UCD School of Computer Science

Model View Control – Graphic User Interface (GUI)

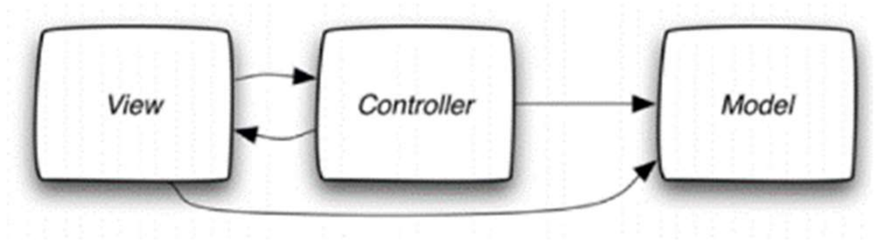
Dr. Abraham(Abey) Campbell

Objective

- Model-View-Control (MVC) model
- Structure our code using the MVC architecture
- Use various GUI components
- Define and use Styles and Themes
- Handle Events
- Development process

Model View Controller

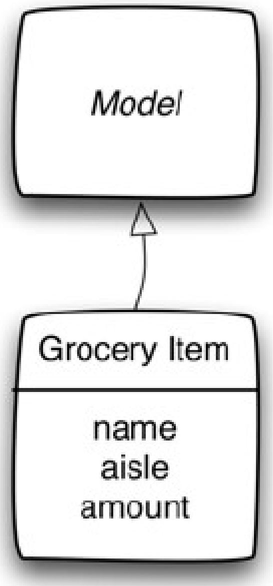
MVC, or model-view-controller, is a common way of **thinking about apps** and is used by web programmers, Windows and Mac programmers, and iPhone app programmers (like you). It's useful for any application that has a user interface.



On MVC – The Model

A model is anything in your app that **isn't related to the user interface**. In applications that store data, models are used to represent the things that you store. **"HOW IT IS GOING TO BE STORED!!"**

For **example**, imagine a grocery list application where you keep track of everything that you want to buy, where it is in the store, and how much of it you need. A model can be used to represent the items you want to buy. Each item has a name (such as "Bananas"), the aisle (such as "Produce"), and the amount you want to buy (such as "One bunch").

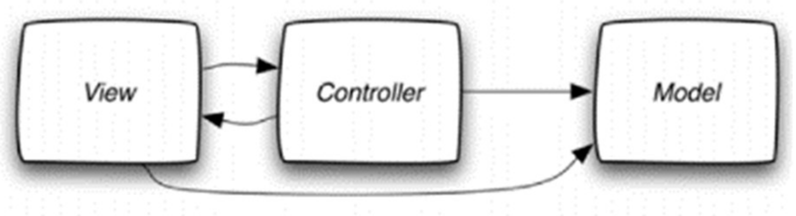


On MVC – The View

- A view is what you see on the screen and interact with. When you draw sketches of iPhone apps or use Interface Builder, you're drawing the view.

- **Each part of the user interface in a separate view. That means each button you click, text box you fill in, or check box you check is its own view.**

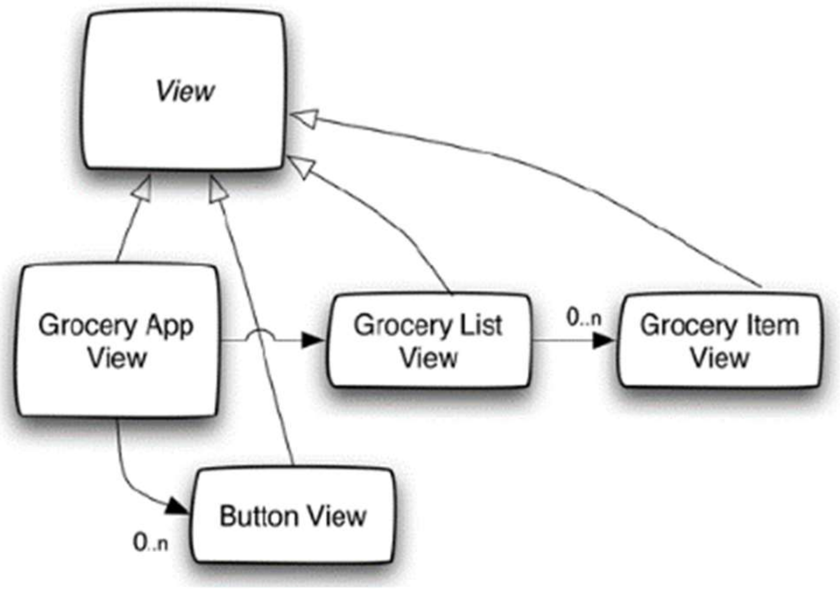
- Views do more than just show the user what is going on—they're also responsible for letting the rest of your application know they're being touched.



On MVC – The View Example

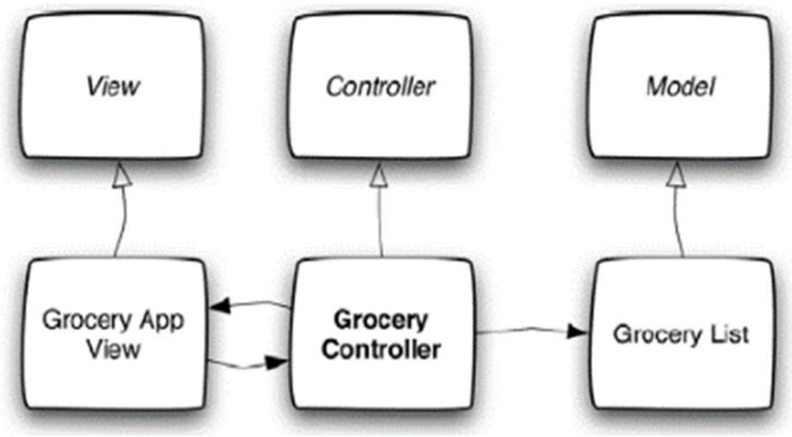
- Views do more than just show the user what is going on—they're also responsible for letting the rest of your application know they're being touched.

- On that grocery list, the list you see is the view. **Each item is in its own view, and when you touch the item to cross it off the list, its view is the part of the app that is first to know.**



On MVC – The Controller

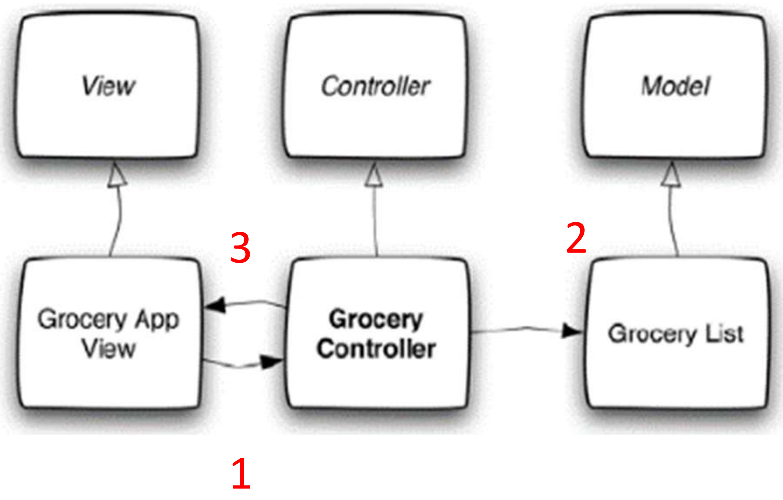
- **A controller is where you coordinate everything**.
- Your controller decides what to do when a button is clicked, when to show a different view, and what models should be changed.
- If you were to draw a flowchart of how your app works, a lot of that would be represented in your controllers



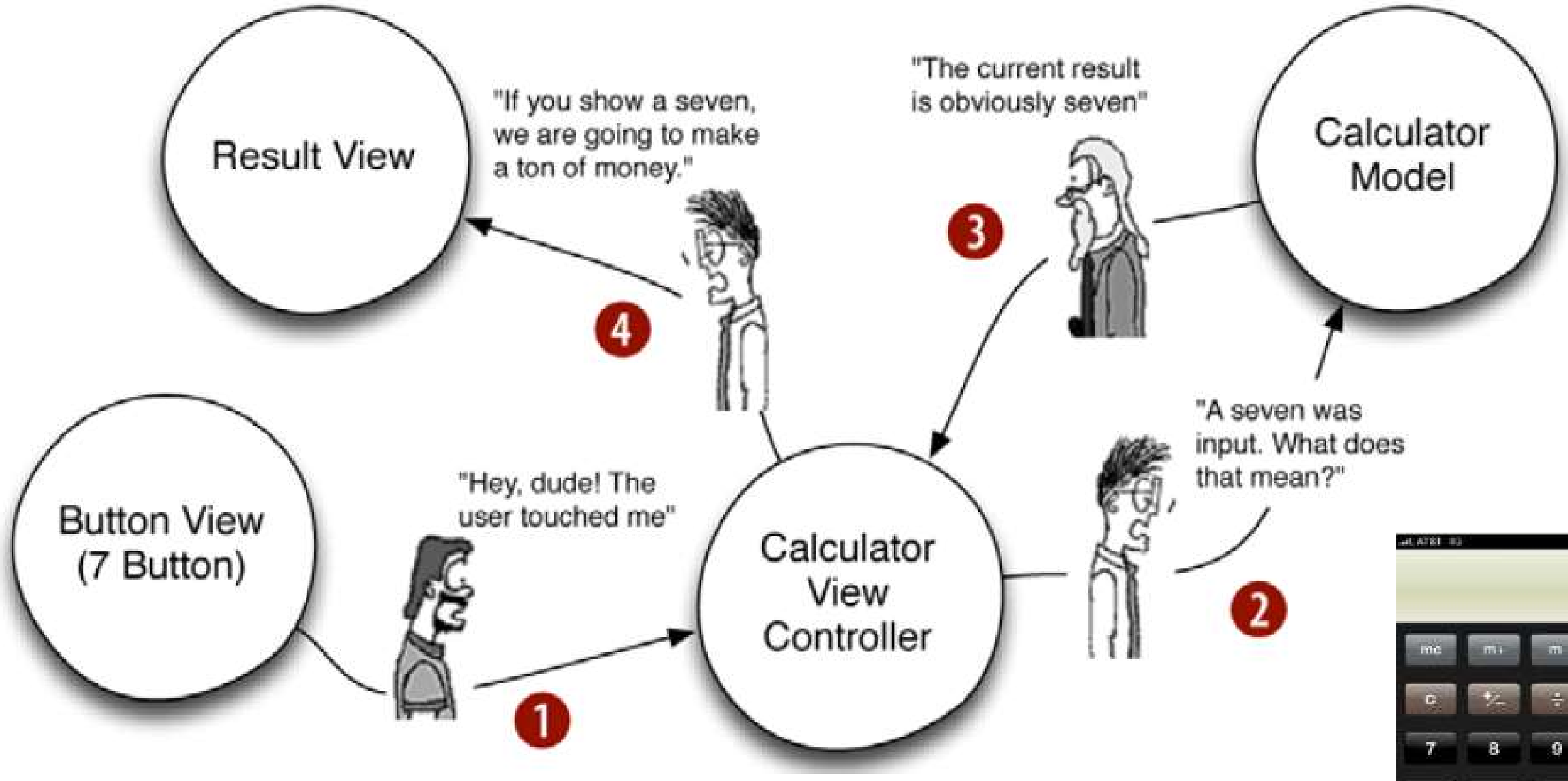
On MVC – The Controller, example

- So, using a grocery list app example again,
  - when you touch the Grocery Item View on your phone to indicate that you've put the item in your shopping cart ..

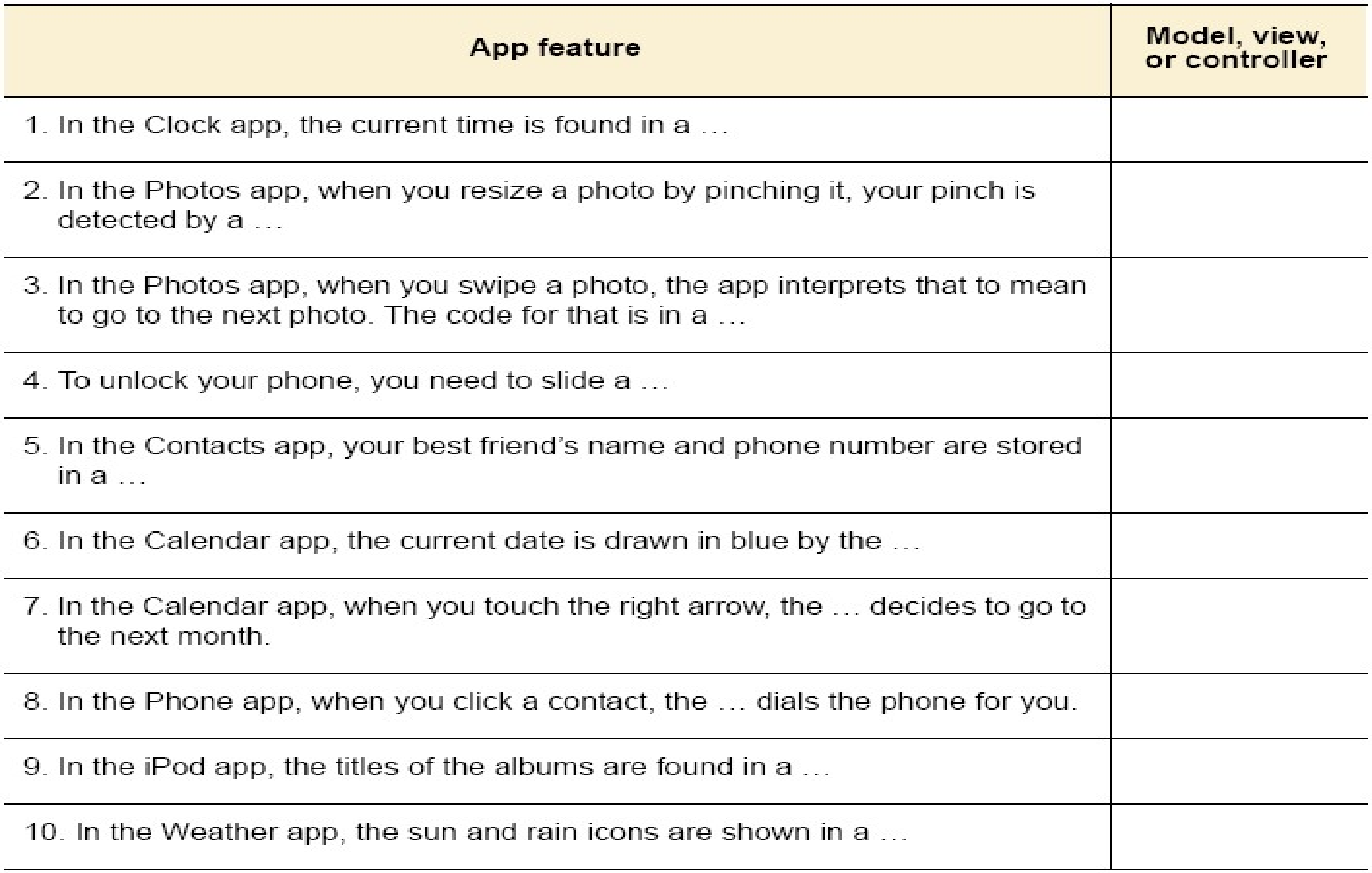
1. the view tells the Grocery Item View Controller that it's been touched.
2. The controller tells the model to remember that it's been taken,
3. and then it tells the view to show it crossed out.



Nice way to look at MVC


Result View
"If you show a seven, we are going to make a ton of money."
"The current result is obviously seven"
Calculator Model
3
4
"A seven was input. What does that mean?"
Button View (7 Button)
"Hey, dude! The user touched me"
Calculator View Controller
2
1

Test yourself…



App feature	Model, view, or controller
1. In the Clock app, the current time is found in a …	
2. In the Photos app, when you resize a photo by pinching it, your pinch is detected by a …	
3. In the Photos app, when you swipe a photo, the app interprets that to mean to go to the next photo. The code for that is in a …	
4. To unlock your phone, you need to slide a …	
5. In the Contacts app, your best friend's name and phone number are stored in a …	
6. In the Calendar app, the current date is drawn in blue by the …	
7. In the Calendar app, when you touch the right arrow, the … decides to go to the next month.	
8. In the Phone app, when you click a contact, the … dials the phone for you.	
9. In the iPod app, the titles of the albums are found in a …	
10. In the Weather app, the sun and rain icons are shown in a …	

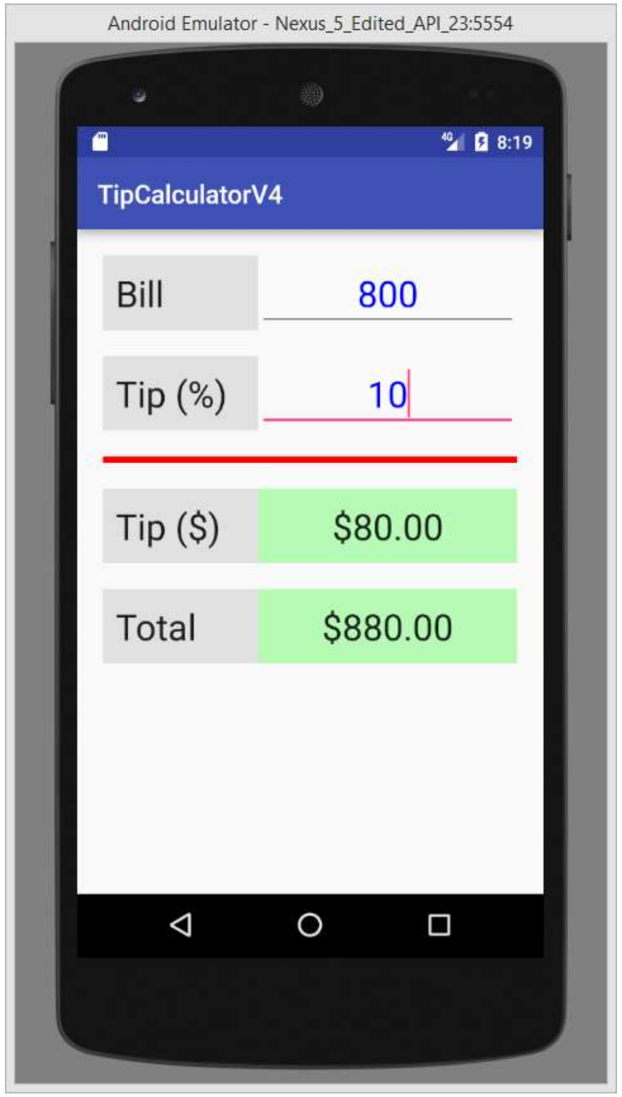
Test yourself…

App feature	Model, view, or controller
1. In the Clock app, the current time is found in a …	M
2. In the Photos app, when you resize a photo by pinching it, your pinch is detected by a …	V
3. In the Photos app, when you swipe a photo, the app interprets that to mean to go to the next photo. The code for that is in a …	C
4. To unlock your phone, you need to slide a …	V
5. In the Contacts app, your best friend's name and phone number are stored in a …	M
6. In the Calendar app, the current date is drawn in blue by the …	V
7. In the Calendar app, when you touch the right arrow, the … decides to go to the next month.	C
8. In the Phone app, when you click a contact, the … dials the phone for you.	C
9. In the iPod app, the titles of the albums are found in a …	M
10. In the Weather app, the sun and rain icons are shown in a …	V

Model View Controller (1 of 2)

- Model: the functionality of the app
- View: The graphical user interface
- Controller: The middleman between the Model and The View
- The user interacts with the View, the Controller sends new data to the Model and asks for some calculations, then displays the result in the View.

Objective


Android Emulator - Nexus_5_Edited_API_23:5554

8:19

TipCalculatorV4

Bill	800
Tip (%)	10
Tip ($)	$80.00
Total	$880.00

Model View Controller (2 of 2)

- Model: TipCalculator.java (the TipCalculator class)
- View: activity_main.xml
- Controller: MainActivity.java (the MainActivity class)

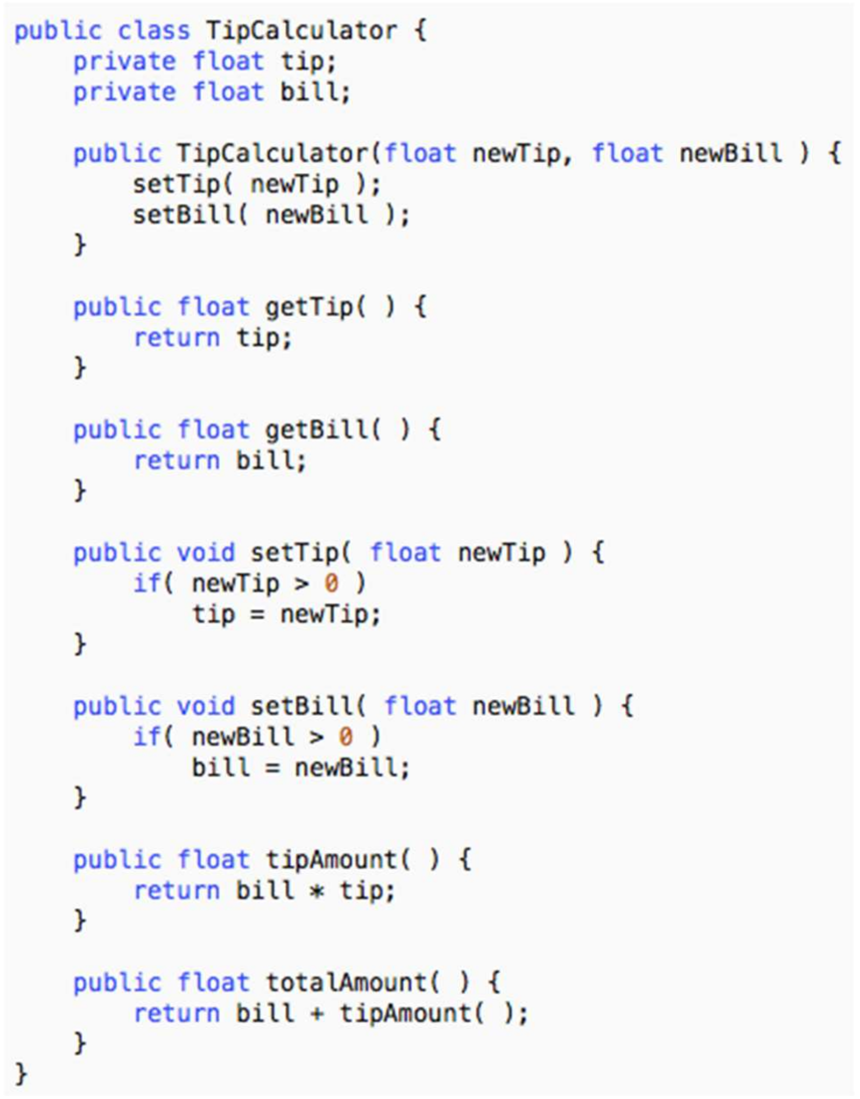
Model

- In this app, we keep the Model to a minimum.
- It includes a tip percentage, a bill, and methods to compute the tip and the total bill.

Tip Calculator

- Model-View-Controller
- GUI components
- TextView, EditText, Button
- XML attributes
- Styles and Themes
- Events and Event Handling

The Model



```java
public class TipCalculator {
    private float tip;
    private float bill;

    public TipCalculator(float newTip, float newBill ) {
        setTip( newTip );
        setBill( newBill );
    }

    public float getTip( ) {
        return tip;
    }

    public float getBill( ) {
        return bill;
    }

    public void setTip( float newTip ) {
        if( newTip > 0 )
            tip = newTip;
    }

    public void setBill( float newBill ) {
        if( newBill > 0 )
            bill = newBill;
    }

    public float tipAmount( ) {
        return bill * tip;
    }

    public float totalAmount( ) {
        return bill + tipAmount( );
    }
}
```

The Model

- Example shows the Model for this app, the TipCalculator class.
- Note that it does not include any GUI-like code. It could be reused for another app, or even a desktop Java application.

View

- The View reflects the data that are stored in the Model.
- It includes GUI components, also called widgets, to gather user input from the user (the tip and the bill), and to display the tip and the total bill.

GUI Components

- GUI components can:
  - Display data
  - Capture user input
  - Allow the user to interact with them, which can trigger some action, i.e., a call to a method

Some GUI Components (1 of 2)

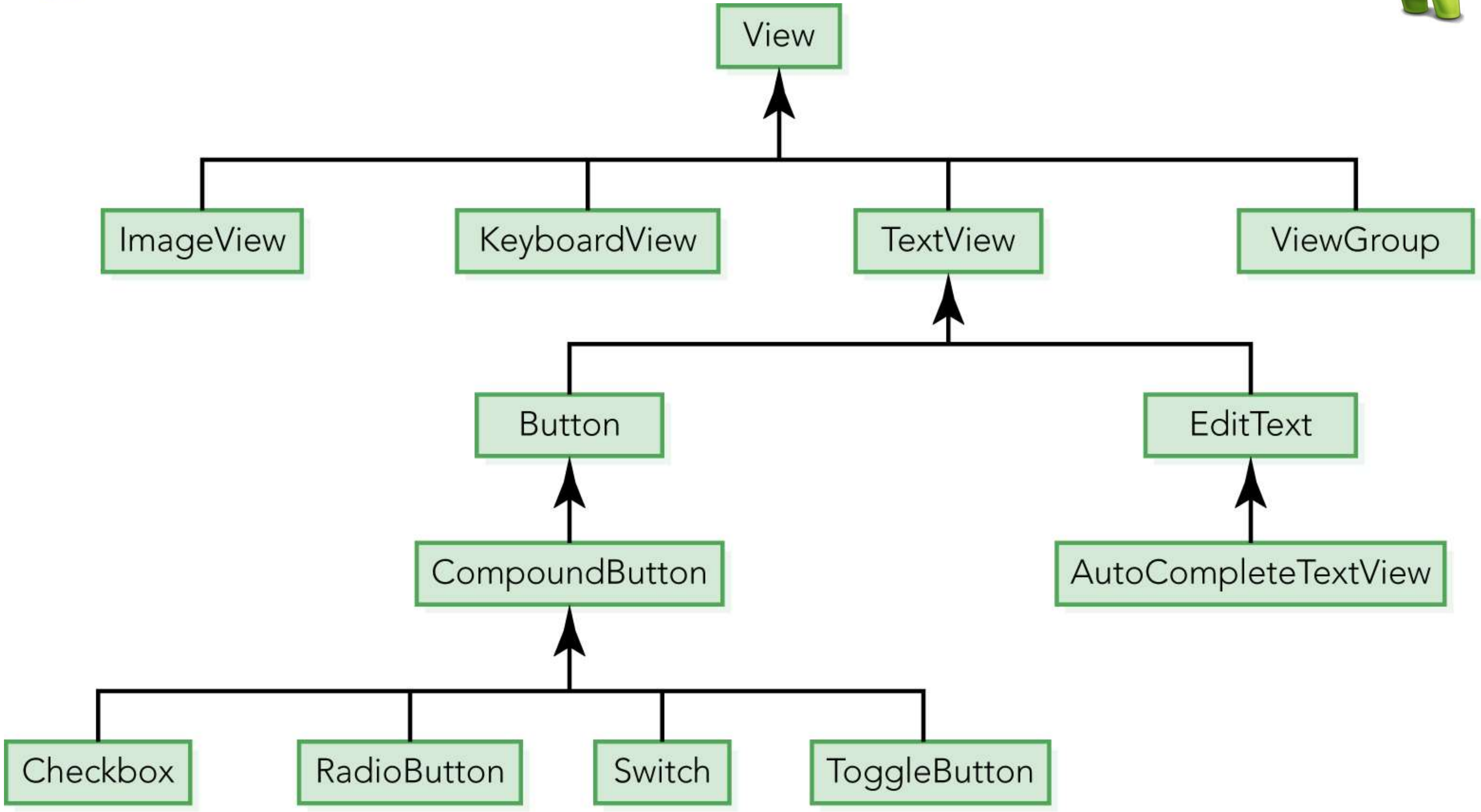
Component	**Class**
Panel	View
Keyboard	KeyboardView
Label	TextView
Text field	EditText

Some GUI Components (2 of 2)

Component	**Class**
Button	Button
Radio button	RadioButton
Checkbox	CheckBox
2-state button	ToggleButton
On-off switch	Switch

GUI Components

- View is the root GUI component.
- The other GUI components inherit from View, either directly or indirectly.
- View is in the android.view package.
- Most GUI components are in the android.widget package.


View
ImageView
KeyboardView
TextView
ViewGroup
Button
EditText
CompoundButton
AutoCompleteTextView
Checkbox
RadioButton
Switch
ToggleButton

Version 0

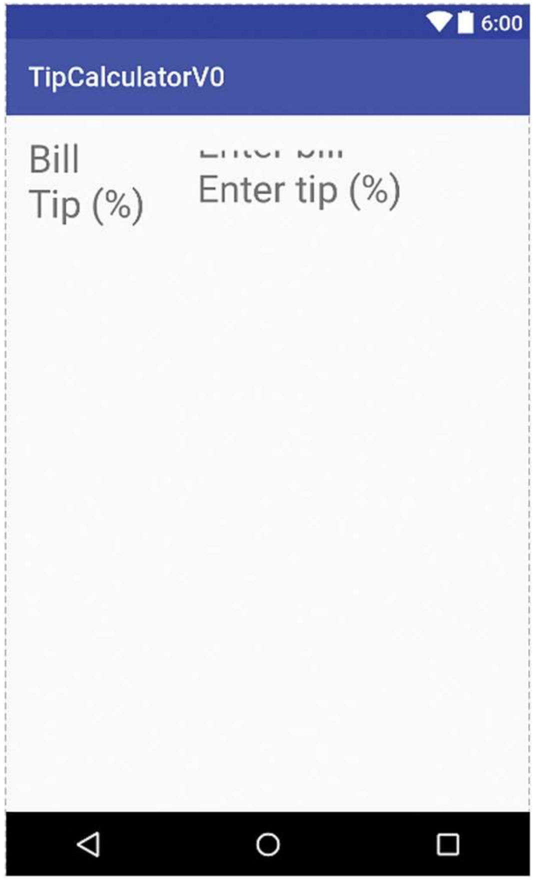
- To keep things simple, we only allow the app to run in vertical orientation.
- ➔ in the AndroidManifest.xml file, inside the activity element, we add an android:screenOrientation attribute and assign to it the value portrait.

AndroidManifest.xml

```
…
<activity
  …
  android:screenOrientation="portrait" >
  …
</activity>
```

Version 0 Preview

- Note the less than perfect positioning.



Version 0 (1 of 4)

- We only display the components:
  - Two EditTexts for the bill and the tip percentage
  - Two TextViews to display labels for the above EditTexts

Version 0 (2 of 4)

- We modify the activity_main.xml file so that it has two EditText and two TextView elements inside the RelativeLayout element.

Version 0 (3 of 4)

- For all widgets, we start by defining the android:layout_width and android:layout_height attributes.

Version 0 (4 of 4)

- We set the width and height of the RelativeLayout to match_parent (that is actually the default for the skeleton code).
- We set the width and height of the four GUI components to wrap_content: the GUI components are only as big as necessary to contain their contents.

Layout Parameters

- Generally, a layout class contains a static inner class, often called LayoutParams, that contains XML attributes that allows us to arrange the components within the layout.

RelativeLayout

- If static class B is an inner class of class A, we refer to it using A.B
- ➔ We use the attributes of RelativeLayout.LayoutParams to position the XML elements.

Identifying XML Elements (1 of 3)

- In order to position an XML element (A – representing a GUI component) relative to another XML element (B – also representing a GUI component), we need to be able to reference B.
- ➔ the android:id attribute allows us to give an id to an XML element.

Identifying XML Elements (2 of 3)

- The syntax for assigning an id to an XML element is

  android:id = "@+id/idValue"
- For example:

  android:id = "@+id/amount_bill"

Identifying XML Elements (3 of 3)

- Assigning an id to an XML element helps us position the GUI components represented by these XML elements relative to each other.
- It also allows us to retrieve the corresponding GUI components (using the findViewById method—more on this later).

RelativeLayout.LayoutParams

Some attributes include (more in Table 2.5)

android:layout_alignLeft
android:layout_alignRight
android:layout_alignBottom
android:layout_alignTop
android:layout_above
android:layout_toLeftOf
…..

Relative Positions

```
<TextView android:id="@+id/label_bill"
 …
<EditText
   android:id="@+id/amount_bill"
   android:layout_toRightOf="@+id/label_bill"
   android:layout_alignBottom="@+id/label_bill
   "

 …
```

Positions of the Four GUI Components

- TextView for the bill: the top left corner of the screen
- EditText for the bill: to the right of the TextView for the bill
- TextView for the tip percentage: below the TextView for the bill
- EditText for the bill: to the right of the TextView for the tip percentage
- ➔ See Example 2.3

Version 0 (1 of 3)

- For all elements of the two EditText, we want to constrain the user to only input valid values, i.e., floating point numbers.
- We do this by using the android:inputType attribute, assigning the value numberDecimal to it.

```
<EditText .. android:inputType="numberDecimal" .. />
```

Version 0 (2 of 3)

- We want to tell the user what to enter in the two EditTexts.
- ➔ We use the android:hint attribute
- Its value will show in light gray, looking like a hint

```
android:hint="@string/amount_bill_hint"
```

Version 0 (3 of 3)

- For the android:text values of the two TextViews and the android:hint values of the two EditTexts, we use string values defined in strings.xml

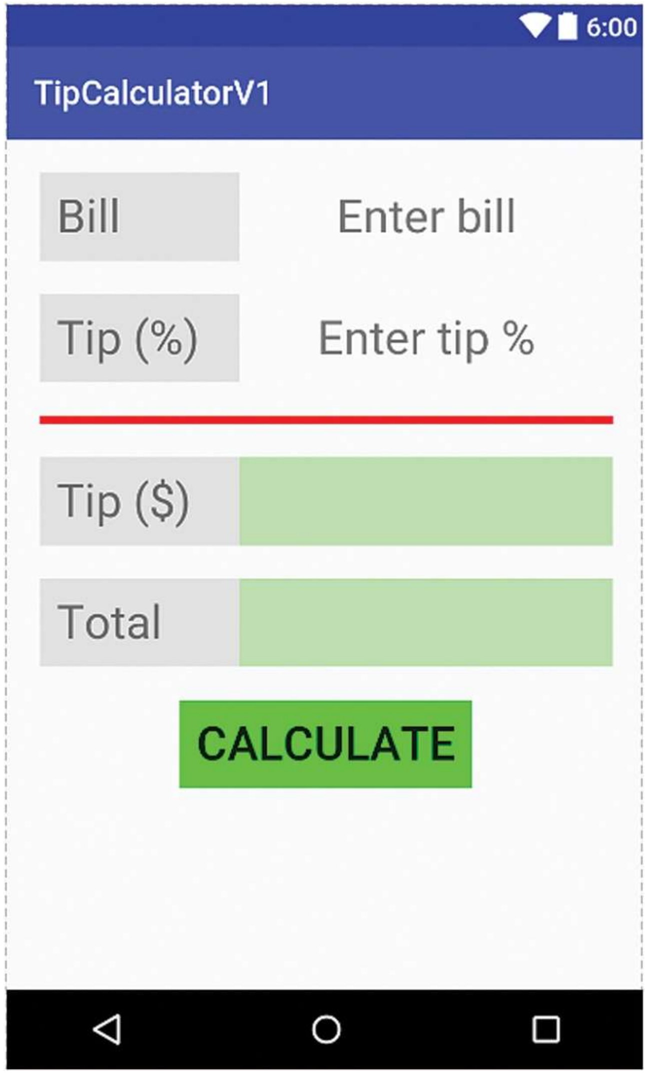
The Four Strings in strings.xm

```
<string name = "label_bill">Bill</string>

<string name = "label_tip_percent">Tip (%)</string>

<string name = "amount_bill_hint">Enter bill</string>

<string name = "amount_tip_percent_hint">Enter tip

</string>
```

Version 1

- In Version 1, we add GUI components and use colors.
- We add a red line and four more TextViews (label and display View for the tip amount and total), as well as a button (to trigger the calculations).

Version 1 Preview

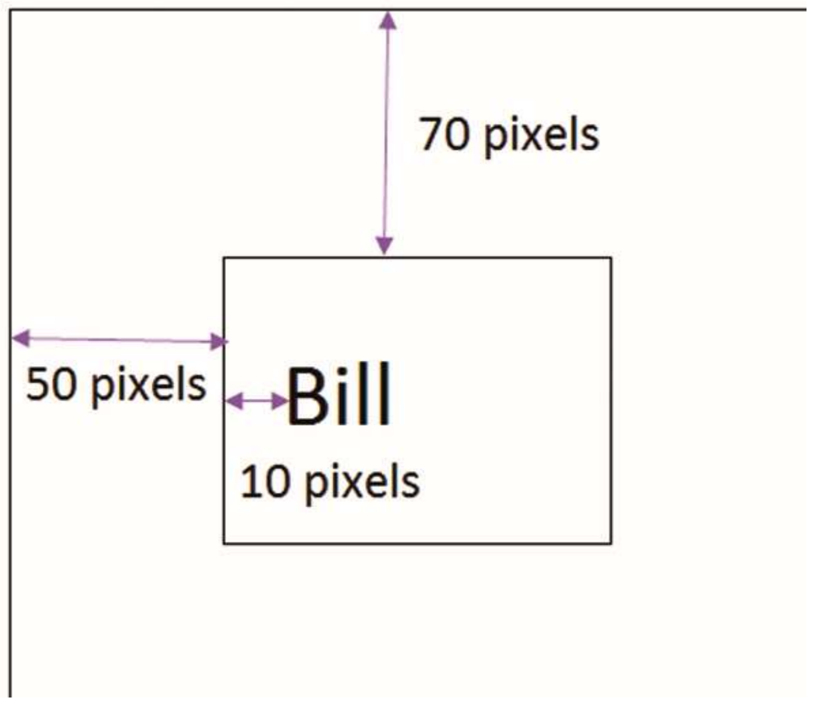


Version 1

- We can use many attributes to specify how a GUI component will look.
- For example, we can specify the background of a component, its foreground color, its text style, its size, the margins around it, or the padding inside it.

Margins and Padding

- The top margin is 70.
- The left margin is 50.
- The left padding is 10.



Version 1 (1 of 2)

We can also specify the margins around a component.

Attribute Name	Related Method	Description
android:layout_marginBottom	setMargins(int, int, int, int)	Set extra space at the bottom of this View
android:layout_marginLeft	setMargins(int, int, int, int)	Set extra space at the left of this View
android:layout_marginRight	setMargins(int, int, int, int)	Set extra space at the right of this View
android:layout_marginTop	setMargins(int, int, int, int)	Set extra space at the top of this View

Version 1 (2 of 2)

We can also specify the padding inside a component.

Attribute Name	Related Method	Description
android:paddingBottom	SetPadding(int, int, int, int)	Set the padding, in pixels, of the View’s bottom edge
android:paddingLeft	SetPadding(int, int, int, int)	Set the padding, in pixels, of the View’s left edge
android:paddingRight	SetPadding(int, int, int, int)	Set the padding, in pixels, of the View’s right edge
android:paddingTop	SetPadding(int, int, int, int)	Set the padding, in pixels, of the View’s top edge

More XML Attributes

Attribute	Class	Description
android:background	View	A drawable resource, for example an image or a color
android:textColor	TextView	The text color
android:textSize	TextView	The text size
android:textStyle	TextView	The text style (bold, ..)

android:textColor (1 of 5)

- Values for the android:textColor attribute can be either a string representing an RGB color, a resource, or a “theme” attribute.
- The string can have many formats:

“#aarrggbb” ➔ aa is the alpha value

“#rrggbb” ➔opaque

“#argb” ➔ equivalent to “#aarrggbb”

“#rgb” ➔ opaque

android:textColor (2 of 5)

Examples:

android:textColor = “#FF49AF32”

android:textColor = “#05F300”

android:textColor = “#C348”

android:textColor = “#4CA”

android:textColor (3 of 5)

- Another way to define a color is to use a resource id using this syntax ([ ] ➔optional)

  "@[+][package:]type:name"

- Assuming that we have created the file colors.xml in the res/values directory

  android:textColor="@color/lightGreen"

android:textColor (4 of 5)

android:textColor="@color/lightGreen"

The lightGreen variable needs to be defined in the colors.xml file (or another file) of the res/values directory

```
<color name="lightGreen">#40F0</color>
```

android:textColor (5 of 5)

```
<color name="lightGreen">#40F0</color>
```

- color is a type of resource, lightGreen is its name, and its value is 40F0

activity_main.xml

- activity_main.xml shows many attributes for various GUI components and a variety of ways to specify their value.

Inserting a Line to Separate GUI Components

- We can define a line as an empty View with a background color and a thickness

```
<View
 …
 android:layout_height="5dip"
android:layout_width="match_parent"
…
android:background="#FF00" />
```

Version 2

- Next, we use styles and themes to organize our project better.
- Styles and themes enable us to separate the look and feel of a View from its content.
- This is similar to the concept of CSS (Cascading Style Sheets) in web design.

Styles (1 of 2)

- We can define styles in a file named styles.xml in the res/values directory.
- The syntax for defining a style is

```
<style name="nameOfStyle"
     [parent="styleThisStyleInheritsFrom"]>

<item name =
 "attributeName">attributeValue</item>
...
</style>
```

Styles (2 of 2)

```
<style name="nameOfStyle"
   [parent="styleThisStyleInheritsFrom"]>
    <item name="attributeName"> attributeValue
    </item>
    ...
</style>
```

- The (optional) parent attribute allows us to create a hierarchy of styles, i.e., styles can inherit from other styles.

A Generic Style for Views Containing Text

```
<style name="TextStyle"
    parent="@android:style/TextAppearance">
    <item name =
    "android:layout_width">wrap_content</item>
    <item name =
    "android:layout_height">wrap_content</item>
    <item name = "android:textSize">32sp</item>
    <item name = "android:padding">10dp</item>
</style>
```

- This style specifies width, height, text size, and padding.

A Style Inheriting from TextStyle

- We can now define style for Buttons that inherit from the TextStyle style, using the parent attribute, and specify more attribute/ value pairs

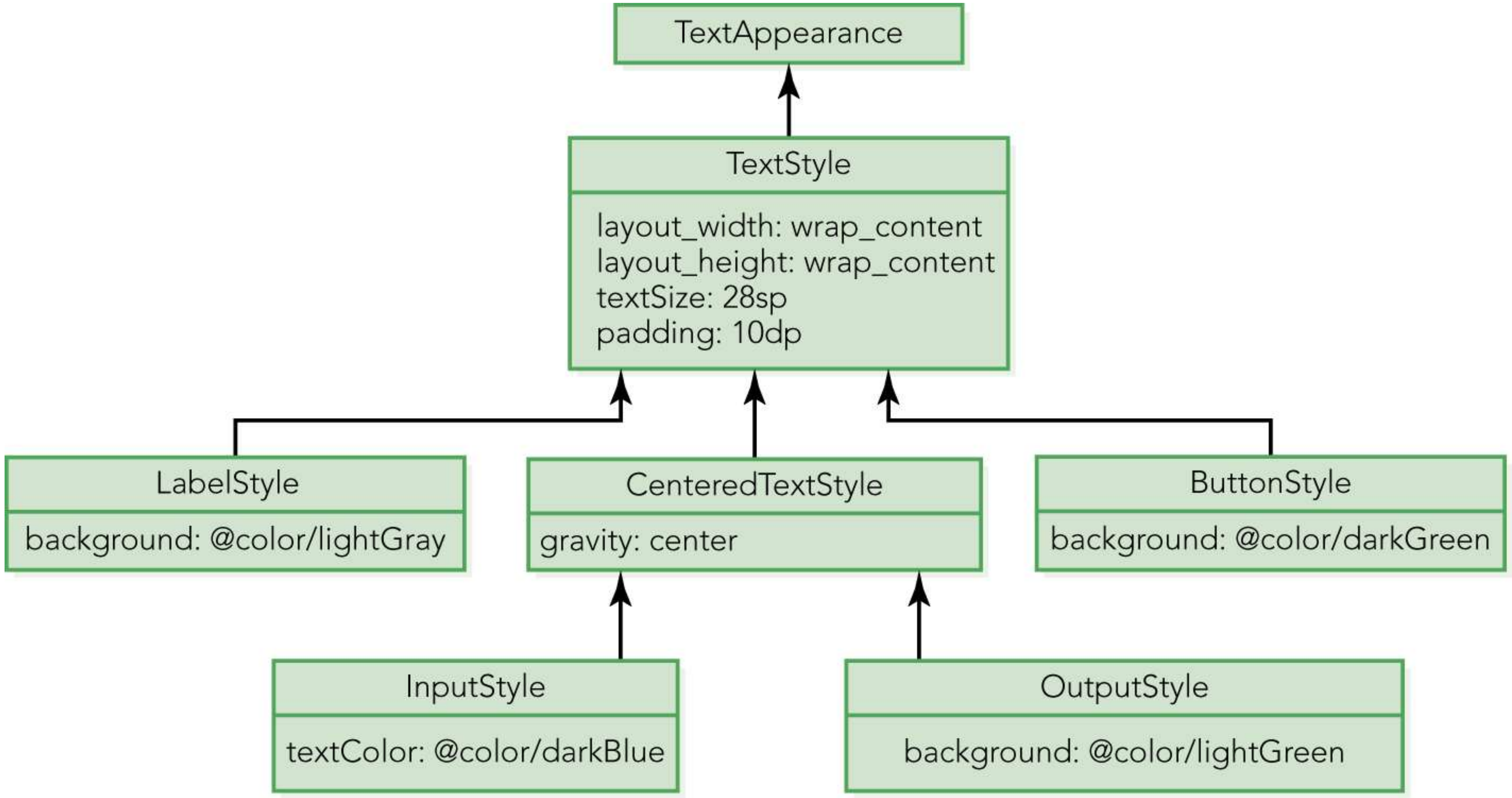
```
<style name="ButtonStyle"
       parent="TextStyle">
…
</style>
```

ButtonStyle

```
<style name="ButtonStyle" parent="TextStyle">
  <item name="android:background">
   @color/darkGreen
  </item>
</style>
```

Styles.xml

- Example shows the complete styles.xml file, defining TextStyle, LabelStyle, CenteredTextStyle, InputStyle, OutputStyle, and ButtonStyle.
- Figure shows the inheritance hierarchy.


TextAppearance
TextStyle
layout_width: wrap_content
layout_height: wrap_content
textSize: 28sp
padding: 10dp
LabelStyle
background: @color/lightGray
CenteredTextStyle
gravity: center
ButtonStyle
background: @color/darkGreen
InputStyle
textColor: @color/darkBlue
OutputStyle
background: @color/lightGreen

Using a Style (1 of 2)

- Now that we have defined styles in styles.xml, we can style a GUI component using the style attribute using this syntax:

```
style = "@style/nameOfStyle"
```

Using a Style (2 of 2)

- In order to style an EditText using the InputStyle style, we write (in the activity_main.xml file)

```
<EditText
 …
    style="@style/InputStyle"
…
```

Benefits of Using Styles (1 of 2)

- If we decide to change the background color of the "Output" TextViews in the app, we only need to do it in one place, the styles.xml file.

Benefits of Using Styles (2 of 2)

- As Example shows, the activity_main.xml is now much simpler and much easier to understand
- Inside activity_main.xml, we can now focus on how the GUI components are organized

Themes (1 of 4)

- A style relates to a GUI component or a View.
- A theme relates to an activity; it can even relate to the whole app.
- We can “theme” an app with a style by editing the AndroidManifest.xml file, changing the android:theme attribute of its application element using this syntax:

```
android:theme="@style/nameOfStyle"
```

Themes (2 of 4)

```
<application     android:allowBackup="true"
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/LabelStyle" >
…
```

Themes (3 of 4)

- To "theme" a whole app, we can also edit the AppTheme style inside the styles.xml file.
- We can “theme” an activity with a style by editing the AndroidManifest.xml file, adding an android:theme attribute to an activity element using the same syntax as before:

```
android:theme="@style/nameOfStyle"
```

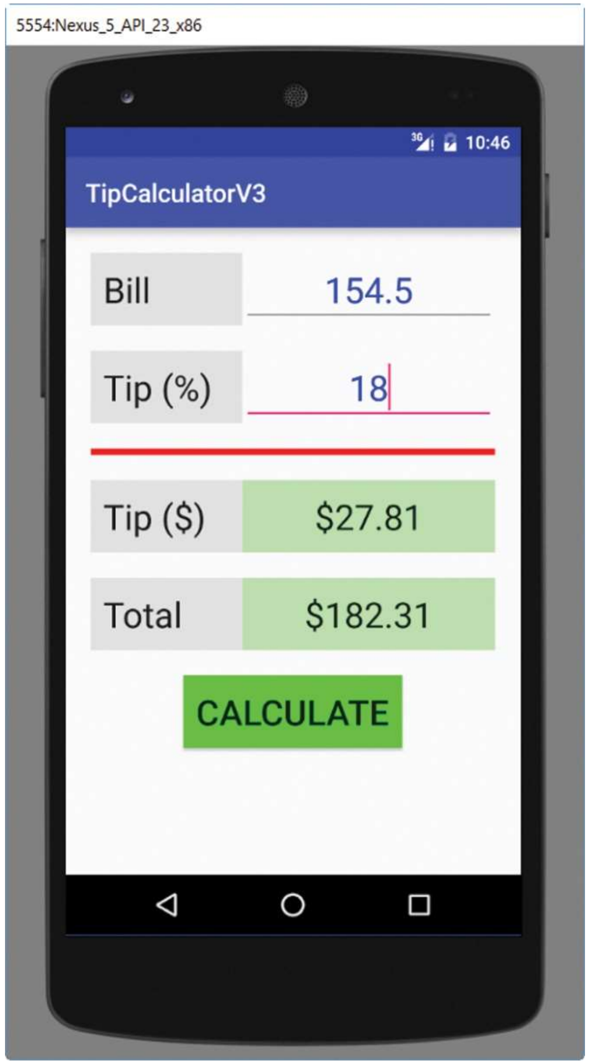
Themes (4 of 4)

```
<activity
  android:name=".MainActivity"
    android:label="@string/app_name"
    android:theme="@style/LabelStyle"
  …
```

Events and Event Handling

- In Version 3, we handle events.
- When the user clicks on the button, we update the value of the tip and the total bill to reflect the values of the tip percentage and bill that the user entered or edited via the keyboard.

Version 3


5554:Nexus_5_API_23_x86

Bill	154.5
Tip (%)	18
Tip ($)	$27.81
Total	$182.31

Using the Model

- In order to update the tip percentage and the bill and calculate the tip and total bill, we use the Model, the TipCalculator class.
- Thus, inside the Activity class, we declare an instance variable of type TipCalculator.

```
private TipCalculator tipCalc;
```

Setting Up Event Handling (1 of 4)

- We can set up event handling in the layout file (xml).
- We can also set up event handling by code (Java).
- In Version 3, we do it in the layout xml file.
- In Version 4, we do it by code.

Setting Up Event Handling (2 of 4)

- To set up a click event on a View, we use the android:onClick attribute of the View and assign to it a method.

  ```
  android:onClick="methodName"
  ```

- If the method is named calculate (see Example 2.10).

  ```
  android:onClick="calculate"
  ```

Setting Up Event Handling (3 of 4)

- The event will be handled inside that method, which must have the following API

```
public void methodName( View v )
```

- v is the View where the event happened.
- If the method is called calculate:

```
public void calculate( View v )
```

Setting Up Event Handling (4 of 4)

- When the user clicks on the button, we execute inside calculate, and its View parameter is the Button; if we have the following statement inside calculate:

```
Log.w( "MainActivity", "v = " + v );
```

- Inside LogCat, we have something like:

```
v = android.widget.Button@425a2e60
```

- The value above identifies the Button.

Event Handling (1 of 2)

- Inside the calculate method, we need to retrieve the bill and the tip percentage in order to calculate the tip and the total bill.
- The android framework enables us to assign an id to a View so that we can retrieve that View using its id; we assign an id to a View using the android:id attribute.

Event Handling (2 of 2)

- We already assigned ids to the various GUI components in the activity_main.xml file.
- Indeed, we used their ids to position the GUI components relative to each other.

The EditText ids

```
<EditText
   android:id="@+id/amount_bill"
   …
 <EditText
   android:id="@+id/amount_tip_percent"
  …
```

The TextView ids

```
<TextView        android:id="@+id/amount_tip"
  …
<TextView
   android:id="@+id/amount_total"
  …
```

Retrieving a View Using Its id (1 of 4)

- We can get a reference to a View that has been given an id using the findViewById method of the Activity class, which has this API:

  View findViewById(int id)

- findViewById returns the View with the specified id within the View controlled by the Activity.

Retrieving a View Using Its id (2 of 4)

- We call the findViewById method inside an Activity class using this syntax:

  findViewById(R.id.idValue)

- findViewById returns the View that is part of the layout xml file that was inflated in the onCreate method of the Activity class.

Retrieving a View Using Its id (3 of 4)

- The findViewById method returns a View; so if we expect to retrieve a TextView and we want to assign the View retrieved to a TextView, we need to cast the View returned by findViewById to a TextView:

```
TextView tipTextView =  ( TextView )
findViewById(   R.id.amount_tip );
```

Retrieving a View Using Its id (4 of 4)

```
EditText billEditText = ( EditText )
   findViewById( R.id. amount_bill );
EditText tipEditText = ( EditText )
   findViewById( R.id.amount_tip_percent );
TextView tipTextView = ( TextView )
   findViewById(   R.id. amount_tip );
TextView totalTextView = ( TextView )
   findViewById( R.id. amount_total );
```

Capturing User Input

- The method getText of the EditText class returns an Editable (an interface)
- We can call the toString method of Editable to get a String

```
String tipString =
 tipEditText.getText( ).toString( );
```

Using the Model (1 of 2)

- Once we have the user inputs as Strings, we convert them to floats using the parseFloat method of the Float wrapper class.
- Then we use the various methods of the Model (the TipCalculator class) to set the tip percentage and bill value of the tipCalc instance variable.
- Then we calculate the tip and total value.

Using the Model (2 of 2)

```
float billAmount = Float.parseFloat( billString );
int tipPercent = Integer.parseInt( tipString );
tipCalc.setBill( billAmount );
tipCalc.setTip( .01f * tipPercent );
tipTextView.setText( "" + tipCalc.tipAmount( ) );
totalTextView.setText( "" + tipCalc.totalAmount( ) );
```

More Event Handling

- We actually do not need a button for this app.
- Pressing a key is an event ..
- .. so every time the user presses a key, i.e., changes either the tip percentage or the bill, we can handle that event and update the tip and total bill accordingly.

More Event Handling—Versior 4

- In Version 4, we set up the event by code, in the Activity class.
- Thus, we delete the Button element in the xml file.

Listening to Key Events (1 of 4)

- The TextWatcher interface, from the android.text package, provides three methods to handle key events: beforeTextChanged, onTextChanged, and afterTextChanged.
- These methods are automatically called when the text inside a TextView changes (provided that event handling is set up).

Listening to Key Events (2 of 4)

- We actually want to handle key events inside the two EditTexts.
- The EditText class is a subclass of TextView, therefore, an EditText “is a” TextView.
- Thus, we can use TextWatcher with EditTexts.

Listening to Key Events (3 of 4)

Generally, to capture and handle an event, we need to do the following:

1. Code an event handling class (typically implementing a listener interface).
2. Declare and instantiate an object of that class.
3. Register that object on one or more GUI components.

Listening to Key Events (4 of 4)

Thus, in order to use TextWatcher so that we get notified that a key event is happening inside one of the two EditTexts, we:

1. Code a class that implements the TextWatcher interface.
2. Declare and instantiate an object of that class.
3. Register that object on the two EditTexts.

Implementing TextWatcher (1 of 6)

- Because we need to access the two EditTexts in several methods (onCreate to set up event handling and when we handle the event), we declare them as instance variables.
- We also declare the two TextViews as instance variables.
- Inside onCreate, we use findViewById to assign a value to all four components.

Implementing TextWatcher (2 of 6)

- We first code a private inner class implementing TextWatcher.
- That class needs to implement the three methods of TextWatcher: beforeTextChanged, onTextChanged, and afterTextChanged.

Implementing TextWatcher (3 of 6)

- We update our View after the key event happened, so we are only interested in afterTextChanged .. but we must implement all three (when inheriting from an interface, it is mandatory to implement all its methods).
- So we implement beforeTextChanged and onTextChanged as “do-nothing” method.

Implementing TextWatcher (4 of 6)

```
private class TextChangeHandler implements
      TextWatcher {
   public void beforeTextChanged( CharSequence s,
            int start, int count, int after ) {
      }
      …
}
```

Implementing TextWatcher (5 of 6)

```
public void onTextChanged( CharSequence
      s, int start, int before, int after ) {
}
public void afterTextChanged( Editable e ) {
      calculate( );
}
```

Implementing TextWatcher (6 of 6)

- Note that calculate no longer takes a View parameter; it is not needed because we do not care about the origin of the event (i.e., which EditText is being edited).
- The code inside calculate is the same as before.

Setting Up Event Handling (1 of 2)

- Now that our event handling class is coded, we need to declare and instantiate an object of that class inside the onCreate method.

```
    TextChangeHandler tch =
     new TextChangeHandler( );
```

Setting Up Event Handling (2 of 2)

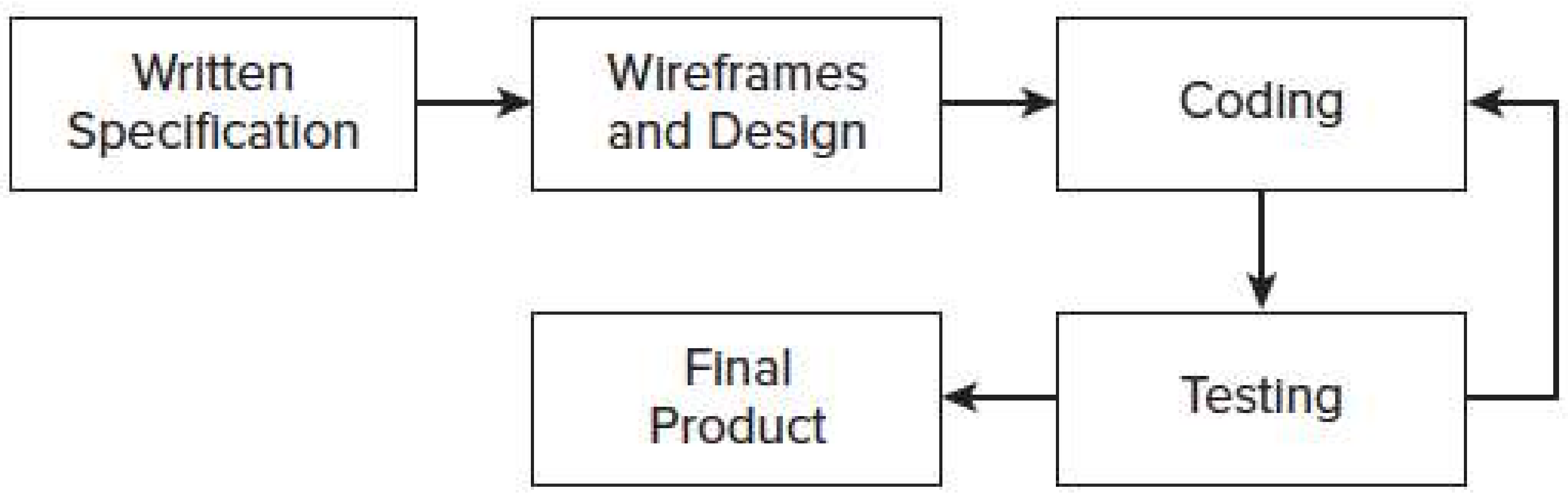
- We now register the listener on the two EditTexts (inside onCreate).

```
billEditText.addTextChangedListener( tch );
tipEditText.addTextChangedListener( tch );
```

Running Version 4

- When running Version 4 of this app, every time we edit either the tip percentage or the bill, the tip and the total value are automatically updated.

Development process



Writing a specification

- The development of an app begins with a concept.
- Put this concept on paper and create a specification.
- At the end of the project
  - come back to the specification document to see how the final product that was created compares with the original specification

Writing a specification

- A short description (200 words or less)
- The target audience/demographic of the users
- How will it be distributed (App Store, or direct to a small number of devices)
- A list of similar competing apps
- The pricing model of competing apps and potential pricing for your app

Wireframes and Design

- Large drawing that contains mockups of each screen of your app as well as lines connecting different screens that indicate the user’s journey through your application.
- Identify flaws in your design early on (before any coding has been done).
- Used to show potential clients how a particular app is likely to look when it’s completed.
- The way to make a wireframe:
  - few sheets of paper and a pen
  - illustration package

Coding

- Using the IDE to type your code.
- Android apps are written in Java
  - consists of several files of Jave code along with resource files (such as images, audio, and video).
  - Combined together by a compilation into a single file that is installed onto the target device.
  - Single file: the application binary or a build

Testing

- Test your app after it has been developed.
- Test your code frequently as you write it.
- Comprehensive test of the entire application as often as possible to ensure things that were working in the past continue to do so.
- Regression testing: a test plan document
  - List all the features that you want to test, and the steps required to carry out each test.
  - list which tests failed. The ones that fail will then need to be fixed.