aligned based on the font ascent and descent values and the second, based on exact text height obtained from getTextBounds() method.

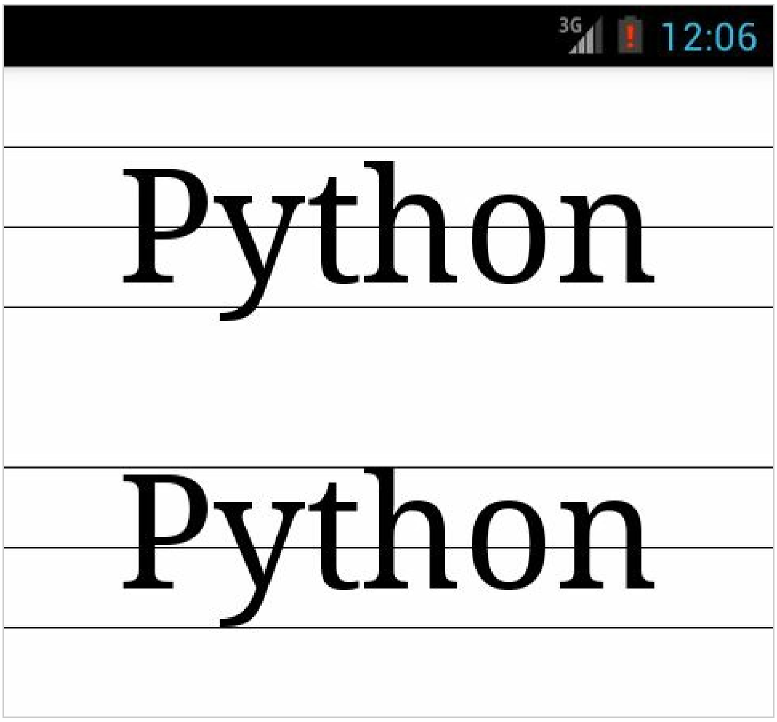


Figure 4-7.

The first method is based on the font ascent and descent values and is implemented in center1() method. First we calculated the y-coordinate offset which is the distance from the baseline to the center.

```
float offset = (paint.ascent() + paint.descent()) / 2;
```

Then we shifted text down by the offset.

```
canvas.drawText(str, x, y-offset, paint);
```

Let's go over math involved in the offset calculation. In Canvas API paint.ascent() value is negative and paint.descent() is positive. It makes sense because Android Canvas API uses the upper-left corner of the screen as the origin and the y-coordinate increases downward. In our example the following values were used.

```
ascent = -92.8
descent = 23.6
offset = (ascent + descent) / 2 = (-92.8 + 23.6) / 2 = -34.6
```

We could have calculated maximum text height first and then added half-height to the ascent to get the same offset value.

```
height = (abs(ascent) + abs(descent)) = 116.4
offset = ascent + height / 2 = -92.8 + 58.2 = -34.6
```

We substracted the offset from the y-coordinate, because the offset is negative and we have to shift text down, or increase the y-coordinate value.

```
canvas.drawText(str, x, y-offset, paint);
```

The second method is similar to the first one. The only difference is that we measure text exactly instead of using maximum ascent and descent values. This method is more accurate. The implementation is provided by center2() method. The getTextBounds() methods is used to measure the text.

```
Rect bounds = new Rect();
paint.getTextBounds(str, 0, str.length(), bounds);
```

It measures both the height and width of the text. In our example we are only interested in the top and bottom fields of the Rect object. The ascent and descent values are the maximum values for a font, but bounds.top and bounds.bottom are exact values of the text being measured. Let's compare the values.

```
ascent = -92.8
descent = 23.6
bounds.top = -76
bounds.bottom = 24
```

The Rect object provides a method to calculate vertical center of a rectangle.

```
float offset = bounds.exactCenterY();
```

We could have used the following formula instead.

```
float offset = (bounds.top + bounds.bottom) / 2.0f;
```

Top Alignment

Similar to center alignment, the y-coordinate of top aligned text can be adjusted based on either font ascent or the exact text height.

The example below shows how to use both methods.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Typeface;
import android.graphics.Paint.Align;
import android.graphics.Rect;
import android.view.View;


public class TextViewAlignTop extends View
{
    private static final int FONT_SIZE = 100;
    private Paint paint;
    private int width;
    private int height;

    public TextViewAlignTop(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint();
        paint.setColor(Color.BLACK);
        paint.setTextSize(FONT_SIZE);
        paint.setTypeface(Typeface.SERIF);
        paint.setTextAlign(Align.CENTER);
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        top1(canvas, "Python", 50);
        top2(canvas, "Python", 250);
    }

    private void top1(Canvas canvas, String str, int y)
    {
        int x = width/2;
        canvas.drawText(str, x, y-paint.ascent(), paint);
        canvas.drawLine(0, y, width, y, paint);
    }

    private void top2(Canvas canvas, String str, int y)
    {
        int x = width/2;

        Rect bounds = new Rect();
        paint.getTextBounds(str, 0, str.length(), bounds);

        canvas.drawText(str, x, y-bounds.top, paint);
        canvas.drawLine(0, y, width, y, paint);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
    }
}
```

A screenshot of the example is shown in Figure 4-8. There are two lines of text. The first line was aligned based on the font ascent and the second on exact text height obtained from getTextBounds() method.

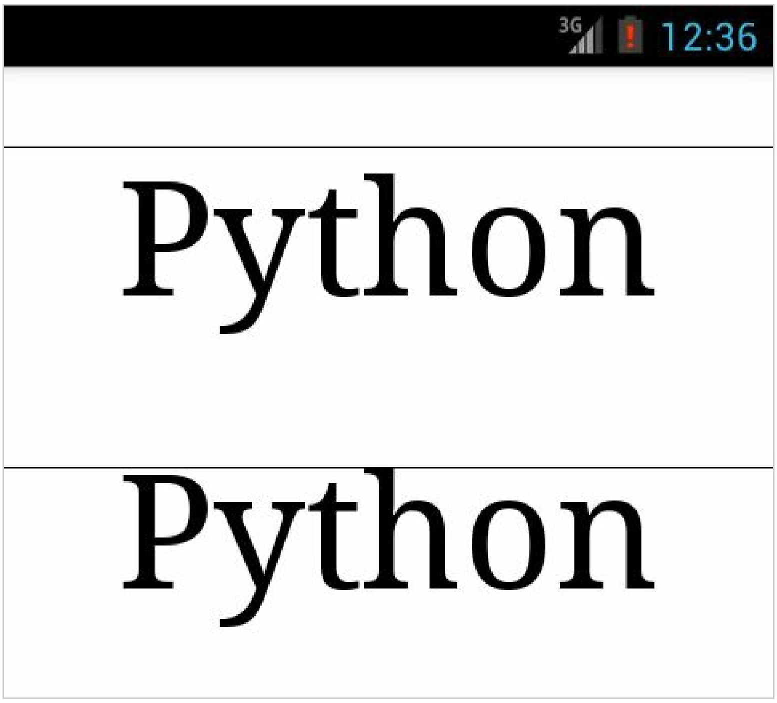


Figure 4-8.

Top alignment based on the font ascent was implemented in top1() method.

```
canvas.drawText(str, x, y-paint.ascent(), paint);
```

The second method based on exact text height obtained from getTextBounds() method was implemented in top2() method.

```
Rect bounds = new Rect();
paint.getTextBounds(str, 0, str.length(), bounds);
canvas.drawText(str, x, y-bounds.top, paint);
```

As you can see from Figure 4-8, the first method leaves a gap between the horizontal line the text is aligned to and the tallest letter. The second method is more accurate.

4.5. Shadow

A shadow is an additional layer painted below the main object such as a text or another graphics primitive. A shadow can improve the appearance of a text in some cases, but it does not mean you have to use shadows all the time.

The Paint object provides the following method to configure shadow appearance.

```
void setShadowLayer(float radius, float dx, float dy, int color)
```

It has several parameters.

- radius - a blur radius in pixels. If radius is 0, the shadow layer is removed. A higher radius

make the image blurrier.
- dx - an offset of a shadow in the horizontal direction in pixels.
- dy - an offset of a shadow in the vertical direction in pixels.
- color - a shadow color.

The example below shows how to use shadows in your application.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.view.View;


public class TextViewShadow extends View
{
    private Paint paint;

    private int width;
    private int height;

    public TextViewShadow(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setColor(Color.BLACK);
        paint.setTextSize(100);
        paint.setTextAlign(Align.CENTER);
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        int x = width/2;
        int y = 120;

        paint.setShadowLayer(10, 5, -5, Color.GRAY);
        canvas.drawText("Shadow", x, y, paint);

        y+= 140;

        paint.setShadowLayer(16, 10, 10, Color.GRAY);
        canvas.drawText("Shadow", x, y, paint);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        width = w;
        height = h;
    }
}
```

A screenshot of the application is shown in Figure 4-9.

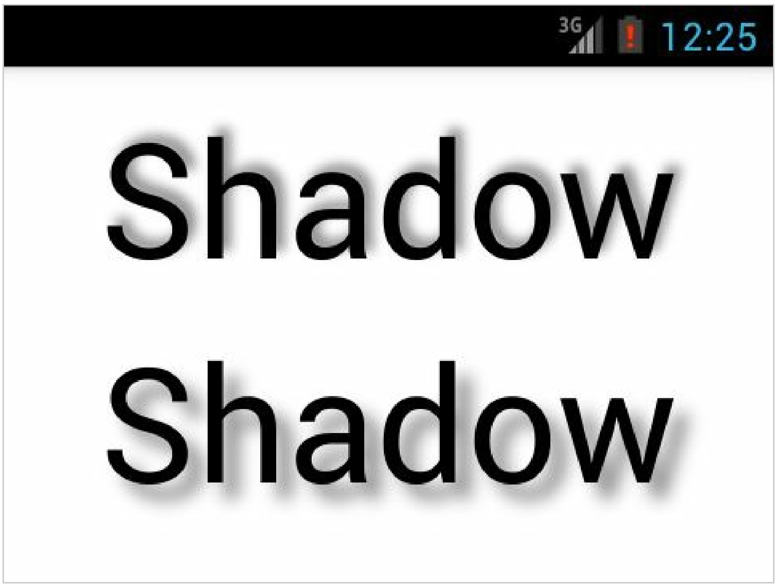


Figure 4-9.

You can experiment with different parameters, and see which values work best with a particular typeface and font size.

5. Rectangles

Canvas API supports two types of rectangles: regular rectangles and rectangles with rounded corners. There are several methods to draw a regular rectangle:

```
void drawRect(float left, float top, float right, float bottom, Paint paint
void drawRect(RectF rect, Paint paint)
void drawRect(Rect rect, Paint paint)
```

and one method to draw a rectangle with rounded corners

```
void drawRoundRect(RectF rect, float rx, float ry, Paint paint)
```

The parameters of all four methods are listed below.

- **left** - the x-coordinate of the left side of the rectangle to be drawn.
- **top** - the y-coordinate of the top side of the rectangle to be drawn.
- **right** - the x-coordinate of the right side of the rectangle to be drawn.
- **bottom** - the y-coordinate of the bottom side of the rectangle to be drawn.
- **rect** - the rectangle to be drawn.
- **rx** - the x-radius of the oval used to round the corners.
- **ry** - the y-radius of the oval used to round the corners.
- **paint** - the paint used to draw the rectangle.

Note that you can also use Path API to draw rectangles.

5.1. Example: Bar Chart

In this example we will draw a bar chart shown in Figure 5-1.

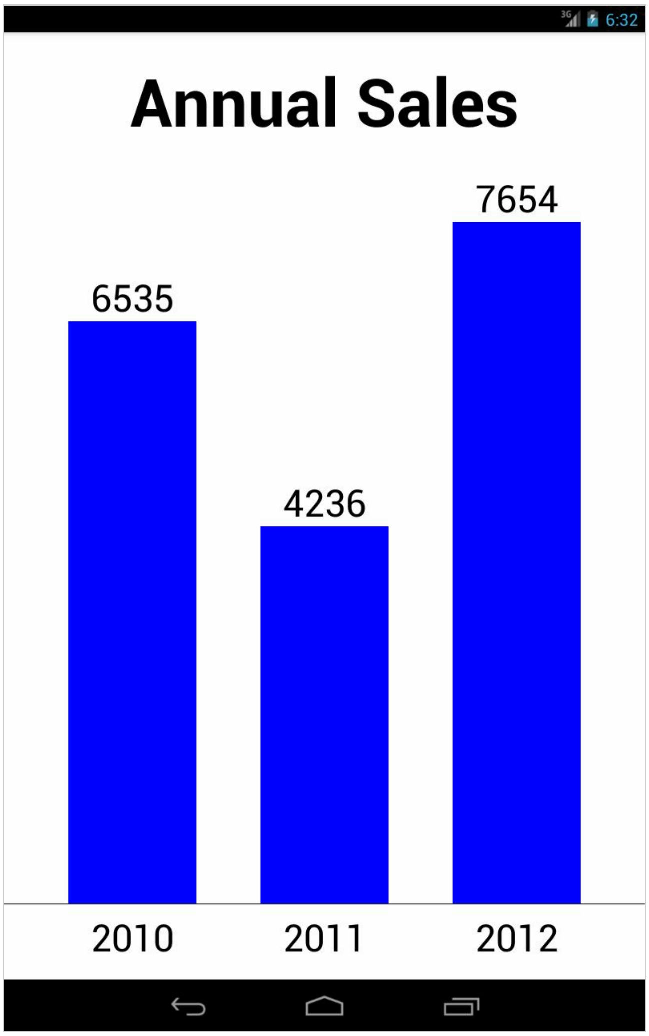


Figure 5-1.

It is a good practice to separate data and its representation. In our application the BarChartModel class provides data and the BarChartView class displays it.

Model

The BarChartModel class is shown below.

```
package com.example.graphics2d;

public class BarChartModel
{
    private String labels[] = { "2010", "2011", "2012" };
    private float values[] = { 6535, 4236, 7654 };
    private float maxValue;


    public BarChartModel()
    {
        // Calculate max value
        maxValue = values[0];
        for(int i = 1; i < values.length; i++)
        {
            if(values[i] > maxValue)
            {
                maxValue = values[i];
            }
        }
    }


    public int size()
    {
        return values.length;
    }


    public String getLabel(int index)
    {
        return labels[index];
    }


    public float getValue(int index)
    {
        return values[index];
    }


    public float maxValue()
    {
        return maxValue;
    }
}
```

The BarChartModel class has the following methods:

- **size()** - returns number of elements or bars in the model.
- **getLabel()** - returns a label for each bar. In our example, it is a year.
- **getValue()** - returns annual sales value for each bar. This value is used to calculate bar height.
- **maxValue()** - returns the maximum value. We will need this value to calculate bar height.

View

The bar chart has several components: a title, labels and the plot area - a part of the view where we draw bars or rectangles. A mock-up of the bar chart view with all its components is shown in Figure 5-2.

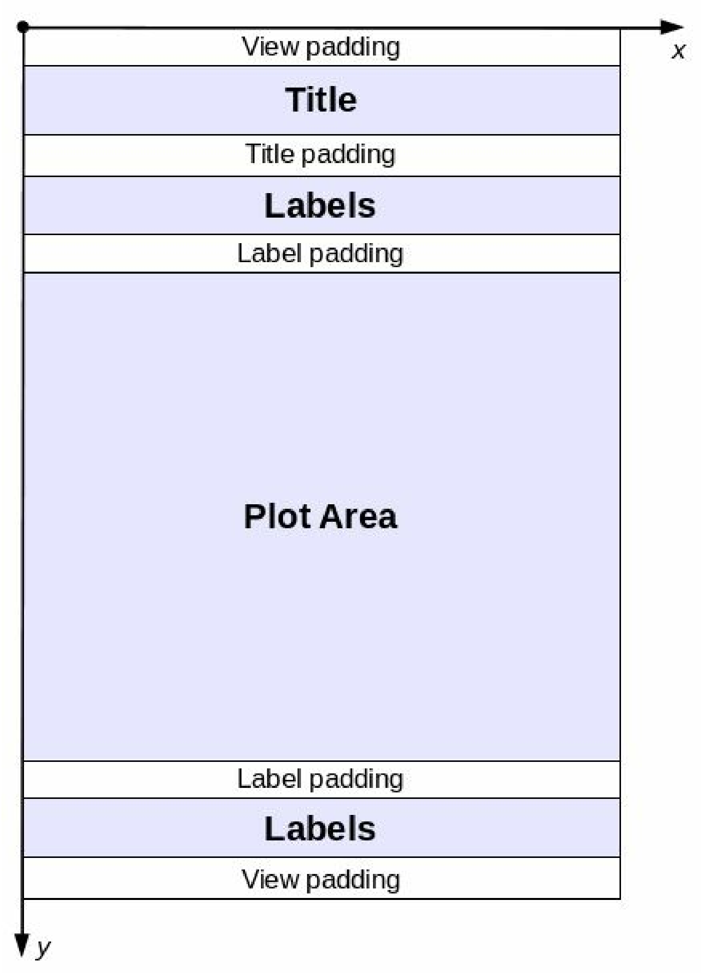


Figure 5-2.

We will calculate coordinates of all components at run-time as ratios or percentages of the view size.

Let's look at the code first and then do a review.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.graphics.Paint.Style;
import android.graphics.RectF;
import android.graphics.Typeface;
import android.view.View;


public class BarChartView extends View
{
    private Paint barPaint;
    private Paint titleTextPaint;
    private Paint labelTextPaint;

    float totalWidth;
    float barWidth;
    float barPading;

    float labelPading;

    float titleY;
    float labelY;
    float plotAreaBottomY;
    float plotAreaHeight;

    private BarChartModel model;
    private RectF bar;


    public BarChartView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        barPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        barPaint.setColor(Color.BLUE);
        barPaint.setStyle(Style.FILL);

        titleTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        titleTextPaint.setColor(Color.BLACK);
        titleTextPaint.setTextAlign(Align.CENTER);
        titleTextPaint.setTypeface(Typeface.DEFAULT_BOLD);

        labelTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        labelTextPaint.setColor(Color.BLACK);
        labelTextPaint.setTextAlign(Align.CENTER);

        model = new BarChartModel();
        bar = new RectF();
    }


    public float getBarLeftX(int barIndex)
    {
        return barPading * (barIndex+1) + barWidth * barIndex;
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        // Title
        canvas.drawText("Annual Sales", totalWidth/2, titleY, titleTextPain

        // Horizontal line below bars
        canvas.drawLine(0, plotAreaBottomY, totalWidth, plotAreaBottomY, la

        // Bars
        for(int i = 0; i < model.size(); i++)
        {
            // Bar coordinates
            float leftX = getBarLeftX(i);
            float rightX = leftX + barWidth;
            float topY = plotAreaBottomY - plotAreaHeight * model.getValue(

            // Bar
            bar.left = leftX;
            bar.right = rightX;
            bar.top = topY;
            bar.bottom = plotAreaBottomY;
            canvas.drawRect(bar, barPaint);

            // Label
            float labelX = (leftX + rightX)/2;
            canvas.drawText(model.getLabel(i), labelX, labelY, labelTextPai

            // Value Text
            String valueText = String.format("%.0f", model.getValue(i));
            canvas.drawText(valueText, labelX, topY-labelPading, labelTextP
        }
    }


    @Override
    protected void onSizeChanged(int viewWidth, int viewHeight, int oldw, i
    {
        totalWidth = viewWidth;

        float viewPading = viewHeight * 0.03f;
        float titlePading = viewHeight * 0.05f;
        labelPading = viewHeight * 0.01f;

        barPading = viewWidth * 0.1f;

        float titleTextSize = viewHeight * 0.07f;
        float labelTextSize = viewHeight * 0.04f;

        titleY = viewPading + titleTextSize;
        labelY = viewHeight - viewPading;

        float plotAreaTopY = viewPading + titleTextSize + titlePading + lab
        plotAreaBottomY = viewHeight - viewPading - labelTextSize - labelPa
        plotAreaHeight = plotAreaBottomY - plotAreaTopY;

        // Bar width
        int numBars = model.size();
        barWidth = (totalWidth - (numBars + 1) * barPading) / numBars;

        // Font sizes
        titleTextPaint.setTextSize(titleTextSize);
        labelTextPaint.setTextSize(labelTextSize);
    }

}
```

Code Review: Class Variables

We have a lot of variables in this example. There are several Paint objects for bars and text.

```
private Paint barPaint;
private Paint titleTextPaint;
private Paint labelTextPaint;
```

The following variables are used to store the width, padding and y-coordinates of different components of the chart. These variables are set in onSizeChanged() method and then used in onDraw() method to draw the chart.

```
float totalWidth;
float barWidth;
float barPading;

float labelPading;

float titleY;
float labelY;
float plotAreaBottomY;
float plotAreaHeight;
```

This is the model for our chart. It will be created in the constructor.

```
private BarChartModel model;
```

Remember that mobile devices have limited resources. You should avoid creating too many objects to save memory. We will reuse one RectF object for all bars.

```
private RectF bar;
```

Code Review: Constructor

In the constructor we set view background color, created paint objects and the chart model.

Code Review: onSizeChanged()

In this method we calculated the width, padding and y-coordinates of different components of the chart. We used percentages in our calculations. For example, the horizontal bar padding is 10% of the view width.

```
barPading = viewWidth * 0.1f;
```

The label font size is 4% of the view height.

```
float labelTextSize = viewHeight * 0.04f;
```

In this example we used the same percentages for both portrait and landscape orientations, but you can use different values in your application.

Plot Area Coordinates

The plot area top coordinate is calculated relative to the view top:

```
plotAreaTopY = viewPading + titleTextSize + titlePading + labelTextSize + l
```

The plot area bottom coordinate is calculated relative to the view bottom:

```
plotAreaBottomY = viewHeight - viewPading - labelTextSize - labelPading;
```

Bar Width

If you look at Figure 4-8 you will see that there are 3 bars and 4 horizontal paddings (spaces). Horizontal padding is fixed. It is 10% of the view width.

```
barPading = w * 0.1f;
```

The total padding is

```
totalPadding = (numBars + 1) * barPading
```

For 3 bars, the total padding as percentage of the view width is:

```
totalPadding = (3 + 1) * 10% = 40%
```

To get the bar width we have to subtract the total padding from the view width (100%) and divide the result by the number of bars (3):

```
barWidth = (100% - 40%) / 3 = 20%
```

The generic formula used in our example is shown below.

```
barWidth = (totalWidth - (numBars + 1) * barPading) / numBars;
```

Code Review: onDraw()

In this method we drew all components of the chart. We started with the title.

```
canvas.drawText("Annual Sales", totalWidth/2, titleY, titleTextPaint);
```

Then we drew a horizontal line below bars (see Figure 4-8).

```
canvas.drawLine(0, plotAreaBottomY, totalWidth, plotAreaBottomY, labelTextP
```

Finally we iterated over bar data, calculated x- and y-coordinates of each bar, drew the bar and labels.

Bar Coordinates

The following method is used to calculate the left coordinate of each bar.

```
public float getBarLeftX(int barIndex)
{
    return barPading * (barIndex+1) + barWidth * barIndex;
}
```

The formula is very simple. We calculate number of paddings and bars in front of each bar. Note, that the barIndex starts from 0. For example, there are 2 paddings and 1 bar in front of the second bar (barIndex = 1):

```
leftX = barPading * (1+1) + barWidth * 1;
```

The right coordinate is calculated as follows:

```
rightX = leftX + barWidth;
```

Now let's talk about y-coordinates. We draw bars in the plot area of the chart view as shown in Figure 5-3.

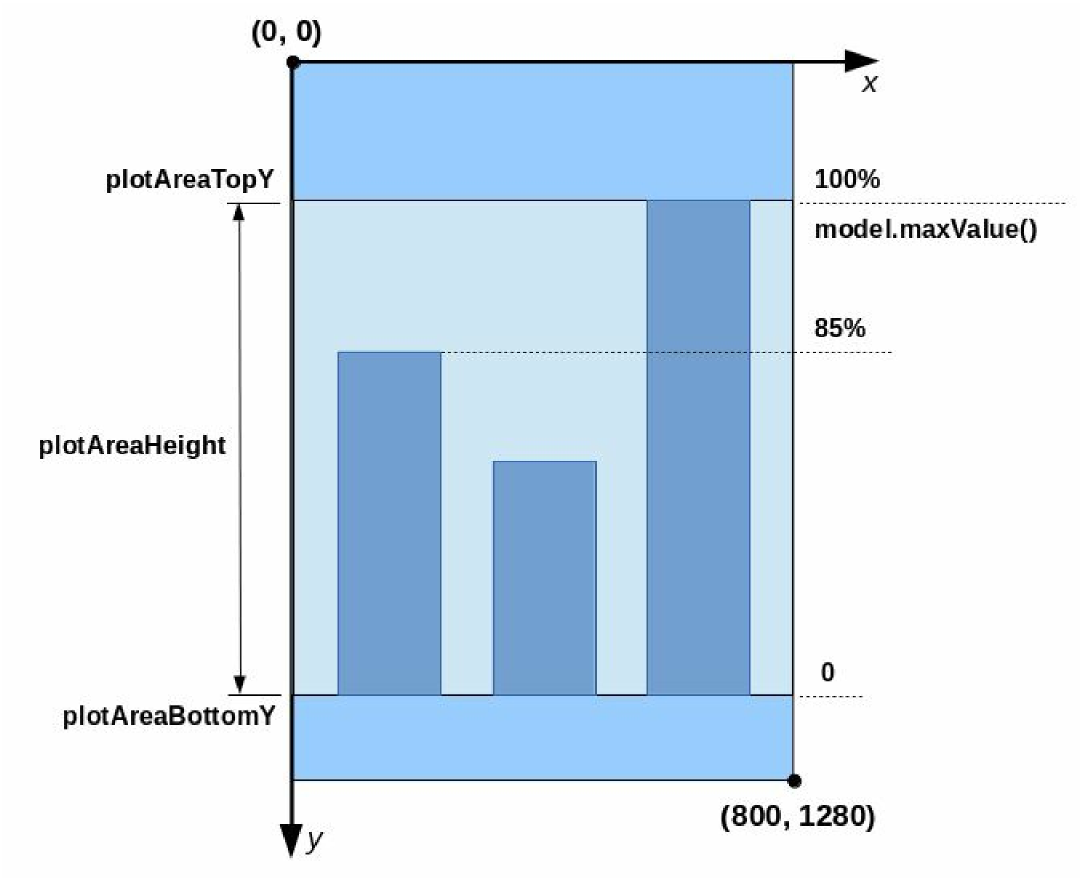


Figure 5-3.

Note that the bar chart view uses device coordinates with the origin at the upper-left corner of the view. The x-coordinate increases to the right, and the y-coordinate increases downward.

The bottom coordinate of each bar is equal to the plot area bottom coordinate. The top coordinate depends on the bar value (sales). Our bar chart model has maxValue() method which returns the value corresponding to the tallest bar on the bar chart. The height of the tallest bar is equal to the height of the plot area. The top coordinate of the tallest bar can be calculated as follows:

```
topY = plotAreaBottomY - plotAreaHeight;
```

We can use ratios or percentages to find the height of a bar relative to the height of the tallest bar. For example, the height of the first bar is 85% of the tallest bar or the plot area height:

```
barHeight = plotAreaHeight * model.getValue(0) / model.maxValue()
          = plotAreaHeight * 6535 / 7654
          = plotAreaHeight * 0.85
```

Then the top coordinate is

```
topY = plotAreaBottomY - barHeight
     = plotAreaBottomY - plotAreaHeight * model.getValue(i) / model.maxValu
```

Labels

For each bar we drew two labels. A year labels below the bar

```
float labelX = (leftX + rightX)/2;
canvas.drawText(model.getLabel(i), labelX, labelY, labelTextPaint);
```

and value label above the bar

```
String valueText = String.format("%.0f", model.getValue(i));
canvas.drawText(valueText, labelX, topY-labelPading, labelTextPaint);
```

Screenshots

Screenshots of the application in the portrait and landscape modes are shown in Figures 5-1 and 5-4.

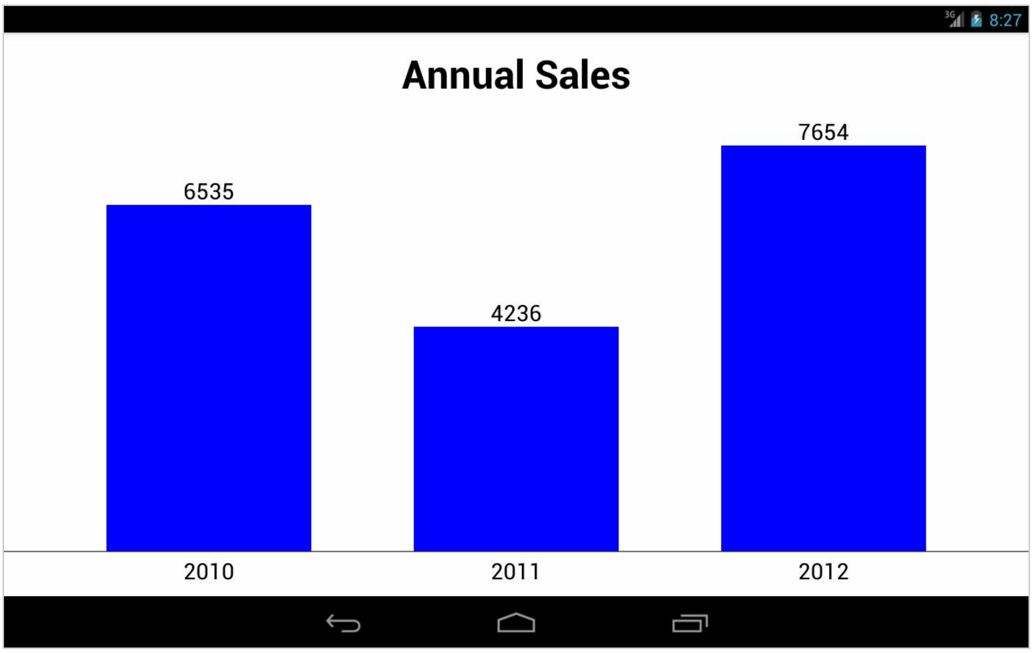


Figure 5-4.

If you replace drawRect() with the drawRoundRect() method

```
canvas.drawRoundRect(rect, 10, 10, barPaint);
```

the bars will have rounded corners, as shown in Figure 5-5.

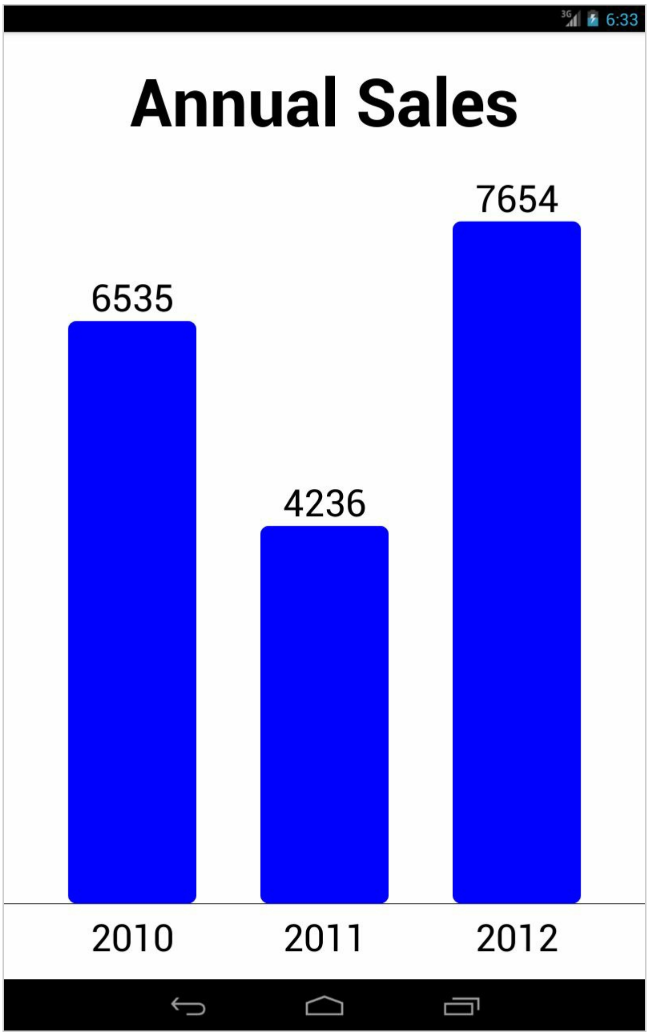


Figure 5-5.

6. Circles, Ovals and Arcs

In this chapter you will learn how to draw circles, ovals and arcs.

6.1. Circles

To draw a circle you can use the drawCircle method of the Canvas class. A circle will be filled or stroked based on the style of a paint.

```
void drawCircle(float cx, float cy, float radius, Paint paint)
```

The drawCircle method has the following parameters:

- **cx, cy** - the x- and y-coordinate of the center of the circle.
- **radius** - the radius of the circle.
- **paint** - a paint object.

The example below shows how to draw a circle.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.view.View;


public class CircleView extends View
{
    private int centerX;
    private int centerY;
    private float radius;

    private Paint paint;


    public CircleView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(0xff000099);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        paint.setStrokeWidth(radius * 0.3f);
        canvas.drawCircle(centerX, centerY, radius, paint);
    }
    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        centerX = w / 2;
        centerY = h / 2;
        radius = Math.min(w, h) * 0.7f / 2;
    }

}
```

Let's review the code.

In the constructor we set the view background color and created a Paint.

In onSizeChanged() method we calculated coordinates of the center of the circle

```
centerX = w / 2;
centerY = h / 2;
```

and the circle radius

```
radius = Math.min(w, h) * 0.7f / 2;
```

We wanted the circle diameter to be 70% of the short side of the view. Therefore the radius is 70% / 2.

In onDraw() method we set the line (stroke) width to 30% of the radius and drew a circle.

```
paint.setStrokeWidth(radius * 0.3f);
canvas.drawCircle(centerX, centerY, radius, paint);
```

A screenshot of the application is shown in Figure 6-1.

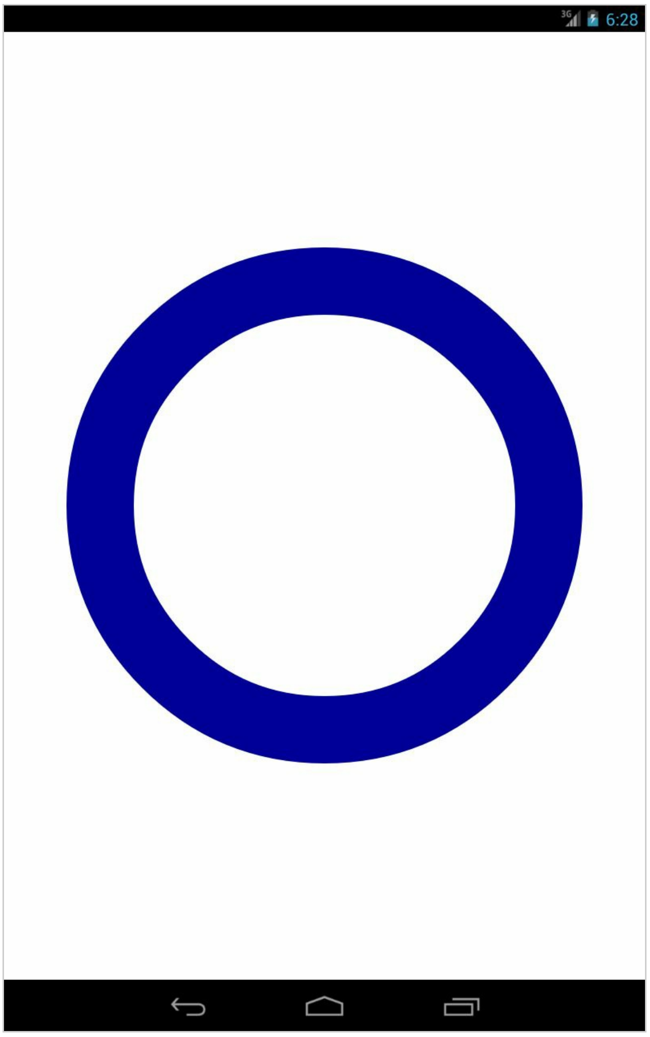


Figure 6-1.

6.2. Ovals

The drawOval method of the Canvas class can be used to draw both ovals and circles.

```
void drawOval(RectF rect, Paint paint)
```

The drawOval method has the following parameters:

- **rect** - a rectangle that defines the bounds of the oval. If the width and height of the rectangle are equal, a circle is drawn. The rectangle is represented by the coordinates of its four edges: left, right, top and bottom. Left and right are x-coordinates, and top and bottom are y-coordinates.
- **paint** - a paint object.

In the following example we will draw both an oval and a rectangle that defines the bounds of the oval.

```
package com.example.graphics2d;
```

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.RectF;
import android.view.View;


public class OvalView extends View
{
    private Paint paint;
    private float padding;
    private RectF rect;


    public OvalView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);

        rect = new RectF();
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        paint.setStrokeWidth(padding);
        paint.setColor(Color.RED);
        canvas.drawOval(rect, paint);

        paint.setStrokeWidth(padding / 12);
        paint.setColor(0xff000077);
        canvas.drawRect(rect, paint);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        padding = Math.max(width, height) * 0.1f;

        rect.left = padding;
        rect.right = width - padding;
        rect.top = padding;
        rect.bottom = height - padding;
    }

}
```

Let's review the code. In the constructor we set the view background color and created a paint and a rectangle.

In onSizeChanged() method we set coordinates of the rectangle that defines the bounds of the oval.

```
padding = Math.max(width, height) * 0.1f;
```

```
rect.left = padding;
rect.right = width - padding;
rect.top = padding;
rect.bottom = height - padding;
```

In onDraw() method we drew the oval first

```
paint.setStrokeWidth(padding);
paint.setColor(Color.RED);
canvas.drawOval(rect, paint);
```

and then the rectangle.

```
paint.setStrokeWidth(padding / 12);
paint.setColor(0xff000077);
canvas.drawRect(rect, paint);
```

A screenshot of the application is shown in Figure 6-2.

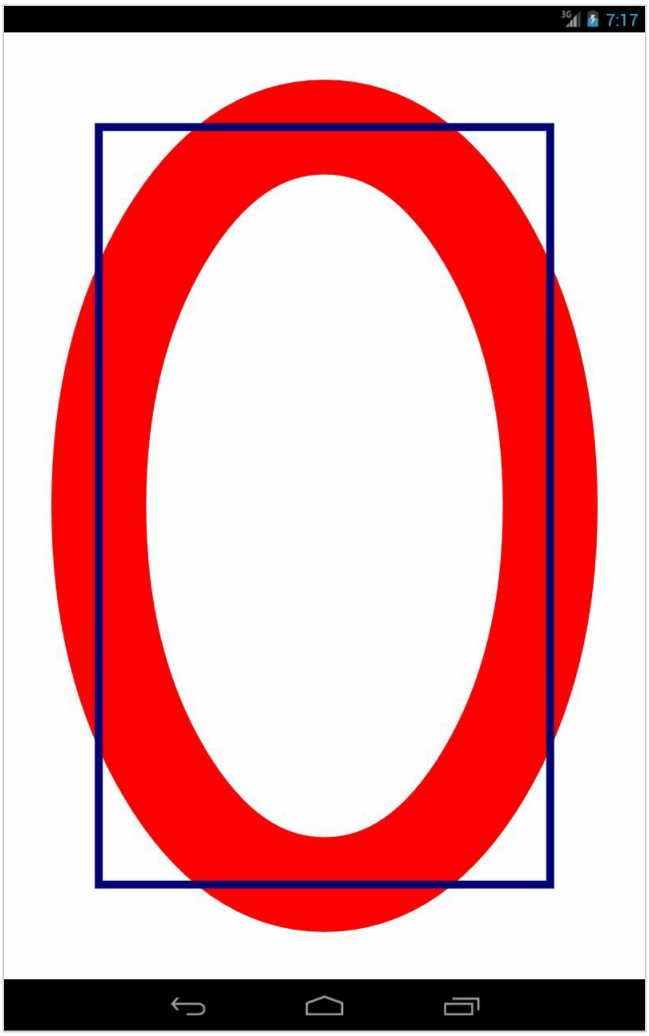


Figure 6-2.

We used different line width for the rectangle and the oval to show you how the oval is positioned relative to its bounding rectangle. As you can see from Figure 6-2, the oval goes half line width outside the bounding rectangle.

6.3. Arcs

Arcs are oval segments. The drawArc method of the Canvas class is used to draw arcs.

```
void drawArc(RectF rect, float startAngle, float sweepAngle, boolean useCen
```

The drawArc method has the following parameters:

- **rect** - a bounding rectangle which defines the shape and size of the arc.
- **startAngle** - an angle in degrees where the arc begins. An angle of zero degrees corresponds to 3 o'clock on a watch. Both positive or negative values can be used. For example, if you specify a value of -90, the arc will begin at 12 o'clock.
- **sweepAngle** - an angle in degrees measured clockwise from the startAngle parameter to ending point of the arc. Both positive or negative values can be used.
- **useCenter** - if true, a wedge is drawn. Based on the style of the paint the wedge can be filled or stroked.
- **paint** - a paint object.

The example below shows how to draw arcs.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Cap;
import android.graphics.Paint.Style;
import android.graphics.RectF;
import android.view.View;


public class ArcView extends View
{
    private int centerX;
    private int centerY;
    private float radius;
    private RectF rect;

    private Paint paint;


    public ArcView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);

        rect = new RectF();
    }
    @Override
    public void onDraw(Canvas canvas)
    {
        paint.setStyle(Style.FILL);
        paint.setColor(Color.LTGRAY);
        canvas.drawOval(rect, paint);

        paint.setStyle(Style.STROKE);
        paint.setStrokeCap(Cap.ROUND);
        paint.setColor(0xff000099);
        canvas.drawArc(rect, 0, 90, false, paint);

        paint.setStyle(Style.FILL_AND_STROKE);
        paint.setColor(0xff339933);
        canvas.drawArc(rect, 180, 90, true, paint);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        centerX = w / 2;
        centerY = h / 2;
        radius = Math.min(w, h) * 0.7f / 2;

        paint.setStrokeWidth(radius * 0.3f);

        rect.left = centerX - radius;
        rect.right = centerX + radius;
        rect.top = centerY - radius;
        rect.bottom = centerY + radius;
    }
}
```

A screenshot of the example is shown in Figure 6-3.

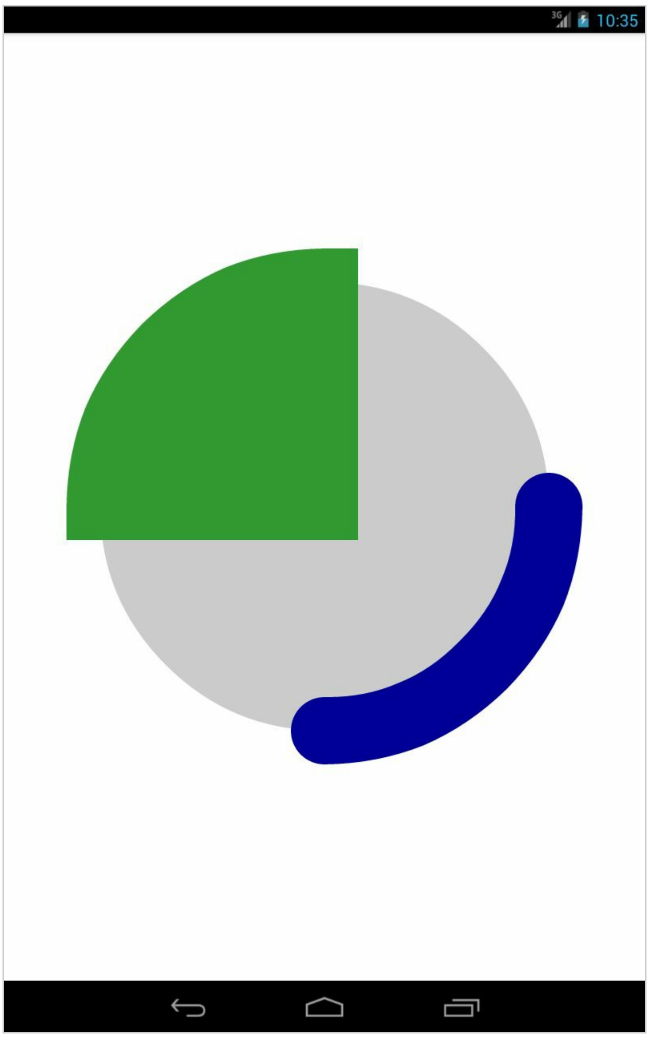


Figure 6-3.

Let's review the code. In the constructor we set the view background color and created a paint and a rectangle.

In onSizeChanged() method we set coordinates of the rectangle that defines the bounds of the circle. The circle diameter is 70% of the short side of the view and consequently the radius is 70% / 2.

```
centerX = w / 2;
centerY = h / 2;
radius = Math.min(w, h) * 0.7f / 2;

rect.left = centerX - radius;
rect.right = centerX + radius;
rect.top = centerY - radius;
rect.bottom = centerY + radius;
```

Also we set the stroke width to 30% of the circle radius

```
paint.setStrokeWidth(radius * 0.3f);
```

In onDraw() method we drew three shapes.

1. A filled circle.

```
paint.setStyle(Style.FILL);
paint.setColor(Color.LTGRAY);
canvas.drawOval(rect, paint);
```

Note that we used the drawOval() method to draw the circle.

2. An arc with round line caps.

```
paint.setStyle(Style.STROKE);
paint.setStrokeCap(Cap.ROUND);
paint.setColor(0xff000099);
canvas.drawArc(rect, 0, 90, false, paint);
```

Note how the arc extends half line width outside the bounding rectangle (the filled circle).

3. A wedge.

```
paint.setStyle(Style.FILL_AND_STROKE);
paint.setColor(0xff339933);
canvas.drawArc(rect, 180, 90, true, paint);
```

6.4. Example: Circular Chart

In this example we will draw a circular chart shown in Figure 6-4. It can be used to display statistical data or as a progress indicator.

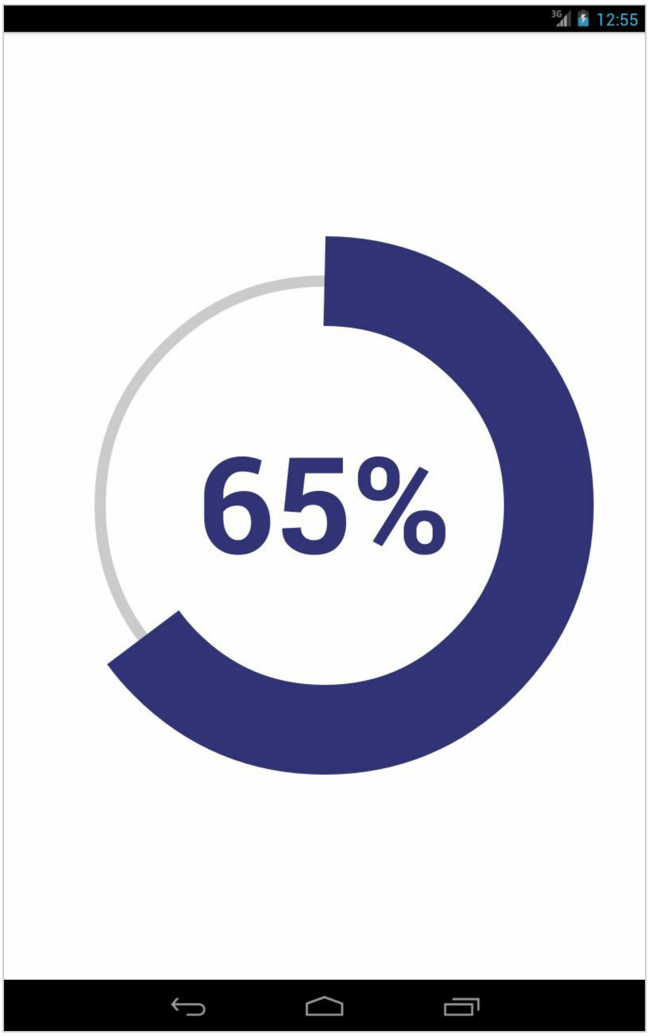


Figure 6-4.

The code of the application content view is shown below.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.Rect;
import android.graphics.RectF;
import android.graphics.Typeface;
import android.view.View;


public class CircularChart extends View
{
    private float value;
    private String label;

    private int centerX;
    private int centerY;
    private float radius;
    private float textX;
    private float textY;

    private RectF arcOval;
    private Rect textBounds;

    private Paint paint;
    private Paint txtPaint;

    private int color1 = Color.LTGRAY;
    private int color2 = 0xff333377;



    public CircularChart(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);

        txtPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        txtPaint.setColor(color2);
        txtPaint.setTypeface(Typeface.DEFAULT_BOLD);

        arcOval = new RectF();
        textBounds = new Rect();

        // Set value
        value = 0.65f;
        label = String.format("%.0f%%", value * 100);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        // Draw circle
        paint.setStrokeWidth(radius * 0.05f);
        paint.setColor(color1);
        canvas.drawCircle(centerX, centerY, radius, paint);

        // Draw arc
        paint.setColor(color2);
        paint.setStrokeWidth(radius * 0.4f);
        canvas.drawArc(arcOval, -90, value * 360, false, paint);

        // Draw label
        canvas.drawText(label, textX, textY, txtPaint);
    }


    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        // Center
        centerX = w / 2;
        centerY = h / 2;

        // Radius
        radius = Math.min(w, h) * 0.7f / 2;

        // Text size
        txtPaint.setTextSize(radius * 0.6f);
        // Center text
        txtPaint.getTextBounds(label, 0, label.length(), textBounds);
        textX = centerX - (textBounds.left + textBounds.right) * 0.5f;
        textY = centerY - (textBounds.top + textBounds.bottom) * 0.5f;

        // Arc oval
        arcOval.left = centerX - radius;
        arcOval.right = centerX + radius;
        arcOval.top = centerY - radius;
        arcOval.bottom = centerY + radius;
    }

}
```

Code Review: Constructor

First, we set the view background color and created Paint objects.

Then we created two temporary rectangles to draw an arc

```
arcOval = new RectF();
```

and to measure text.

```
textBounds = new Rect();
```

These rectangles are used in onSizeChanged() method.

Finally, we set the chart value and its corresponding label (65%).

```
value = 0.65f;
label = String.format("%.0f%%", value * 100);
```

Code Review: onSizeChanged()

First, we calculated the x- and y-coordinates of the center of the view

```
centerX = w / 2;
centerY = h / 2;
```

and the radius of the circle

```
radius = Math.min(w, h) * 0.7f / 2;
```

We wanted the circle diameter to be 70% of the short side of the view. Therefore the radius is 70% / 2.

Then we set a text size to 60% of the radius.

```
txtPaint.setTextSize(radius * 0.6f);
```

After that we measured the label size

```
txtPaint.getTextBounds(label, 0, label.length(), textBounds);
```

and calculated the x- and y-coordinates to center-align the label horizontally and vertically.

```
textX = centerX - (textBounds.left + textBounds.right) * 0.5f;
textY = centerY - (textBounds.top + textBounds.bottom) * 0.5f;
```

We explained these formulas in chapter 3 when discussed horizontal (section 3.3) and vertical (section 3.4) text alignment.

Finally we set the arc coordinates.

```
arcOval.left = centerX - radius;
arcOval.right = centerX + radius;
arcOval.top = centerY - radius;
arcOval.bottom = centerY + radius;
```

Code Review: onDraw()

First we drew a circle to outline the chart.

```
paint.setStrokeWidth(radius * 0.05f);
paint.setColor(color1);
canvas.drawCircle(centerX, centerY, radius, paint);
```

Then we drew an arc. The sweep angle of the arc is proportional to the value it represents. In our example, the full circle (360 degrees) represents 100%. Therefore the angle corresponding to 65% is 360 * 0.65. We wanted the arc to start at 12 o'clock, therefore we used the start angle value of -90 degrees.

```
paint.setColor(color2);
paint.setStrokeWidth(radius * 0.4f);
canvas.drawArc(arcOval, -90, value * 360, false, paint);
```

Finally we drew the label.

```
canvas.drawText(label, textX, textY, txtPaint);
```

7. Gradients

So far, in our previous examples, we filled shapes with solid colors. In this chapter you will learn how to use gradients. A gradient fill is a fill that varies from one color to another. Canvas API supports several gradient types. Let's discuss most common ones.

7.1. Linear Gradients

A linear gradient varies along the line, also called gradient axis, between two points. Figure 7-1 shows an example of a linear gradient which varies along the vertical line that goes from the top of a rectangle to its bottom.

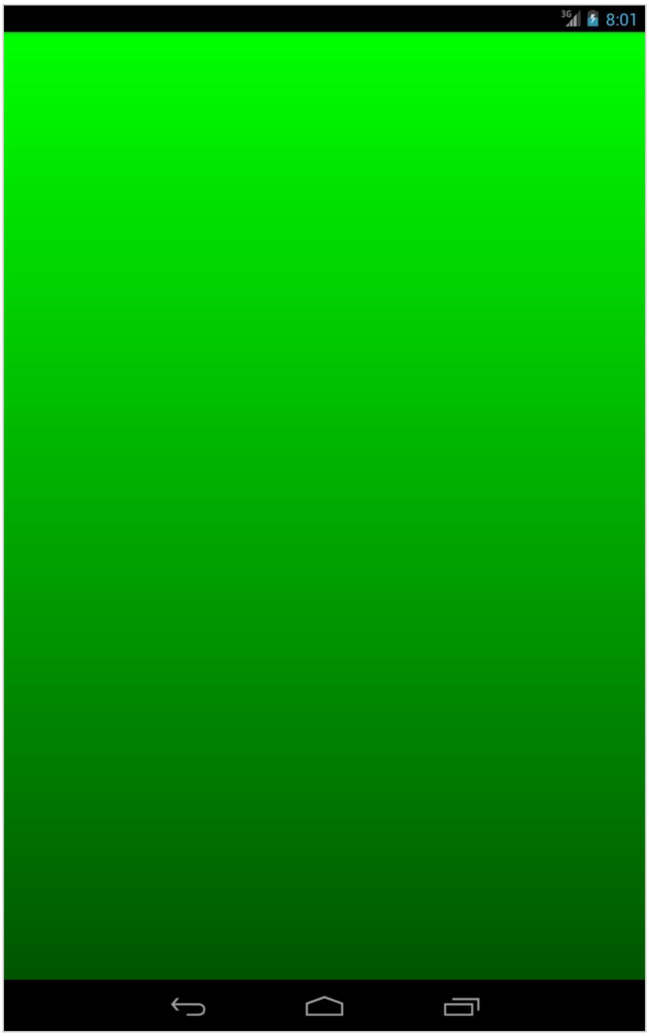


Figure 7-1.

To fill a shape with a linear gradient you have to:

1. Create an instance of the LinearGradient class.
2. Configure a Paint object to use the gradient you created.

The next several examples show how to use different linear gradients.

Example: Vertical Gradient

In this example we will draw a gradient shown in Figure 7-2. The code of the application content view is shown below.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.LinearGradient;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.Shader;
import android.view.View;


public class LinearGradientView extends View
{
    private int color1 = 0xff00FF00;
    private int color2 = 0xff005500;

    private int width, height;
    private LinearGradient gradient;
    private Paint paint;


    public LinearGradientView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.FILL);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawRect(0, 0, width, height, paint);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        this.width = width;
        this.height = height;

        gradient = new LinearGradient(0, 0, 0, height, color1, color2, Shad
        paint.setShader(gradient);
    }
}
```

Code Review

First, we defined two colors we will use to create a gradient.

```
private int color1 = 0xff00FF00;
private int color2 = 0xff005500;
```

In the constructor we set the background color of the view, created a paint object and set its style to FILL.

In onSizeChanged() method we stored the view width and height we will use to draw (fill) a rectangle.

```
this.width = width;
this.height = height;
```

Then we created an instance of the LinearGradient class.

```
gradient = new LinearGradient(0, 0, 0, height, color1, color2, Shader.TileM
```

We used the LinearGradient constructor shown below.

```
LinearGradient(float x1, float y1, float x2, float y2,
               int color1, int color2, Shader.TileMode tile)
```

The constructor has the following parameters:

- **x1, y1** - the x- and y-coordinate of the starting point of the gradient line.
- **x2, y2** - the x- and y-coordinate of the end point of the gradient line.
- **color1** - the color at the start of the gradient line.
- **color2** - the color at the end of the gradient line.
- **tile** - the tile mode specifies how to draw a gradient when the area being filled is bigger than the area defined by the gradient. In this example, both areas are the same, so this parameter does not affect the gradient. The tile mode can have one of the following values: CLAMP, MIRROR, REPEAT. We will discuss this parameter later in this section.

The gradient axis in this example is a vertical line which goes from the top of a rectangle to its bottom.

After we created the gradient, we configured the paint object to use this gradient to fill a rectangle.

```
paint.setShader(gradient);
```

Note that LinearGradient class extends the base class called Shader. The Paint class uses setShader() method to set any shader object, including LinearGradient.

Finally, in onDraw() method we filled the rectangle with the gradient created in onSizeChanged() method.

```
canvas.drawRect(0, 0, width, height, paint);
```

A screenshot of this application is shown in Figure 7-1.

Example: Horizontal Gradient

In this example we will draw a horizontal gradient shown in Figure 7-2.

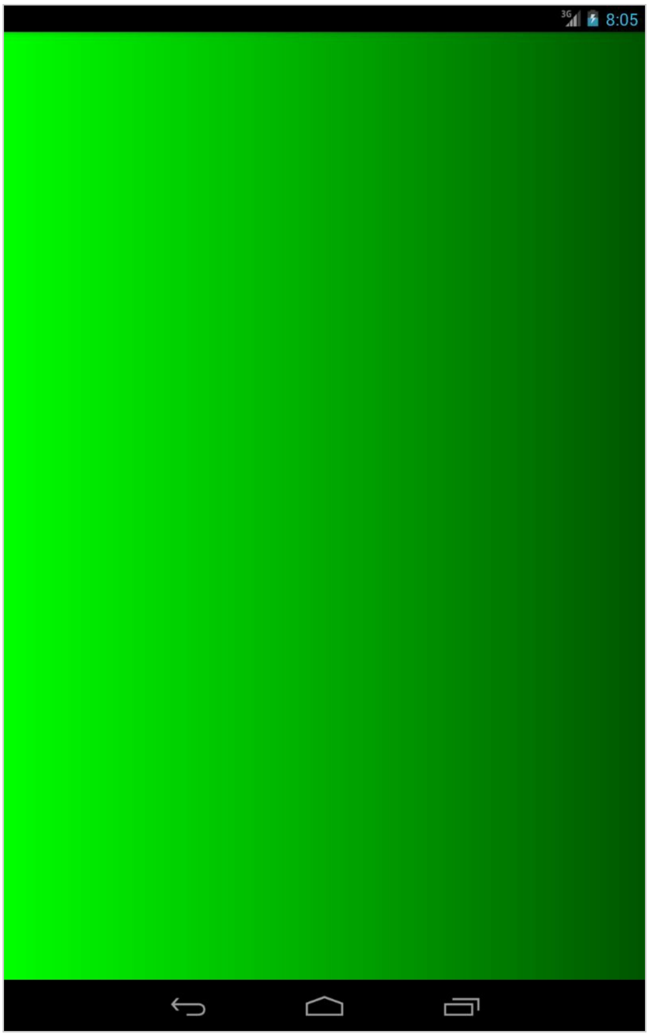


Figure 7-2.

To create this gradient we have to change the LinearGradient constructor as shown below.

```
gradient = new LinearGradient(0, 0, width, 0, color1, color2, Shader.TileMo
```

In this example we used horizontal gradient axis which goes from the left side of the rectangle to its right side.

Example: Diagonal Gradient

To draw the gradient shown in Figure 7-3 we have to change the LinearGradient constructor again.

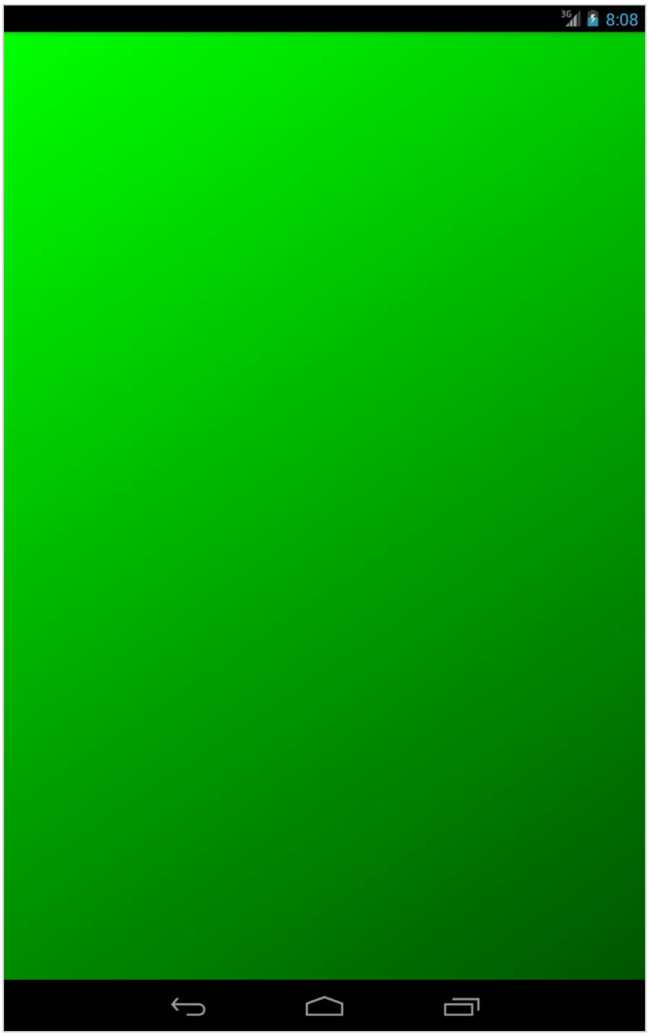


Figure 7-3.

```
gradient = new LinearGradient(0, 0, width, height, color1, color2, Shader.T
```

In this example we used the gradient axis which goes from the upper-left corner of the rectangle to its lower-right corner.

Tile Mode

The gradient tile mode specifies how to draw a gradient when the area being filled is bigger than the area defined by the gradient. The tile mode can have one of the following values:

- **CLAMP** - use the terminal colors of the gradient to fill the remainder of the region.
- **MIRROR** - mirror the same linear gradient along the gradient axis before the starting point and after the end point.
- **REPEAT** - repeat the same linear gradient along the gradient axis before the starting point and after the end point.

The next several examples show how to use different gradient tile modes.

CLAMP Mode

In this example we will draw a vertical gradient shown in Figure 7-4.

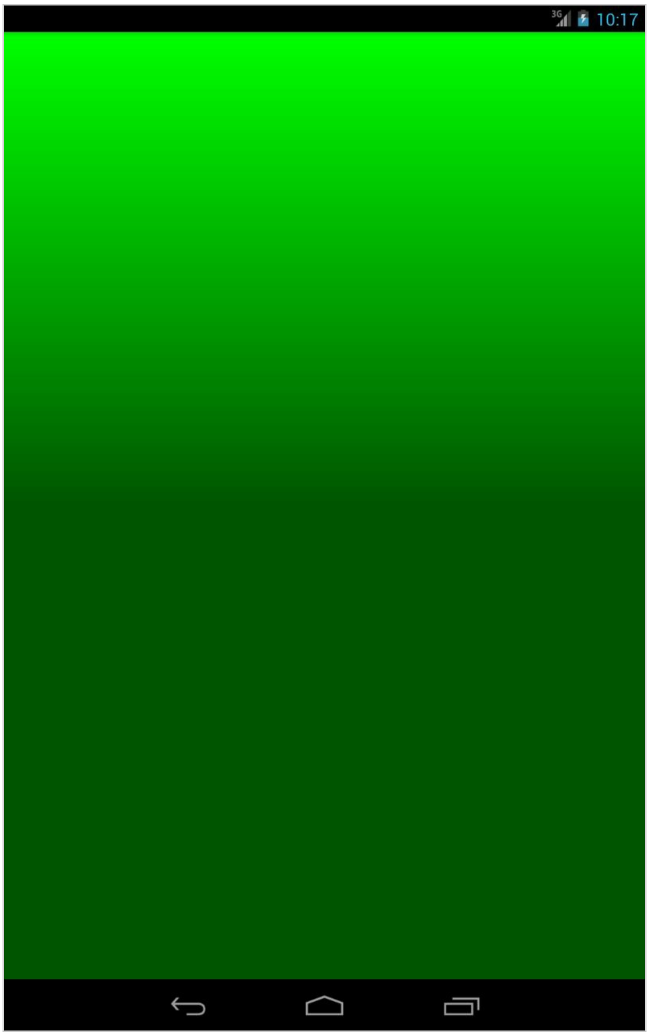


Figure 7-4.

The modified version of the constructor is shown below.

```
gradient = new LinearGradient(0, 0, 0, height/2, color1, color2, Shader.Til
```

The vertical gradient axis goes from the top of a rectangle to its center (height/2). The remainder of the rectangle (from height/2 to height) is filled with the end color (color2).

MIRROR Mode

In this example we will draw a vertical gradient shown in Figure 7-5.

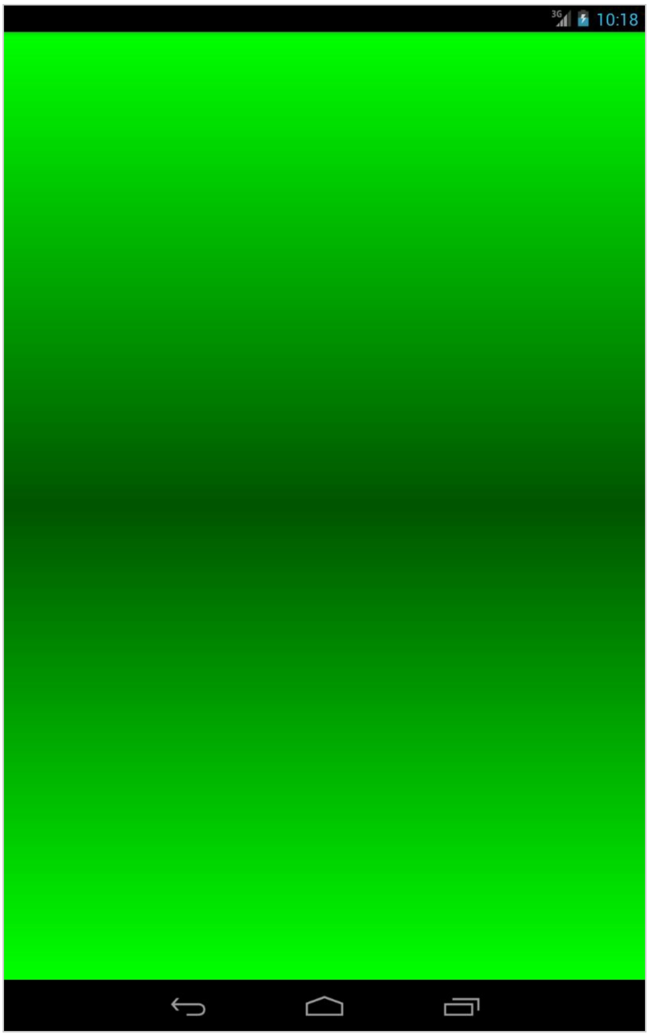


Figure 7-5.

New version of the constructor is shown below.

```
LinearGradient gradient = new LinearGradient(0, 0, 0, height/2, color1, col
```

Similar to the previous example the vertical gradient axis goes from the top of a rectangle to its center (height/2). The remainder of the rectangle (from height/2 to height) is filled with the mirrored gradient.

REPEAT Mode

In this example we will draw a vertical gradient shown in Figure 7-6.

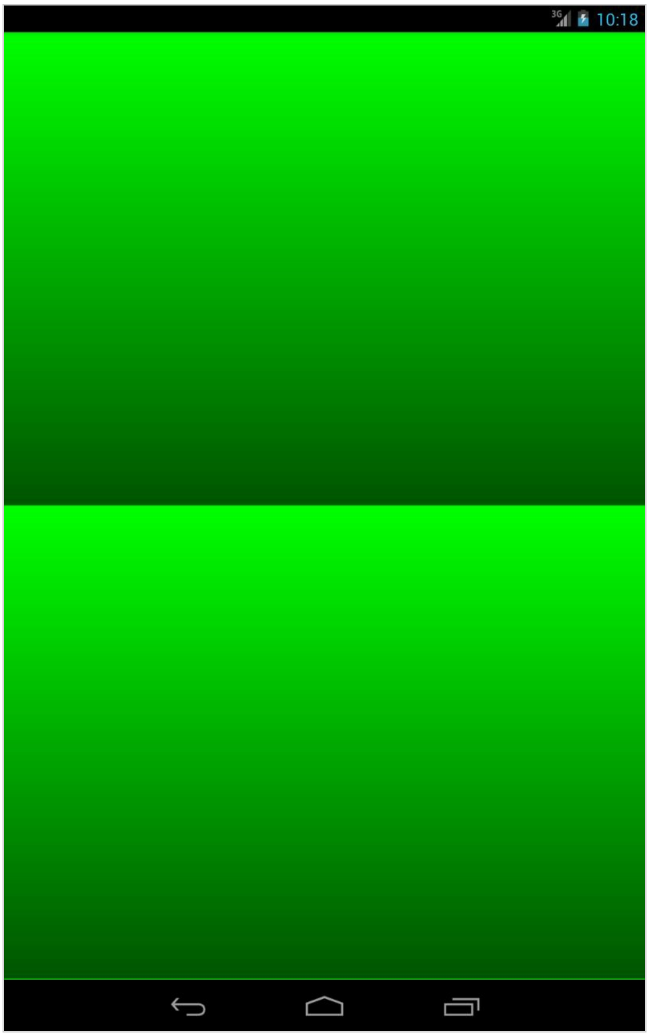


Figure 7-6.

The modified version of the constructor is shown below.

```
LinearGradient gradient = new LinearGradient(0, 0, 0, height/2, color1, col
```

Similar to the previous example the vertical gradient axis goes from the top of a rectangle to its center (height/2). The remainder of the rectangle (from height/2 to height) is filled with the same gradient.

7.2. Radial Gradients

A radial gradient is a fill that varies radially along the radius of a circle as shown in Figure 7-7. The next several examples show how to use different radial gradients.

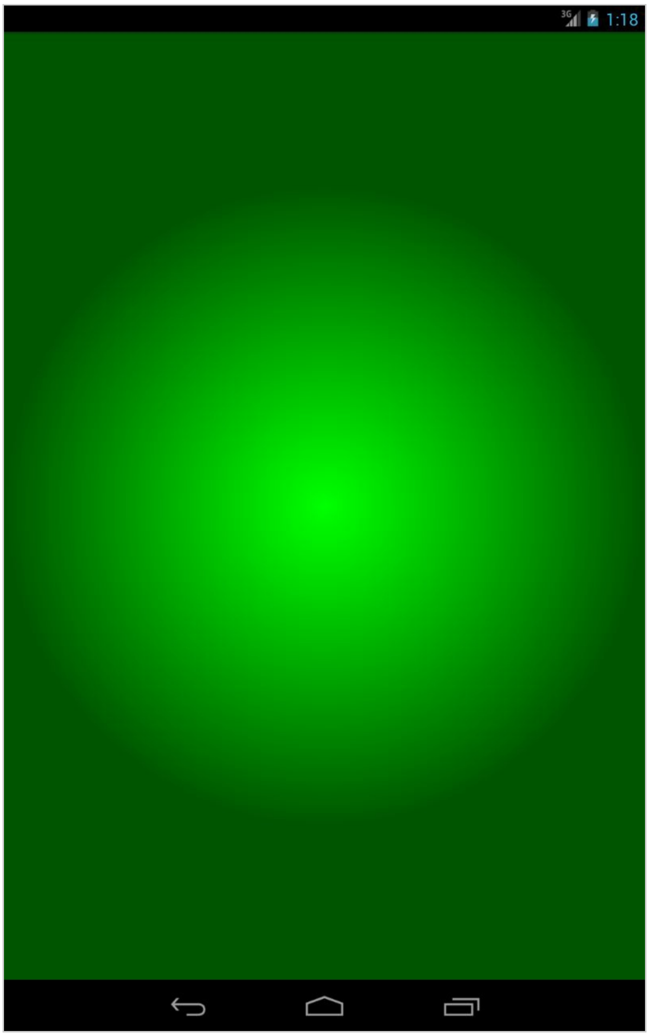


Figure 7-7.

Example: CLAMP Mode

The CLAMP tile mode is the most common for radial gradients. In this example we will draw a radial gradient shown in Figure 7-7. The code of the application content view is shown below.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.RadialGradient;
import android.graphics.Paint.Style;
import android.graphics.Shader;
import android.view.View;


public class RadialGradientView extends View
{
    private int color1 = 0xff00FF00;
    private int color2 = 0xff005500;
    private int width, height;

    private RadialGradient gradient;
    private Paint paint;


    public RadialGradientView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.FILL);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawRect(0, 0, width, height, paint);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        this.width = width;
        this.height = height;

        float centerX = width / 2.0f;
        float centerY = height / 2.0f;
        float radius = Math.min(width, height) / 2.0f;

        gradient = new RadialGradient(centerX, centerY, radius, color1, col
        paint.setShader(gradient);
    }
}
```

Code Review

We used the same colors as in linear gradient examples.

```
private int color1 = 0xff00FF00;
private int color2 = 0xff005500;
```

The constructor and onDraw() method are also the same.

In onSizeChanged() method we stored the view width and height

```
this.width = width;
this.height = height;
```

and calculated the center coordinates and the radius of the radial gradient

```
float centerX = width / 2.0f;
float centerY = height / 2.0f;
float radius = Math.min(width, height) / 2.0f;
```

Then we created an instance of the RadialGradient class.

```
gradient = new RadialGradient(centerX, centerY, radius, color1, color2, Sha
```

We used the RadialGradient constructor shown below.

```
RadialGradient(float cx, float cy, float radius, int color1, int color2, Sh
```

The constructor has the following parameters:

- **cx, cy** - the x- and y- coordinate of the center of the gradient circle.
- **radius** - the radius of the gradient circle.
- **color1** - the color at the center of the circle.
- **color2** - the color at the edge of the circle.
- **tile** - the tile mode which specifies how to draw a gradient when the area being filled is bigger than the area defined by the gradient.

After we created the gradient, we configured the paint object to use this gradient to fill a rectangle.

```
paint.setShader(gradient);
```

In this example we used CLAMP tile mode, therefore the area outside the gradient circle is filled with color2.

A screenshot of this application is shown in Figure 7-7.

Example: 3D Ball

Radial gradients can be used to create 3D effects. In this example we will draw a 3D ball shown in Figure 7-8.

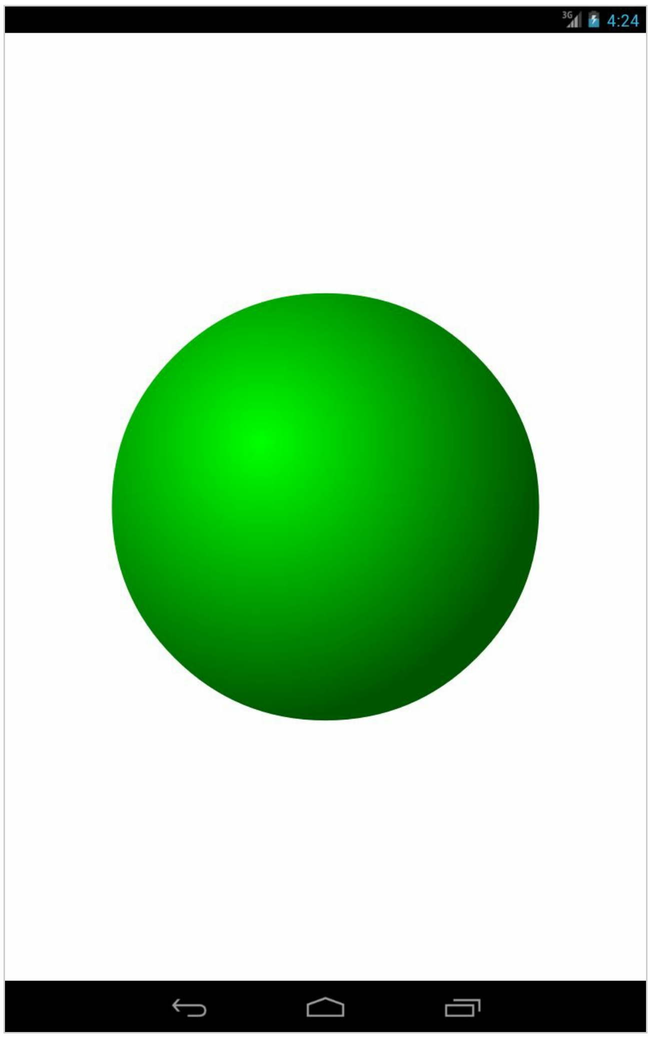


Figure 7-8.

Let's look at the code first and then review it.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.RadialGradient;
import android.graphics.Paint.Style;
import android.graphics.Shader;
import android.view.View;


public class RadialGradientView2 extends View
{
    private int color1 = 0xff00FF00;
    private int color2 = 0xff005500;

    private float centerX, centerY, radius;

    private RadialGradient gradient;
    private Paint paint;
    public RadialGradientView2(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.FILL);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawCircle(centerX, centerY, radius, paint);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        centerX = width / 2.0f;
        centerY = height / 2.0f;
        radius = Math.min(width, height) / 3.0f;

        gradient = new RadialGradient(centerX-radius*0.3f, centerY-radius*0
        paint.setShader(gradient);
    }
}
```

Code Review

We used the same colors as in previous examples.

```
private int color1 = 0xff00FF00;
private int color2 = 0xff005500;
```

The constructor is also the same.

In onSizeChanged() method we calculated coordinates of the center of the view and the circle radius.

```
centerX = width / 2.0f;
centerY = height / 2.0f;
radius = Math.min(width, height) / 3.0f;
```

Then we created an instance of RadialGradient class and configured the Paint object to use this gradient to fill the circle.

```
gradient = new RadialGradient(centerX-radius*0.3f, centerY-radius*0.3f, rad
paint.setShader(gradient);
```

Note how we shifted the center of the gradient left and up 30% of the radius relative to the center of the circle.

```
gradientCx = centerX-radius*0.3f;
gradientCy = centerY-radius*0.3f;
```

Also we increased the radius of the gradient 30% to compensate for the shift.

```
gradientRadius = radius*1.3f;
```

In onDraw() method we drew (filled) the circle.

```
canvas.drawCircle(centerX, centerY, radius, paint);
```

Example: REPEAT Mode

You can also use radial gradients to create interesting effects. In this example we will draw the intricate circular pattern shown in Figure 7-9.

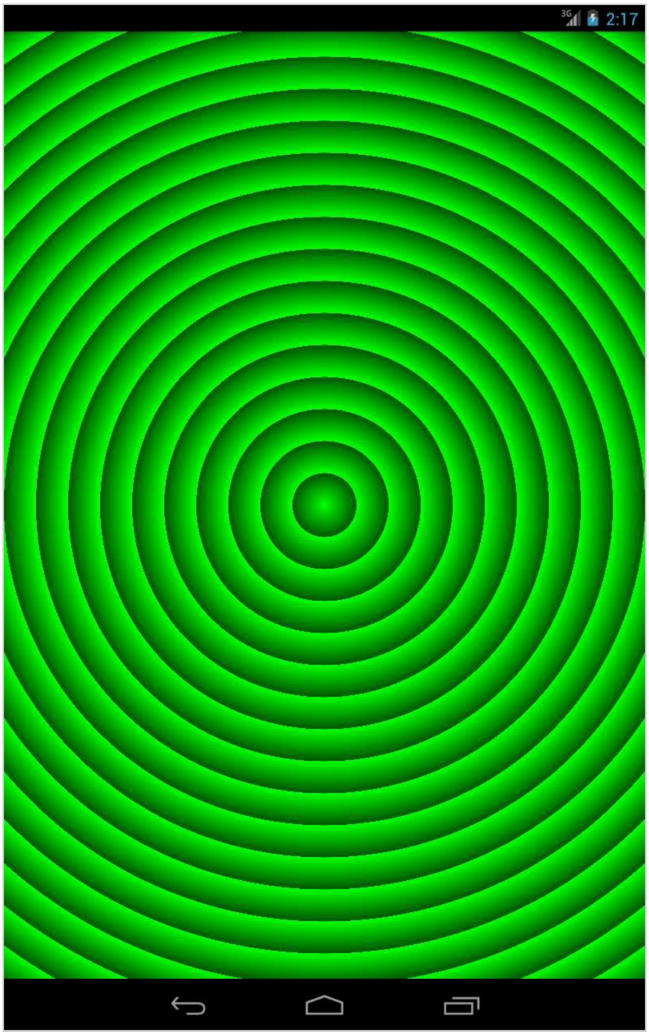


Figure 7-9.

To draw this gradient we reused the code from the first radial gradient example and modified the radius and the tile mode as shown below.

```
float radius = Math.min(width, height) / 20.0f;
gradient = new RadialGradient(centerX, centerY, radius, color1, color2, Sha
```

7.3. Sweep Gradients

A sweep gradient also known as angle gradient is a fill that varies in a clockwise sweep around the center as shown in Figure 7-10.

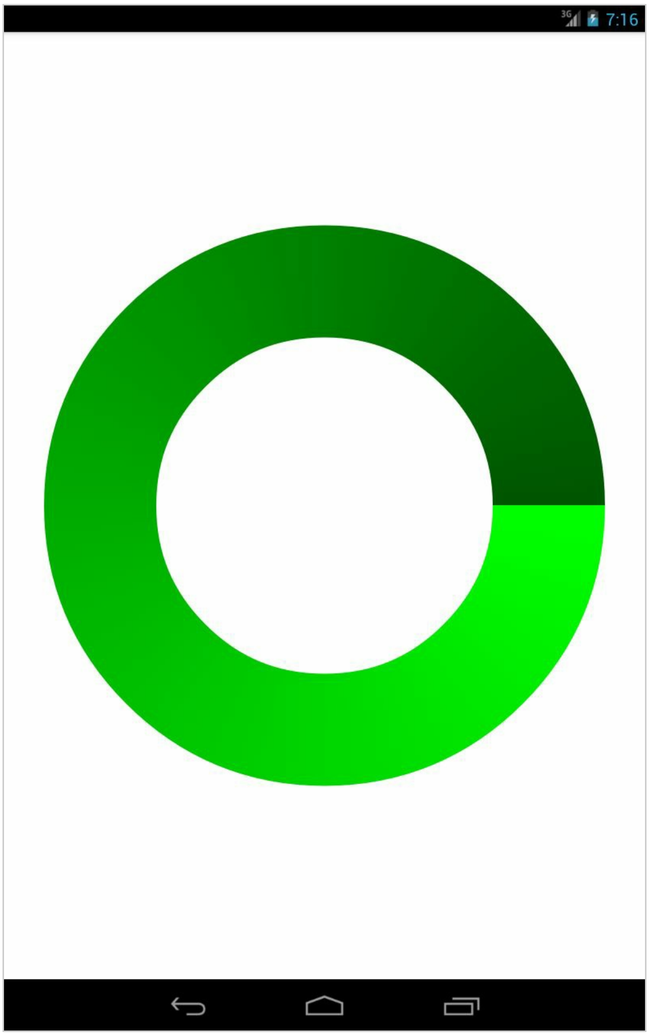


Figure 7-10.

Example

In this example we will draw a sweep gradient shown in Figure 7-10. The code of the application content view is shown below.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.SweepGradient;
import android.view.View;


public class SweepGradientView extends View
{
    private int color1 = 0xff00FF00;
    private int color2 = 0xff005500;

    private SweepGradient gradient;
    private Paint paint;

    private float centerX;
    private float centerY;
    private float radius;

    public SweepGradientView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        centerX = width / 2.0f;
        centerY = height / 2.0f;
        radius = Math.min(width, height) * 0.7f / 2.0f;

        gradient = new SweepGradient(centerX, centerY, color1, color2);

        paint.setShader(gradient);
        paint.setStrokeWidth(radius/2);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawCircle(centerX, centerY, radius, paint);
    }
}
```

Code Review

We used the same colors as in linear and radial gradient examples.

```
private int color1 = 0xff00FF00;
private int color2 = 0xff005500;
```

In the constructor we set the view background color, created a paint object and set its style to STROKE.

In onSizeChanged() method we calculated the center coordinates and the radius of a circle we will drive.

```
centerX = width / 2.0f;
centerY = height / 2.0f;
radius = Math.min(width, height) * 0.7f / 2.0f;
```

Then we created a SweepGradient object.

```
gradient = new SweepGradient(centerX, centerY, color1, color2);
```

We used the SweepGradient constructor shown below.

```
SweepGradient(float cx, float cy, int color1, int color2)
```

The constructor has the following parameters:

- **cx, cy** - the x- and y-coordinate of the gradient center.
- **color1** - the color at the start of the sweep.
- **color2** - the color at the end of the sweep.

After we created the gradient, we configured the paint object to use this gradient to draw a circle.

```
paint.setShader(gradient);
```

Finally we set the line width to 50% of the radius.

```
paint.setStrokeWidth(radius/2);
```

In onDraw() method we drew the circle.

```
canvas.drawCircle(centerX, centerY, radius, paint);
```

A screenshot of the application is shown in Figure 7-10. As you can see from the screenshot, the sweep starts at 0 degrees (3 o'clock) and the SweepGradient constructor does not have any parameters to change this value. Fortunately there is a workaround.

Custom Sweep Angle

You can change the sweep angle of a SweepGradient by modifying its local transformation matrix. We will discuss coordinate transformations and transformation matrices in Chapters 8 and 9. If you want, you can read those chapters first to better understand this section.

In this example we will use a sweep angle of -45 degrees to draw a gradient shown in Figure 7-11.

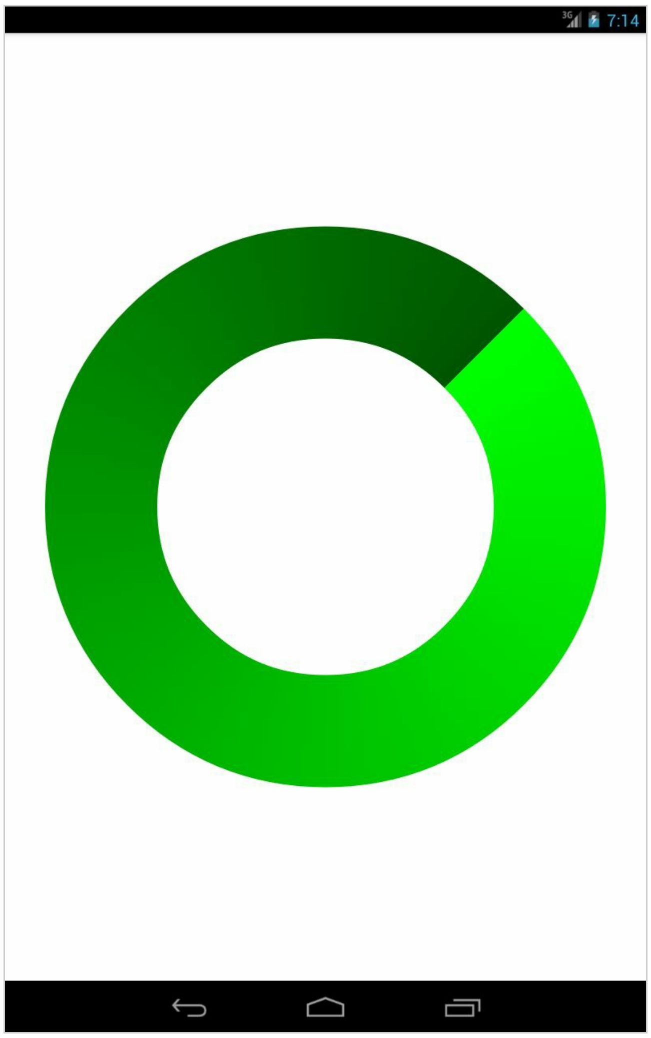


Figure 7-11.

We will reuse the code from the previous example and modify its onSizeChanged() method as shown below.

```
@Override
protected void onSizeChanged(int width, int height, int oldw, int oldh)
{
    centerX = width / 2.0f;
    centerY = height / 2.0f;
    radius = Math.min(width, height) * 0.7f / 2.0f;

    gradient = new SweepGradient(centerX, centerY, color1, color2);

    Matrix matrix = new Matrix();
    matrix.postTranslate(-centerX, -centerY);
    matrix.postRotate(-45);
    matrix.postTranslate(centerX, centerY);
    gradient.setLocalMatrix(matrix);

    paint.setShader(gradient);
    paint.setStrokeWidth(radius/2);
}
```

Let's review the changes. After creating the gradient we changed its transformation matrix to rotate the gradient 45 degrees counterclockwise.

```
Matrix matrix = new Matrix();
matrix.postTranslate(-centerX, -centerY);
matrix.postRotate(-45);
matrix.postTranslate(centerX, centerY);
gradient.setLocalMatrix(matrix);
```

We used 3 transformation operations:

1. Move (translate) the origin of the local coordinate system to (centerX, centerY).

```
matrix.postTranslate(-centerX, -centerY);
```

2. Rotate the coordinate system 45 degrees counterclockwise (negative value) around the new origin (centerX, centerY).

```
matrix.postRotate(-45);
```

3. Move the origin back.

```
matrix.postTranslate(centerX, centerY);
```

8. Introduction to Coordinate Transformations

In Chapter 2 you learned that there are two types of coordinate systems used in computer graphics: device or screen coordinate systems and logical coordinate systems. Many computer graphics libraries, including Android Canvas API, use device coordinate system by default. Usually it is very difficult to work with screen coordinates directly. That is why many programmers use different logical coordinate systems for different tasks. For example, to draw a stock chart, it is more convenient to use a logical coordinate system with days on horizontal axis and stock prices on vertical axis.

In this chapter we will discuss the math behind coordinate transformations and learn how to transform coordinates without matrices. I hope that information from this chapter will help you better understand transformation matrices discussed in Chapter 9.

8.1. Transformation Operations

There are several different ways to change coordinates, such as translation or move, scaling, and rotation. In this chapter we will talk about translation and scaling only. Translation, scaling, and rotation are called simple transformation operations. Several simple transformations can be applied sequentially to achieve more complex transformations.

Translation

Translation also known as move, involves shifting the origin of the coordinate system horizontally and vertically by a specific amount. Transformation formulas for x and y coordinates are shown below.

```
newX = moveX + x;
newY = moveY + y;
```

Scaling

Scaling lets you stretch or shrink coordinates along the x and y axes independently. Transformation formulas for x and y coordinates are shown below.

```
newX = scaleX * x;
newY = scaleY * y;
```

Now we have enough information to try several examples.

8.2. Example: Triangle

In this example we will create a simple Android application to draw a triangle.

First, let's define a logical coordinate system for the triangle. We will use ratios or percentages for the width and height of a view. This type of coordinate system is called normalized coordinate system. Also, our logical coordinate system has the origin in the lower-left corner. Figure 8-1 shows an example of a device coordinate system with dimensions of 1280x800 pixels and the corresponding logical coordinate system.

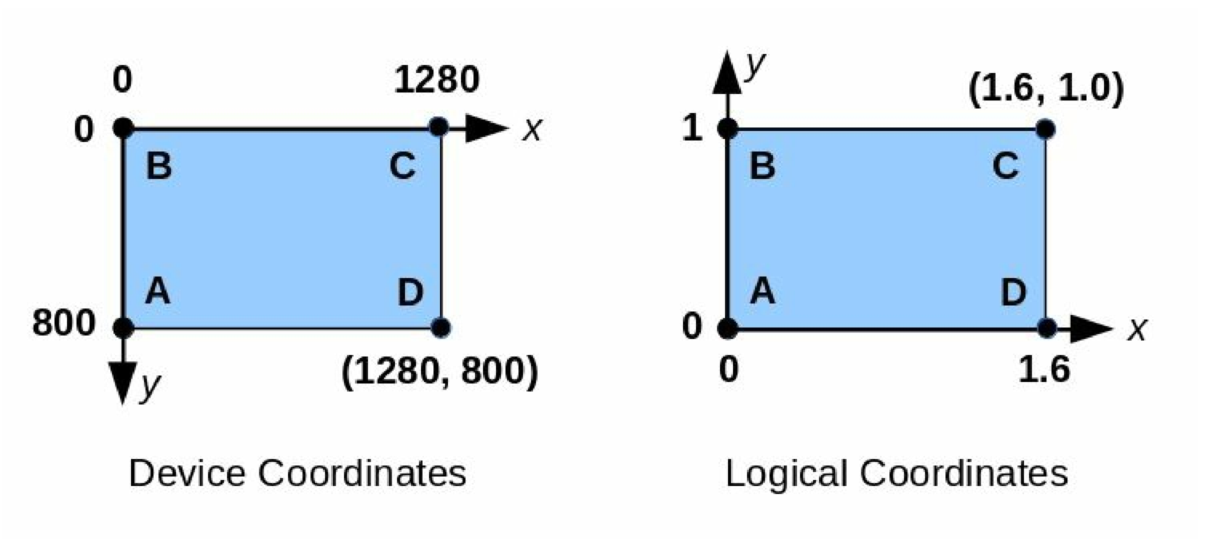


Figure 8-1.

As you can see, the normalized height is 1 or 100% and the normalized width is 1.6 (1280/800) or 160%. Figure 8-1 also shows four points A, B, C, D and its coordinates in both the device and logical coordinate systems.

By default, all drawing functions of Canvas API use device coordinates, so we have to convert all logical coordinates to device coordinates. Figure 8-2 shows how our logical coordinate system can be transformed into the device coordinate system.

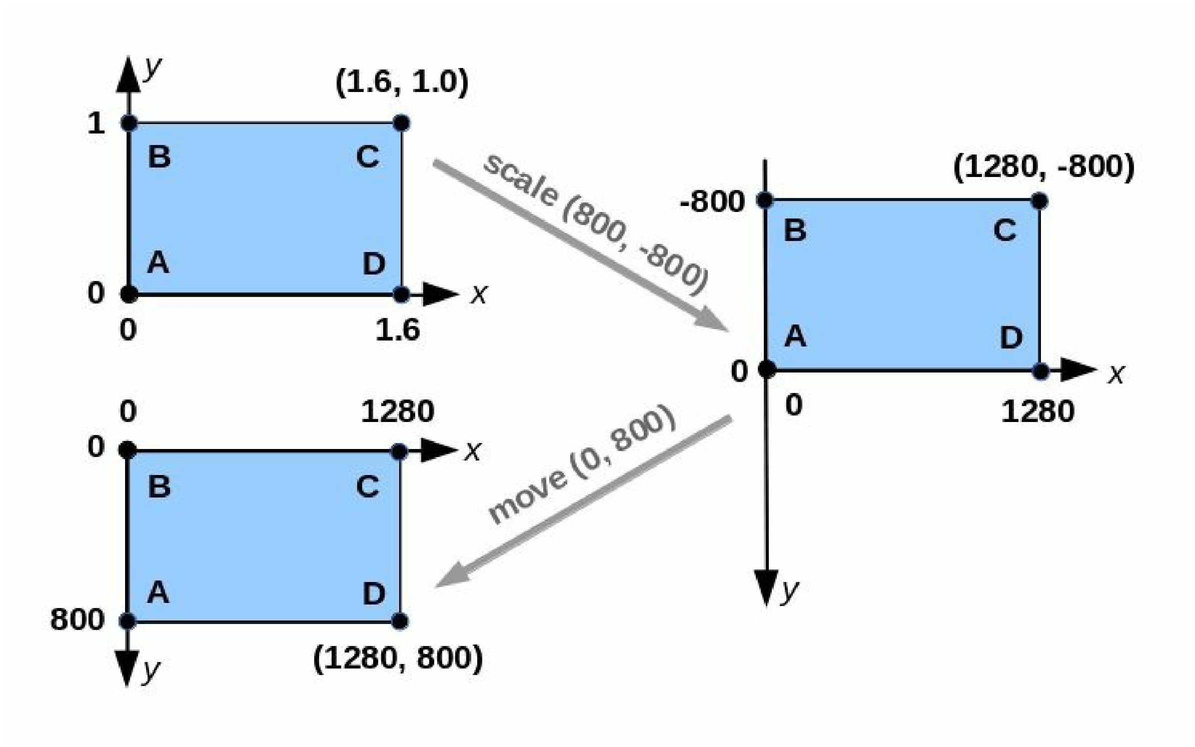


Figure 8-2.

We perform the scale first, followed by the move. Note that in order to flip our coordinate system vertically, we use a negative value (-800) for y-axis scale.

Now, let's define a triangle in the logical coordinate system we just discussed, as shown in Figure 8-3. This is the triangle we want to draw in this example.

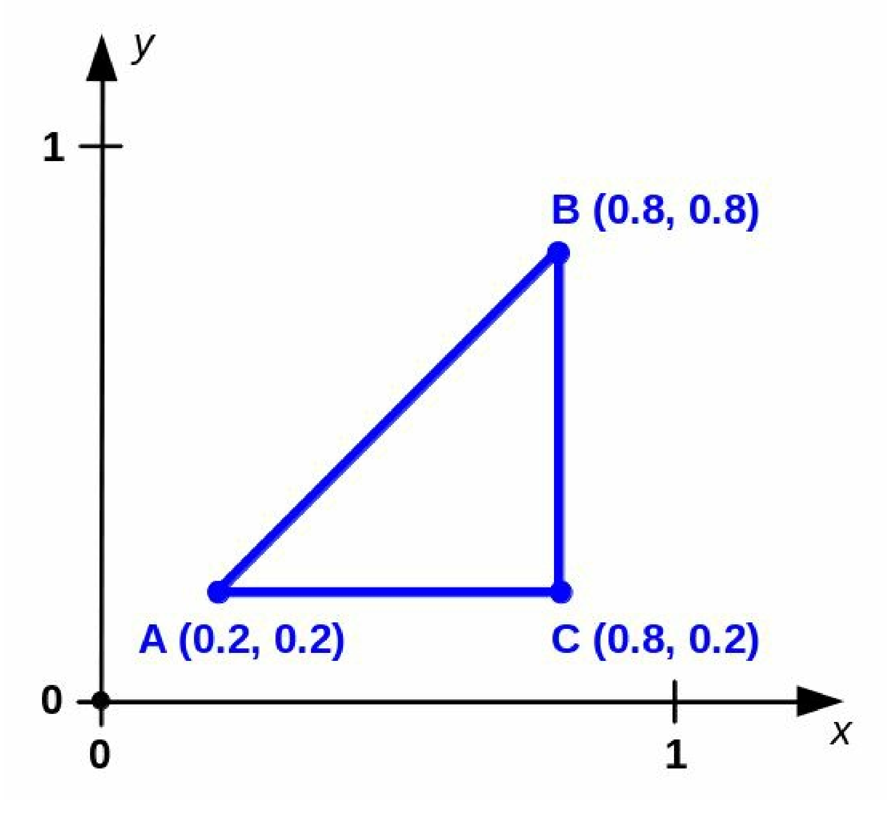


Figure 8-3.

Transform

We will use the following Java class to do transformations described in Figure 8-2.

```
package com.example.graphics2d;

public class Transform
{
    private float moveX;
    private float moveY;

    private float scaleX;
    private float scaleY;

    public Transform()
    {
        setMove(0, 0);
        setScale(1, 1);
    }

    public void setMove(float x, float y)
    {
        moveX = x;
        moveY = y;
    }

    public void setScale(float x, float y)
    {
        scaleX = x;
        scaleY = y;
    }

    public float transformX(float x)
    {
        return moveX + scaleX * x;
    }

    public float transformY(float y)
    {
        return moveY + scaleY * y;
    }
}
```

The Transform class has two groups of methods.

- The setMove() and setScale() methods setup transformation operations.
- The transformX() and transformY() methods transform x and y coordinates from the logical coordinate system to the device coordinate system.

To transform coordinates you have to create an instance of the Transform class and setup transformation operations you want to do.

```
Transform transform = new Transform();
transform.setScale(800, -800);
transform.setMove(0, 800);
```

Then you can convert one or more coordinates.

```
float newX = transform.transformX(0.8f);
float newY = transform.transformY(0.2f);
```

Activity

We will use the following activity with this example.

```
package com.example.graphics2d;

import android.app.Activity;
import android.os.Bundle;

public class GraphicsActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);

        TriangleView view = new TriangleView(this);
        this.setContentView(view);
    }
}
```

You should be familiar with this code already. To save some space we will not show the title or the action bar which takes a lot of space in Android 4.x.

```
this.requestWindowFeature(Window.FEATURE_NO_TITLE);
```

View

The TriangleView is the main view of our application. It does the drawing and coordinate transformations.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Path;
import android.graphics.Paint.Style;
import android.view.View;


public class TriangleView extends View
{
    private Transform transform;
    private Path path;
    private Paint paint;
    public TriangleView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        transform = new Transform();

        paint = new Paint();
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(25);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }


    @Override
    protected void onSizeChanged(int deviceWidth, int deviceHeight, int old
    {
        // Init transform

        // Landscape mode
        if(deviceWidth > deviceHeight)
        {
            transform.setScale(deviceHeight, -deviceHeight);
            transform.setMove(0, deviceHeight);
        }

        else
        {
            transform.setScale(deviceWidth, -deviceWidth);
            transform.setMove(0, deviceHeight);
        }

        // Convert coordinates
        float Ax = transform.transformX(0.2f);
        float Ay = transform.transformY(0.2f);

        float Bx = transform.transformX(0.8f);
        float By = transform.transformY(0.8f);

        float Cx = transform.transformX(0.8f);
        float Cy = transform.transformY(0.2f);

        // Create path
        path = new Path();
        path.moveTo(Ax, Ay);
        path.lineTo(Bx, By);
        path.lineTo(Cx, Cy);
        path.close();
    }
}
```

Code Review: Constructor

In the constructor we set the view background color

```
setBackgroundColor(Color.WHITE);
```

and created a Paint object

```
paint = new Paint();
paint.setStyle(Style.STROKE);
paint.setColor(Color.BLUE);
paint.setStrokeWidth(25);
```

Also we created an instance of the Transform class. It will be initialized later in onSizeChanged() method.

```
transform = new Transform();
```

Code Review: onSizeChanged()

The onSizeChanged() method is called by Android framework when TriangleView size changes, either after the view was created or when Android device orientation changes. This is the best place to initialize the transform object.

The scale transformation is a little different for the landscape and portrait orientation. The short side of a view (the height in landscape mode and the width in portrait mode) is used for this operation.

```
// Landscape mode
if(deviceWidth > deviceHeight)
{
    transform.setScale(deviceHeight, -deviceHeight);
    transform.setMove(0, deviceHeight);
}
// Portrait mode
else
{
    transform.setScale(deviceWidth, -deviceWidth);
    transform.setMove(0, deviceHeight);
}
```

The move transformation is the same for both the landscape and portrait orientation.

```
transform.setMove(0, deviceHeight);
```

Once the transform object is initialized we can convert logical coordinates of three points A, B, and C (see Figure 8-3) to device coordinates.

```
float Ax = transform.transformX(0.2f);
float Ay = transform.transformY(0.2f);

float Bx = transform.transformX(0.8f);
float By = transform.transformY(0.8f);

float Cx = transform.transformX(0.8f);
float Cy = transform.transformY(0.2f);
```

Next, we created a path to draw three lines AB, BC, and CA.

```
path = new Path();
path.moveTo(Ax, Ay);
path.lineTo(Bx, By);
path.lineTo(Cx, Cy);
path.close();
```

Note that close() method adds CA line segment to the path. You don't have to call

```
path.lineTo(Ax, Ay);
```

Code Review: onDraw()

The onDraw() method is very simple. It draws the path created in onSizeChanged() method.

```
canvas.drawPath(path, paint);
```

Screenshots

Figures 8-4 and 8-5 show screenshots of the application in the portrait and landscape modes.

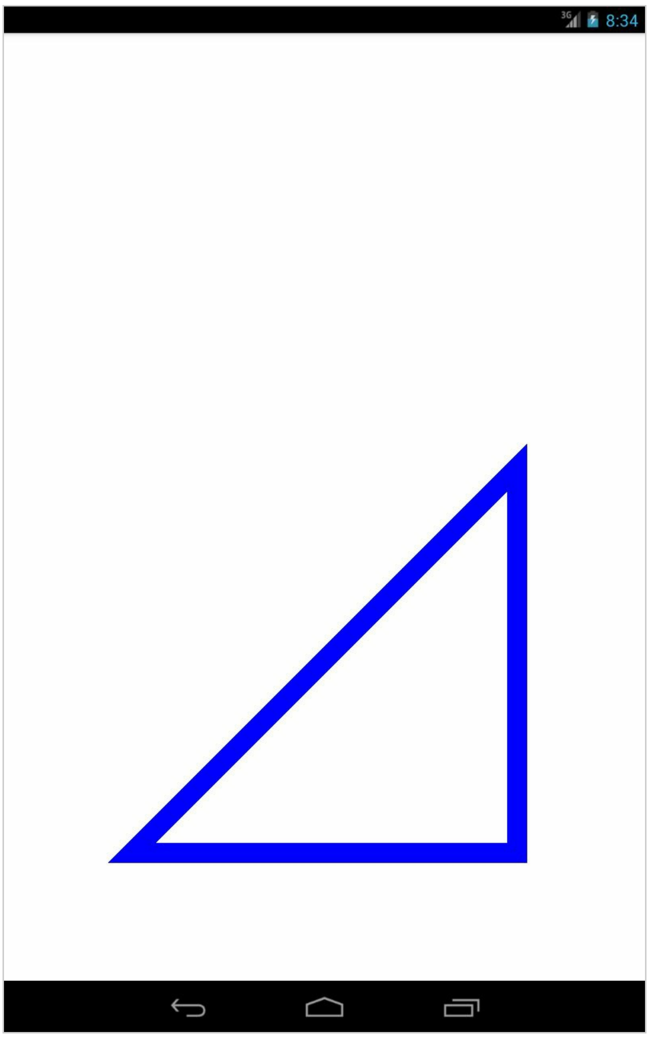


Figure 8-4.

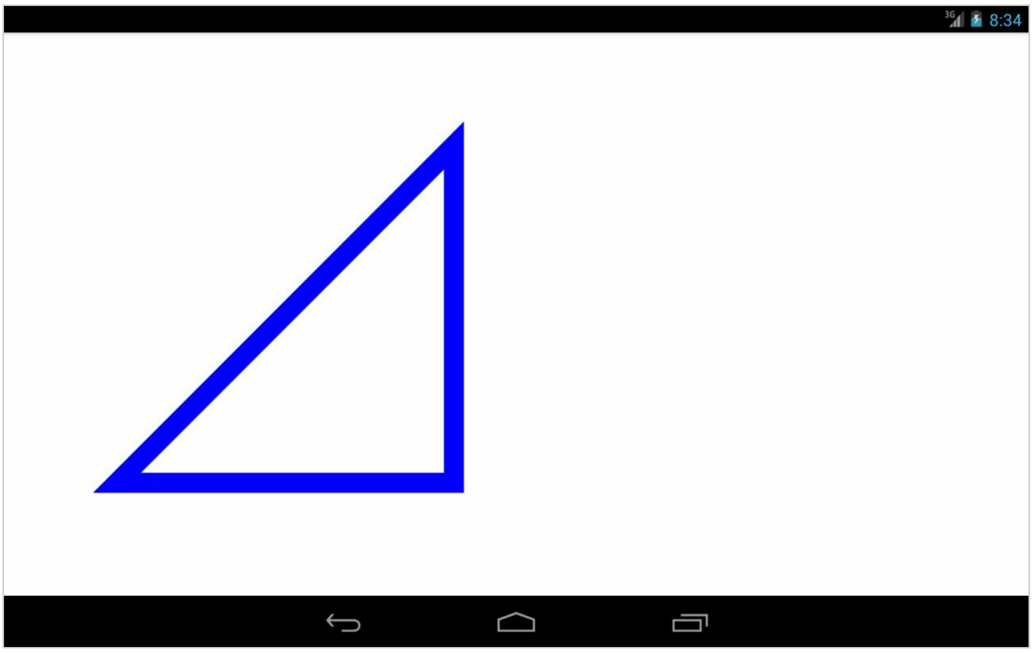


Figure 8-5.

Remember that the logical coordinate system shown in Figure 8-1 has the origin in the lower-left corner of the view and the view height and width are different. That's why the triangle in Figure 8-4 is bottom aligned and in the Figure 8-5 is left-aligned. If you want to center the triangle, you have to adjust the move transformation by (deviceWidth - deviceHeight)/2 as shown below.

```
// Landscape mode
if(deviceWidth > deviceHeight)
{
    transform.setScale(deviceHeight, -deviceHeight);
    transform.setMove((deviceWidth - deviceHeight)/2, deviceHeight);
}
// Portrait mode
else
{
    transform.setScale(deviceWidth, -deviceWidth);
    transform.setMove(0, deviceHeight - (deviceHeight-deviceWidth)/2);
}
```

In landscape mode, the x coordinate is adjusted, because the width is greater than the height. In portrait mode, the height is greater than the width, therefore the y coordinate is adjusted.

Figures 8-6 and 8-7 show the center aligned triangle in the portrait and landscape modes.

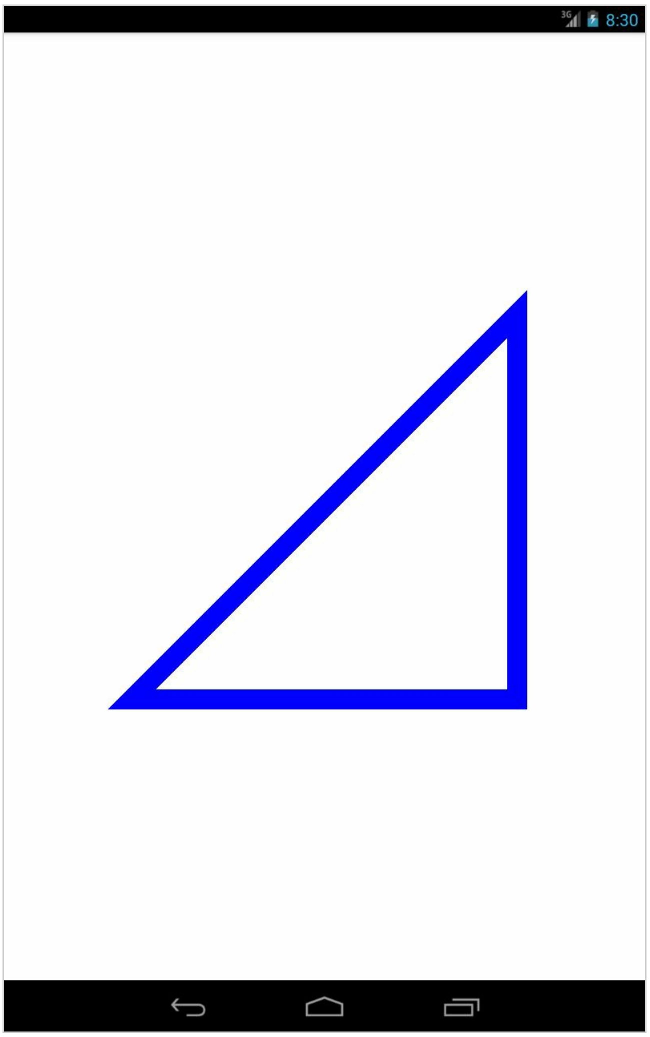


Figure 8-6.

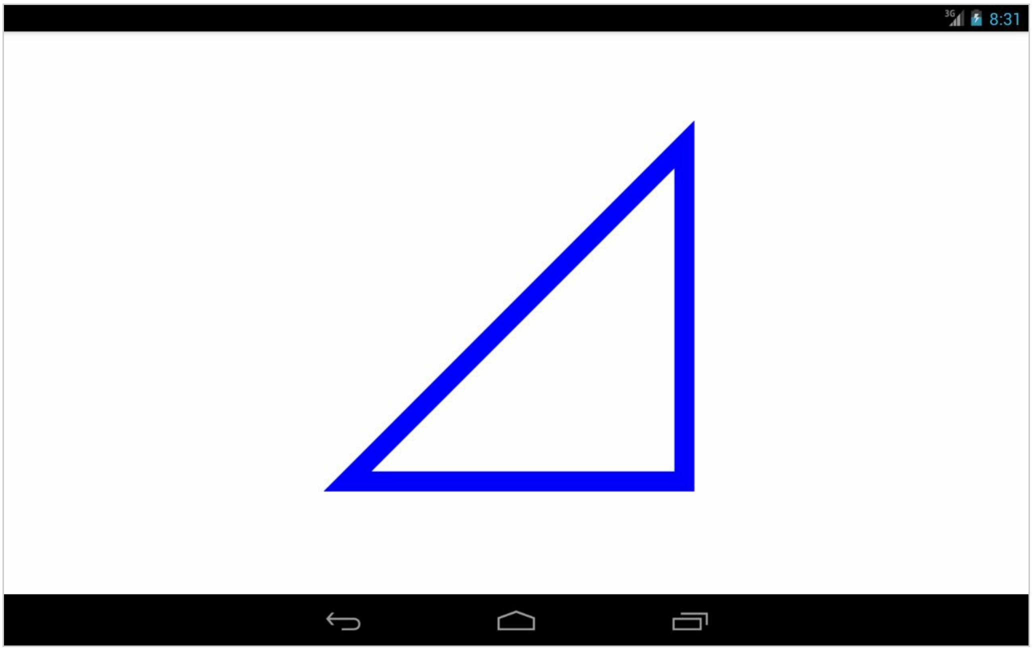


Figure 8-7.

8.3. Example: Stock Chart

In this example we will draw a chart of daily closing prices of S&P 500 index. The S&P 500 is one of the most commonly used benchmarks of the overall U.S. stock market. It is an index of 500 stocks chosen by a team of analysts and economists at Standard & Poor's. Historical stock and index prices are available for free from many web sites, such as Yahoo Finance or Google Finance. In this example we will use March 2013 daily closing prices shown below.

```
private float prices[] =
{
    1518.20f, 1525.20f, 1539.79f, 1541.46f, 1544.26f,
    1551.18f, 1556.22f, 1552.48f, 1554.52f, 1563.23f,
    1560.70f, 1552.10f, 1548.34f, 1558.71f, 1545.80f,
    1556.89f, 1551.69f, 1563.77f, 1562.85f, 1569.19f
};

private String dates[] =
{
    "2013-03-01", "2013-03-04", "2013-03-05", "2013-03-06", "2013-03-07",
    "2013-03-08", "2013-03-11", "2013-03-12", "2013-03-13", "2013-03-14",
    "2013-03-15", "2013-03-18", "2013-03-19", "2013-03-20", "2013-03-21",
    "2013-03-22", "2013-03-25", "2013-03-26", "2013-03-27", "2013-03-28"
};
```

We already saw an example of a stock chart in Figure 2-4 in Chapter 2 when we discussed logical coordinate systems. In this example we will use similar logical coordinates shown in Figure 8-8.

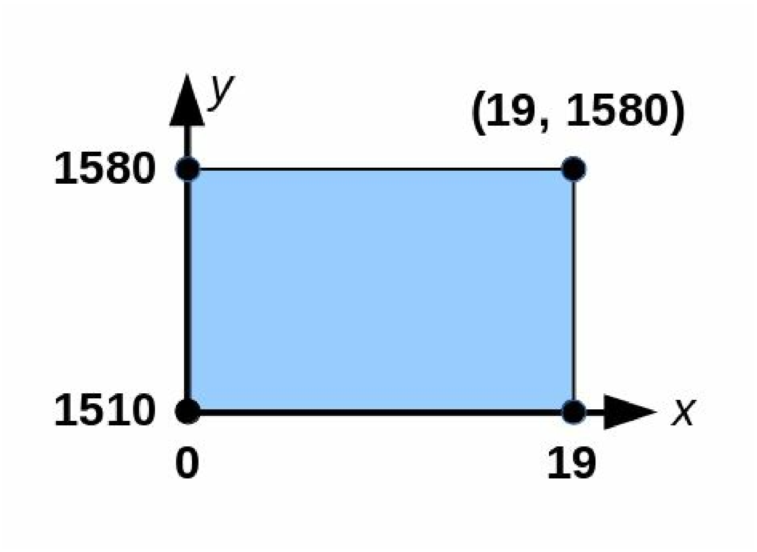


Figure 8-8.

The horizontal axis represents days, or more precisely, an index in prices[] or dates[] array. The index can have values from 0 to 19. The vertical axis represents prices which range from 1510 to 1580.

By default, all drawing functions of Canvas API use device coordinates, so we have to convert all logical coordinates to device coordinates. To draw a chart we will use all the space available on the screen (view). That means that the height-to-width ratio of a chart will depend on a screen resolution of a device. Figure 8-9 shows three transformations required to convert logical coordinates to device coordinates. Note that all numbers in Figure 8-9 are based on the device coordinate system with dimensions of 1280x800 pixels.

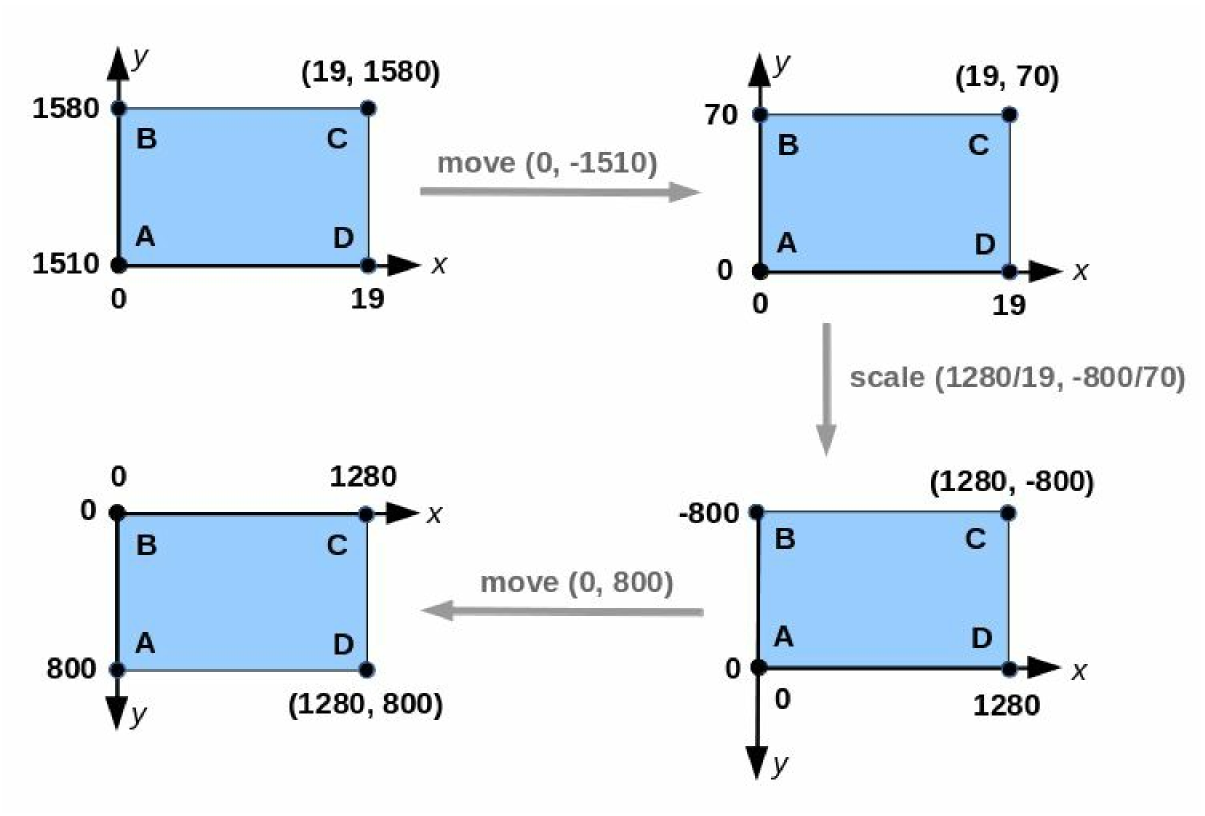


Figure 8-9.

Let's quickly review each transformation.

- **move(0, -1510)** This transformation moves the origin from (0, 0) to (0, 1510).
- **scale(1280/19, -800/70)** As in the previous example, this transformation flips coordinates vertically and changes the width of the view from 19 to 1280 and the height from 70 to 800.
- **move(0, 800)** This transformation moves the origin from (0, -800) to (0, 0). This is the same transformation we did in the previous example.

Transform

In the previous example we created the Transform class to perform a scale and move operations. We have to modify this class to support an additional move transformation as shown below.

```
package com.example.graphics2d;

public class Transform
{
    private float moveBeforeScaleX;
    private float moveBeforeScaleY;

    private float moveAfterScaleX;
    private float moveAfterScaleY;

    private float scaleX;
    private float scaleY;

    public Transform()
    {
        setMoveBeforeScale(0, 0);
        setMoveAfterScale(0, 0);
        setScale(1, 1);
    }

    public void setMoveBeforeScale(float x, float y)
    {
        moveBeforeScaleX = x;
        moveBeforeScaleY = y;
    }

    public void setMoveAfterScale(float x, float y)
    {
        moveAfterScaleX = x;
        moveAfterScaleY = y;
    }

    public void setScale(float x, float y)
    {
        scaleX = x;
        scaleY = y;
    }

    public float transformX(float x)
    {
        return moveAfterScaleX + scaleX * (moveBeforeScaleX + x);
    }
    public float transformY(float y)
    {
        return moveAfterScaleY + scaleY * (moveBeforeScaleY + y);
    }
}
```

In addition to the scale transformation, the new version of Transform class supports two move operations: move before scale and move after scale. New transformX() and transformY() methods are shown below.

```
public float transformX(float x)
{
    return moveAfterScaleX + scaleX * (moveBeforeScaleX + x);
}

public float transformY(float y)
{
    return moveAfterScaleY + scaleY * (moveBeforeScaleY + y);
}
```

Chart Model

It is common to separate data from its visual representation. In our example, the ChartView class is responsible for visual representation of the data and the ChartModel class provides the API to access the data.

```
package com.example.graphics2d;

public class ChartModel
{
    private float prices[] =
    {
        1518.20f, 1525.20f, 1539.79f, 1541.46f, 1544.26f,
        1551.18f, 1556.22f, 1552.48f, 1554.52f, 1563.23f,
        1560.70f, 1552.10f, 1548.34f, 1558.71f, 1545.80f,
        1556.89f, 1551.69f, 1563.77f, 1562.85f, 1569.19f
    };

    private float minPrice = 1510;
    private float maxPrice = 1580;

    public ChartModel()
    {
    }

    public float getMinPrice()
    {
        return minPrice;
    }

    public float getMaxPrice()
    {
        return maxPrice;
    }

    public int getNumberOfDays()
    {
        return prices.length;
    }

    public int getMinDay()
    {
        return 0;
    }

    public int getMaxDay()
    {
        return prices.length - 1;
    }

    public float getPrice(int index)
    {
        return prices[index];
    }
}
```

To make this example simpler we hardcoded min and max prices instead of calculating them.

Activity

The ChartActivity is similar to the activity from the previous example.

```
package com.example.graphics2d;

import android.app.Activity;
import android.content.pm.ActivityInfo;
import android.os.Bundle;
import android.view.Window;

public class ChartActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSC
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);

        ChartView view = new ChartView(this);
        this.setContentView(view);
    }
}
```

A stock chart looks much better in the landscape mode, so we will enforce this screen orientation by calling the following method.

```
this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

Also, to have more space on the screen we will hide the action bar, which is pretty big in Android 4.x.

```
this.requestWindowFeature(Window.FEATURE_NO_TITLE);
```

Chart View

The ChartView class is responsible for drawing the chart.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Cap;
import android.graphics.Paint.Join;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.view.View;


public class ChartView extends View
{
    private Transform transform;
    private ChartModel model;

    private Paint paint;
    private Path path;


    public ChartView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(5);
        paint.setStrokeCap(Cap.ROUND);
        paint.setStrokeJoin(Join.ROUND);

        transform = new Transform();
        model = new ChartModel();
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }


    @Override
    protected void onSizeChanged(int deviceWidth, int deviceHeight, int old
    {
        // Init transform
        transform.setMoveBeforeScale(0, -model.getMinPrice());

        float scaleX = (float)deviceWidth /
                       (model.getMaxDay() - model.getMinDay());
        float scaleY = (float)deviceHeight /
                       (model.getMaxPrice() - model.getMinPrice());
        transform.setScale(scaleX, -scaleY);
        transform.setMoveAfterScale(0, deviceHeight);

        // Create path
        path = new Path();

        float x = transform.transformX(0);
        float y = transform.transformY(model.getPrice(0));
        path.moveTo(x, y);

        for(int i = 1; i < model.getNumberOfDays(); i++)
        {
            x = transform.transformX(i);
            y = transform.transformY(model.getPrice(i));
            path.lineTo(x, y);
        }
    }
}
```

Code Review: Constructor

In the constructor we set the view background color and created a Paint object. We used round stroke cap and join styles and anti-alias flag to make the line look smoother.

```
paint = new Paint(Paint.ANTI_ALIAS_FLAG);
...
paint.setStrokeCap(Cap.ROUND);
paint.setStrokeJoin(Join.ROUND);
```

Also we created the Transform and ChartModel objects.

```
transform = new Transform();
model = new ChartModel();
```

Code Review: onSizeChanged()

Most of the work is done in this method.

Before we can do any coordinate transformations, we have to initialize the transform object. There are three transformation operations we have to do (see Figure 8-9).

1. Move the origin from (0, 0) to (0, minPrice).

```
transform.setMoveBeforeScale(0, -model.getMinPrice());
```

2. Scale. Negative scaleY value is used to flip coordinates vertically.

```
float scaleX = (float)deviceWidth /
               (model.getMaxDay() - model.getMinDay());
float scaleY = (float)deviceHeight /
               (model.getMaxPrice() - model.getMinPrice());
transform.setScale(scaleX, -scaleY);
```

3. Move the origin back to (0, 0).

```
transform.setMoveAfterScale(0, deviceHeight);
```

Once the transform object is initialized we can convert logical coordinates to device coordinates. Device coordinates are used to create a Path.

```
path = new Path();

float x = transform.transformX(0);
float y = transform.transformY(model.getPrice(0));
path.moveTo(x, y);

for(int i = 1; i < model.getNumberOfDays(); i++)
{
    x = transform.transformX(i);
    y = transform.transformY(model.getPrice(i));
    path.lineTo(x, y);
}
```

Note that in this example we did not close the path.

Code Review: onDraw()

In onDraw() method we drew the path created in onSizeChanged().

```
canvas.drawPath(path, paint);
```

Screenshot

A screenshot of the application is shown in Figure 8-10.

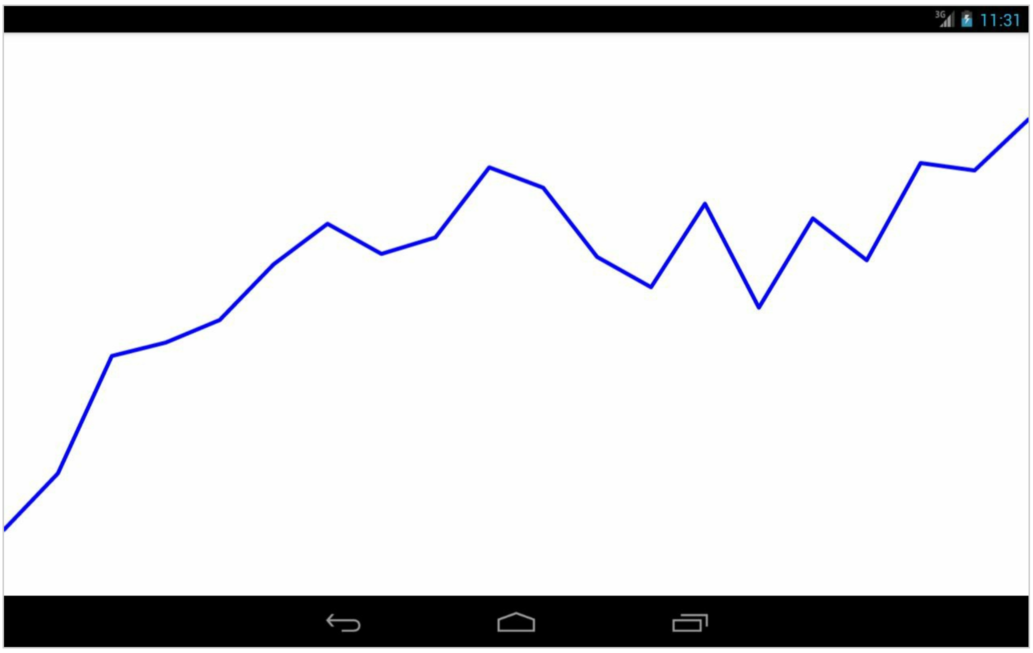


Figure 8-10.

9. Transformation Matrices

In the previous chapter you learned that there are several simple transformation operations, such as translation, scaling and rotation. Each transformation operation can be represented by a 3x3 matrix shown below.

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

The last row of a matrix always contains the constant values [0 0 1]. These values are required to perform matrix concatenation which is explained later in this chapter.

A coordinate can be represented by a column vector.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The last element of a vector always contains the constant value of 1. This extra value is required for matrix multiplication. Two matrices can be multiplied only if the number of columns in the first matrix equals the number of rows in the second matrix. Therefore 3x3 matrix can be only multiplied by 3x1 vector.

To transform source coordinates (x, y) to destination coordinates (x', y') we have to multiply a transformation matrix by a source coordinate vector.

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

The result of the multiplication is shown below.

$$\begin{aligned} x' &= m_{11}x + m_{12}y + m_{13} \\ y' &= m_{21}x + m_{22}y + m_{23} \end{aligned}$$

9.1. Transformation Operations

Let's review matrices for several simple transformation operations.

Identity Matrix

The identity matrix is the matrix in which all the elements on the main diagonal are equal to 1 and all other elements are equal to 0.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

It does not perform any transformations.

$$\begin{aligned} x' &= x \\ y' &= y \end{aligned}$$

Translation

Translation, also known as move, involves shifting the origin of the current coordinate system horizontally and vertically by a specific amount. Transformation matrix for a translation operation is shown below.

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

These are the transformation equations:

$$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \end{aligned}$$

Scaling

Scaling lets you stretch or shrink coordinates along the x and y axes independently. Transformation matrix for a scaling operation is shown below.

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These are the transformation equations:

$$x' = s_x x$$
$$y' = s_y y$$

Rotation

Rotation operation moves coordinates by the specified angle. Transformation matrix for a rotation operation is shown below.

$$\begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These are the transformation equations:

$$x' = cos(\alpha)x - sin(\alpha)y$$
$$y' = sin(\alpha)x + cos(\alpha)y$$

9.2. Matrix Concatenation

One of the advantages of using matrices is that you can combine several transformations by multiplying corresponding matrices.

For example, if you want to combine a scale and translate operations, you multiply the scale and translate matrices to produce a composite matrix.

$$M_c = M_s M_t$$

The process of combining several transformation matrices is called concatenation.

The order in which matrices are concatenated is important. Matrix multiplication is not commutative. The result of multiplying matrix A by matrix B does not necessarily equal the result of multiplying matrix B by matrix A.

9.3. Matrix Class

The Matrix class holds a 3x3 matrix for transforming coordinates. This class has many methods, but usually you would use just a few of them in most programs. Let's review some common methods.

Constructor

To create a Matrix object you call its constructor which creates an identity matrix.

```
Matrix matrix = new Matrix();
```

Concatenation Methods

One of the advantages of using matrices is that you can combine several transformations and store them in a single matrix. Usually you create a Matrix object and then concatenate it with one or more transformation operations.

There are two groups of concatenation methods: pre- and post-concatenation methods.

- Pre-concatenation methods such as preConcat(), preTranslate(), preScale(), preRotate() apply new operation before the preceding operation.
- Post-concatenation methods such as postConcat(), postTranslate(), postScale(), postRotate() apply new operation after the preceding operation.

A code snippet below shows scale and translate operations discussed in section 8.2 of the previous chapter.

```
matrix.postScale(800, -800);
matrix.postTranslate(0, 800);
```

Coordinate Transformation

The Matrix class provides several methods to transform coordinates. All these methods start with "map" prefix, such as mapPoints(), mapVectors(), mapRadius(), mapRect().

The following code snippet shows how to apply the matrix to the array of points specified by src, and write the transformed points into the array of points specified by dst.

```
float[] src;
float[] dst;
...
matrix.mapPoints(dst, src);
```

The points in a Path object can also be transformed as shown below.

```
Path path;
```

```
...
path.transform(matrix);
```

9.4. Example: Triangle

In this section we will modify the example we created in section 8.2 of the previous chapter to use transformation matrices instead of the Transform class we wrote ourselves. Let's look at the modified version of the TriangleView class first and then discuss the changes.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.Path;
import android.graphics.Paint.Style;
import android.view.View;


public class TriangleViewMatrix extends View
{
    private Matrix matrix;
    private Path path;
    private Paint paint;


    public TriangleViewMatrix(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint();
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(25);

        matrix = new Matrix();
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }


    @Override
    protected void onSizeChanged(int deviceWidth, int deviceHeight, int old
    {
        // Init transformation matrix
        matrix.reset();

        if(deviceWidth > deviceHeight)
        {
            matrix.postScale(deviceHeight, -deviceHeight);
            matrix.postTranslate(0, deviceHeight);
        }
        else
        {
            matrix.postScale(deviceWidth, -deviceWidth);
            matrix.postTranslate(0, deviceHeight);
        }

        // Create path
        path = new Path();
        path.moveTo(0.2f, 0.2f);
        path.lineTo(0.8f, 0.8f);
        path.lineTo(0.8f, 0.2f);
        path.close();

        // Convert coordinates
        path.transform(matrix);
    }
}
```

Code Review: Constructor

We replaced the Transform object

```
transform = new Transform();
```

with the Matrix object.

```
matrix = new Matrix();
```

The rest of the code is the same.

Code Review: onSizeChanged()

First we initialized the transformation matrix with the identity matrix.

```
matrix.reset();
```

Then we concatenated it with scale and translate operations.

```
if(deviceWidth > deviceHeight)
{
    matrix.postScale(deviceHeight, -deviceHeight);
    matrix.postTranslate(0, deviceHeight);
}
else
{
    matrix.postScale(deviceWidth, -deviceWidth);
    matrix.postTranslate(0, deviceHeight);
}
```

So far the new code is very similar to the old code from Chapter 8.

Once the transformation matrix is initialized we can convert logical coordinates to device coordinates and create a Path object.

In the old code we converted coordinates first and then created a Path object with the device coordinates.

In the new code, we used logical coordinates to create a Path

```
path = new Path();
path.moveTo(0.2f, 0.2f);
path.lineTo(0.8f, 0.8f);
path.lineTo(0.8f, 0.2f);
path.close();
```

and then converted the points in the Path to device coordinates.

```
path.transform(matrix);
```

Code Review: onDraw()

The onDraw() method didn't change. It draws the path created in onSizeChanged() method.

```
canvas.drawPath(path, paint);
```

Screenshots

Figures 9-1 and 9-2 show screenshots of the application in the portrait and landscape modes. Not surprisingly, the screenshots look the same as in Chapter 8.

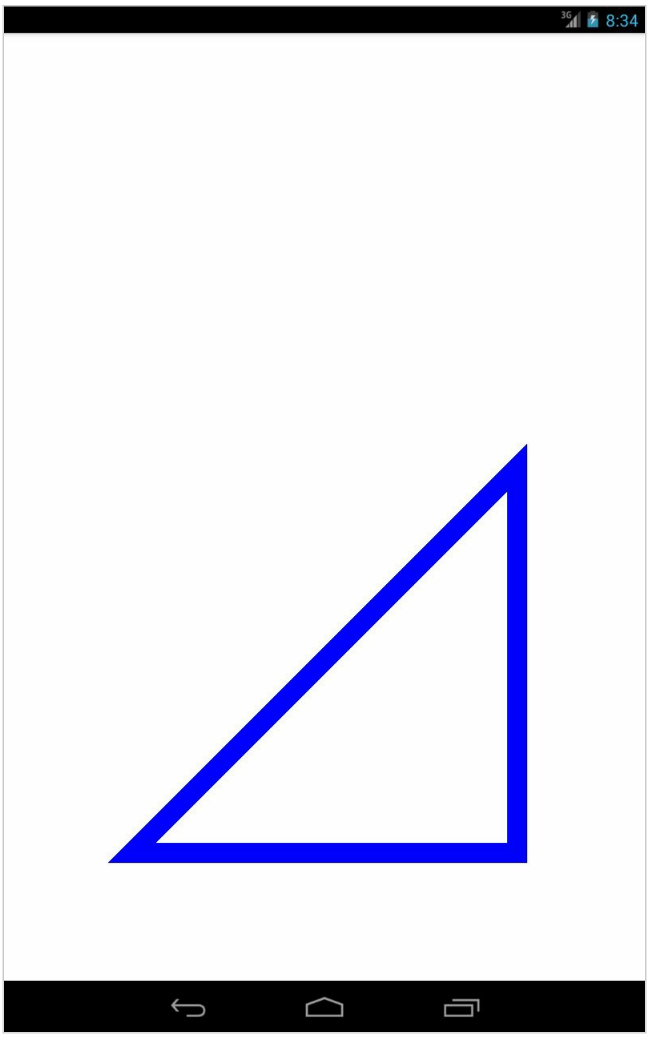


Figure 9-1.

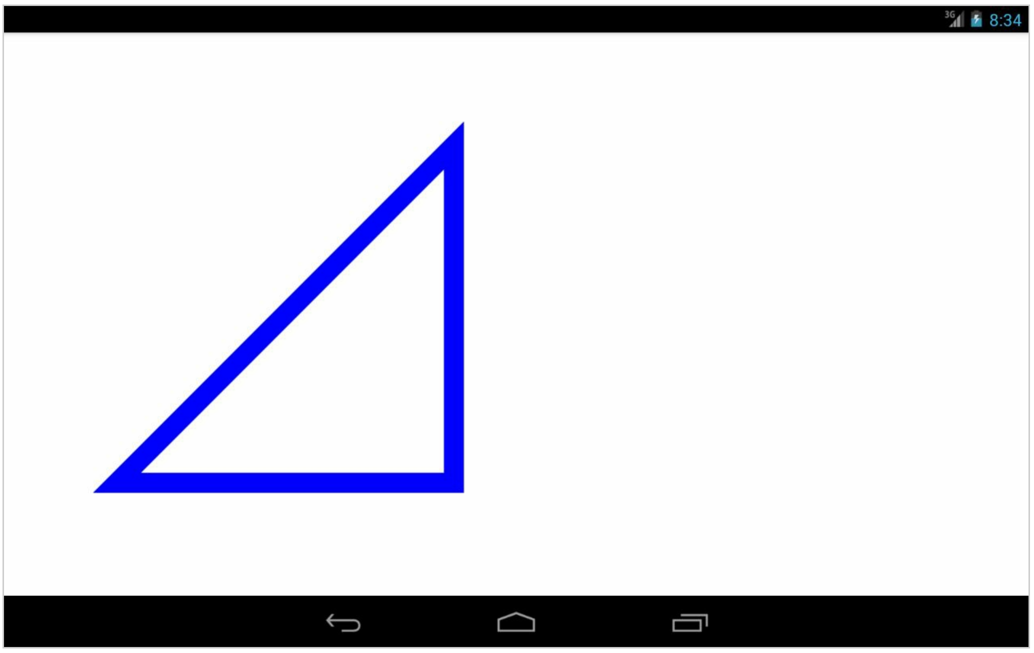


Figure 9-2.

9.5. Example: Stock Chart

In this section we will modify the example we created in section 8.3 of the previous chapter to use transformation matrices instead of the Transform class we wrote ourselves. Let's look at the modified version of the ChartView class first and then review the changes.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.Paint.Cap;
import android.graphics.Paint.Join;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.view.View;


public class ChartViewMatrix extends View
{
    private Matrix matrix;
    private ChartModel model;

    private Paint paint;
    private Path path;

    public ChartViewMatrix(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(5);
        paint.setStrokeCap(Cap.ROUND);
        paint.setStrokeJoin(Join.ROUND);

        matrix = new Matrix();
        model = new ChartModel();
    }

    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawPath(path, paint);
    }

    @Override
    protected void onSizeChanged(int deviceWidth, int deviceHeight, int old
    {
        // Init transformation matrix
        matrix.reset();

        matrix.postTranslate(0, -model.getMinPrice());

        float scaleX = (float)deviceWidth /
                       (model.getMaxDay() - model.getMinDay());
        float scaleY = (float)deviceHeight /
```

```
                    (model.getMaxPrice() - model.getMinPrice());
        matrix.postScale(scaleX, -scaleY);

        matrix.postTranslate(0, deviceHeight);

        // Create path
        path = new Path();
        path.moveTo(0, model.getPrice(0));

        for(int i = 1; i < model.getNumberOfDays(); i++)
        {
                path.lineTo(i, model.getPrice(i));
        }

        // Convert coordinates
        path.transform(matrix);
    }

}
```

Code Review: Constructor

We replaced the Transform object

```
transform = new Transform();
```

with the Matrix object.

```
matrix = new Matrix();
```

The rest of the code is the same.

Code Review: onSizeChanged()

Before we can do any coordinate transformations, we have to initialize the transformation matrix.

First we initialized the transformation matrix with the identity matrix.

```
matrix.reset();
```

Next we concatenated the matrix with three transformation operations shown in Figure 9-3.

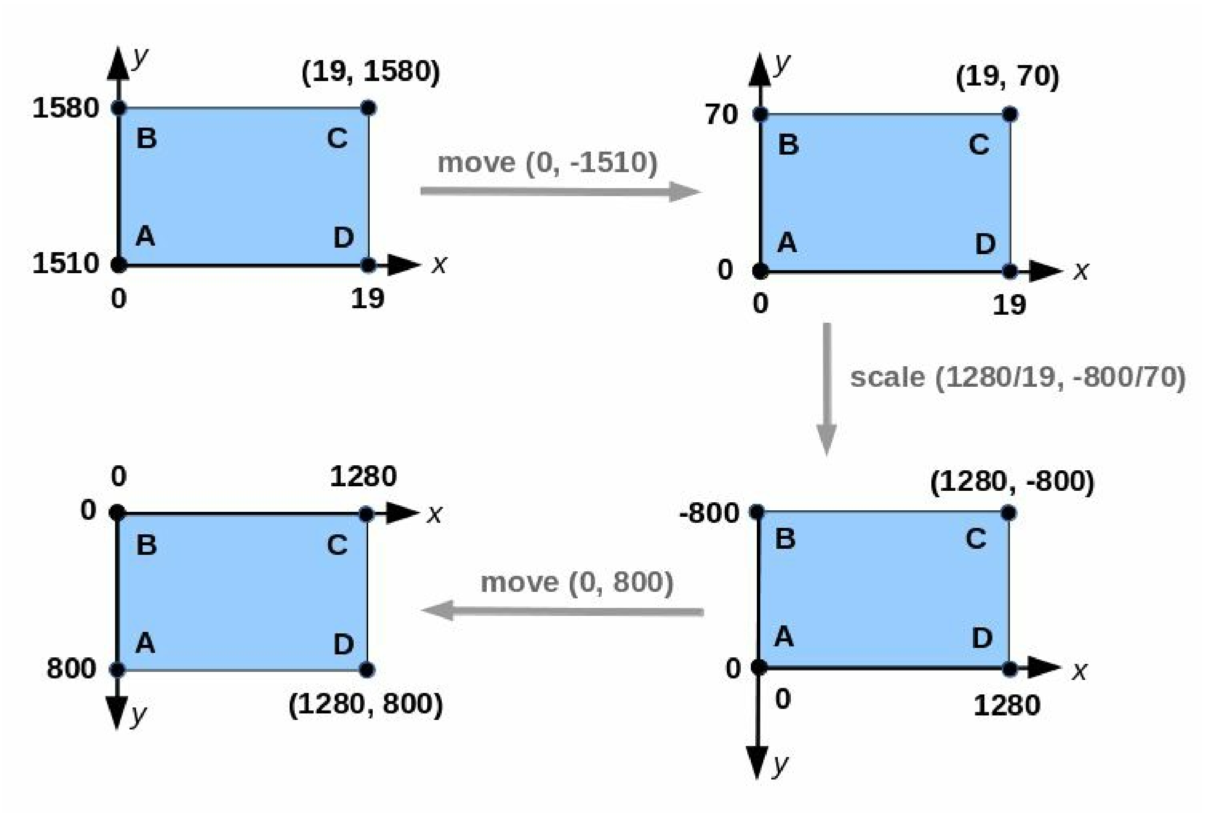


Figure 9-3.

1. Move the origin from (0, 0) to (0, minPrice).

```
matrix.postTranslate(0, -model.getMinPrice());
```

2. Scale. Negative scaleY value is used to flip coordinates vertically.

```
float scaleX = (float)deviceWidth /
               (model.getMaxDay() - model.getMinDay());
float scaleY = (float)deviceHeight /
               (model.getMaxPrice() - model.getMinPrice());
matrix.postScale(scaleX, -scaleY);
```

3. Move the origin back to (0, 0).

```
matrix.postTranslate(0, deviceHeight);
```

Once the transformation matrix is initialized we can convert logical coordinates to device coordinates and create a Path object.

In the old code from the previous chapter we converted coordinates first and then used device coordinates to create a Path.

In this example we used logical coordinates to create a Path

```
path = new Path();
```

```
path.moveTo(0, model.getPrice(0));

for(int i = 1; i < model.getNumberOfDays(); i++)
{
    path.lineTo(i, model.getPrice(i));
}
```

and then converted the points in the Path to device coordinates.

```
path.transform(matrix);
```

Code Review: onDraw()

The onDraw() method didn't change. It draws the path created in onSizeChanged() method.

```
canvas.drawPath(path, paint);
```

Screenshot

A screenshot of the application is shown in Figure 9-4.

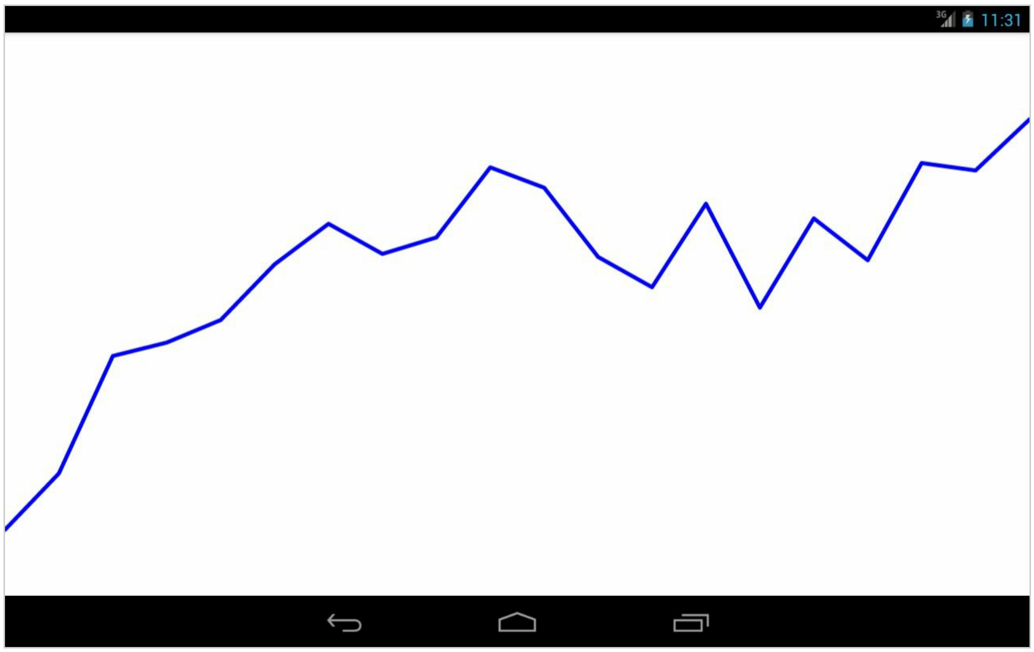


Figure 9-4.

9.6. Current Transformation Matrix (CTM)

Each Canvas object has a transformation matrix called the Current Transformation Matrix or CTM. The CTM is applied to any drawing performed by the Canvas object. After a Canvas object is created, the CTM is the identity matrix and does not affect default device coordinate system.

There are several methods such as translate(), scale(), rotate() and concat() which can be used to change the Current Transformation Matrix. The following example shows how this can be done.

Example

We will use the same logical coordinate system as in stock chart example (see Figure 9-3) and try to draw a rectangle.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.RectF;
import android.view.View;

public class CTMView extends View
{
    private int minX = 0;
    private int maxX = 19;
    private int minY = 1510;
    private int maxY = 1580;

    private Paint paint;
    private RectF rect;

    private int deviceHeight;
    private int deviceWidth;

    public CTMView(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(5);

        rect = new RectF(5, 1520, 14, 1570);
    }


    @Override
    protected void onSizeChanged(int deviceWidth, int deviceHeight, int old
    {
        this.deviceWidth = deviceWidth;
        this.deviceHeight = deviceHeight;
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        // Transform 3
        canvas.translate(0, deviceHeight);

        // Transform 2
        float scaleX = (float)deviceWidth / (maxX - minX);
        float scaleY = (float)deviceHeight / (maxY - minY);
        canvas.scale(scaleX, -scaleY);

        // Transform 1
        canvas.translate(0, -minY);

        // Draw
        canvas.drawRect(rect, paint);
    }
}
```

Code Review: Class Variables

First we defined min and max values of x and y coordinates in the logical coordinate system. In stock chart example we got those values from the chart model.

```
private int minX = 0;
private int maxX = 19;
private int minY = 1510;
private int maxY = 1580;
```

Code Review: Constructor

In the constructor we set the background color and created a Paint object. Also we created a rectangle we want to draw. Note that we used logical coordinates.

```
rect = new RectF(5, 1520, 14, 1570);
```

Code Review: onSizeChanged()

In the onSizeChanged() method we just stored the view width and height in class variables.

```
this.deviceWidth = deviceWidth;
this.deviceHeight = deviceHeight;
```

Code Review: onDraw()

In section 9.3 you learned that the Matrix class has pre- and post-concatenation methods. In all our examples so far we used post-concatenation methods which we called in the same order we wanted to apply transformation operations.

The translate(), scale() and rotate() methods which can be used to change the Current Transformation Matrix are pre-concatenation methods. Therefore we have to apply transformation operations in the reverse order.

```
// Transform 3
canvas.translate(0, deviceHeight);
```

```
// Transform 2
float scaleX = (float)deviceWidth / (maxX - minX);
float scaleY = (float)deviceHeight / (maxY - minY);
canvas.scale(scaleX, -scaleY);

// Transform 1
canvas.translate(0, -minY);
```

After changing the CTM, all subsequent drawing methods such as drawRect(), drawLine(), drawPath() will use our local coordinate system. So, now we can draw the rectangle.

```
canvas.drawRect(rect, paint);
```

Screenshot

If you run this example application you should see an unusual rectangular shape as shown in Figure 9-5.

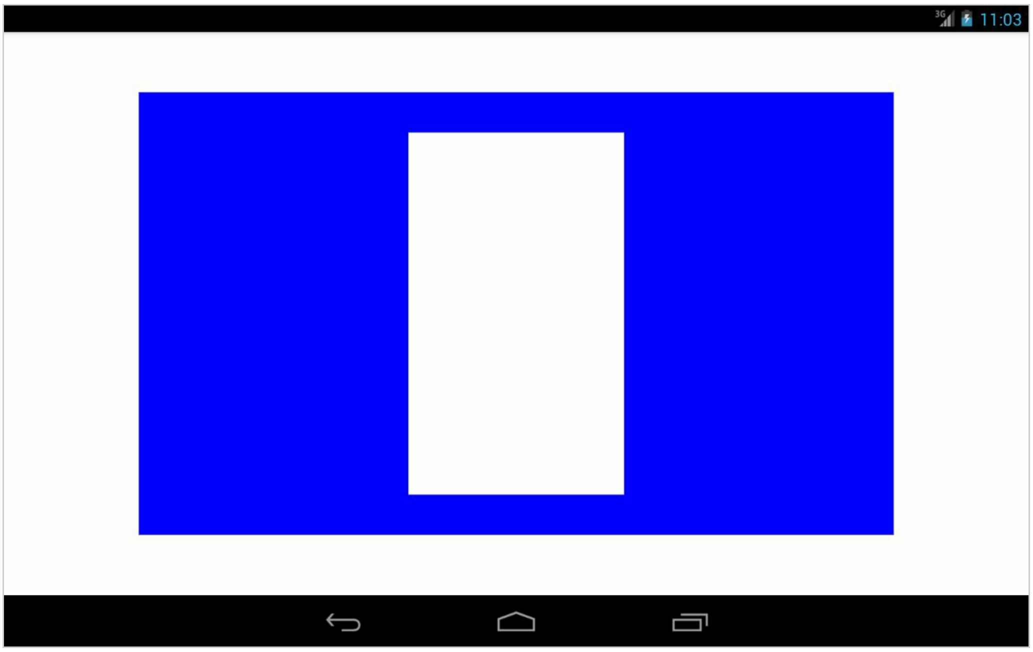


Figure 9-5.

So, what just happened?

One of the main reasons, you should never change the Current Transformation Matrix is that it not only transforms the coordinates of vertices or points of drawing primitives, but also the line width.

In our example, the horizontal and vertical scale factors are different, therefore horizontal and vertical lines have different width. When we called

```
paint.setStrokeWidth(5);
```

we expected the line to be 5 pixels wide, but instead, this value was scaled. If we assume that the view width is 1280 pixels and the height is 800 pixels, we will get the following values of the vertical and horizontal line widths:

```
vertical = 1280 / 19 * 5 = 337
horizontal = 800 / 70 * 5 = 57
```

This explains why vertical lines are much wider than horizontal.

So, how can we fix it? The only way to fix the line width is to convert logical coordinates to device coordinates, like we did in previous examples, without changing the CTM. The following example shows hot to do it.

Fixed Example

Below is the modified version of the previous example which does not use the CTM.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.RectF;
import android.view.View;

public class CTMViewFixed extends View
{
    private int minX = 0;
    private int maxX = 19;
    private int minY = 1510;
    private int maxY = 1580;

    private Matrix matrix;
    private Paint paint;
    private RectF rect;

    public CTMViewFixed(Context context)
    {
        super(context);
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);

        matrix = new Matrix();
    }


    @Override
    protected void onSizeChanged(int deviceWidth, int deviceHeight, int old
    {
        matrix.reset();

        // Transform 1
        matrix.postTranslate(0, -minY);
        // Transform 2
        float scaleX = (float)deviceWidth / (maxX - minX);
        float scaleY = (float)deviceHeight / (maxY - minY);
        matrix.postScale(scaleX, -scaleY);

        // Transform 3
        matrix.postTranslate(0, deviceHeight);

        // Convert coordinates
        rect = new RectF(5, 1520, 14, 1570);
        matrix.mapRect(rect);

        // Set line width
        paint.setStrokeWidth(deviceHeight * 0.05f);
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        canvas.drawRect(rect, paint);
    }
}
```

Code Review: Constructor

Instead of using the CTM we created a Matrix object in the constructor.

```
matrix = new Matrix();
```

Code Review: onSizeChanged()

We initialized the transformation matrix the way we did it in the stock chart example.

```
matrix.reset();

// Transform 1
matrix.postTranslate(0, -minY);

// Transform 2
float scaleX = (float)deviceWidth / (maxX - minX);
float scaleY = (float)deviceHeight / (maxY - minY);
matrix.postScale(scaleX, -scaleY);

// Transform 3
matrix.postTranslate(0, deviceHeight);
```

After that we created a new rectangle using logical coordinates

```
rect = new RectF(5, 1520, 14, 1570);
```

and then converted logical coordinates to device coordinates.

```
matrix.mapRect(rect);
```

Finally we set the line width to 5% of the view height.

```
paint.setStrokeWidth(deviceHeight * 0.05f);
```

Code Review: onDraw()

In onDraw() method we drew the rectangle created in onSizeChanged() method.

```
canvas.drawRect(rect, paint);
```

Screenshot

A screenshot of the application is shown in Figure 9-6.

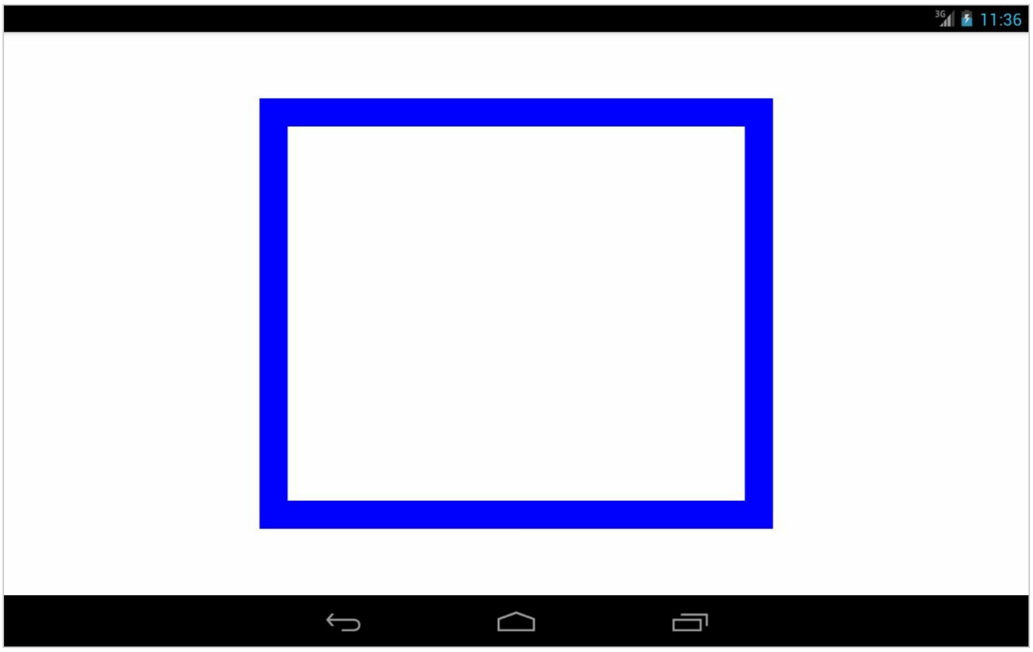


Figure 9-6.

10. Multithreading

In this chapter we will discuss two common approaches to writing multithreaded graphics applications in Android. The first approach uses the Handler and Message classes to communicate between threads. The second approach utilizes the SurfaceView class.

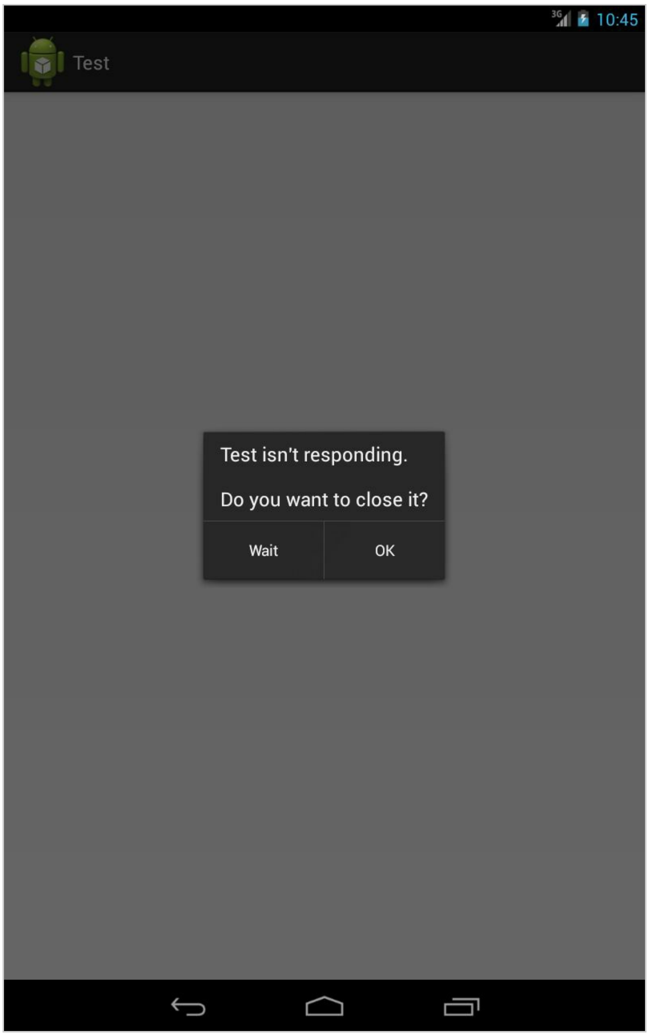
10.1. Processes and Threads

Every Android application runs in its own process. A process consists of the executing program code, a set of resources such as open files, memory, and one or more threads of execution.

A thread of execution is the smallest sequence of instructions to which an operating system allocates processor time. Threads exist within a process and share its resources, including memory and open files.

An application starts with just one thread, called the main thread or UI thread. This thread is responsible for dispatching events, such as drawing and touch events to user interface widgets. This is where all of the updates to the UI are made.

If your application performs long operations such as network access or database queries in the main thread, the whole UI will wait until those operations are completed, because no events can be dispatched. If an application cannot respond to user input because the UI thread is blocked, Android operating system can display the "application not responding" dialog shown in Figure 10-1.



Test isn't responding.

Do you want to close it?

Wait OK

Figure 10-1.

You should execute long running tasks in separate threads also called worker or background threads. Unfortunately the Android UI toolkit is not thread-safe. Therefore, you must update user interface only from the UI thread. Worker threads can communicate with the main UI thread through a Handler class.

10.2. Handler

The Handler class in **android.os** package can be used to send messages from worker threads to the main UI thread. When new Handler object is created it is associated with the message queue of the current thread. All messages sent to this Handler from different threads are delivered to the thread in which the Handler was created. The Handler class takes care of thread synchronization.

Example: Multithreaded Stock Chart

In this example we will add a worker thread to the stock chart application from the previous chapter. The worker thread is responsible for fetching market data and creating a stock chart model. Once a model object is created it is sent to the main UI thread to be displayed.

We will also change the UI. We will add a text field and Go button at the top of the screen as shown in Figure 10-2.

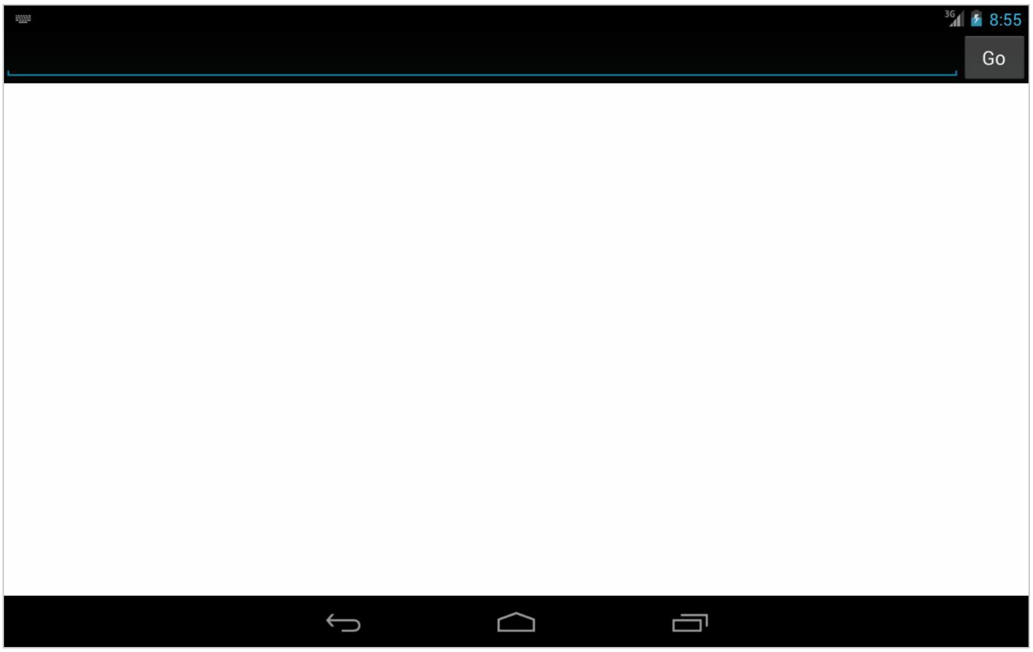


Figure 10-2.

After you enter a stock ticker, such as IBM or AAPL and hit Go button, the text field and Go button are disabled and "Loading..." message is displayed as shown in Figure 10-3.

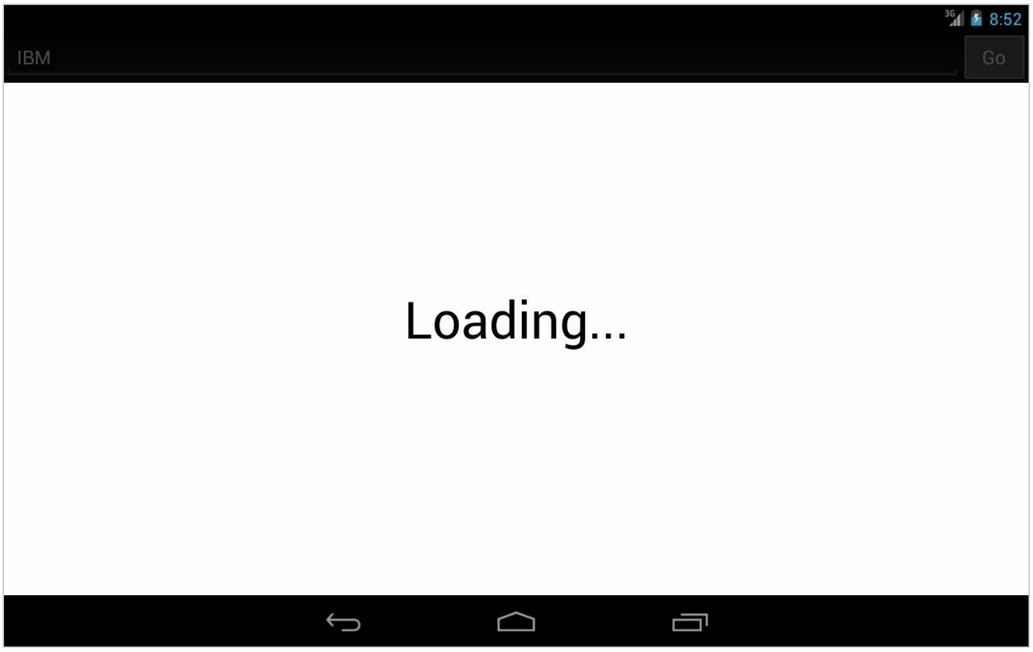


Figure 10-3.

After several seconds the stock chart is displayed and the text field and Go button are enabled again as shown in Figure 10-4.

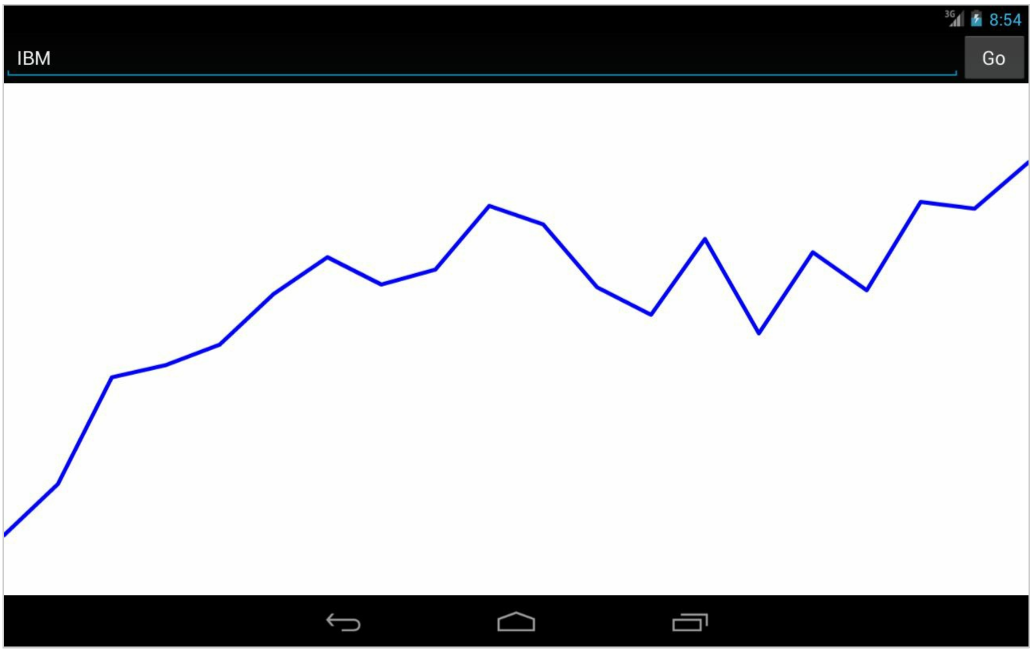


Figure 10-4.

Let's review classes and interfaces we used in this example.

MessageListener

This interface defines a method which will be called by a Handler when a ChartModel object is delivered from the worker thread to the main UI thread.

```
package com.example.graphics2d;

public interface MessageListener
{
    public void onChartData(ChartModel model);
}
```

MessageHandler

The MessageHandler class extends the Handler class and overrides its handleMessage() method.

```
package com.example.graphics2d;

import android.os.Handler;
import android.os.Message;

public class MessageHandler extends Handler
{
    public static int MSG_CHART_DATA = 1;

    private MessageListener listener;


    public MessageHandler()
    {
    }


    public void setMessageListener(MessageListener l)
    {
        this.listener = l;
    }


    @Override
    public void handleMessage(Message msg)
    {
        if(listener == null) return;

        if(msg.what == MSG_CHART_DATA)
        {
            ChartModel model = (ChartModel)msg.obj;
            listener.onChartData(model);
        }
    }
}
```

The handleMessage() method is called when a message is delivered to the Handler object. Subclasses of the Handler class must implement this method to receive messages.

The Message is a simple data structure in **android.os** package which is used to send data to a Handler. It has several public variables and methods to store data such as **Message.what** to store a message code and **Message.obj** to store an arbitrary object.

Messages are sent by a worker thread by calling **sendMessage()** method of the Handler class.

It is common to define a code for each message and assign it to the Message.what variable. It helps the recipient to identify the message. We defined MSG_CHART_DATA code in the MessageHandler class.

```
public static int MSG_CHART_DATA = 1;
```

If there were more messages we could have used a separate interface to define all messages, such as Messages.

In handleMessage() we check the message code and if it is a MSG_CHART_DATA message, we extract a ChartModel from the message and notify the listener that chart data is available.

```
if(msg.what == MSG_CHART_DATA)
{
    ChartModel model = (ChartModel)msg.obj;
    listener.onChartData(model);
}
```

ChartDataThread

The ChartDataThread class is a worker thread responsible for fetching market data.

```
package com.example.graphics2d;
```

```
import android.os.Message;


public class ChartDataThread extends Thread
{
    private MessageHandler handler;
    private String ticker;

    public ChartDataThread(MessageHandler handler, String ticker)
    {
        this.handler = handler;
        this.ticker = ticker;
    }

    public void run()
    {
        try
        {
            Thread.sleep(5000);
        }
        catch(Exception ex)
        {
        }

        ChartModel data = new ChartModel();

        Message msg = Message.obtain();
        msg.what = MessageHandler.MSG_CHART_DATA;
        msg.obj = data;

        handler.sendMessage(msg);
    }
}
```

The constructor has two parameters: MessageHandler to send messages to the main thread and a stock ticker. To simplify this example we will use the same hard-coded S&P 500 market data for all queries. Therefore a stock ticker is not used. Also we added a 5 seconds delay to simulate a network communication delay as shown below.

```
Thread.sleep(5000);
```

You can modify this example to get real market data from Yahoo! Finance of another source.

After the delay, a ChartModel is created. This is the same class we used in the previous chapter. It has hard-coded historical S&P 500 market data. If you decide to fetch market data from Yahoo! Finance or another source, you will have to modify the ChartModel to store the data and to calculate minimum and maximum prices.

Next, we got a Message object.

```
Message msg = Message.obtain();
```

Although the Message class has a public constructor, the preferred way to get a Message is to call **Message.obtain()** method which returns an object from the global pool. If you send a lot of messages in your application it will reduce the load on a garbage collector and improve the application performance.

Next, we set a message type. This step is optional, but usually it helps the recipient to identify what this message is about.

```
msg.what = MessageHandler.MSG_CHART_DATA;
```

Then we stored a ChartModel in the Message.

```
msg.obj = data;
```

Finally, the message is sent to the handler.

```
handler.sendMessage(msg);
```

Layout

The following layout is used by the application. It is stored in **res/layout/chart_mt.xml** file.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <EditText
        android:id="@+id/txtTicker"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
    <Button
        android:id="@+id/btnGo"
        android:text="Go"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
</LinearLayout>
<com.example.graphics2d.ChartViewMT
    android:id="@+id/chartView"
    android:layout_weight="1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
</LinearLayout>
```

It has a text view, Go button and a chart view.

Activity

The ChartActivity class implements two interfaces: the OnClickListener to handle clicks on "Go" button and the MessageListener to display a chart when a ChartModel object is delivered from the worker thread to the main UI thread.

```
package com.example.graphics2d;

import android.app.Activity;
import android.content.pm.ActivityInfo;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.Window;
import android.widget.Button;
import android.widget.EditText;


public class ChartActivity extends Activity implements OnClickListener, Mes
{
    private Button btnGo;
    private EditText txtTicker;
    private ChartViewMT chartView;

    private MessageHandler handler;


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

        this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSC
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);

        this.setContentView(R.layout.chart_mt);

        txtTicker = (EditText)findViewById(R.id.txtTicker);

        btnGo = (Button)findViewById(R.id.btnGo);
        btnGo.setOnClickListener(this);

        chartView = (ChartViewMT)findViewById(R.id.chartView);

        // Handler
        handler = new MessageHandler();
        handler.setMessageListener(this);
    }


    @Override
    public void onClick(View v)
    {
        btnGo.setEnabled(false);
        txtTicker.setEnabled(false);
        chartView.setLoading(true);

        String ticker = txtTicker.getText().toString();

        ChartDataThread thread = new ChartDataThread(handler, ticker);
        thread.start();
    }


    @Override
    public void onChartData(ChartModel model)
    {
        chartView.setModel(model);
        btnGo.setEnabled(true);
        txtTicker.setEnabled(true);
    }
}
```

In the constructor we set the preferred screen orientation and removed the title.

```
this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
this.requestWindowFeature(Window.FEATURE_NO_TITLE);
```

In previous examples we created a content view programmatically, but in this example we loaded the view from the layout XML.

```
this.setContentView(R.layout.chart_mt);
```

After the content view is created, we stored references to the text field, Go button and chart view in class variables.

```
txtTicker = (EditText)findViewById(R.id.txtTicker);
btnGo = (Button)findViewById(R.id.btnGo);
chartView = (ChartViewMT)findViewById(R.id.chartView);
```

We also set OnClickListener of the Go button.

```
btnGo.setOnClickListener(this);
```

Finally we created the MessageHandler and set its message listener.

```
handler = new MessageHandler();
handler.setMessageListener(this);
```

The MessageHandler is created in the main UI thread, therefore all messages sent from a worker thread will be delivered to the main UI thread.

The onClick() method is called when Go button is clicked. This method is defined in OnClickListener interface.

```
@Override
public void onClick(View v)
{
    btnGo.setEnabled(false);
    txtTicker.setEnabled(false);
    chartView.setLoading(true);

    String ticker = txtTicker.getText().toString();

    ChartDataThread thread = new ChartDataThread(handler, ticker);
    thread.start();
}
```

In this method we disabled the text field and Go button to prevent a user from submitting a new request before the previous request is completed.

```
btnGo.setEnabled(false);
txtTicker.setEnabled(false);
```

Also we requested the chart view to show the "Loading..." message.

```
chartView.setLoading(true);
```

Next we got a stock ticker from the EditText control. To simplify this example we don't validate the ticker.

```
String ticker = txtTicker.getText().toString();
```

Finally, we created and started a worker thread.

```
ChartDataThread thread = new ChartDataThread(handler, ticker);
thread.start();
```

We passed the handler and the stock ticker to the thread. The ChartDataThread fetches market data for the ticker, creates a ChartModel and sends a message with the ChartModel to the handler. The handler processes the message and calls the onChartData() method with the ChartModel as a parameter.

```
@Override
public void onChartData(ChartModel model)
{
    chartView.setModel(model);

    btnGo.setEnabled(true);
    txtTicker.setEnabled(true);
}
```

The onChartData() method is defined in the MessageListener interface and implemented by the ChartActivity. It runs in the main UI thread. In this method we requested the chart view to display a chart

```
chartView.setModel(model);
```

and then enabled the text field and Go button.

```
btnGo.setEnabled(true);
txtTicker.setEnabled(true);
```

ChartView

This class is responsible for displaying a stock chart and the "Loading..." message. It is similar to the chart view from the previous chapter.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.graphics.Paint.Cap;
import android.graphics.Paint.Join;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.util.AttributeSet;
import android.view.View;


public class ChartViewMT extends View
{
    private Matrix matrix;
    private ChartModel model;

    private Paint paint, textPaint;
    private Path path;

    private int deviceWidth, deviceHeight;
    private boolean loading = false;


    public ChartViewMT(Context context)
    {
        super(context);
        init();
    }


    public ChartViewMT(Context context, AttributeSet attrs)
    {
        super(context, attrs);
        init();
    }


    private void init()
    {
        setBackgroundColor(Color.WHITE);

        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(5);
        paint.setStrokeCap(Cap.ROUND);
        paint.setStrokeJoin(Join.ROUND);

        textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        textPaint.setTextAlign(Align.CENTER);

        matrix = new Matrix();
    }


    @Override
    public void onDraw(Canvas canvas)
    {
        if(model == null)
        {
            if(loading)
            {
                canvas.drawText("Loading...", deviceWidth/2, deviceHeight/2
            }
        }
        else
        {
            canvas.drawPath(path, paint);
        }
    }


    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        deviceHeight = h;
        deviceWidth = w;

        textPaint.setTextSize(h * 0.1f);
    }


    public void setLoading(boolean b)
    {
        this.loading = b;
        this.model = null;
        invalidate();
    }

    public void setModel(ChartModel m)
    {
        loading = false;
        this.model = m;
        updatePath();
        invalidate();
    }


    private void updatePath()
    {
        // Init transformation matrix
        matrix.reset();

        matrix.postTranslate(0, -model.getMinPrice());

        float scaleX = (float)deviceWidth / (model.getMaxDay() - model.getM
        float scaleY = (float)deviceHeight / (model.getMaxPrice() - model.g
        matrix.postScale(scaleX, -scaleY);

        matrix.postTranslate(0, deviceHeight);

        // Create path
        path = new Path();
        path.moveTo(0, model.getPrice(0));

        for(int i = 1; i < model.getNumberOfDays(); i++)
        {
            path.lineTo(i, model.getPrice(i));
        }

        // Convert coordinates
        path.transform(matrix);
    }
```

```
}
```

We added one more constructor which is called when the view is created from an XML file, supplying attributes.

```
public ChartViewMT(Context context, AttributeSet attrs)
{
    super(context, attrs);
    init();
}
```

Also we moved all initialization code to a separate method.

```
private void init()
{
    setBackgroundColor(Color.WHITE);

    paint = new Paint(Paint.ANTI_ALIAS_FLAG);
    paint.setStyle(Style.STROKE);
    paint.setColor(Color.BLUE);
    paint.setStrokeWidth(5);
    paint.setStrokeCap(Cap.ROUND);
    paint.setStrokeJoin(Join.ROUND);

    textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    textPaint.setTextAlign(Align.CENTER);

    matrix = new Matrix();
}
```

We added loading flag and setLoading() method.

```
private boolean loading = false;
...
public void setLoading(boolean b)
{
    this.loading = b;
    this.model = null;
    invalidate();
}
```

After we set loading and model variables, we called the invalidate() method to request the repaint of the view. This will trigger a call to onDraw() method.

```
@Override
public void onDraw(Canvas canvas)
{
    if(model == null)
    {
        if(loading)
        {
            canvas.drawText("Loading...", deviceWidth/2, deviceHeight/2, te
        }
    }
    else
    {
        canvas.drawPath(path, paint);
    }
}
```

If model variable is not null, a stock chart is displayed. If model variable is null and loading flag is set the "Loading..." message is displayed.

We also added setModel() method.

```
public void setModel(ChartModel m)
{
    loading = false;
    this.model = m;
    updatePath();
    invalidate();
}
```

It sets loading flag to false, stores a ChartModel in model variable, creates new Path object and calls invalidate() to refresh the view.

The updatePath() method creates a Path from the ChartModel. This is the same code we used in the previous chapter. Before it was in onSizeChanged() method.

```
private void updatePath()
{
    // Init transformation matrix
    matrix.reset();

    matrix.postTranslate(0, -model.getMinPrice());

    float scaleX = (float)deviceWidth / (model.getMaxDay() - model.getMinDa
    float scaleY = (float)deviceHeight / (model.getMaxPrice() - model.getMi
    matrix.postScale(scaleX, -scaleY);

    matrix.postTranslate(0, deviceHeight);

    // Create path
    path = new Path();
    path.moveTo(0, model.getPrice(0));

    for(int i = 1; i < model.getNumberOfDays(); i++)
    {
        path.lineTo(i, model.getPrice(i));
    }

    // Convert coordinates
    path.transform(matrix);
}
```

In the onSizeChanged() method we stored the view width and height and calculated text size for the "Loading..." message.

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh)
{
    deviceHeight = h;
    deviceWidth = w;

    textPaint.setTextSize(h * 0.1f);
}
```

10.3. SurfaceView

In addition to the Handler class you can use the SurfaceView class to update the UI from a worker thread. The SurfaceView extends the View class, therefore you can override its onDraw() method to do the drawing as we did in previous chapters. In addition to that, the SurfaceView provides another drawing surface located behind the view. You can access this second drawing surface via the SurfaceHolder interface as shown below.

```
SurfaceView view;
...
SurfaceHolder holder = view.getHolder();
Canvas canvas = holder.lockCanvas();
```

Once the Canvas object is locked by a thread, only this thread can draw on the Canvas. Note that this is not the same Canvas you get in onDraw() method. The lock is released by calling the following method.

```
holder.unlockCanvasAndPost(canvas);
```

After this call, the UI will be updated and other threads can access the surface again.

You can update the drawing surface provided by the SurfaceHolder from any thread. If you want to update other views, such as buttons or text fields, or the main drawing area of the SurfaceView inherited from the View class you have to do this in the main UI thread.

Remember that the second drawing surface is behind a window (main drawing area) of the SurfaceView. When you set the SurfaceView background color, it is applied to the window of a view. If you use an opaque color, such as Color.WHITE (0xFFFFFFFF) you would not see the surface underneath the window (main drawing area).

If you want to receive information about changes to the surface, you have to implement the **SurfaceHolder.Callback** interface. It defines the following methods:

```
void surfaceCreated(SurfaceHolder holder);
void surfaceChanged(SurfaceHolder holder, int format, int width, int height
void surfaceDestroyed(SurfaceHolder holder);
```

The surfaceCreated() is called immediately after the surface is first created. It can be used to set a background color and show "Loading..." message.

The surfaceChanged() method is called when the format or size of a surface changes. This method is called at least once. It can be used instead of onSizeChanged() method to determine the size of a surface view.

The surfaceDestroyed() is called immediately before a surface is destroyed.

Example: Progress Indicator

In this example we will create a circular progress indicator similar to the circular chart example discussed in Section 6.4 of Chapter 6. We will use a SurfaceView to draw the indicator and a separate thread to update it. A progress thread starts when the application starts. It increases the progress value by 10% every half second until it reaches 100%. Below are several screenshots of the application.

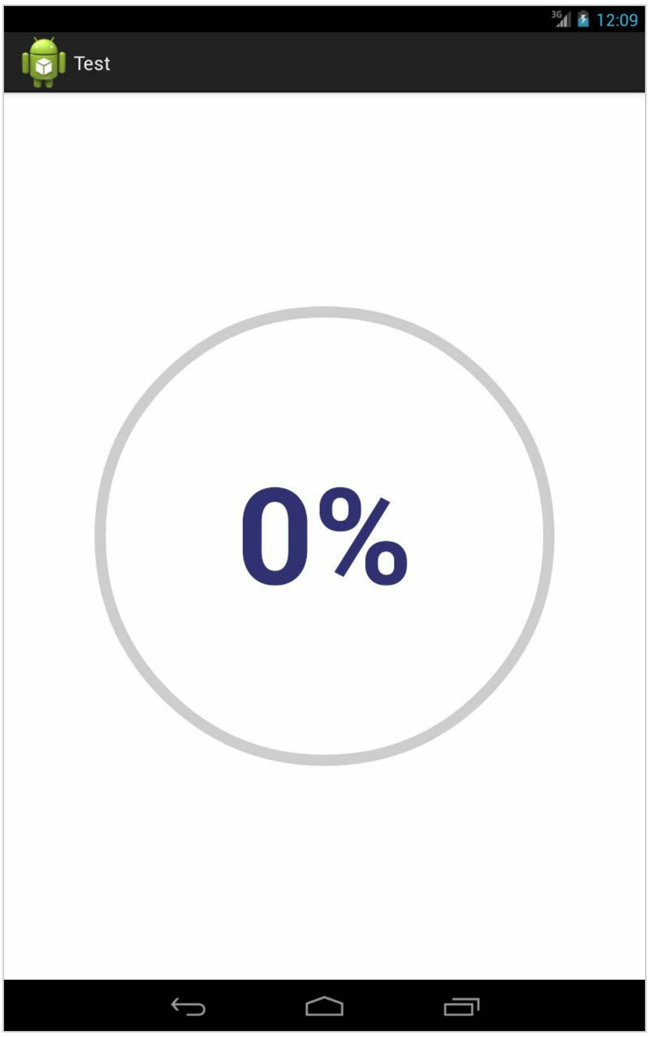


Figure 10-5.

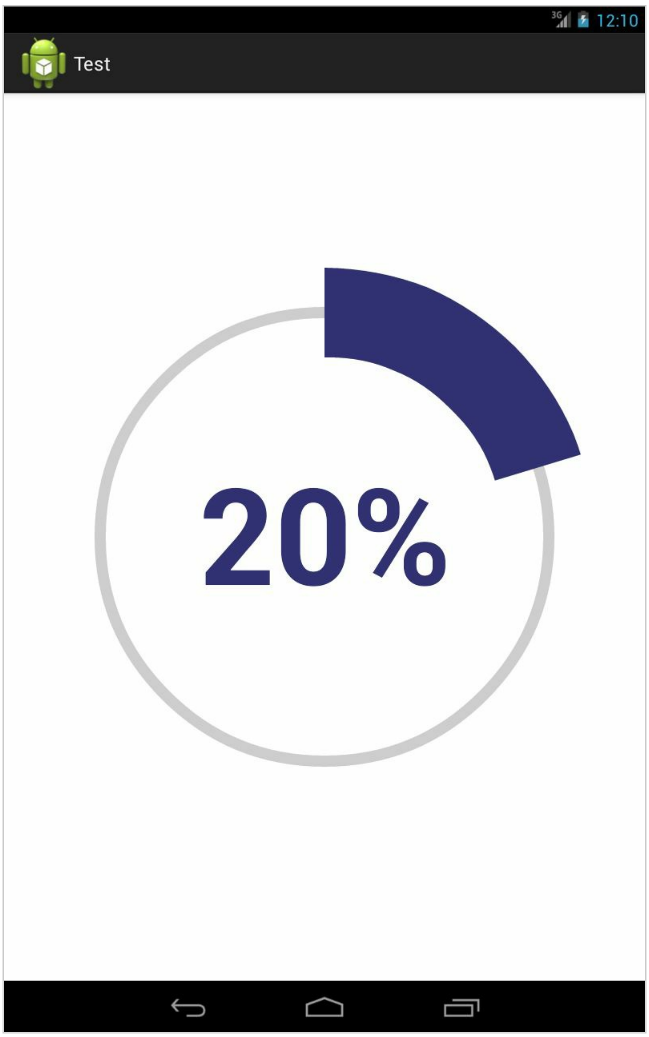


Figure 10-6.

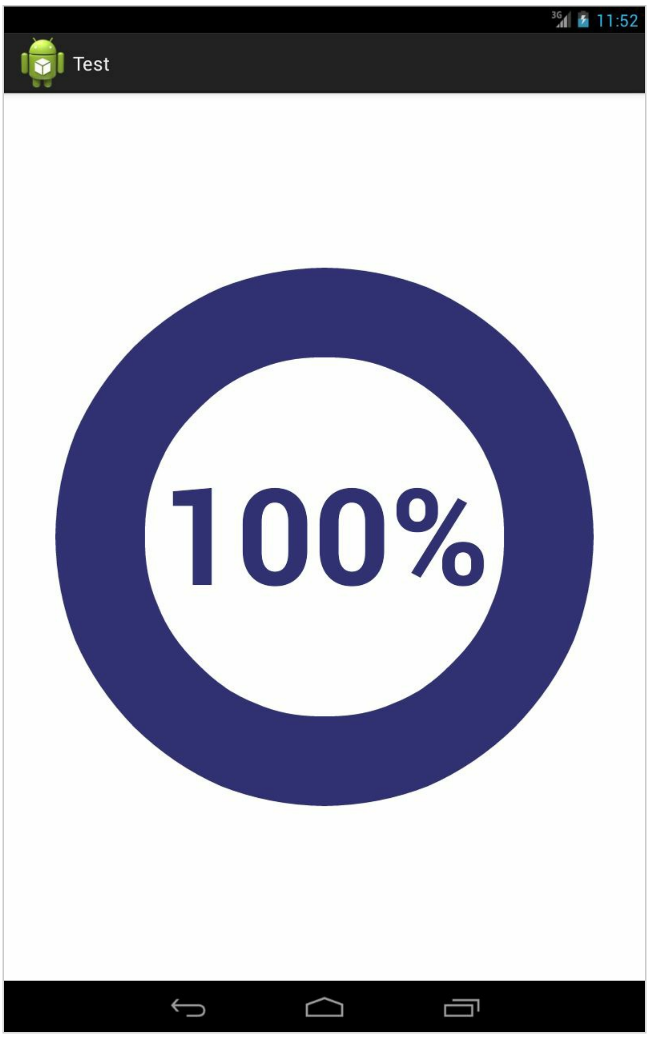


Figure 10-7.

Let's review different classes we used in this example.

ProgressActivity

This is the main activity of our application.

```
package com.example.graphics2d;

import android.app.Activity;
import android.os.Bundle;

public class ProgressActivity extends Activity
{
    private ProgressView view;
    private ProgressThread thread;


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

        view = new ProgressView(this);
        this.setContentView(view);
    }


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

        thread = new ProgressThread(view);
        thread.start();
    }


    @Override
    protected void onStop()
    {
        super.onStop();

        if(thread == null) return;

        thread.requestStop();

        // Wait for the thread to stop
        boolean retry = true;
        while(retry)
        {
            try
            {
                thread.join();
                retry = false;
            }
            catch(InterruptedException e)
            {
            }
        }

        thread = null;
    }
}
```

The onCreate() method is called when the activity is first created. In this method we created the ProgressView programmatically and set it as the content view of the activity.

```
view = new ProgressView(this);
this.setContentView(view);
```

The onStart() method is called when the activity is becoming visible to the user. It is a convenient place to create and start a progress thread.

```
thread = new ProgressThread(view);
thread.start();
```

The onStop() method is called when the activity is no longer visible to the user. This is a convenient place to stop a progress thread if it is still running. We used the following method to stop the thread.

```
thread.requestStop();
```

We will discuss it when we review the ProgressThread class.

The following code is used to wait for a progress thread to stop.

```
boolean retry = true;
while(retry)
{
    try
    {
        thread.join();
        retry = false;
    }
    catch(InterruptedException e)
    {
    }
}
```

The thread.join() method waits for the progress thread to stop, but it can throw an InterruptedException. Therefore we added a while loop to retry the thread.join() in case of an exception.

ProgressHelper

The ProgressHelper class is responsible for drawing the progress indicator.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.graphics.Paint.Style;
import android.graphics.Rect;
import android.graphics.RectF;
import android.graphics.Typeface;


public class ProgressHelper
{
    private float centerX;
    private float centerY;
    private float radius;
    private float textY;

    private RectF arcOval;
    private Rect textBounds;

    private Paint paint;
    private Paint txtPaint;

    private int color1 = Color.LTGRAY;
    private int color2 = 0xff333377;


    public ProgressHelper(Context context)
    {
        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        txtPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        txtPaint.setColor(color2);
        txtPaint.setTypeface(Typeface.DEFAULT_BOLD);
        txtPaint.setTextAlign(Align.CENTER);

        arcOval = new RectF();
        textBounds = new Rect();
    }


    public void draw(Canvas canvas, float value)
    {
        // Draw circle
        paint.setStrokeWidth(radius * 0.05f);
        paint.setColor(color1);
        canvas.drawCircle(centerX, centerY, radius, paint);

        // Draw arc
        paint.setColor(color2);
        paint.setStrokeWidth(radius * 0.4f);
        canvas.drawArc(arcOval, -90, value * 360, false, paint);

        // Draw label
        String label = String.format("%.0f%%", value * 100);
        canvas.drawText(label, centerX, textY, txtPaint);
    }


    public void changeSize(int w, int h)
    {
        // Center
        centerX = w / 2.0f;
        centerY = h / 2.0f;

        // Radius
        radius = Math.min(w, h) * 0.7f / 2;

        // Text size
        txtPaint.setTextSize(radius * 0.6f);

        // Center text
        String label = "100";
        txtPaint.getTextBounds(label, 0, label.length(), textBounds);
        textY = centerY - (textBounds.top + textBounds.bottom) * 0.5f;

        // Arc oval
        arcOval.left = centerX - radius;
        arcOval.right = centerX + radius;
        arcOval.top = centerY - radius;
        arcOval.bottom = centerY + radius;
    }
}
```

This class is similar to the CircularChart class from Chapter 6. Unlike the CircularChart it doesn't extend a View, but everything else is almost the same. In the constructor we initialized Paint objects and rectangles. We let Android align text horizontally, instead of calculating textX coordinate ourselves like we did in Chapter 6.

```
txtPaint.setTextAlign(Align.CENTER);
```

In the changeSize() method which is similar to onSizeChanged() from Chapter 6, we calculated the x- and y-coordinates of the center of the view and the radius of the circle. We set the text size and calculated the textY coordinate to center text vertically. We used the hard-coded value of 100 to measure the top and bottom of a label.

```
String label = "100";
txtPaint.getTextBounds(label, 0, label.length(), textBounds);
textY = centerY - (textBounds.top + textBounds.bottom) * 0.5f;
```

Usually all digits of a font have the same top and bottom values, so instead of 100 we could have used any single digit. Finally we set the arc oval coordinates.

The draw() method takes two parameters: the Canvas to draw on and a value to draw. The value can be any number from 0 to 1. The code is similar to the onDraw() method of the CircularChart class from Section 6.4 of Chapter 6. In the CircularChart example we hard-coded the value to 0.65 or 65%, but in the ProgressHelper class we pass this value as a parameter.

ProgressView

This is the content view of the application.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.view.SurfaceHolder;
import android.view.SurfaceView;


public class ProgressView extends SurfaceView
{
    private ProgressHelper progress;


    public ProgressView(Context context)
    {
        super(context);
        progress = new ProgressHelper(context);
    }


    @Override
    protected void onSizeChanged(int width, int height, int oldw, int oldh)
    {
        progress.changeSize(width, height);
    }


    public void drawProgress(int value)
    {
        SurfaceHolder holder = this.getHolder();

        Canvas canvas = null;
        try
        {
            canvas = holder.lockCanvas();
            if(canvas != null)
            {
                canvas.drawColor(Color.WHITE);
                progress.draw(canvas, value/100.0f);
            }
        }
        finally
        {
            if(canvas != null)
            {
                holder.unlockCanvasAndPost(canvas);
            }
        }
    }
}
```

The ProgressView class extends the SurfaceView. It delegates most of the drawing to the ProgressHelper class.

In the constructor we created a ProgressHelper class.

```
progress = new ProgressHelper(context);
```

The onSizeChanged() method overrides the corresponding method of the View class. We delegated all calculations to the ProgressHelper class.

```
progress.changeSize(width, height);
```

The drawProgress() method is called by a worker thread to update the UI. This method takes a progress value in 0 to 100 range as a parameter.

Before we can draw, we have to get a reference to the Canvas object from the SurfaceHolder.

```
SurfaceHolder holder = this.getHolder();
Canvas canvas = null;
...
canvas = holder.lockCanvas();
```

After we got the reference to the Canvas object, we cleared the view

```
canvas.drawColor(Color.WHITE);
```

and delegated drawing of the progress indicator to the ProgressHelper class.

```
progress.draw(canvas, value/100.0f);
```

We used try-finally statement to make sure that even in case of an exception, the Canvas is unlocked.

```
try
{
    ...
}
```

```
finally
{
    ...
    holder.unlockCanvasAndPost(canvas);
}
```

ProgressThread

This class acts as a timer. It increases the progress value by 10% every half second until it reaches 100% and updates the progress view.

```
package com.example.graphics2d;


public class ProgressThread extends Thread
{
    private volatile boolean stopRequested = false;
    private ProgressView view;
    private int progress;


    public ProgressThread(ProgressView view)
    {
        this.view = view;
    }


    public void run()
    {
        while(!stopRequested && progress = 100)
        {
            view.drawProgress(progress);

            try
            {
                Thread.sleep(500);
            }
            catch(Exception ex)
            {
                break;
            }

            progress+=10;
        }
    }


    public void requestStop()
    {
        stopRequested = true;
    }
}
```

In the constructor we stored the reference to the ProgressView.

```
public ProgressThread(ProgressView view)
{
    this.view = view;
}
```

In the run() method, which overrides the corresponding method of the Thread class, we loop while either stop is requested or the progress value reaches 100%.

```
while(!stopRequested && progress = 100)
{
   ...
}
```

In the loop we draw the progress indicator

```
view.drawProgress(progress);
```

sleep for half a second

```
Thread.sleep(500);
```

and increase the progress value by 10%.

```
progress+=10;
```

Let's also discuss the stopRequested flag and requestStop() method.

```
volatile boolean stopRequested = false;
...
public void requestStop()
{
    stopRequested = true;
}
```

The common way to stop a Thread which performs repeating operations in a loop is to check a boolean flag inside the loop. The flag can be changed from another thread. The volatile keyword is used to make sure that the value of the stopRequested flag changed in another thread is immediately visible to the ProgressThread.

10.4. Example: SurfaceView and Handler in the same Application

You can use both the SurfaceView and Handler in the same application.

In this example we will change the stock chart application from Section 10.2 to use the SurfaceView to draw a chart and the "Loading..." message. The SurfaceView can be updated from any thread. We will draw a chart in a worker thread and the "Loading..." message in the main UI thread.

The rest of the UI can be only updated from the main UI thread. We will use the Handler class to send messages from a worker thread to the main UI thread to update the text field and Go button.

Let's review modified classes and interfaces.

MessageListener

This interface defines a method which will be called by a Handler after a chart is drawn to enable the text field and Go button.

```
package com.example.graphics2d;

public interface MessageListener
{
    public void afterDrawChart();
}
```

MessageHandler

The MessageHandler class extends the Handler class and overrides its handleMessage() method.

```
package com.example.graphics2d;

import android.os.Handler;
import android.os.Message;

public class MessageHandler extends Handler
{
    public static int MSG_AFTER_DRAW_CHART = 1;

    private MessageListener listener;


    public MessageHandler()
    {
    }


    public void setMessageListener(MessageListener l)
    {
        this.listener = l;
    }


    @Override
    public void handleMessage(Message msg)
    {
        if(listener == null) return;

        if(msg.what == MSG_AFTER_DRAW_CHART)
        {
            listener.afterDrawChart();
        }
    }
}
```

It is similar to the MessageHandler from Section 10.2. The message has changed. It doesn't have a ChartModel anymore. After a MSG_AFTER_DRAW_CHART message is received, we notify the listener.

```
public static int MSG_AFTER_DRAW_CHART = 1;
...
if(msg.what == MSG_AFTER_DRAW_CHART)
{
    listener.afterDrawChart();
}
```

ChartSurfaceView

This class is responsible for displaying a stock chart and the "Loading..." message.

```
package com.example.graphics2d;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.graphics.Paint.Cap;
import android.graphics.Paint.Join;
import android.graphics.Paint.Style;
import android.graphics.Path;
import android.util.AttributeSet;
import android.view.SurfaceHolder;
import android.view.SurfaceView;


public class ChartSurfaceView extends SurfaceView implements SurfaceHolder.
{
    private Matrix matrix;

    private Paint paint, textPaint;
    private Path path;

    private int deviceWidth, deviceHeight;


    public ChartSurfaceView(Context context)
    {
        super(context);
        init();
    }


    public ChartSurfaceView(Context context, AttributeSet attrs)
    {
        super(context, attrs);
        init();
    }


    private void init()
    {
        paint = new Paint(Paint.ANTI_ALIAS_FLAG);
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
        paint.setStrokeWidth(5);
        paint.setStrokeCap(Cap.ROUND);
        paint.setStrokeJoin(Join.ROUND);
        textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        textPaint.setTextAlign(Align.CENTER);
        textPaint.setColor(Color.BLACK);

        matrix = new Matrix();

        this.getHolder().addCallback(this);
    }


    public void drawLoading()
    {
        SurfaceHolder holder = this.getHolder();
        Canvas canvas = null;

        try
        {
            canvas = holder.lockCanvas();
            if(canvas != null)
            {
                canvas.drawColor(Color.WHITE);
                canvas.drawText("Loading...", deviceWidth/2, deviceHeight/2
            }
        }
        finally
        {
            if(canvas != null)
            {
                holder.unlockCanvasAndPost(canvas);
            }
        }
    }


    public void drawChart(ChartModel model)
    {
        SurfaceHolder holder = this.getHolder();
        Canvas canvas = null;

        try
        {
            canvas = holder.lockCanvas();
            if(canvas != null)
            {
                updatePath(model);
                canvas.drawColor(Color.WHITE);
                canvas.drawPath(path, paint);
            }
        }
        finally
        {
            if(canvas != null)
            {
                holder.unlockCanvasAndPost(canvas);
            }
        }
    }


    private void updatePath(ChartModel model)
    {
        // Init transformation matrix
        matrix.reset();

        matrix.postTranslate(0, -model.getMinPrice());
        float scaleX = (float)deviceWidth / (model.getMaxDay() - model.getM
        float scaleY = (float)deviceHeight / (model.getMaxPrice() - model.g
        matrix.postScale(scaleX, -scaleY);

        matrix.postTranslate(0, deviceHeight);

        // Create path
        path = new Path();
        path.moveTo(0, model.getPrice(0));

        for(int i = 1; i < model.getNumberOfDays(); i++)
        {
            path.lineTo(i, model.getPrice(i));
        }

        // Convert coordinates
        path.transform(matrix);
    }


    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        deviceHeight = h;
        deviceWidth = w;

        textPaint.setTextSize(h * 0.1f);
    }


    @Override
    public void surfaceCreated(SurfaceHolder holder)
    {
        Canvas canvas = null;

        try
        {
            canvas = holder.lockCanvas();
            if(canvas != null)
            {
                canvas.drawColor(Color.WHITE);
            }
        }
        finally
        {
            if(canvas != null)
            {
                holder.unlockCanvasAndPost(canvas);
            }
        }
    }


    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width,
    {
    }


    @Override
    public void surfaceDestroyed(SurfaceHolder holder)
    {
    }
```

```
}
```

The ChartSurfaceView extends the SurfaceView. We use its drawing surface instead of the main drawing area inherited from the View class.

Constructors are the same as before.

In the init() method we removed the setBackgroundColor() method.

```
setBackgroundColor(Color.WHITE);
```

The Color.WHITE is opaque. If you keep the setBackgroundColor() method, you would not see the surface underneath the window (main drawing area).

We added drawLoading() method to draw "Loading..." label and drawChart() method to draw a chart. Both methods get a Canvas from the SurfaceHolder.

```
SurfaceHolder holder = this.getHolder();
Canvas canvas = null;
...
canvas = holder.lockCanvas();
```

After drawing is done, the canvas is unlocked and UI is refreshed.

```
holder.unlockCanvasAndPost(canvas);
```

The code for drawing on the Canvas is the same as before.

The ChartSurfaceView implements the SurfaceHolder.Callback interface. We used its surfaceCreated() method to set the surface background color, right after the surface is created.

```
@Override
public void surfaceCreated(SurfaceHolder holder)
{
    Canvas canvas = null;

    try
    {
        canvas = holder.lockCanvas();
        if(canvas != null)
        {
            canvas.drawColor(Color.WHITE);
        }
    }
    finally
    {
        if(canvas != null)
        {
            holder.unlockCanvasAndPost(canvas);
        }
    }
}
```

ChartDataThread

The ChartDataThread class is a worker thread responsible for fetching market data and drawing a chart.

```
package com.example.graphics2d;

import android.os.Message;


public class ChartDataThread extends Thread
{
    private String ticker;
    private ChartSurfaceView view;
    private MessageHandler handler;


    public ChartDataThread(ChartSurfaceView view, MessageHandler handler, S
    {
        this.view = view;
        this.handler = handler;
        this.ticker = ticker;
    }


    public void run()
    {
        try
        {
            Thread.sleep(5000);
        }
        catch(Exception ex)
        {
        }

        ChartModel data = new ChartModel();
        view.drawChart(data);

        Message msg = Message.obtain();
        msg.what = MessageHandler.MSG_AFTER_DRAW_CHART;
        handler.sendMessage(msg);
    }
}
```

As before, we used 5 seconds delay to simulate a network communication delay.

```
Thread.sleep(5000);
```

After that we created a ChartModel with hard-coded S&P500 market data used for all queries

```
ChartModel data = new ChartModel();
```

and drew the chart.

```
view.drawChart(data);
```

Next we created a MSG_AFTER_DRAW_CHART message and sent it to the MessageHandler to notify the main UI thread that the chart is drawn and that the main UI thread can enable the text field and Go button.

```
Message msg = Message.obtain();
msg.what = MessageHandler.MSG_AFTER_DRAW_CHART;
handler.sendMessage(msg);
```

The main difference between this class and the ChartDataThread from Section 10.2 is that in this example we drew the chart in the worker thread. In Section 10.2 we sent a ChartModel to the main UI thread and drew the chart there.

Layout

We changed the layout from Section 10.2 to use new ChartSurfaceView. Everything else is the same.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <EditText
        android:id="@+id/txtTicker"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
    <Button
        android:id="@+id/btnGo"
        android:text="Go"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
</LinearLayout>
<com.example.graphics2d.ChartSurfaceView
    android:id="@+id/chartView"
    android:layout_weight="1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
</LinearLayout>
```

Activity

The ChartActivity is also almost the same. It implements the OnClickListener to handle clicks on Go button and the MessageListener to get notifications from the MessageHandler.

```
package com.example.graphics2d;

import android.app.Activity;
import android.content.pm.ActivityInfo;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.Window;
import android.widget.Button;
import android.widget.EditText;


public class ChartActivity extends Activity implements OnClickListener, Mes
{
    private Button btnGo;
    private EditText txtTicker;
    private ChartSurfaceView chartView;
    private MessageHandler handler;


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

        this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSC
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);

        this.setContentView(R.layout.chart_mt);

        txtTicker = (EditText)findViewById(R.id.txtTicker);

        btnGo = (Button)findViewById(R.id.btnGo);
        btnGo.setOnClickListener(this);

        chartView = (ChartSurfaceView)findViewById(R.id.chartView);

        handler = new MessageHandler();
        handler.setMessageListener(this);
    }


    @Override
    public void onClick(View v)
    {
        btnGo.setEnabled(false);
        txtTicker.setEnabled(false);

        chartView.drawLoading();

        String ticker = txtTicker.getText().toString();

        ChartDataThread thread = new ChartDataThread(chartView, handler, ti
        thread.start();
    }


    @Override
    public void afterDrawChart()
    {
        btnGo.setEnabled(true);
        txtTicker.setEnabled(true);
    }
}
```

We didn't change the constructor.

In the onClick() method which is called when Go button is clicked, we disabled the text field and Go button to prevent a user from submitting a new request before the previous request is completed.

```
btnGo.setEnabled(false);
txtTicker.setEnabled(false);
```

Next we drew the "Loading..." message.

```
chartView.drawLoading();
```

After getting a stock ticker from the EditText control we created and started a worker thread.

```
ChartDataThread thread = new ChartDataThread(chartView, handler, ticker);
thread.start();
```

The worker thread fetches market data for a stock ticker and draws a chart. After that it sends a MSG_AFTER_DRAW_CHART message to the MessageHandler. The MessageHandler handles the message and calls the afterDrawChart() method of the MessageListener interface.

```
@Override
public void afterDrawChart()
{
    btnGo.setEnabled(true);
    txtTicker.setEnabled(true);
}
```

In this method we enabled the text field and Go button.

The main difference between this class and the old version from Section 10.2 is that before we drew a chart in the main UI thread in the onChartData() method by calling

```
chartView.setModel(model);
```

In this example we drew a chart in a worker thread.