COMP-304
Fall 2018

Review of Lecture 4

- **Using Menus**:
  - **options menu**
  - **context menu**
  - **popup menu**
  - A menu resource file is stored in /res/menu folder
  - Menu items are defined in item tags
  - **Handling Options Menu:**
    - Override onCreateOptionsMenu to show the menu
    - Override onOptionsItemSelected to handle menu item event
- **Handling Context Menu:**
  - Register the View to which the context menu should be associated by calling registerForContextMenu() and pass it the View.
  - Implement the onCreateContextMenu() method in your Activity or Fragment.
  - Implement onContextItemSelected() in your activity

Review of Lecture 4

- **Validation**:
  - Using **attributes** of Android controls, **InputFilter** class, and other controls (**Spinner**, etc)
- **CheckBox control**
  - **setOnClickListener** to register
  - **onClick** to handle check event
  - **isChecked** to find out the state of control
- **RadioButton** and **RadioGroup Controls**
  - **setOnCheckedChangeListener** to register the group
  - **onCheckedChanged** to handle click event
  - **checkedId** to find out which button is selected
- **Progress Indicators**
  - **indeterminate** progress bar
  - **horizontal progress bar**
  - **setProgress** method

Review of Lecture 4

- **Threads in Android Apps**:
  - When an application is launched, the "main." thread or **UI thread**
  - Performing long operations will block the whole UI
  - Android UI toolkit is not thread-safe
- Two rules to Android's single thread model:
  1. **Do not block the UI thread**
  2. **Do not access the Android UI toolkit from outside the UI thread**
- Android offers **several ways to access the UI thread** from other threads

- **Using Handler class**

```
Handler handler = new Handler();
//do some work in background thread
new Thread(new Runnable()
{
    public void run()
    {
        //do some work here—-
        while (progressStatus < 500)
        {
            progressStatus = doSomeWork();
            // do not call setProgress here
            //*****************************
            //—-Update the progress bar—-
            handler.post(new Runnable()
            {
                public void run() {
                //update the bar here, is OK!
progressBar.setProgress(progressStatus);
                } //
            });
        }
    } //
} ).start();
```

Review of Lecture 4

- **SeekBar**:
  - **setOnSeekBarChangeListener**
  - **onProgressChanged** to handle progress changed event
- **RatingBar**:
  - **setOnRatingBarChangeListener**
  - **onRatingChanged**
- **TimePicker** :
  - **setOnTimeChangedListener** to register the control
  - **onTimeChanged** to handle time changed event
  - **Date** class to get Date information
- **DatePicker**:
  - **OnDateChangedListener** to register the control
  - **onDateChanged** to handle a date change
  - year, month, day arguments
- **Styles**
  - **separate the design from the content**
  - <style> tag **/res/values/**styles.xml
  - Use attribute**:** **style**=”@style/style_name”
  - **Themes**: Applied to the whole activity

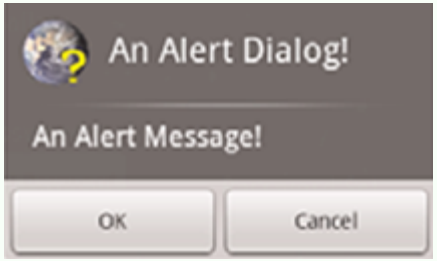
Data-Driven Containers and Drawing in Android apps

Objectives:

- **Use Dialog boxes**
- **Use Data-Driven Containers in Android UIs**
  - ArrayAdapter and CursorAdapter
  - ListView and ListActivity
- **Using Drawings in Android Apps**
  - Drawing on Screen
  - Drawing on ImageView
  - Drawing Shapes
  - Drawing Text

Working with Dialogs

- A dialog is a small window that does not fill the screen and **prompts the user to make a decision** or **enter additional information**.
- The **Dialog** class is the base class for dialogs, but you should avoid instantiating Dialog directly, use one of the following subclasses:
- **AlertDialog**
  - A dialog that can show:
    - a **title**,
    - up to three **buttons**,
    - a list of selectable **items**
    - or a **custom layout**.



- **DatePickerDialog** or **TimePickerDialog**
  - A dialog with a pre-defined UI that allows the user to select a date or time.

DialogFragment

- Google recommends using **DialogFragment**
- In **onCreateDialog,** use an **AlertDialog** builder to create a simple AlertDialog with Yes/No confirmation buttons. Less code!
- To build an AlertDialog:
  - Instantiate an AlertDialog.Builder with its constructor

```
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
```

  - Chain together various setter methods to set the dialog characteristics

```
builder.setMessage(R.string.dialog_message)
       .setTitle(R.string.dialog_title);
```

  - Get the AlertDialog from create()

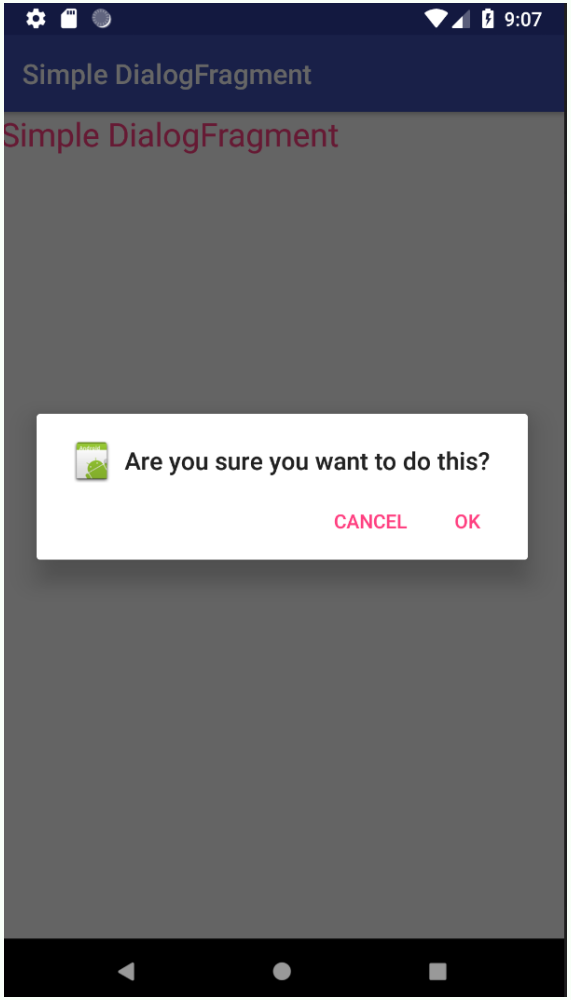
```
AlertDialog dialog = builder.create();
```

DialogFragment

- To add action buttons call the setPositiveButton() and setNegativeButton() methods:

```
// Add the buttons
builder.setPositiveButton(R.string.ok,
new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            // User clicked OK button
        }
    });

builder.setNegativeButton(R.string.cancel,
new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            // User cancelled the dialog
        }
    });
```



Showing Time Passage with the Chronometer

- You can use the **Chronometer** control as a timer
- The Chronometer control can be formatted with text, as shown in this XML layout resource definition:

```
<Chronometer
android:id="@+id/Chronometer01"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:format="Timer: %s" />
```

- A Chronometer won't show the passage of time until its **start()** method is called.
- To stop it, simply call its **stop()** method
- Use **setBase()** method to set it to **count from a particular time** in the past instead of from the time it's started.

Showing Time Passage with the Chronometer

- In this example code, the timer is retrieved from the View by its resource identifier.
  - We then check its base value and set it to 0.
  - Finally, we start the timer counting up from there.

```
final Chronometer timer =
(Chronometer)findViewById(R.id.Chronometer01);
long base = timer.getBase();
Log.d(ViewsMenu.debugTag, “base = “+ base);
timer.setBase(0);
timer.start();
```

Displaying the Time

- There are **two clock controls** available to display the time:
  - the **DigitalClock** and **AnalogClock** controls
- The **DigitalClock** control is a compact text display of the current time in standard numeric format based on the users' settings.
  - It is a **TextView**, so anything you can do with a TextView you can do with this control, except change its text.
  - You can **change the color and style of the text**.
- By default, the **DigitalClock** control shows the seconds and automatically updates as each second ticks by.
- Here is an example of an XML layout resource definition for a DigitalClock control:

```
<DigitalClock
android:id="@+id/DigitalClock01"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

Displaying the Time

- The **AnalogClock** control is a dial-based clock with a basic **clock face with two hands**, one for the minute and one for the hour.
  - It updates automatically as **each minute** passes
- Here is an example of an XML layout resource definition for an AnalogClock control:

```
<AnalogClock
android:id=”@+id/AnalogClock01”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content” />
```

- The AnalogClock control’s clock face is simple
  - You can set its minute and hour hands.
- You can also set the clock face to specific drawable resources

Using Data-Driven Containers

- Data-Driven Containers are all types of **AdapterView** controls.
- An **AdapterView** control contains a set of **child View controls to display data from some data source**.
- An **Adapter reads data from some data source** and **provides a View object** based on some rules, depending on the type of Adapter used.
  - This **View is used to populate the child View objects** of a particular AdapterView

Using Data-Driven Containers

- The most common Adapter classes are the **CursorAdapter** and the **ArrayAdapter**.
- The **CursorAdapter** gathers data from a **Cursor**, whereas the ArrayAdapter gathers data from an **array**.
  - A CursorAdapter is a good choice to use when using **data from a database**.
- The **ArrayAdapter** is a good choice to use when there is only **a single column of data** or when the data comes from **a resource array**.

Using the **ArrayAdapter**

- An **ArrayAdapter** binds each element of the array to a single View object within the layout resource.
- Here is an example of creating an ArrayAdapter:

```
String[] programs = {
        "Software Engineering Technology",
        "Interactive Gaming",
        "Health Informatics Technology",
        "Software Systems Design"
    };
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_dropdown_item_1line, programs);

AutoCompleteTextView textView = (AutoCompleteTextView)
        findViewById(R.id.txtPrograms);
textView.setThreshold(3);
textView.setAdapter(adapter);
```

Binding Data to the **AdapterView**

- Apply the Adapter object to one of the AdapterView controls.
- Here is an example of this with a **ListView**:

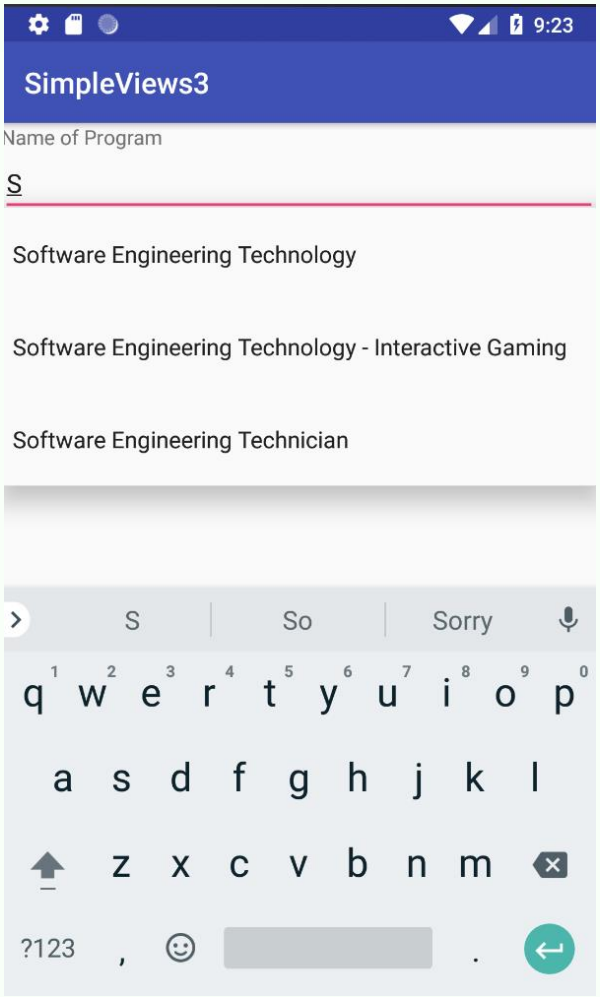
```
((ListView)findViewById(R.id.list)).setAdapter(adapter);
```

- The call to the **setAdapter()** method of the AdapterView, a ListView in this case, should come **after your call to setContentView()**.
- This is all that is required to bind data to your AdapterView.

Binding Data to the **AdapterView**

Figure below shows binding data to an ArrayAdapter

(**SimpleViews3** example)



Using the **ListActivity**

- The **ListView** control is commonly used for **full-screen menus** or lists of items from which a user selects.
- As such, you might consider using **ListActivity as the base class** for such screens.
- First, to **handle item events**, you now need to provide an implementation in your **ListActivity**.
  - For instance, the equivalent of onListItemClickListener is to implement the **onListItemClick()** method within your ListActivity.
- Second, to **assign an Adapter**, you need a call to the **setListAdapter()** method.
- You do this **after the call to the setContentView()** method.

Using the **ListActivity**

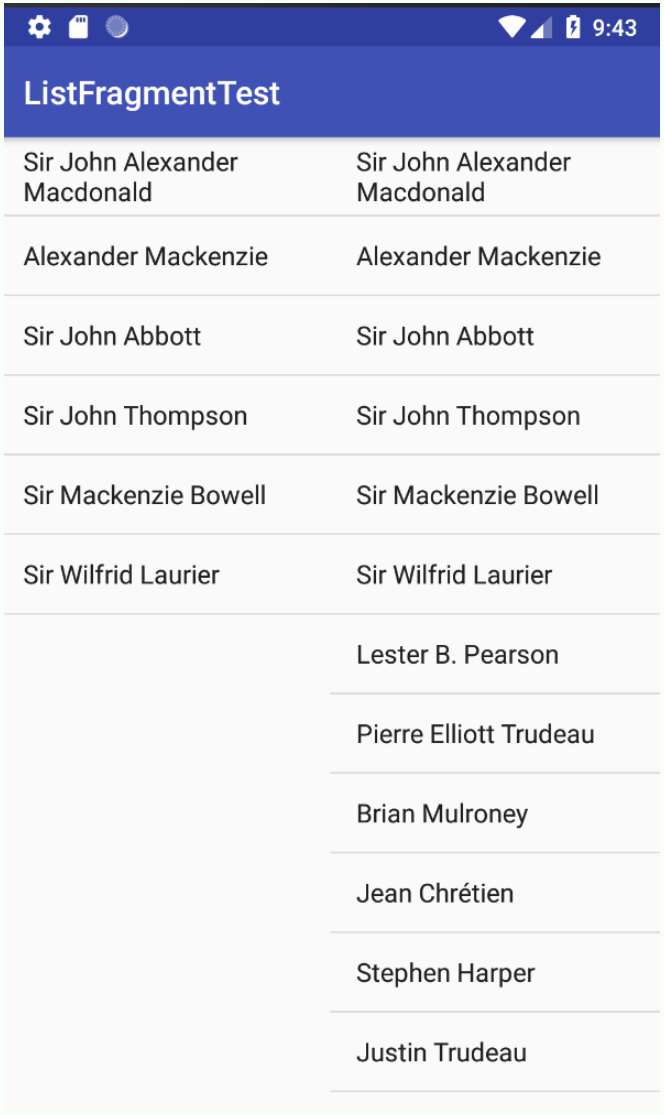
- To use ListActivity, the layout that is set with the setContentView() method must contain a ListView with the identifier set to **android:list**; this cannot be changed.
- Second, you can also have a View with an identifier set to android:empty to have a View display when no data is returned from the Adapter.
- Finally, this **works only with ListView controls**, so it has limited use.

Handling Selection Events

- ListView, GridView, and Gallery—enable your application to monitor for click events in the same way.
- You need to call **setOnItemClickListener**() on your AdapterView and pass in an implementation of the **AdapterView.OnItemClickListener** class.
- Here is an example implementation of this class:

```
av.setOnItemClickListener(
new AdapterView.OnItemClickListener() {
    public void onItemClick(
        AdapterView<?> parent, View view,
        int position, long id) {
        Toast.makeText(Scratch.this, “Clicked _id=”+id,
        Toast.LENGTH_SHORT).show();
    }
});
```

ListFragmentTest and ListViewExample



Drawing on the Screen

- Canvas objects are used for drawing.
- Create a subclass of the **View** class and write the drawing code in **onDraw()** method.

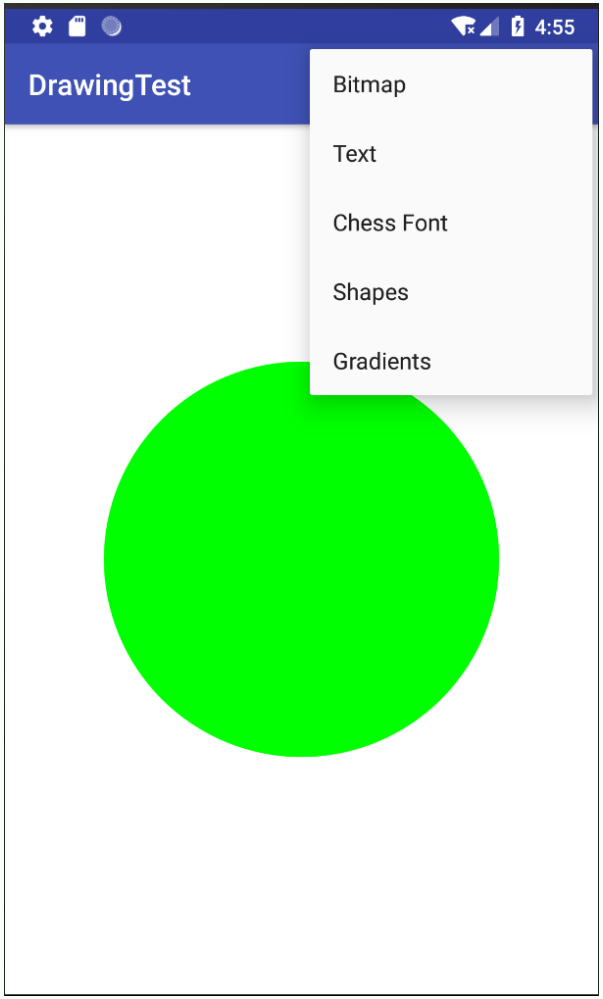
```
private static class ViewWithRedDot extends View {
        public ViewWithRedDot(Context context) {
        super(context);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.BLACK);
        Paint circlePaint = new Paint();
        circlePaint.setColor(Color.RED);
        canvas.drawCircle(canvas.getWidth()/2,
        canvas.getHeight()/2,
        canvas.getWidth()/3, circlePaint);
    }
}
```

Drawing on the Screen

- To use the canvas just display the View like any other layout.
- In the onCreate() method use the following code:

```
setContentView(new ViewWithRedDot(this));
```

Drawing on the Screen



Drawing on the Screen

- The **Canvas** (android.graphics.Canvas) contains the necessary methods available for drawing images, text, shapes, and support for clipping regions.
- The dimensions of the **Canvas** are bound by the container view.
- You can retrieve the size of the Canvas using the getHeight() and getWidth() methods.
- **Paint** (android.graphics.Paint) class encapsulates the **style** and **complex color** and rendering information, which can be applied to a drawable like a **graphic**, **shape**, or piece of **text** in a given Typeface.

Drawing on the Screen

- You can set the color of the Paint using the setColor() method.
- Standard colors are predefined within the android.graphics.**Color** class.
- For example, the following code sets the paint color to red:

```
Paint redPaint = new Paint();
redPaint.setColor(Color.RED);
```

- The following code instantiates a Paint object with antialiasing enabled:

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

Working with Paint Styles

- Paint style controls how an object is filled with color.
- For example, the following code instantiates a Paint object and sets the Style to STROKE, which signifies that the object should be painted as a **line drawing and not filled** (the default):

```
Paint linePaint = new Paint();
linePaint.setStyle(Paint.Style.STROKE);
```

Working with Paint Gradients

- You can create a gradient of colors using one of the gradient subclasses.
- The different gradient classes, including LinearGradient, RadialGradient, and SweepGradient, are available under the superclass android.graphics.Shader.
- All gradients need at least two colors—a **start color** and an **end color**—but might contain any number of colors in an array.
- The different types of gradients are differentiated by the direction in which the gradient "flows."
- Gradients can be set to mirror and repeat as necessary.
- You can set the Paint gradient using the setShader() method.

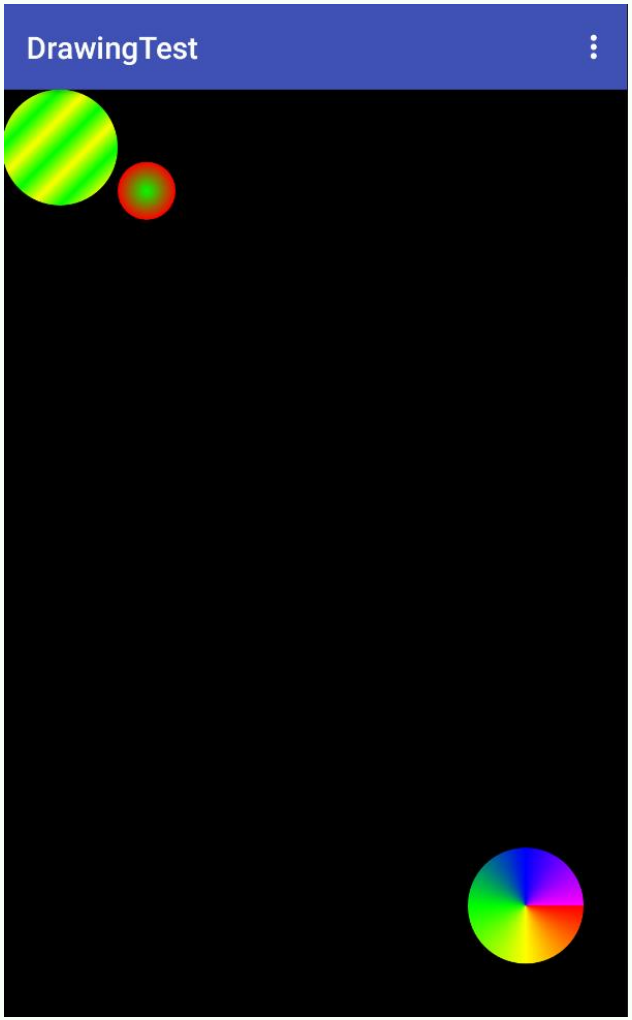


Figure 9.2 An example of a LinearGradient (top), a RadialGradient (right), and a SweepGradient (bottom).

Working with Linear Gradients

- A **linear** gradient is one that **changes colors along a single straight line**.
- You can achieve this by creating a **LinearGradient** and setting the Paint method **setShader()** before drawing on a Canvas, as follows:

```
Paint circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
LinearGradient linGrad = new LinearGradient(0, 0, 25, 25,
Color.RED, Color.BLACK,
Shader.TileMode.MIRROR);
circlePaint.setShader(linGrad);
canvas.drawCircle(100, 100, 100, circlePaint);
```

Working with Text

- Android provides several default **font typefaces** and styles.
- Applications can also use **custom fonts** by including font files as application assets and loading them using the AssetManager.
- By default, Android uses the Sans Serif typeface, but Monospace and Serif typefaces are also available.
- The following code draws some antialiased text in the default typeface (Sans Serif) to a Canvas:

Working with Text

```
Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
Typeface mType;
mPaint.setTextSize(16);
mPaint.setTypeface(null);
canvas.drawText("Default Typeface", 20, 20, mPaint);
```

- You can also load a different typeface, such as Monotype:

```
Typeface mType = Typeface.create(Typeface.MONOSPACE,
Typeface.NORMAL);
```

- If you prefer *italic text, simply set the style of the typeface* and the font family:

```
Typeface mType = Typeface.create(Typeface.SERIF,
Typeface.ITALIC);
```

- You can set properties of a typeface such as antialiasing, underlining, and strikethrough using the **setFlags()** method of the Paint object:

```
mPaint.setFlags(Paint.UNDERLINE_TEXT_FLAG);
```

Working with Text

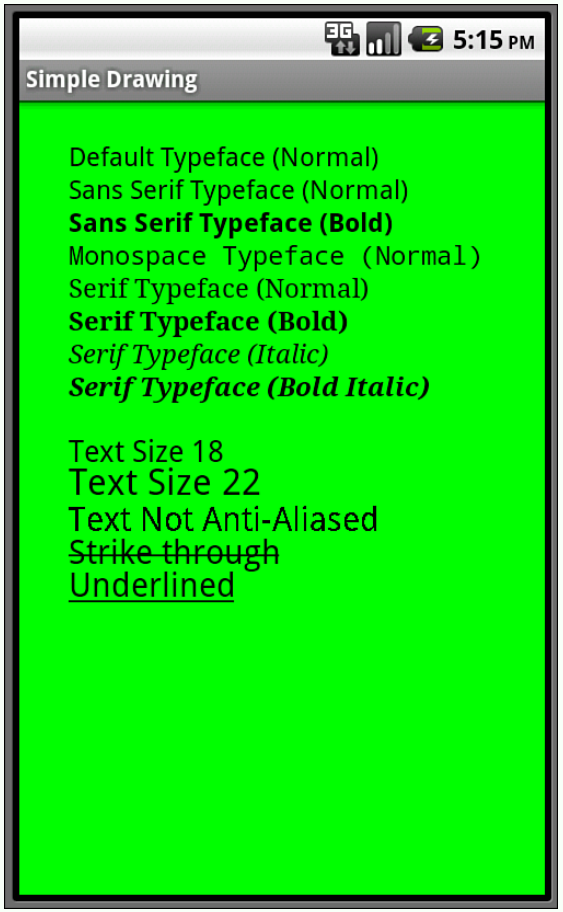


Figure 9.3 Some typefaces and typeface styles available on Android.

Drawing Bitmap Graphics on a Canvas

- The core class for bitmaps is android.graphics.**Bitmap**.
- You can draw bitmaps onto a valid Canvas, such as **within the onDraw() method** of a View, using one of the drawBitmap() methods.
- For example, the following code loads a Bitmap resource and draws it on a canvas:

```
Bitmap pic =
    BitmapFactory.decodeResource(getResources(),
R.drawable.bluejay);
canvas.drawBitmap(pic, 0, 0, null);
```

Transforming Bitmaps Using Matrixes

- Use the Matrix class to perform tasks such as **mirroring** and **rotating** graphics, among other actions.
- The following code uses the **createBitmap()** method to generate a new Bitmap that is a mirror of an existing Bitmap called pic:

```
Matrix mirrorMatrix = new Matrix();
mirrorMatrix.preScale(-1, 1);
Bitmap mirrorPic = Bitmap.createBitmap(pic, 0, 0,
pic.getWidth(), pic.getHeight(), mirrorMatrix, false);
```

- You can perform a 30-degree rotation in addition to mirroring by using this Matrix instead:

```
Matrix mirrorAndTilt30 = new Matrix();
mirrorAndTilt30.preRotate(30);
mirrorAndTilt30.preScale(-1, 1);
```

Transforming Bitmaps Using Matrixes

Figure 9.5 A single-source bitmap: scaled, tilted, and mirrored using Android Bitmap classes.

Working with Shapes

- You can define and draw primitive shapes such as rectangles and ovals using the **ShapeDrawable** class in conjunction with a variety of specialized **Shape** classes.
- You can define **Paintable** drawables as XML resource files, but more often, especially with more complex shapes, this is done programmatically.

Defining Shape Drawables as XML Resources

- The following resource file called /res/drawable/green_rect.xml describes a simple, green rectangle shape drawable:

```
<?xml version=”1.0” encoding=”utf-8”?>
<shape xmlns:android=
“http://schemas.android.com/apk/res/android”
android:shape=”rectangle”>
<solid android:color=”#0f0”/>
</shape>
```

- You can then load the shape resource and set it as the Drawable as follows:

```
ImageView iView =
    (ImageView)findViewById(R.id.ImageView1);
iView.setImageResource(R.drawable.green_rect);
```

Defining Shape Drawables as XML Resources

- Many **Paint properties can be set via XML as part of the Shape definition**.
- For example, the following **Oval** shape is defined with a linear gradient (red to white) and stroke style information:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="oval">
<solid android:color="#f00"/>
<gradient android:startColor="#f00"
android:endColor="#fff"
android:angle="180"/>
<stroke android:width="3dp" android:color="#00f"
android:dashWidth="5dp" android:dashGap="3dp"/>
</shape>
```

Defining Shape Drawables Programmatically

- You can also define these **ShapeDrawable** instances programmatically.
- The different shapes are available as classes within the android.graphics.drawable.**shapes** package.
- For example, you can programmatically define the aforementioned green rectangle as follows:

```
ShapeDrawable rect = new ShapeDrawable(new RectShape());
rect.getPaint().setColor(Color.GREEN);
```

- You can then set the Drawable for the ImageView directly:

```
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rect);
```

Drawing Different Shapes

- Some of the different shapes available within the android.graphics.drawable.shapes package include:
  - Rectangles (and squares)
  - Rectangles with rounded corners
  - Ovals (and circles)
  - Arcs and lines
  - Other shapes defined as paths

Drawing Rectangles and Squares

- Drawing rectangles and squares (rectangles with equal height/width values) is simply a matter of creating a ShapeDrawable from a RectShape object.
- The **RectShape** object has no dimensions but is bound by the container object- in this case, the ShapeDrawable.
- You can set some basic properties of the ShapeDrawable, such as the Paint **color** and the default **size**.

Drawing Rectangles and Squares

```
ShapeDrawable rect = new ShapeDrawable(new
    RectShape());
rect.setIntrinsicHeight(2);
rect.setIntrinsicWidth(100);
rect.getPaint().setColor(Color.MAGENTA);
ImageView iView =
    (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rect);
```

Drawing Rectangles with Rounded Corners

- Simply create a ShapeDrawable from a RoundRectShape object.
- The RoundRectShape requires an array of eight float values, which signify the radii of the rounded corners.

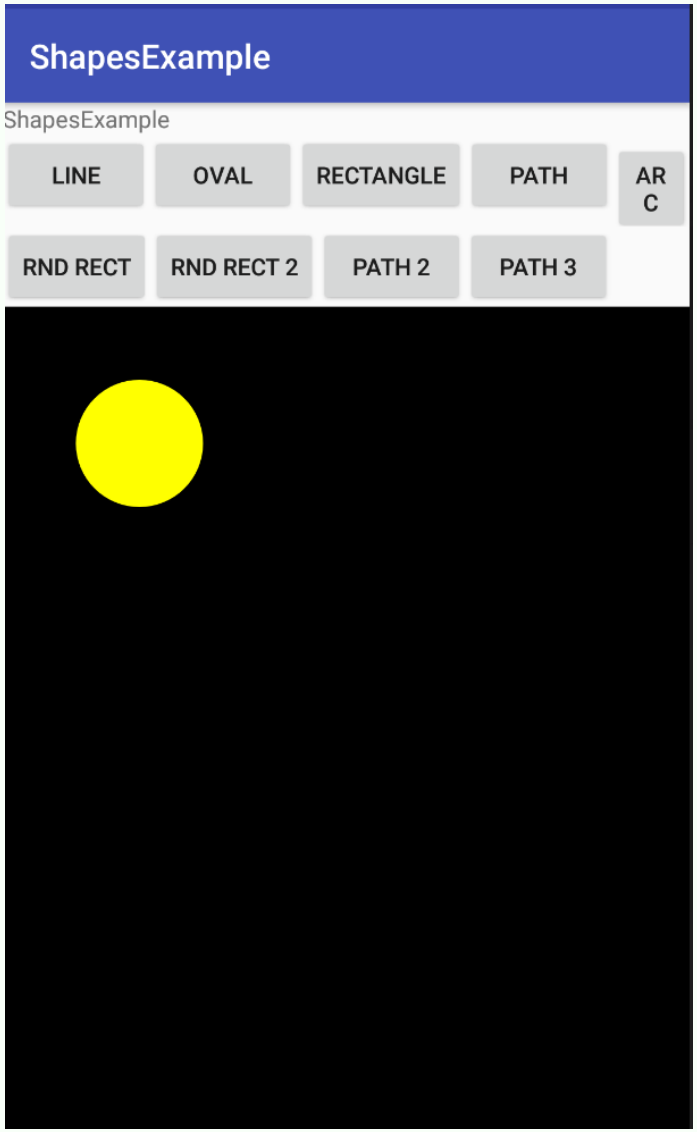
```
ShapeDrawable rndrect = new ShapeDrawable(
new RoundRectShape( new float[] { 5, 5, 5, 5, 5, 5, 5, 5 },
null, null));
rndrect.setIntrinsicHeight(50);
rndrect.setIntrinsicWidth(100);
rndrect.getPaint().setColor(Color.CYAN);
ImageView iView =
    (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rndrect);
```

Drawing Ovals and Circles

- You can create ovals and circles (which are ovals with equal height/width values) by creating a ShapeDrawable using an OvalShape object.
- The **OvalShape** object has no dimensions but is bound by the container object—in this case, the ShapeDrawable.
- You can set some basic properties of the ShapeDrawable, such as the Paint color and the default size.
- For example, here we create a red oval that is 40-pixels high and 100-pixels wide, which looks like a Frisbee:

```
ShapeDrawable oval = new ShapeDrawable(new OvalShape());
oval.setIntrinsicHeight(40);
oval.setIntrinsicWidth(100);
oval.getPaint().setColor(Color.RED);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(oval);
```

Drawing Ovals and Circles



Drawing Arcs

- You can draw arcs depending on the sweep angle you specify.
- You can create arcs by creating a ShapeDrawable by using an **ArcShape** object.
- The ArcShape object requires two parameters: a **startAngle** and a **sweepAngle**.
- The following code creates an arc that looks like a magenta Pac-Man:

```
ShapeDrawable pacMan =
new ShapeDrawable(new ArcShape(0, 345));
pacMan.setIntrinsicHeight(100);
pacMan.setIntrinsicWidth(100);
pacMan.getPaint().setColor(Color.MAGENTA);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(pacMan);
```

Drawing Arcs

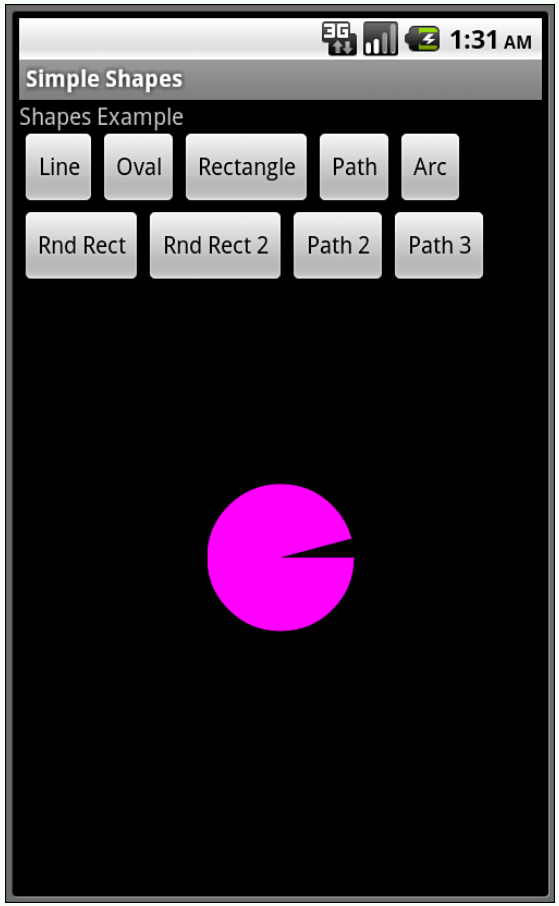


Figure 9.10 A magenta arc of 345 degrees (resembling Pac-Man).

Drawing Paths

- You can specify any shape you want by breaking it down into a **series of points along a path**.
- The android.graphics.**Path** class encapsulates a series of lines and curves that make up some larger shape.
- For example, the following Path defines a rough five-point star shape:

```
Path p = new Path();
p.moveTo(50, 0);
p.lineTo(25,100);
p.lineTo(100,50);
p.lineTo(0,50);
p.lineTo(75,100);
p.lineTo(50,0);
```

Drawing Paths

- You can then encapsulate this star Path in a **PathShape**, create a ShapeDrawable, and paint it yellow.

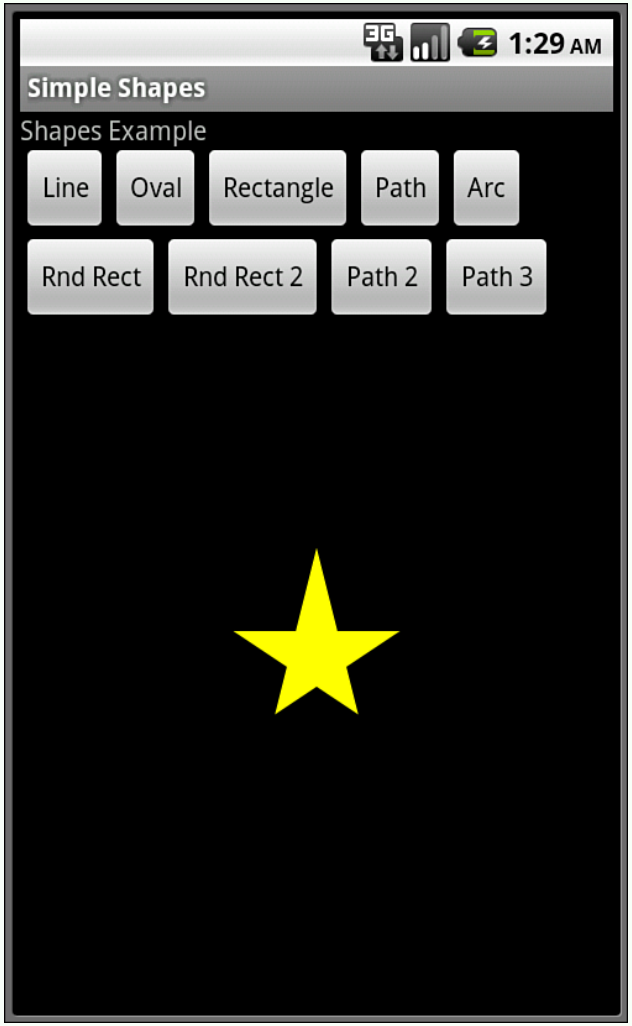
```
ShapeDrawable star =
new ShapeDrawable(new PathShape(p, 100, 100));
star.setIntrinsicHeight(100);
star.setIntrinsicWidth(100);
star.getPaint().setColor(Color.YELLOW);
```

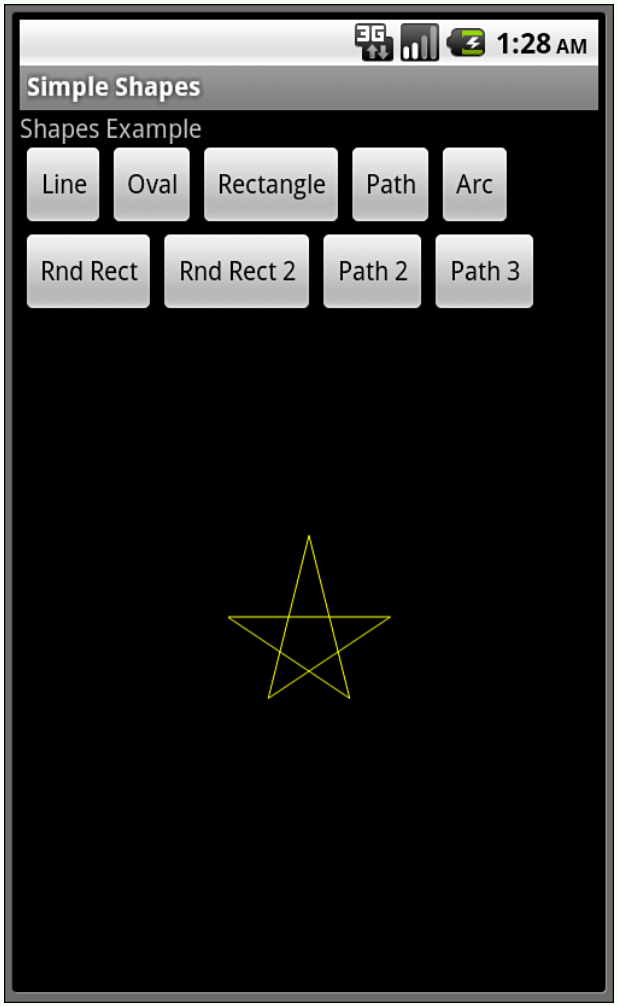
- By default, this generates a star shape filled with the Paint color yellow.
- You can set the Paint style to Stroke for a line drawing of a star.

```
star.getPaint().setStyle(Paint.Style.STROKE);.
```

Drawing Paths – ShapesExample

- Star shapes





Drawing on ImageView

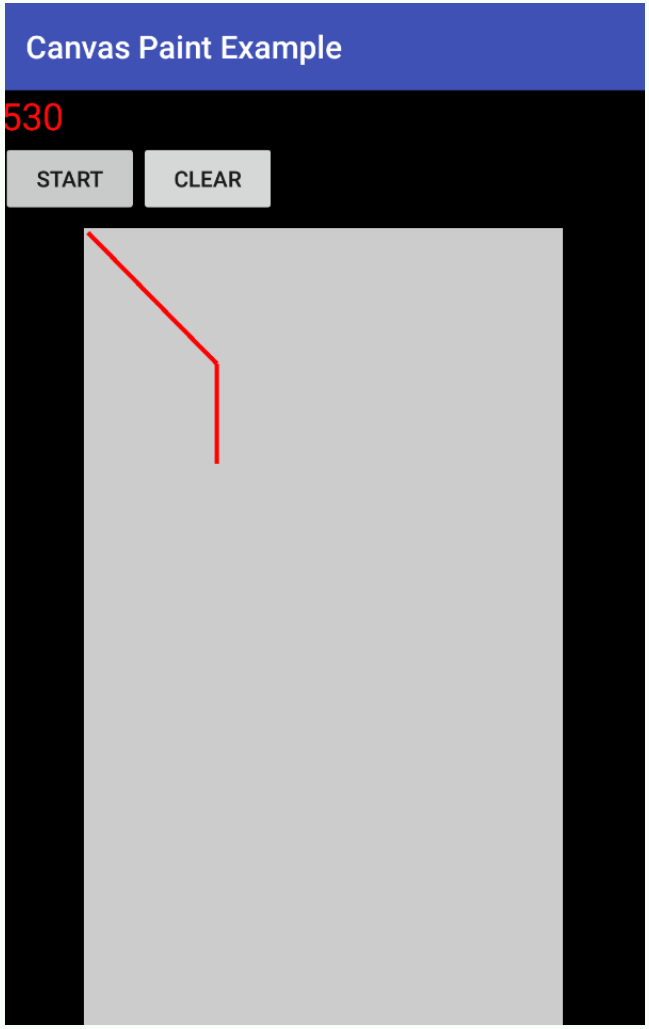
- Create a BitmMap as content view for the image
- Construct a canvas with the specified bitmap to draw into
- Create a Paint object
- Use Canvas methods (drawLine, etc.) to draw on the image

  canvas.**drawLine**(startx, starty, endx, endy, paint);
- You may use keys on your device to control your drawings, etc.

Drawing on ImageView

```
//Activate the DPAD on emulator:
//change the settings in config.ini file in .android folder for your emulator
//hw.dPad=yes
//hw.mainKeys=yes
public boolean onKeyDown(int keyCode, KeyEvent event)
{
      if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN)
      {
            reusableImageView.setFocusable(true);
            reusableImageView.requestFocus();
            endy=endy+5;
            drawLine( keyCode,canvas);
            reusableImageView.invalidate();
            return true;
      }
      return false;
   }
```

CanvasPaintExample example



References

- Textbook
- Android Documentation
- Android Wireless Application Development book (Shane Conder and Lauren Darcey)