

## INHALTSVERZEICHNIS

## Kapitel 4

# Arbeitsweise von Android-Views, Events, CallBacks, ToolBar

Wir haben bislang vorwiegend Funktionalitäten von Android-Apps untersucht, die in **XML-Ressourcen** definiert wurden. An den in Java implementierten *Activities* haben wir dagegen nur wenige Änderungen vorgenommen. In diesem Kapitel werden nun die Zusammenhänge

- zwischen XML-Ressourcen und Activity-Klassen,
- zwischen Activity-Klassen und View-Klassen,
- zwischen View-Klassen und ihren Listenern, sowie
- zwischen Views und der Ereignisbehandlung mit Hilfe des `android:onClick`-Attributs

näher betrachtet.

### 4.1 Zusammenspiel Activities und Views

Die bislang in XML-Layouts konfigurierten Views besitzen auch korrespondierende Klassen in Java. Activities können auch direkt in Java auf

Methoden und Attribute der von ihnen angezeigten Views zugreifen.  
Das Zusammenspiel wird zunächst grob im folgenden Klassendiagramm  
gezeigt:

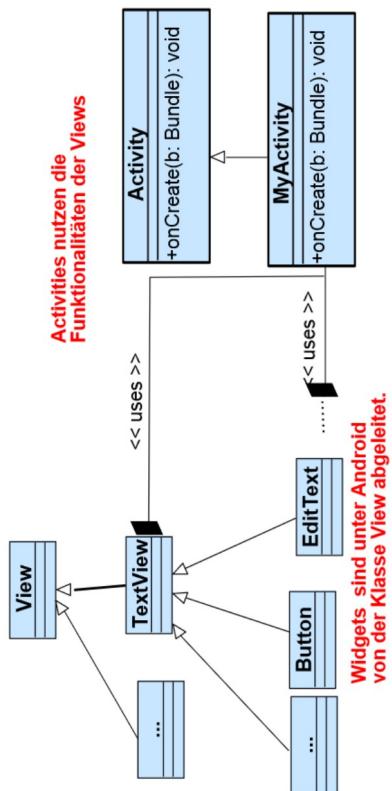


Abbildung 4.1: Klassenhierarchie Views und Activities

Das Klassendiagramm ist folgendermaßen zu interpretieren:

- Activities sind hierbei immer direkt oder indirekt von der Klasse *Activity* abgeleitet.
- Views sind immer direkt oder indirekt von der Klasse *View* abgeleitet.
- Activities verwenden in ihren Methoden (z. B. *onCreate()*) die in ihren XML-Layouts konfigurierten View-Objekte.
- Eine Activity kann diese Objekte über ihre in XML vergebene ID identifizieren – dies geschieht über die schon erläuterte Klasse *R*. Den Mechanismus dazu werden wir in den folgenden Kapiteln genauer erklären.

## 4.2 Event-Modell in Android

Android-Applikationen können auf zwei Arten ihr Dialogverhalten implementieren:

- Menüpunkte des ToolBar auswerten. Dies geschieht in einer separaten Callback-Methode der Activity: `onOptionsItemSelected(...)`
- **android:onClick-Attribut** nutzen: Hier wird die Reaktionsmethode direkt in der Activity-Klasse implementiert. Ein Querverweis auf die Methode erfolgt im XML-Layout der Activity über das `android:onClick-` Attribut der entsprechenden Controls.
- **Listener**: Dieser Mechanismus wurde schon ausführlich im SWT-Widget-Set erläutert. Wir werden hier dennoch auf die Android-spezifischen Listener-Mechanismen eingehen.

### 4.2.1 ToolBar und Menüs

Ab Android 4 hat sich das „Look-and-Feel“ und auch die Verankerung von ToolBars und Menüs in Android-Apps noch einmal stark gewandelt. In diesem Kapitel werden wir betrachten:

- Das Sichtbarmachen / Verstecken des ToolBar über das Theme der App,
- Die Layoutgestaltung eines ToolBar,
- Den Aufbau des ToolBar im Java-Code der Activity,
- Die Auswertung von Menüoptionen des ToolBar im Java-Code der Activity, und
- Komplettbeispiel eines ToolBar mit Menü über die Android-Support-Library

#### 4.2.1.1 Das Sichtbarmachen / Verstecken des ToolBar über das Theme der App

Falls der ToolBar standardmäßig **nicht** angezeigt wird, so liegt das häufig an den Einstellungen im Theme der App: Hier müssen ggf. folgende Einträge gesetzt werden:

- `android:windowActionBar` auf `true` → sorgt dafür, dass der ToolBar angezeigt wird.
- `android:windowNoTitle` auf `false` → sorgt dafür, dass der Titeltext/das Titel-Icon der App nicht versteckt wird.

Die Abbildung auf der folgenden Seite zeigt das Vorgehen hierzu.

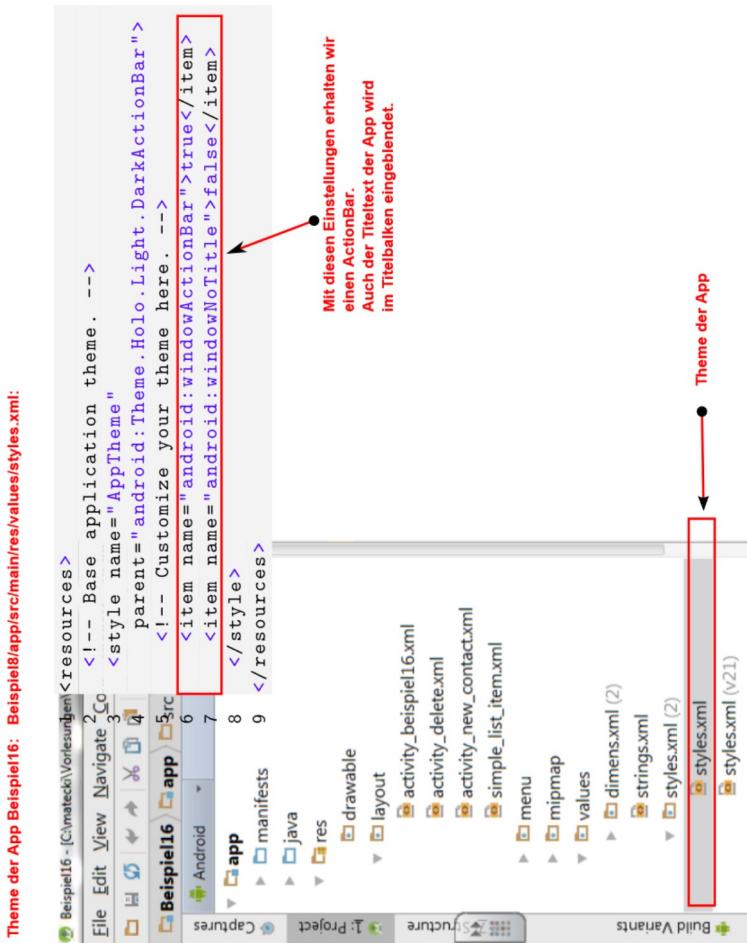


Abbildung 4.2: ToolBar-Einstellung im Theme *Beispiel16*

#### 4.2.1.2 Layoutgestaltung beim ToolBar

Die Ausgestaltung der Items erfolgt in einer weiteren Layout-Datei, die im Ordner IhrProjekt/app/src/main/res/menu liegt. In Variante 1 sieht das Menü-Layout folgendermaßen aus:

```
1 <menu xmlns:android="http://schemas.android.
  com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/
    tools"
3   tools:context=".Beispiel16Activity">
4     <item android:id="@+id/new_item"
5       android:title="@string/new_item"
6       android:icon="@mipmap/ic_new"
7       android:orderInCategory="100"
8       android:showAsAction="always"/>
9     <item android:id="@+id/delete_item"
10       android:icon="@mipmap/ic_delete"
11       android:title="@string/delete_item"
12       android:orderInCategory="100"
13       android:showAsAction="always"/>
14 </menu>
```

Listing 4.1: Variante 1: Layout für sichtbare Buttons im ToolBar

Für jeden Menüeintrag/Button haben wir hier einen *item*-Eintrag. Für die direkte Darstellung auf dem ToolBar sind hier zwei Einträge in den *item*-Definitionen verantwortlich:

- `android:showAsAction="always"` sagt aus, dass das Item *immer* auf dem ToolBar sichtbar sein soll. Dies ist nur möglich, wenn das Item nicht in einem Klapp-Menü versteckt wird.
- `android:icon="@mipmap/ic_new"` sagt aus, dass das Item mit einem Icon aus dem *mipmap*-Ordner dargestellt werden soll. Würde diese Angabe fehlen, so wäre der Titeltext aus dem Attribut `android:title="@string/new_item"` „ständig“ sichtbar.

In Variante 2 sieht die Darstellung folgendermaßen aus:

```
1 <menu xmlns:android="http://schemas.android.
  com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/
    tools"
3   tools:context=".Beispiel16Activity">
4     <item android:id="@+id/new_item"
5       android:title="@string/new_item"
6       android:icon="@mipmap/ic_new"
7       android:orderInCategory="100"
8       android:showAsAction="collapseActionView
      "/>
9     <item android:id="@+id/delete_item"
10       android:icon="@mipmap/ic_delete"
11       android:title="@string/delete_item"
12       android:orderInCategory="100"
13       android:showAsAction="collapseActionView
      "/>
14 </menu>
```

Listing 4.2: Variante 2: Layout für Klappmenü im ToolBar

Hier wurde das Attribut **android:showAsAction** so gesetzt, dass das Item im Klappmenü der rechten oberen Ecke verschwindet.

Insgesamt können für dieses Attribut folgende Werte angegeben werden:

<b>Attribut android:showAsAction</b>	
<b>Attribut-Wert</b>	<b>Erklärung</b>
<code>always</code>	erzwingt die direkt sichtbare Darstellung des Items auf dem ToolBar (vgl. ToolBar in anderen Widget-Sets). Ist in neueren Android- Versionen nicht mehr üblich.
<code>ifRoom</code>	stellt die Items, falls möglich, direkt auf dem ToolBar dar (wie bei <code>always</code> ).
<code>never</code>	Item wird überhaupt nicht dargestellt – weder direkt auf dem ToolBar noch indirekt im Klappmenü. Dies ist die Standardeinstellung für die vorgefertigten Menüoptionen – daher sehen wir, sofern wir die Items nicht bearbeiten auch keine Menüoptionen in der App.
<code>collapseActionView</code>	versteckt das Item im Klappmenü in der rechten oberen Ecke.

Tabelle 4.1: Attribut android:showAsAction

#### 4.2.1.3 Aufbau des ToolBar im Java-Code der Activity

Im Regelfall leiten wir unsere Activity von *AppCompatActivity* ab, sofern wir einen ToolBar gestalten wollen. Diese Superklasse wird an späterer Stelle noch genauer erläutert. Genau so, wie es eine Methode *onCreate()* gibt, die wir im Code unserer Activity erben und überschreiben, gibt es auch eine solche Methode für den Aufbau unserer Menüoptionen im ToolBar. Sie heißt *onCreateOptionsMenu()* und wird bei der Erzeugung unserer Activity von der Entwicklungsumgebung mit generiert. Sie sieht immer etwa folgendermaßen aus:

```
1 public class MyActivity
2     extends AppCompatActivity {
3     // ... onCreate() wird hier
4     // noch nicht dargestellt ...
5     @Override
6     public boolean onCreateOptionsMenu(
7             Menu menu) {
8         // Inflate the menu; this adds
9         // items to the action bar if
10        // it is present.
11        getMenuInflater().inflate(
12            R.menu.menu_my, menu);
13        return true;
14    }
15
16    // ... weitere Methoden
17 }
```

Listing 4.3: Aufbau des ToolBar von der Activity aus

Die generierte Klasse *R* hat für das Menü-Layout ebenfalls eine ID, die hier der ererbten Methode *getMenuInflater().inflate(...)* übergeben wird. Dieser Aufruf sorgt für die ordnungsgemäße Anzeige des ToolBar mit all seinen Items.

#### 4.2.1.4 Auswertung der Menüoptionen im Java-Code der Activity

Die Auswertung der Items geschieht in der ererbten Methode `onOptionsItemSelected(...)`, die wir mit dem Auswertungscode implementieren:

```
1 public class MyActivity
2     extends AppCompatActivity {
3     // ... onCreate() und
4     // onCreateOptionsMenu() wird
5     // nicht dargestellt ...
6
7     @Override
8     public boolean onOptionsItemSelected(
9             MenuItem item) {
10        // Gedreckte Option im ToolBar
11        int id = item.getItemId();
12
13        // Auswertung, falls item1 selektiert
14        if (id == R.id.item1) {
15            // Code fuer item1 ..
16        }
17        // Auswertung, falls item2 selektiert
18        else if(id == R.id.item2){
19            // Code fuer item2 ..
20        }
21        // .. ggf weitere Faelle ..
22
23        // Gib return-Code der
24        // Eltern-Methode zurueck
25        return super.onOptionsItemSelected(item);
26    } // end method
27    // ... weitere Methoden
28 } // end class
```

Listing 4.4: Auswertung der Menüoptionen des ToolBar

Jedes der Items hat im Menü-Layout eine eigene ID bekommen. Über

diese IDs geschieht hier in einer Fallunterscheidung die Auswertung, welches Item gedrückt wurde. In jedem der verschiedenen Fälle steht nun der Code, der sonst in einen eigenständigen Listener ausgelagert worden wäre.

### Achtung

Wir schreiben hier keinen separaten Listener. Die Auswertung des Event-Verhaltens der ToolBar-Items geschieht ausschließlich in dieser Methode!

#### 4.2.1.5 Komplettbeispiel mit dem ToolBar der Support-Library

Die Support-Library wird verwendet, um Android-Apps versions-unabhängiger zu machen. Diese Bibliothek stellt „Wrapper“ zur Verfügung. Diese liefern – sofern es für eine niedrigere Android-Version beispielsweise noch keinen ToolBar gibt, diesen in einer Eigenimplementierung. So reicht dann eine einzige App-Implementierung für viele Android-Versionen aus. Wir zeigen die Anwendung dieser Library mit der Nutzung eines **ToolBars**, der auch in den niedrigeren Android-Versionen funktioniert. Unsere Beispiel-App wird mit ihren Menüoptionen einfach die Schriftfarbe unserer zentralen *TextView* ändern:

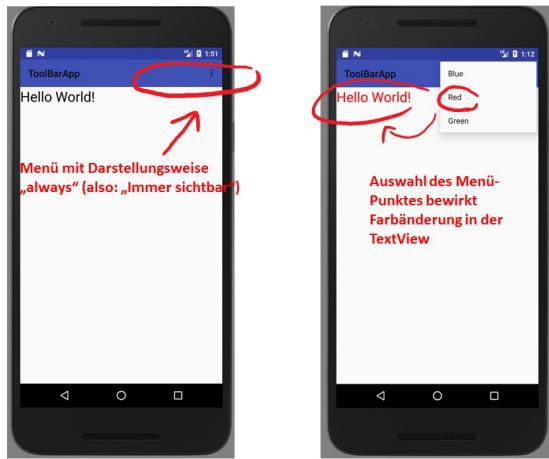


Abbildung 4.3: ToolBar-App mit Klappmenü – Items sind in *collapseActionView*-Variante dargestellt

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

---

In der etwas älteren Darstellungweise haben wir die Items des ToolBar offen dargestellt. Jedes Item ist hier mit einem Icon hinterlegt, welches aber nur in der offenen Sichtweise dargestellt wird:

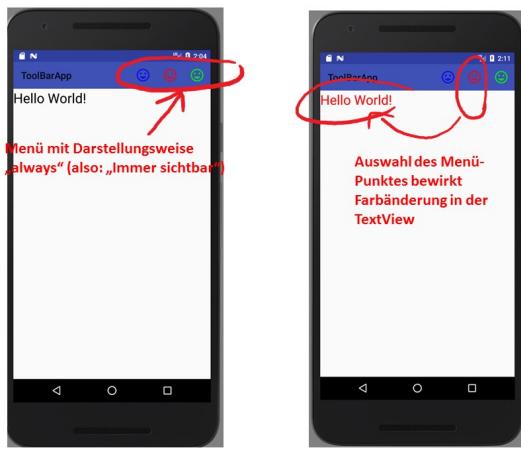


Abbildung 4.4:ToolBar-App mit Klappmenü – Items sind in *always*-Variante dargestellt

Wir legen die Activity dieser App als **Basic Activity** an. Auf diese Weise werden schon viele Bausteine, die wir für die spätere Menü-Darstellung benötigen, vorgefertigt. Die Menü-Definition der App sieht folgendermaßen aus:

menu/menu\_main\_activity\_toolbar.xml:

```
1 <menu
2   xmlns:android="http://schemas.android.com/apk/res/
3     android"
4   xmlns:app="http://schemas.android.com/apk/res-auto"
5   xmlns:tools="http://schemas.android.com/tools"
6   tools:context="gui.inf.hisas.de.toolbarapp.
7     MainActivityToolbar">
8   <item
9     android:id="@+id/action_blue"
10    android:orderInCategory="100"
11    android:title="@string/action_blue"
12    android:icon="@drawable/ic_action_blue"
13    app:showAsAction="collapseActionView"/>
14   <item
15     android:id="@+id/action_red"
16     android:orderInCategory="100"
17     android:title="@string/action_red"
18     android:icon="@drawable/ic_action_red"
19     app:showAsAction="collapseActionView"/>
20   <item
21     android:id="@+id/action_green"
22     android:orderInCategory="100"
23     android:title="@string/action_green"
24     android:icon="@drawable/ic_action_green"
25     app:showAsAction="collapseActionView"/>
26 </menu>
```

Wir haben hier die Sichtbarkeit der einzelnen Menü-Items auf *collapseActionView* gesetzt.

Das Layout hat sich gegenüber früher ebenfalls etwas geändert:

`layout/activity_main_toolbar.xml:`

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/apk/res/
4         android"
5     xmlns:app="http://schemas.android.com/apk/res-auto"
6     xmlns:tools="http://schemas.android.com/tools"
7     android:layout_width="match_parent"
8     android:layout_height="match_parent"
9     android:orientation="vertical"
9     tools:context="gui.inf.hsas.de.toolbarapp.
    MainActivityToolbar">
```

Die Support-Library bietet auch diverse, komplexere Layouts an. Uns genügt hier ein lineares, vertikales Layout. Wir verwenden hier die Android-Tools und geben den Package- Pfad unserer Activity an.

```
10    <android.support.v7.widget.Toolbar
11        android:id="@+id/toolbar"
12        android:layout_width="match_parent"
13        android:layout_height="?attr/actionBarSize"
14        android:background="?attr/colorPrimary"
15        app:popupTheme="@style/AppTheme.PopupOverlay"/>
```

Wir verwenden den ToolBar der Support-Library. Für ihn wird eine Standard-Höhe, eine Standard-Farbe und ein vordefiniertes Theme angegeben. Danach wird nur noch das Test-Textfeld für den Farbwechsel angegeben:

```
16    <TextView
17        android:id="@+id/textView"
18        android:layout_width="match_parent"
19        android:layout_height="match_parent"
20        android:text="@string/test_string"
21        android:textSize="28sp"
22        android:textColor="#000000"/>
23 </LinearLayout>
```

Die Implementierung der Activity-Klasse gestaltet sich wie folgt:

**MainActivityToolBar:**

```
1 package gui.inf.hsas.de.toolbarapp;
2
3 import android.graphics.Color;
4 import android.os.Bundle;
5 import android.support.v7.app.AppCompatActivity;
6 import android.support.v7.widget.Toolbar;
7 import android.view.Menu;
8 import android.view.MenuItem;
9 import android.widget.TextView;
10
11 public class MainActivityToolbar
12         extends AppCompatActivity {
13
14     // Textfeld zum Färben
15     private TextView tv;
```

Die Activity ist nun von einer Activity der Support-Library abgeleitet:  
von **AppCompatActivity**.

```
16  @Override  
17  protected void onCreate(Bundle savedInstanceState) {  
18      super.onCreate(savedInstanceState);  
19      setContentView(R.layout.activity_main_toolbar);  
20      // ToolBar abfragen  
21      Toolbar toolbar =  
22          (Toolbar) findViewById(R.id.toolbar);  
23      // Textfeld abfragen  
24      tv = (TextView) findViewById(R.id.textView);  
25      // ToolBar sichtbar machen mit ererbter  
26      // Methode  
27      setSupportActionBar(toolbar);  
28  } // end method
```

Wir fragen beim Rendern der Activity die Referenzen von ToolBar und Textfeld ab. Der ToolBar wird „sichtbar“ geschaltet.

Wir belassen die Rendering-Methode für unser Menü so, wie sie von der Entwicklungsumgebung erzeugt wurde:

```
31  @Override  
32  public boolean onCreateOptionsMenu(Menu menu) {  
33      // Menu-Balken aufklappen  
34      getMenuInflater().inflate(  
35          R.menu.menu_main_activity_toolbar, menu);  
36      return true;  
37  } // end method
```

Der letzte Abschnitt zeigt die Auswertung der Nutzeraktionen im Menü:

```
38  @Override
39  public boolean onOptionsItemSelected(
40      MenuItem item) {
41      // Auswertung Menu-Items
42      int id = item.getItemId();
43
44      // Rot-/Gruen- oder Blaufärbung
45      if (id == R.id.action_red) {
46          tv.setTextColor(Color.RED);
47          return true;
48      }
49      else if (id == R.id.action_green) {
50          tv.setTextColor(Color.GREEN);
51          return true;
52      }
53      else if (id == R.id.action_blue) {
54          tv.setTextColor(Color.BLUE);
55      }
56      return super.onOptionsItemSelected(item);
57  } // end method
58 } // end class
```

#### 4.2.2 Attribut android:onClick von Android-Views – *Beispiel10Android*

Das Beispiel, an dem wir diesen Mechanismus betrachten, zeigt eine *TextView* und *Button*-Objekte:

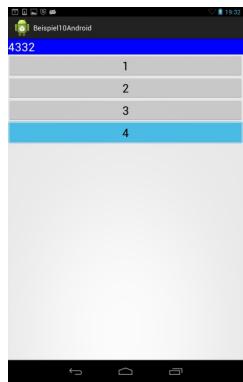


Abbildung 4.5: Screenshot *Beispiel10Android*

Das Layout der App besteht aus:

- einem *LinearLayout*,
- einer *TextView*, sowie
- vier *Buttons*

Sobald einer der Buttons gedrückt wird, wird seine Beschriftung an die *TextView* angehängt. Das Reaktionsverhalten der vier Buttons wurde innerhalb der Java-Klasse der Haupt-Activity in einer Reaktionsmethode implementiert.

Die Verknüpfung dieser Methode mit den zugehörigen Buttons geschieht  
in der XML-Layoutdefinition der Activity:

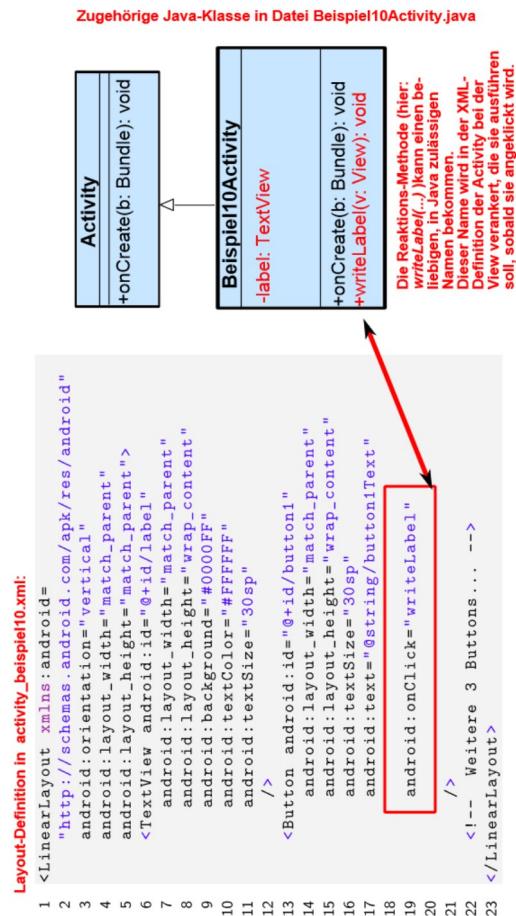


Abbildung 4.6: Reaktions-Methode in *Beispiel10Android*

Die Java-Implementierung der Activity sieht nun folgendermaßen aus:

```
1 package guidev.inf.hdas.de.beispiel10neu;
2
3 import android.os.Bundle;
4 import android.app.Activity;
5 import android.view.View;
6 import android.widget.Button;
7 import android.widget.TextView;
8
9 public class Beispiel10Activity
10     extends Activity {
11
12     // Das TextView wird ueber seine ID
13     // an diese private Variable zugewiesen
14     private TextView label;
```

In diesem ersten Code-Abschnitt sehen wir die Klassenvereinbarung unserer Activity. Die Klasse enthält als einziges privates Attribut eine *TextView*-Referenz.

Im folgenden Code-Abschnitt sehen wir die Implementierung der Methode `onCreate()`. In ihr wird die Referenz des Textfeldes über die ererbte Methode `findViewById(...)` ermittelt. Sie

- bekommt die in der generierten Klasse *R* verankerte ID der *TextView* übergeben und
- liefert die Referenz auf das *TextView*-Objekt zurück.

```
16     @Override
17     protected void onCreate(
18         Bundle savedInstanceState) {
19
20         super.onCreate(savedInstanceState);
21
22         setContentView(
23             R.layout.activity_beispiel10);
24
25         // Widgets anhand ihrer ID an die
26         // Objektvariablen zuweisen: Dies
27         // geschieht mit der ererbten
28         // Methode findViewById()
29         this.label =
30             (TextView) findViewById(R.id.
31             label);
31     } // end method onCreate()
```

Da die Methode `onCreate()` beim Start bzw. beim Neu-Rendering der Oberfläche zuerst aufgerufen wird (vgl. Konstruktoren in „Nicht-Android“-Klassen), ist die hier abgefragte Referenz für alle später von uns aufgerufenen Methoden zugreifbar.

Der letzte Teil der Activity beinhaltet die Reaktionsmethode, die bereits in Abbildung 4.6 eingeführt wurde:

```
32 // Die Reaktions-Methode muss immer
33 // eine Referenz vom Typ View als
34 // Parameter bekommen. Sie wird auf
35 // den echten Datentyp gecastet.
36 public void writeLabel(View v) {
37
38     // hole uebergebene Referenz --
39     // muss einer unserer Buttons sein
40     Button b = (Button) v;
41
42     // Beschriftung des Labels wird
43     // als CharSequence geliefert.
44     String s = label.getText().toString();
45
46     // Haenge die Beschriftung des
47     // Buttons an den String an
48     s += b.getText().toString();
49
50     // Setze neuen String ins Label
51     label.setText(s);
52 } // end method writeLabel()
53
54 } // end class
```

Reaktions-Methoden für das `android:onClick`-Attribut bekommen immer die Referenz der View, von welcher sie ausgelöst wurden, als Parameter übergeben. Der Name der Methode ist jedoch frei wählbar. Er muss lediglich mit dem Eintrag in der Layout-XML-Datei übereinstimmen (hier:

`Beispiel10Android/res/layout/activity_beispiel10.xml`).

#### 4.2.3 Listener-Mechanismus von Android– *Beispiel-11Android*

Der Listener-Mechanismus von Android ist dem aus SWT sehr ähnlich. Haupt-Unterschied: Die Listener-Interfaces sind in der Regel als „Inner Interfaces“ – analog zu „Inner Classes“ in der Klasse *View* und deren Kindklassen verankert:

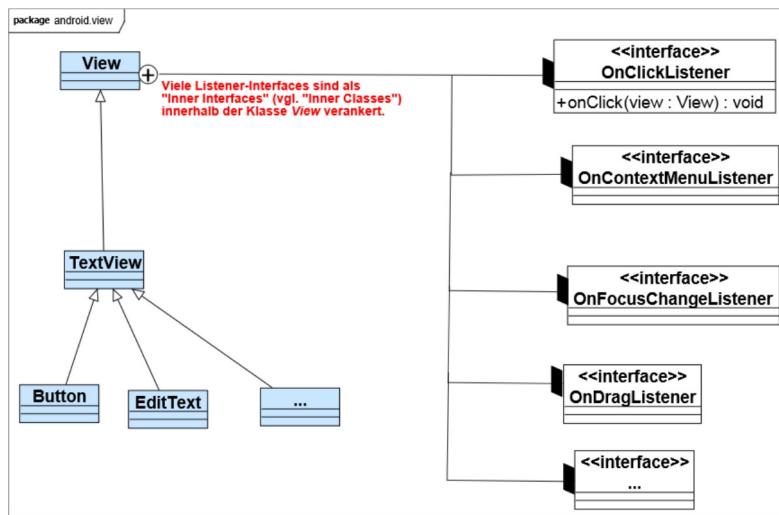


Abbildung 4.7: Listener-Interfaces in Android

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

Das folgende Beispiel *Beispiel11* ist äußerlich sehr ähnlich zum vorhergehenden *Beispiel10Android*: Auch bei diesem Beispiel wird, sobald einer



Abbildung 4.8: Screenshot *Beispiel11*

der vier Buttons gedrückt wird, seine Beschriftung an die der *TextView* angehängt. Unterschiede zu *Beispiel10Android*:

- Das Reaktionsverhalten der Buttons wird in einem ausgelagerten, nicht anonymen Listener implementiert.
- Die Buttons sind nicht in der XML-Layout-Datei (hier: `Beispiel11/app/src/main/res/layout/activity_beispiel11.xml`) definiert.
- Statt dessen werden die Buttons rein programmatisch innerhalb der `onCreate(...)`-Methode der Haupt-Activity-Klasse *Beispiel11Activity* erzeugt und in das Layout eingehängt.

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

Die Klassenhierarchie dieses Beispiels lässt sich folgendermaßen beschreiben:

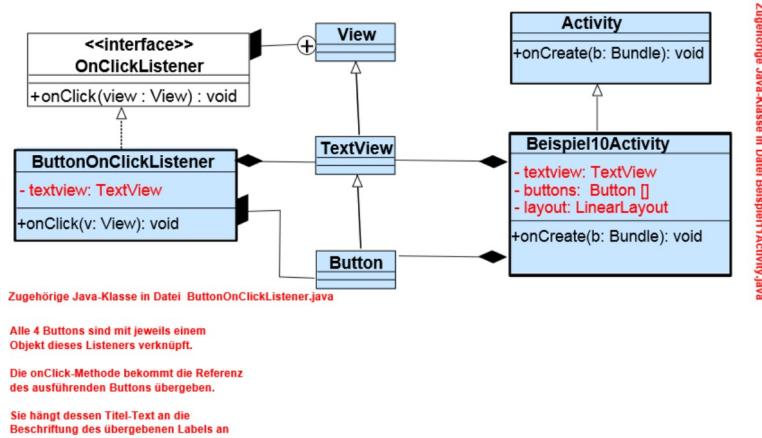


Abbildung 4.9: Klassenhierarchie *Beispiel11*

Wir zeigen zunächst die sehr geschrumpfte Layout-Vereinbarung von  
*Beispiel11Android*:

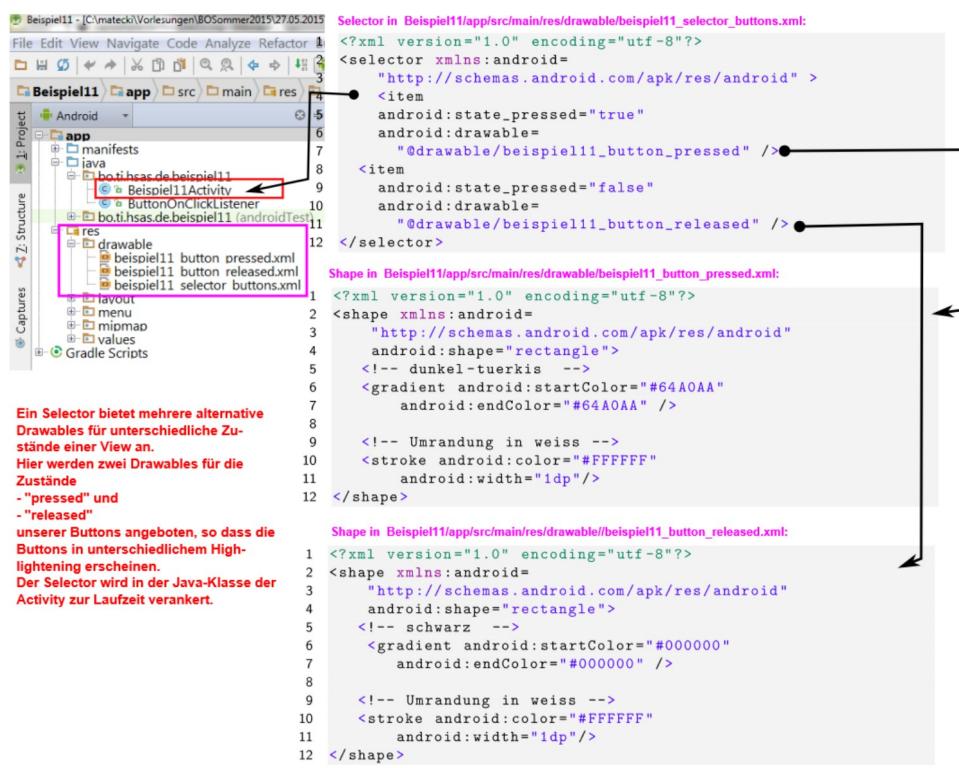
**activity\_beispiel11.xml:**

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk
3     /res/android"
4   android:orientation="vertical"
5   android:layout_width="match_parent"
6   android:layout_height="match_parent"
7   android:id="@+id/layout_beispiel11"
8 >
9   <TextView android:id="@+id/label"
10    android:layout_width="match_parent"
11    android:layout_height="wrap_content"
12    android:background="#0000FF"
13    android:textColor="#FFFFFF"
14    android:textSize="30sp"
15  />
16 </LinearLayout>
```

Wir sehen, dass in diesem Layout keiner der vier Buttons definiert ist.  
Sie werden später im Java-Code der Activity zur Laufzeit erzeugt. Statt  
dessen haben wir noch die Definition für das Highlightening unserer  
Buttons als „Drawable“ hinterlegt. Ihre Verankerung erfolgt ebenfalls im  
Java-Code der Activity:

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

---



The screenshot shows the Android Studio interface with the project 'Beispiel11' open. The file tree on the left shows the project structure with files like 'AndroidManifest.xml', 'Activity.java', and 'res/drawable/\*.xml'. A red box highlights the 'ButtonOnTouchListener' class in the Java package 'boti.has.de.beispiel11'. A purple box highlights the 'res/drawable' directory, which contains three XML files: 'beispiel11\_button\_pressed.xml', 'beispiel11\_button\_released.xml', and 'beispiel11\_selector\_buttons.xml'. The code for each is shown on the right:

```

Selector in Beispiel11/app/src/main/res/drawable/beispiel11_selector_buttons.xml:
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android=
    "http://schemas.android.com/apk/res/android" >
    <item
        android:state_pressed="true"
        android:drawable=
            "@drawable/beispiel11_button_pressed" />
    <item
        android:state_pressed="false"
        android:drawable=
            "@drawable/beispiel11_button_released" />
</selector>

Shape in Beispiel11/app/src/main/res/drawable/beispiel11_button_pressed.xml:
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <!-- dunkel-tuerkis -->
    <gradient android:startColor="#64A0AA"
              android:endColor="#64A0AA" />
    <!-- Umrundung in weiss -->
    <stroke android:color="#FFFFFF"
            android:width="1dp"/>
</shape>

Shape in Beispiel11/app/src/main/res/drawable/beispiel11_button_released.xml:
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <!-- schwarz -->
    <gradient android:startColor="#000000"
              android:endColor="#000000" />
    <!-- Umrundung in weiss -->
    <stroke android:color="#FFFFFF"
            android:width="1dp"/>
</shape>

```

**Ein Selector bietet mehrere alternative Drawables für unterschiedliche Zustände einer View an. Hier werden zwei Drawables für die Zustände "pressed" und "released" unserer Buttons angeboten, so dass die Buttons in unterschiedlichem Highlighting erscheinen. Der Selector wird in der Java-Klasse der Activity zur Laufzeit verankert.**

Abbildung 4.10: Selector für das Highlighting der Buttons

Die Implementierung der Activity ist nun etwas umfangreicher geworden:

```
1 package guidev.inf.hdas.de.beispiel1neu;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5
6 import android.graphics.Color;
7 import android.widget.Button;
8 import android.widget.LinearLayout;
9 import android.widget.TextView;
10
11 public class Beispiel11Activity
12     extends Activity {
13
14     // Anzahl der einzufuegenden Buttons
15     private final int NO_OF_BUTTONS=4;
16
17     // Array fuer die Buttons
18     private Button [] buttons;
19
20     //Referenz des Layouts
21     // der Haupt-Activity
22     LinearLayout mainLayout;
23
24     // Referenz auf die TextView
25     TextView textView;
```

Wir haben hier folgende private Referenzen hinterlegt:

- Ein Array für 4 *Button*-Referenzen. Diese entstehen zur Laufzeit.
- Eine Referenz für das *LinearLayout* aus der XML-Definition. Die erzeugten *Button*-Referenzen müssen zur Laufzeit mit dem Layout verknüpft werden.
- Die *TextView*, die anzeigen soll, welcher Button gedrückt wurde. Diese *TextView* ist bereits im Layout verankert. Sie wird – wie

gewohnt – über ihre ID aus Klasse *R* verankert.

Die Methode *onCreate()* beginnt wie gewohnt:

```
26
27     @Override
28     protected void onCreate(
29         Bundle savedInstanceState) {
30
31         // onCreate()-Methode der Superklasse
32         // aufrufen
33         super.onCreate(savedInstanceState);
34
35         // Das Layout der Haupt-Activity als
36         // Inhalt bestimmen
37         setContentView(
38             R.layout.activity_beispiel11);
39
40         // TextView holen
41         textView = (TextView)this.findViewById
42             (
43                 R.id.label);
44
45         // Layout der Haupt-Activity holen
46         mainLayout =
47             (LinearLayout)this.
48                 findViewById(
49                     R.id.layout_beispiel11
50                 );
```

Zuerst wird die ererbte *onCreate()*-Methode aufgerufen und der Inhalt der Activity bestimmt. Danach werden die privaten Referenzen, deren Views bereits im XML-Layout definiert wurden, über die Klasse *R* mit diesen verknüpft.

Es folgt die Erzeugung und Verankerung der Buttons:

```
48     // Array fuer die Buttons anlegen
49     buttons = new Button[NO_OF_BUTTONS];
50
51     // Schleife zum Anlegen der Buttons
52     for (int i=0; i<buttons.length;i++) {
53
54         // Button erzeugen -- geschieht
55         // hier
56         // vollstaendig programmatisch
57         buttons[i] = new Button(this);
58         buttons[i].setTextSize(30);
59
60         // Button seinen Hintergrund-Style
61         // als Drawable mitgeben
62         buttons[i].setBackgroundResource(
63             R.drawable.
64                 beispiel11_selector_buttons
65             );
66
67         // Button seine Textfarbe geben
68         buttons[i].setTextColor(
69             Color.parseColor("#FFFFFF"
70             ));
71
72         // Button mit Nummer beschriften
73         String title = new String(""+i);
74         buttons[i].setText(title);
75
76         // Button mit Listener verknuepfen
77         buttons[i].setOnClickListener(
78             new ButtonOnClickListener(
79                 textview));
80
81         // Button ins Layout einhaengen
82     }
83 }
```

```
78         mainLayout.addView(buttons[i]);  
79     } // end for  
80 } // end method onCreate()  
81 } // end class
```

Nachdem das Array für die 4 Button-Referenzen erzeugt wurde, werden in einer Schleife die Buttons erzeugt und mit dem umrahmenden Layout verknüpft. Wichtig sind hier zwei Statements des Schleifenrumpfes:

- Zeile 60-63: Hier wird der gerade erzeugte Button mit seinem Selector für das Highlightening-Verhalten verknüpft.
- Zeile 74-75: Hier wird der erzeugte Button mit einem Listener-Objekt der selbst geschriebenen Klasse *ButtonOnClickListener* verknüpft. Diese Klasse wird in den nächsten Abschnitten erläutert.

Die folgende Listener-Klasse implementiert das Verhalten der vier Buttons:

```
1 package guidev.inf.hsas.de.beispiel1neu;  
2 import android.view.View;  
3 import android.view.View.OnClickListener;  
4 import android.widget.Button;  
5 import android.widget.TextView;  
6  
7 public class ButtonOnClickListener  
8     implements OnClickListener {  
9  
10    TextView textview;  
11  
12    // Der Listener bekommt die  
13    // TextView uebergeben, in die  
14    // er hineinmalen soll.  
15    public ButtonOnClickListener(  
16        TextView textview) {  
17        super();  
18        this.textview = textview;  
19    }
```

Der Konstruktor bekommt die Referenz des Textfeldes mit, in welches beim Drücken eines der vier Buttons hinein geschrieben wird.

Die Methode `onClick()` tut nun das Gleiche, wie in *Beispiel10Android* die Reaktionsmethode, welche direkt innerhalb der Activity programmiert war:

```
21  @Override
22  public void onClick(View view) {
23
24      // Uebergebene View ist der reagierende
25      // Button
26      Button btn = (Button) view;
27
28      // Text aus Button holen
29      String buttonText =
30          btn.getText().toString();
31
32      // Text aus TextView holen
33      String textviewText =
34          textview.getText().toString();
35
36      // Button-Beschriftung an
37      // TextView-Beschriftung haengen
38      textviewText += buttonText;
39
40      // TextView-Beschriftung neu
41      // setzen
42      textview.setText(textviewText);
43
44  } // end method onClick()
45 } // end class ButtonOnClickListener
```

#### 4.2.4 Touch-Events und Wischgesten

Ein großer Unterschied in der Haptik von mobilen Applikationen liegt in der Verarbeitung von Berührungen auf dem Display. Aus der Kombination der „richtigen“ Display-Berührungen (Touch-Events) ergeben sich Gesten, wie z. B. Wischen, Zoomen, und viele mehr.

Um überhaupt Gesten analysieren zu können, müssen wir zunächst verstehen, wie Touch-Events verarbeitet werden. Sobald eine Berührung mit dem Finger auf dem Display erfolgt, wird ein **MotionEvent** geworfen. In diesem Event-Objekt merkt die App sich, an welcher x/y-Koordinate die Berührung stattfand:

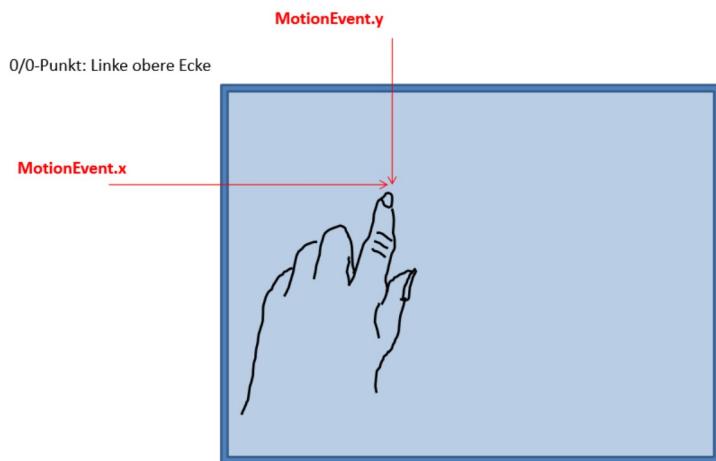


Abbildung 4.11: *MotionEvent* bei Berührung des Displays

Derartige MotionEvents werden immer **von der View, die gerade bearbeitet wird**, geworfen.

Mit der Methode ***getAction()*** können wir den int-Code des genauen Ereignisses, welches erkannt wurde, abfragen. Hier die wichtigsten Codes:

Wichtigste Codes von MotionEvents	
Code	Erklärung
ACTION_CANCEL	Geste wurde abgebrochen
ACTION_DOWN	Geste wurde gestartet (Erster Finger auf das Display gelegt)
ACTION_MOVE	Geste ist in Bewegung (auf dem Display)
ACTION_UP	Geste wurde beendet (Finger weg vom Display)
ACTION_POINTER_DOWN	Zweiter Finger wurde auf das Display gelegt (Multitouch-Gesten)
ACTION_POINTER_UP	Zweiter Finger wurde vom Display genommen (Multitouch-Gesten)

Tabelle 4.2: Wichtigste Codes von MotionEvents

#### 4.2.4.1 Einfache Touch-Events verarbeiten

Unsere erste App mit Touch-Verarbeitung arbeitet folgendermaßen:

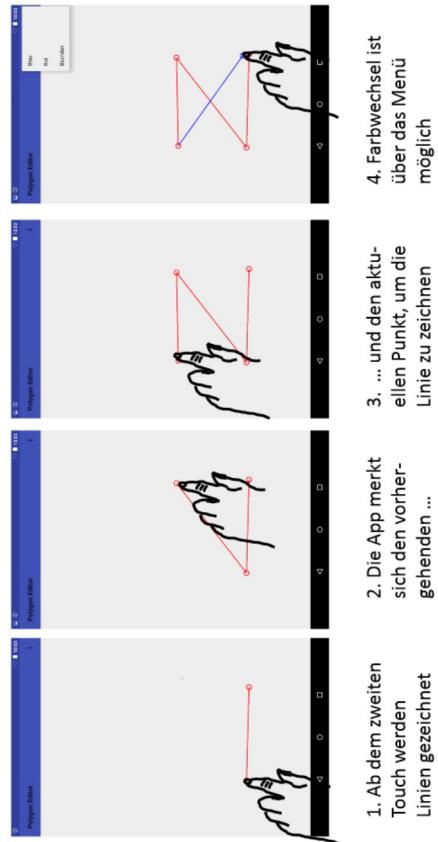


Abbildung 4.12: PolygonApp mit Touch-Events

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

Bei der ersten Berührung in der ImageView wird ein Punkt gezeichnet. Ab der zweiten Berührung werden die gezeichneten Punkte jeweils über eine Gerade mit ihrem Vorgänger verbunden. Wir haben also einen kleinen Polygoneditor. Die Klassenhierarchie ist hier sehr einfach: Die (einzige)

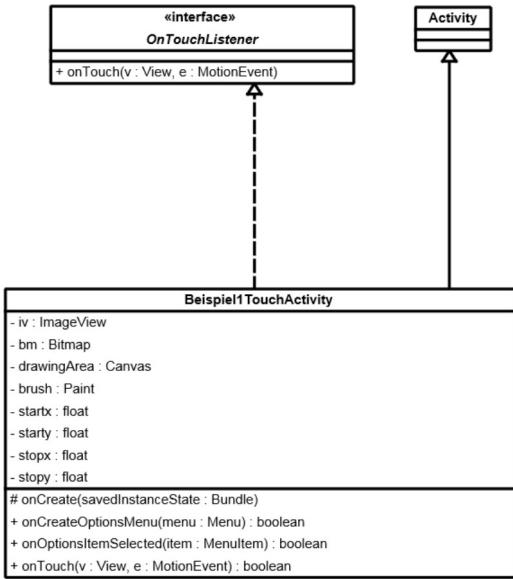


Abbildung 4.13: Klassenhierarchie der PolygonApp

Activity ist von der Klasse *Activity* abgeleitet. Zusätzlich implementiert sie das Interface *OnTouchListener*. Auf diese Weise ist die Reaktionsmethode auf Berührungen des Displays einfach in die Activity als weitere Methode integriert.

Wir zeigen zunächst das sehr einfach gehaltene Layout: Die Implementierung der Activity ist nun etwas umfangreicher geworden:

`layout/activity_beispiel1_touch.xml:`

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/
4         apk/res/android"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:orientation="vertical"
8     >
9         <ImageView
10            android:layout_width="match_parent"
11            android:layout_height="match_parent"
12            android:id="@+id/draw" />
13     </LinearLayout>
```

Die Beschriftungen der App sind (wie immer) in unserer externen XML-Ressource gebündelt:

`values/strings.xml:`

```
1 <resources>
2     <string name="app_name">Polygon Editor</
3         string>
4     <string name="red">Rot</string>
5     <string name="blue">Blau</string>
6     <string name="bye">Beenden</string>
7 </resources>
```

Das Layout ist ergänzt durch ein Menü für Farbauswahl und Beendigung  
der App:

menu/menu\_beispiel1\_touch.xml:

```
1 <menu xmlns:android="http://schemas.android.
  com/apk/res/android"
2   xmlns:app="http://schemas.android.com/apk/
    res-auto"
3   xmlns:tools="http://schemas.android.com/
    tools"
4   tools:context=".Beispiel1TouchActivity">
5     <item android:id="@+id/action_blue"
6       android:title="@string/blue"
7       android:orderInCategory="100"
8       app:showAsAction="collapseActionView" />
9     <item android:id="@+id/action_red"
10       android:title="@string/red"
11       android:orderInCategory="100"
12       app:showAsAction="collapseActionView" />
13     <item android:id="@+id/action_bye"
14       android:title="@string/bye"
15       android:orderInCategory="100"
16       app:showAsAction="collapseActionView" />
17 </menu>
```

Der interessante Teil der App verbirgt sich in der Java-Implementierung  
der Haupt-Activity:

```
1 package guidev.inf.hsas.de.  
    beispieltouch1androidneu;  
2  
3 import android.app.Activity;  
4 import android.graphics.Bitmap;  
5 import android.graphics.Canvas;  
6 import android.graphics.Color;  
7 import android.graphics.Paint;  
8 import android.graphics.Point;  
9 import android.os.Bundle;  
10 import android.view.Display;  
11 import android.view.Menu;  
12 import android.view.MenuItem;  
13 import android.view.MotionEvent;  
14 import android.view.View;  
15 import android.view.View.OnTouchListener;  
16 import android.widget.ImageView;  
17  
18 public class Beispiel1TouchActivity  
19     extends Activity  
20     implements OnTouchListener {
```

Die Klasse *Beispiel1TouchActivity* ist – wie gewohnt – von *Activity* abgeleitet. Sie implementiert aber zusätzlich das Listener-Interface *OnTouchListener*. Dieses gibt die weiter unten besprochene Methode *onTouch()* vor.

Folgende private Variablen wurden vereinbart:

```
21 // Referenz auf die im Layout eingebaute  
22     ImageView  
23     private ImageView iv;  
24  
25 // Bitmap, die von der ImageView angezeigt  
26     wird  
26     private Bitmap bm;  
27  
28 // Malfläche auf der Bitmap  
29     private Canvas drawingArea;  
30  
31 // Pinsel zum Malen  
32     private Paint brush;  
33  
34 // Startpunkt zum Malen  
35     private float startx=-1;  
36     private float starty=-1;  
37  
38 // Endpunkt zum Malen  
39     private float stopx=-1;  
40     private float stopy=-1;
```

In der `onCreate()`-Methode geschieht nun etwas mehr:

```
42 @Override
43     protected void onCreate(
44         Bundle savedInstanceState) {
45
46     // ererbte Superklassen-Methode
47     // aufrufen
48     super.onCreate(savedInstanceState);
49
50     // Layout laden
51     setContentView(
52         R.layout.activity_beispiel1_touch);
53
54     // Referenz auf ImageView
55     // aus Layout abfragen
56     iv = (ImageView)
57         this.findViewById(R.id.draw);
```

Die Referenz der `ImageView` aus der Layoutvereinbarung wird zunächst  
– wie üblich – über ihre ID abgefragt.

Um in der ImageView zeichnen zu können, wird jedoch eine „Malfläche“ – eine *Canvas* mit einer *Bitmap* benötigt. Die Größe der Malfläche wird durch die in ihr platzierte *Bitmap* bestimmt. Sie wird so breit wie das Display und so hoch, dass sie unter den ToolBar passt.

```
59     // Geometrie des Geraete-Displays
60     // abfragen
61     Display display =
62         getWindowManager().getDefaultDisplay()
63         ;
64     Point size = new Point();
65     display.getSize(size);
66     int width = size.x;
67
68     // Hoehe der Bitmap so hoch
69     // wie die ImageView
70     // unterhalb des ActionBar
71     int height = size.y / 10 * 9;
72
73     // Bitmap erzeugen --
74     // Farbmodell RGB
75     bm = Bitmap.createBitmap(
76         width, height,
77         Bitmap.Config.ARGB_8888);
78
79     // Malflaeche auf Bitmap erzeugen
80     drawingArea = new Canvas(bm);
81
82     // Bitmap in ImageView einsetzen
83     iv.setImageBitmap(bm);
```

Danach wird die *ImageView* mit ihrem *OnTouchListener* verknüpft – in diesem Fall mit der Referenz auf DIESE Activity, die ja die Implementierung der *onTouch*-Methode enthält. Zuletzt wird noch das Malwerkzeug erstellt:

```
83
84     // Diese Activity ist auch
85     // das Objekt mit der
86     // onTouch-Reaktionsmethode
87     iv.setOnTouchListener(this);
88
89     // Malpinsel erstellen
90     brush = new Paint();
91     brush.setStyle(Paint.Style.STROKE);
92     brush.setColor(Color.RED);
93     brush.setStrokeWidth(3.0f);
94 } // end method onCreate()
```

Die Methoden für den Menü-Aufbau sind ähnlich, wie in den letzten Beispielen:

```
95 @Override
96 public boolean onCreateOptionsMenu(
97     Menu menu) {
98     // Menu aufbauen
99     getMenuInflater().inflate(
100         R.menu.menu_beispiel1_touch, menu);
101     return true;
102 } // end method onCreateOptionsMenu()
103
104 @Override
105 public boolean onOptionsItemSelected(
106     MenuItem item) {
107
108     // Welche Menu-Option wurde gedrueckt?
109     int id = item.getItemId();
110
111     // Beenden
```

```
112     if (id == R.id.action_bye){
113         this.finish();
114         return true;
115     }
116     // Pinsel auf Rot umschalten
117     else if(id==R.id.action_red) {
118         brush.setColor(Color.RED);
119         return true;
120     }
121     // Pinsel auf blau umschalten
122     else if(id==R.id.action_blue) {
123         brush.setColor(Color.BLUE);
124         return true;
125     }
126
127     return
128     super.onOptionsItemSelected(item);
129 } // end method onOptionsItemSelected()
```

Interessant wird die Implementierung der *onTouch()*-Methode:

```
130 @Override  
131     public boolean onTouch(View v,  
132                           MotionEvent e) {  
133         // Radius des zu zeichnenden  
134         // Punktes  
135         int radius=10;  
136  
137         // Genaue Aktion innerhalb  
138         // des MotionEvents abfragen  
139         int action = e.getAction();
```

Sie bekommt die Referenz auf die auslösende View (hier: unsere *ImageView*) übergeben. Der zweite Parameter ist das *MotionEvent*, welches beim Touch auf das Display innerhalb der *ImageView* geworfen wird. Von diesem Event wird nun der genaue Code abgefragt. In unserem Falle ist nur *ACTION\_DOWN* interessant – also der Start des Touch:

```
141     switch (action) {  
142  
143         // Finger auf die Malflaeche  
144         // gesetzt  
145         case MotionEvent.ACTION_DOWN:  
146  
147             // Erster Touch:  
148             // Nur Punkt zeichnen  
149             if(startx == -1 && starty == -1) {  
150                 startx = e.getX();  
151                 starty = e.getY();  
152  
153                 drawingArea.drawCircle(  
154                     startx,starty,10,brush);  
155  
156             } // end if  
157             // Ab dem zweiten Touch:  
158             // Linie und danach Punkt  
159         else {
```

```
160         stopx = startx;
161         stopy = starty;
162         startx = e.getX();
163         starty = e.getY();
164
165
166         drawingArea.drawLine(
167             stopx, stopy,
168             startx, starty, brush);
169         drawingArea.drawCircle(
170             startx, starty,
171             radius, brush);
172     } // end else
```

Wir unterscheiden hier den ersten Touch von allen weiteren: Bei der ersten Display-Berührung soll nur ein Punkt gezeichnet werden. Bei den nachfolgenden Ereignissen soll dann zuerst eine Gerade vom Vorgänger-Punkt zur aktuellen Koordinate gezeichnet werden. Danach soll auf die aktuelle Koordinate wieder ein Punkt gezeichnet werden. Zum Schluß wird noch der Event-Puffer geleert, so dass das nächste Stück Zeichnung *jetzt* auf dem Display erscheint:

```
174
175         // Puffer JETZT leeren und
176         // auf der ImageView sichtbar
177         // machen
178         iv.invalidate();
179         break;
180     default:
181         break;
182     } // end switch
183     return true;
184 } // end method onTouch
185 } // end class
```

#### 4.2.4.2 Einfache Wisch-Gesten verarbeiten

Wisch-Gesten sind nichts anderes als zeitliche Abfolgen von *MotionEvent*-Events. Wir verdeutlichen dies in folgender Abbildung:

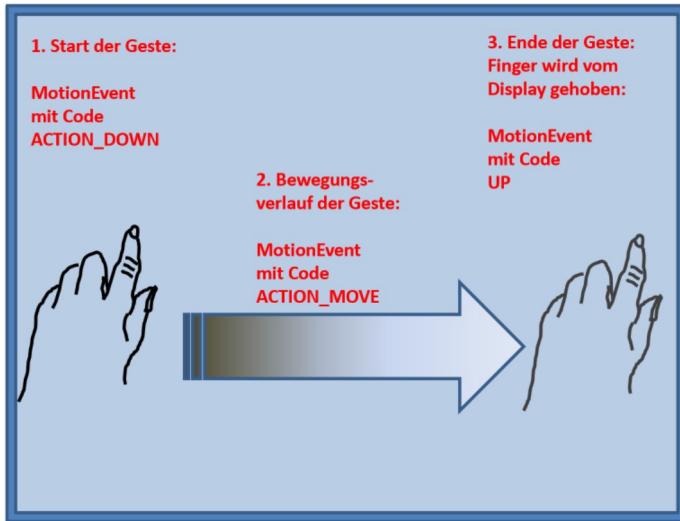


Abbildung 4.14: Wischgeste als Abfolge von *MotionEvent*s

Mit anderen Worten: Bei einer Wischgeste wird ein `onTouch`-Listener mehrfach hintereinander aufgerufen. Softwaretechnisch wird es nun etwas komplizierter. Um die Analyse der zeitlichen Abfolgen derartiger Events nicht „von Hand“ implementieren zu müssen, hat Android hier bereits zwei Klassen und einige Interfaces zur Erkennung gängiger Gesten vorgesehen:

- Die Klasse *GestureDetector* ist für die Erkennung gängiger Gesten, wie z.B.

- Wischen (*Fling*)
- Einfacher Klick (*Tap*)
- Doppelklick (*DoubleTap*)

und einige mehr zuständig. Sie hat hierfür einige *Inner Interfaces* vorgesehen. Der *GestureDetector* bekommt i.d.R. von einem *OnTouchListener* die einzelnen *MotionEvent*s geliefert und versucht aus ihnen, die jeweilige Art der Geste zu erkennen.

- Die Interfaces *OnContextClickListener*, *OnDoubleTapListener*, und *OnGestureListener*. Diese Listener sind dann dafür zuständig, das Reaktionsverhalten der App auf eine bestimmte Art von Geste hin zu implementieren. Das häufig benutzte Interface *GestureDetector.SimpleOnGestureListener* hält Methodendefinitionen für viele Arten von Gesten bereit. Für uns ist hier interessant: Die Wisch-Geste (*Fling*)
- Die Klasse *SimpleOnGestureListener*. Sie ist eine Adapter-Klasse, die bereits die drei oben genannten Interfaces implementiert. Eigene Gesten-Listener werden häufig von ihr abgeleitet und müssen dann nur noch die interessierenden Methoden überschreiben.

Die folgende Abbildung zeigt das Zusammenspiel zwischen auslösender View, *GestureDetector*, *GestureListener* und *OnTouchListener*:

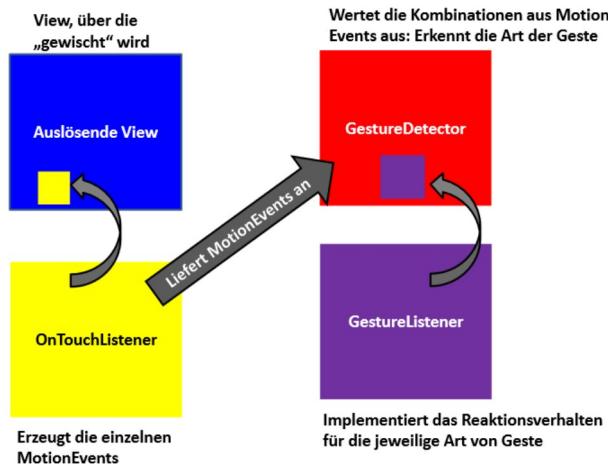


Abbildung 4.15: Zusammenspiel *GestureDetector* mit den unterschiedlichen Listenern

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

Die nachfolgende App zeigt ein einfaches, horizontales Wisch-Verhalten mit einer TextView:

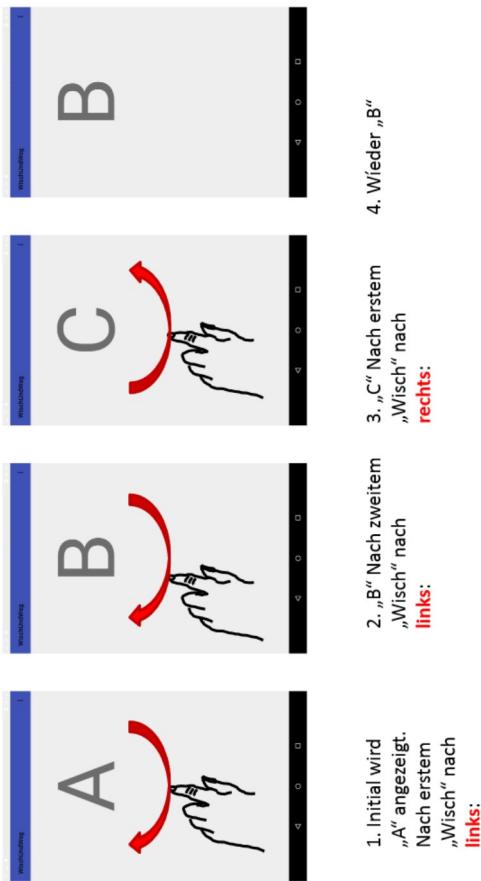


Abbildung 4.16: Verhalten der „WischUndWeg“-App

## KAPITEL 4. ARBEITSWEISE VON ANDROID-VIEWS, EVENTS, CALLBACKS, TOOLBAR

Die Klassenhierarchie sieht hier folgendermaßen aus:

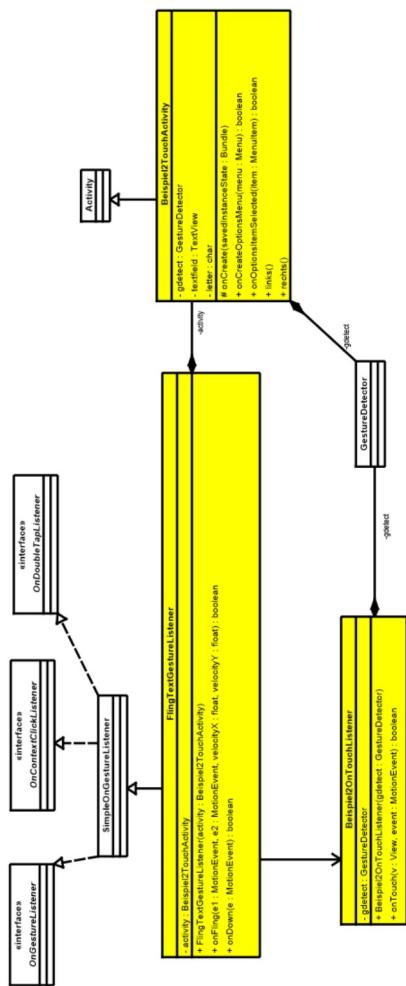


Abbildung 4.17: Klassen der „WischUndWeg“-App

Wir beginnen wieder mit dem Layout:

`layout/layout/activity_beispiel2_touch.xml`

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/
4         apk/res/android"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:orientation="vertical">
8     <TextView
9         android:layout_width="match_parent"
10        android:layout_height="match_parent"
11        android:id="@+id/text"
12        android:textSize="250sp"
13        android:text="@string/start_letter"
14        android:textAlignment="center"/>
15 </LinearLayout>
```

Im linearen Layout ist nur eine *TextView* mit sehr großem Font eingebettet. Die Beschriftungen der App sind (wie immer) in unserer externen XML-Ressource gebündelt:

`values/strings.xml`

```
1 <resources>
2     <string name="app_name">WischUndWeg</string>
3     <string name="action_bye">Beenden</string>
4     <string name="start_letter">A</string>
5 </resources>
```

Das Menü im ToolBar ist ebenfalls sehr einfach mit nur einer Menüoption:  
menu/menu\_beispiel2\_touch.xml

```
1 <menu
2   xmlns:android="http://schemas.android.com/
3     apk/res/android"
4   xmlns:app="http://schemas.android.com/apk/
5     res-auto"
6   xmlns:tools="http://schemas.android.com/
7     tools"
8   tools:context=".Beispiel2TouchActivity">
9     <item android:id="@+id/action_bye"
10       android:title="@string/action_bye"
11       android:orderInCategory="100"
12       app:showAsAction="collapseActionView" />
13 </menu>
```

Auch hier beginnt der interessante Teil in der Java-Implementierung. Wir beginnen mit der Implementierung der Activity:

```
1 package guidev.inf.hsas.de.  
    beispiel2touchandroidneu;  
2  
3 import android.app.Activity;  
4 import android.os.Bundle;  
5 import android.view.GestureDetector;  
6 import android.view.Menu;  
7 import android.view.MenuItem;  
8 import android.widget.TextView;  
9  
10 public class Beispiel2TouchActivity  
11     extends Activity {  
12  
13     // Erkennt Wisch-Geste und reicht  
14     // sie weiter an Gesten-Listener  
15     private GestureDetector gdetect;  
16  
17     private TextView textfield;  
18     private char letter = 'A';
```

Diese Activity implementiert keines der Listener-Interfaces, da hier *OnTouchListener* und *GestureListener* in die eigenen Klassen

- *Beispiel2OnTouchListener* und
- *FlingTextGestureListener*

ausgelagert wurden.

Die `onCreate()`-Methode ist hier etwas einfacher aufgebaut:

```
19  @Override
20  protected void onCreate(
21      Bundle savedInstanceState) {
22
23      super.onCreate(savedInstanceState);
24
25      setContentView(
26          R.layout.activity_beispiel2_touch);
27      // Referenz des Textfeldes abfragen
28      textfield =
29          (TextView) this.findViewById(
30              R.id.text);
31
32      // Erzeuge Gesten-Detektor und
33      // verknuepfe ihn mit Listener fuer
34      // Auswertung der Wisch-Geste
35      this.gdetect = new GestureDetector(
36          this,new FlingTextGestureListener(this
37              ));
38
39      // Externer OnTouchListener
40      this.textfield.setOnTouchListener(
41          new Beispiel2OnTouchListener(
42              gdetect));
42 } // end method onCreate()
```

Sie ermittelt anhand der ID die Referenz des Textfeldes zum „Wischen“. Danach erzeugt sie einen `GestureDetector`, der sogleich mit seinem Listener verknüpft wird. Zuletzt geben wir dem Textfeld noch seinen `OnTouchListener` mit – diesmal als Objekt einer ausgelagerten Klasse.

Die Menügestaltung sieht aus, wie gewohnt:

```
43 @Override
44     public boolean onCreateOptionsMenu(
45             Menu menu) {
46         getMenuInflater().inflate(
47             R.menu.menu_beispiel2_touch,
48             menu);
49         return true;
50     } // end onCreateOptionsMenu()
51
52 @Override
53     public boolean onOptionsItemSelected(
54         MenuItem item) {
55
56         int id = item.getItemId();
57
58         // Beenden der App
59         if (id == R.id.action_bye) {
60             this.finish();
61             return true;
62         }
63
64         return
65             super.onOptionsItemSelected(
66                     item);
67     } // end onOptionsItemSelected()
```

Interessant sind noch die beiden Methoden *links()* und *rechts()*: Sie implementieren das Wischen links herum und rechts herum:

```
68 // Blättern links herum
69 public void links(){
70     letter++;
71     if (letter > 'Z'){
72         letter = 'A';
73     }
74
75     String s = "" + letter;
76     textfield.setText(s);
77 } // end method links()
78
79 // Blättern rechts herum
80 public void rechts(){
81     letter--;
82     if (letter < 'A'){
83         letter = 'Z';
84     }
85
86     String s = "" + letter;
87     textfield.setText(s);
88 } // end method rechts()
89 } // end class
```

Als nächstes betrachten wir den (sehr kleinen) *OnTouchListener* für das Textfeld:

```
1 package guidev.inf.hisas.de.  
    beispiel2touchandroidneu;  
2  
3 import android.view.GestureDetector;  
4 import android.view.MotionEvent;  
5 import android.view.View;  
6 import android.view.View.OnTouchListener;  
7  
8  
9 public class Beispiel2OnTouchListener  
10    implements OnTouchListener {  
11  
12    private GestureDetector gdetect;  
13  
14    public Beispiel2OnTouchListener(  
15        GestureDetector gdetect) {  
16        this.gdetect = gdetect;  
17    } // end constructor
```

Er bekommt die Referenz des *GestureDetector*-Objektes von der Activity aus über den Konstruktor hineingereicht. Die *onTouch()*-Methode tut nichts anderes, als ihr *MotionEvent* an eben diesen *GestureDetector* weiterzurichten:

```
18 @Override  
19 public boolean onTouch(View v,  
20                      MotionEvent event) {  
21    // reiche MotionEvent an  
    // GestureDetector weiter.  
    // Er sammelt seine MotionEvents  
    // und baut aus ihnen die Gesten  
    // zusammen.  
22    return  
23        this.gdetect.onTouchEvent(event);  
24    } // end method onTouch()
```

29 } // end class

Der letzte Baustein ist der *GestureListener*, welcher mit dem *GestureDetector* verknüpft wurde. Sobald der *GestureDetector* eine Geste erkannt hat, wird die entsprechende Reaktionsmethode des *GestureListener*-Objektes aufgerufen. Zunächst der obere Teil des Listeners:

```
1 package guidev.inf.hsas.de.  
2     beispiel2touchandroidneu;  
3  
4 import android.view.GestureDetector.  
5     SimpleOnGestureListener;  
6 import android.view.MotionEvent;  
7  
8 // Gesten-Listener fuer Wischgesten  
9 public class FlingTextGestureListener  
10    extends SimpleOnGestureListener {  
11  
12    // Listener bekommt Activity-Referenz  
13    // Er ruft bei Wisch-Gesten deren links()  
14    // oder rechts()-Methode auf.  
15    private Beispiel2TouchActivity activity;  
16  
17    public FlingTextGestureListener(  
18        Beispiel2TouchActivity activity) {  
19        this.activity = activity;  
20    } // end constructor
```

Über den Konstruktor wird die Referenz auf die Activity hineingereicht. Auf diese Weise können die Reaktionsmethoden des Listeners die *links()*- und *rechts()*-Methoden der Activity aufrufen, sobald „gewischt“ wird.

Wir überschreiben zwei Methoden aus der Superklasse *SimpleOnGestureListener*:

```
19 // e1: MotionEvent bei Start der Wischgeste
20 // e2: MotionEvent bei Ende der Wischgeste
21 // velocityX: Geschwindigkeit der Wischgeste
22 //           in horizontaler Richtung
23 // velocityY: Geschwindigkeit der Wischgeste
24 //           in vertikaler Richtung
25 @Override
26 public boolean onFling(MotionEvent e1,
27                         MotionEvent e2,
28                         float velocityX,
29                         float velocityY) {
30
31     // Erst bei Wischgesten einer gewissen
32     // Groesse reagieren
33     float abWannReagieren = 20;
```

Die Methode *onFling()* ist die Reaktionsmethode, die vom *GestureDetector* für Wischgesten aktiviert wird. Sie bekommt zwei *MotionEvent*s übergeben. Das erste (hier: *e1*) hat typischerweise den Code *ACTION\_DOWN*. Es wird bei Start der Wischgeste geworfen. Das zweite Event hat typischerweise den Code *ACTION\_UP*. Es wird bei Ende der Wischgeste geworfen. Außerdem wird die Wischgeschwindigkeit in x- und in y-Richtung übergeben. Zunächst wird ein Schwellwert *abWannReagieren* festgelegt. Er sorgt dafür, dass die App nicht schon bei versehentlichen „Mini-Wischs“ reagiert.

Nun folgt die eigentliche Auswertung der Wischgeste. Mit dieser Abfrage unterscheiden wir, ob von links nach rechts oder von rechts nach links gewischt wurde:

```
34 // Wir checken nur links und
35 // rechts herum ab
36 if((e1.getX() - e2.getX()) >
37         abWannReagieren){
38     activity.links();
39 }
40 else if((e2.getX() - e1.getX()) >
41         abWannReagieren){
42     activity.rechts();
43 }
44 } // end else
```

Am Ende der Methode geben wir das Ergebnis des Aufrufes der Superklassenmethode (immer true) zurück:

```
46     return super.onFling(
47         e1, e2,
48         velocityX, velocityY);
49 } // end method onFling()
```

Die zweite Methode, die wir in diesem Listener überschreiben, ist die `onDown()`-Methode:

```
50  @Override
51  public boolean onDown(MotionEvent e) {
52
53      // muss IMMER true zurueckgeben
54      // wird bei Start der Geste aufgerufen.
55      // Falls FALSE, wird der Rest der
56      // nachfolgenden Geste nicht erkannt.
57      return true;
58  } // end method onDown()
59 } // end class
```

Sie wird beim Start der Geste aufgerufen und muss immer true zurückgeben, damit der Rest der Geste erkannt wird.

#### 4.2.5 Zusammenfassung zum Event-Modell in Android

Wir fassen zusammen:

##### Merke

Dialogverhalten kann für Android-Apps auf folgenden Wegen hinterlegt werden:

- **android:onClick-Attribut** nutzen: Hier wird die Reaktionsmethode direkt in der Activity-Klasse implementiert. Ein Querverweis auf die Methode erfolgt im XML-Layout der Activity über das android:onClick- Attribut der entsprechenden Controls.
- **Listener**: Hier können – wie bereits bei SWT gelernt – ausgelagerte Listener- Klassen hinterlegt werden. Sie werden zur Laufzeit innerhalb der Java-Implementierung ihrer Activity mit den Views verknüpft, welche sie auslösen sollen. Diese Listener erlauben eine genauere Auswertung von Events (nicht nur onClick!).
- **Items auf dem ToolBar**: Ihre Selektion wird in der ererbten Methode onOptionsItemSelected(...) mit einer einfachen Fallunterscheidung ausgewertet.
- **Touch-Events**: Sie werden für die App auswertbar durch Implementierung des Interfaces OnTouchListener. Bei jedem Touch auf das Display wird ein MotionEvent geworfen.

- **Gesten:** Gesten sind zeitliche Abfolgen von `MotionEvent`-Objekten. Sie müssen mit einem `GestureDetector` ausgewertet werden. Dieser muss mit einem Listener z.B. vom Typ `SimpleOnGestureListener` verknüpft werden. Außerdem muss er von einem `OnTouchListener` die `MotionEvent`-Objekte gemeldet bekommen.

## Kapitel 5

# Activities, Intents und SubActivities

**Bisher entwickelt:** Apps, welche aus einer einzigen Activity bestanden.

**Neu:**

- Apps, bestehend aus mehreren Activities (Haupt-Activity startet über ein **Intent** eine Sub-Activity)
- Apps, deren Activities über **Intents** Informationen austauschen

Grundgedanke dahinter:

- Activities sollen unabhängig voneinander existieren können (Entkopplung)
- Ein Intent formuliert die „Absicht“ der App, was die Activities miteinander tun sollen.
- →Activities können immer wieder neu miteinander kombiniert werden. Wichtig ist nur, welche Informationen sie eventuell von einer anderen Activity benötigen.

Sobald eine App aus mehreren Activities besteht, die sich gegenseitig starten können, stehen für den Entwickler folgende Arbeitsschritte an:

- Jede Activity wird mit ihrer Layout-Datei und ihrer eigenen Java-Klasse implementiert.
- Jede Activity wird in die Manifest-Datei eingetragen.

In der Regel werden diese Arbeitsschritte von der Entwicklungsumgebung zu einem großen Teil automatisiert.

## 5.1 Activities und Manifest-Datei

Die folgende Abbildung zeigt einen Auszug aus der Manifest-Datei einer App mit einer Haupt- und einer Sub-Activity:

**Auszug aus Beispiel6/AndroidManifest.xml**

```
1 <!-- ... weitere Definitionen ... -->
2 <!-- application-Definition im Manifest -->
3 <application
4     android:allowBackup="true"
5     android:icon="@mipmap/ic_launcher"
6     android:label="@string/app_name"
7     android:theme="@style/AppTheme" >
8
9     <activity
10        android:name=".Beispiel16Activity"
11        android:label="@string/app_name" >
12         <intent-filter>
13             <action android:name="android.intent.action.MAIN" />
14             <category android:name="android.intent.category.LAUNCHER" />
15         </intent-filter>
16     </activity>
17
18     <activity
19        android:name=".NewContactActivity"
20        android:label="@string/title_activity_new_contact" >
21         <intent-filter>
22             <action android:name="android.intent.action.DELETE" />
23             <category android:name="android.intent.category.DEFAULT" />
24         </intent-filter>
25     </activity>
26
27 </application>
```

Abbildung 5.1: Manifest-Datei mit mehreren Activities

Sobald eine Android-App aus mehreren Activities besteht, müssen diese in der Manifest-Datei eingetragen werden. Hierbei wird immer über ein Intent vermerkt, welche Activity die **Haupt-Activity** ist. Hierbei handelt es sich um die Activity, welche beim Start der App erscheint.

## 5.2 Start von (Sub-)Activities über Intents

Wie wir im obigen Manifest-Auszug gesehen haben, können Intents fest in der Manifest-Datei eingetragen werden. Weitauß häufiger werden sie jedoch direkt im Java-Code einer Activity erzeugt und verwendet. Im Zusammenhang mit Sub-Activities werden Intents benutzt für ...

- ... den Start einer Sub-Activity,
- ... das Mitgeben von Werten beim Start einer Sub-Activity über Bundles, sowie
- ... für das Beenden einer Activity, die einen Ergebniswert zurückliefern soll.

### 5.2.1 Start von Sub-Activities ohne Ergebnisauswertung

Der Start einer einfachen Sub-Activity über ein Intent funktioniert im Java-Code der aktuell angezeigten Activity folgendermaßen:

#### **Codefragment aktuellen Activity**

```
1 public class AktuelleActivity
2     extends Activity {
3         // ...
4         // ... in irgendeiner Methode:
5         Intent intent =
6             new Intent(AktuelleActivity.this,
7                         NeueSubActivity.class);
8
9         this.startActivity(intent);
10        // ... Rest der Methode ...
11        // ... weitere Methoden ...
12    } // end class AktuelleActivity
```

Listing 5.1: Start einer Sub-Activity

In diesem einfachen Fall wird das Ergebnis der Sub-Activity noch nicht weiter ausgewertet. Das Intent-Objekt bekommt über seinen Konstruktor:

- Die Referenz des Objekts der aktuell angezeigten Activity und
- Eine Referenz der Klasseninformation der neu zu startenden Sub-Activity.

Danach ruft die aktuell angezeigte Activity ihre ererbte Methode *startActivity()* auf. Sie bekommt das Intent-Objekt übergeben.

**Codefragment der neu gestarteten Sub-Activity**

Sie ist programmiert wie jede andere Activity auch. Wenn sie ihre Arbeit beendet hat, muss sie jedoch ihre ererbte *finish()*-Methode aufrufen. Ein Code-Fragment dazu:

```
1 public class NeueSubActivity
2     extends Activity {
3         // ... weitere Methoden ...
4         // ... Code einer Reaktionsmethode:
5         void reagiereAufEndeButton(View button){
6             // Code der vor Beendigung zu erledigen
7             // ist ...
8             this.finish();
9         } // end method reagiereAufEndeButton
10 } // end class NeueSubActivity
```

Listing 5.2: Beenden einer Activity

### 5.2.2 Start von Sub-Activities mit Ergebnisauswertung

Häufig wollen wir nachfragen, ob eine Sub-Activity mit Erfolg abgeschlossen wurde. Der Start der Sub-Activity erfolgt in diesem Falle mit der ererbten Methode `startActivityForResult()`. Sie bekommt außer dem Start-Intent noch einen ID-Code übergeben, der aussagt, welche Art von Activity gestartet wird. Häufig ist dieser Code `public static` in der Sub-Activity-Klasse hinterlegt.

#### **Codefragment der aktuellen Activity**

```
1 public class AktuelleActivity
2     extends Activity {
3     // ...
4     // ... in irgendeiner Methode:
5     Intent intent =
6         new Intent(AktuelleActivity.this,
7                 NeueSubActivity.class);
8     this.startActivityForResult(intent,
9         NeueSubActivity.SELBST_DEF_REQUESTCODE);
10    // ... Rest der Methode ...
```

Listing 5.3: Start einer Sub-Activity mit Ergebnisauswertung

Wir befinden uns weiterhin in der Klasse der übergeordneten Activity. In ihr überschreiben wir nun die ererbte Methode `onActivityResult()`. Sie wertet aus, mit welchem Ergebnis die Sub-Activity beendet wurde:

```
11 // Wird aufgerufen, sobald die Sub-Activity
12 // beendet wird
13 protected void onActivityResult(
14     int requestCode,
15     int resultCode,
16     Intent data) {
17
18     // ererbte Methode aufrufen
19     super.onActivityResult(requestCode,
20                           resultCode, data);
21
22
23     // Wenn Result von NeueSubActivity kommt ...
24     if(requestCode ==
25         NeueSubActivity.SELBST_DEF_REQUESTCODE) {
26
27         // Wenn die gewünschte Activity vor ihrem
28         // finish() nicht das korrekte Result
29         // geliefert hat ...
30         if(resultCode != Activity.RESULT_OK) {
31             // Fehlerbehandlung ...
32         } // end if resultCode ...
33     } // end if requestCode ...
34 } // end method onActivityResult
35
36 } // end class AktuelleActivity
```

Listing 5.4: Ergebnisauswertung in der übergeordneten Activity

***Codefragment der neu gestarteten Sub-Activity***

```
1 public class NeueSubActivity
2     extends Activity {
3         // ... weitere Methoden ...
4         // ... Code einer Reaktionsmethode:
5         void reagiereAufEndeButton(View button){
6             // Code der vor Beendigung zu erledigen
7             // ist ...
8
8             // Intent fuer Activity beenden
9             Intent intent = new Intent();
10
11
12             // ererbte Methode zum Setzen des
13             // Activity-Ergebnisses aufrufen.
14             // Bei Misserfolg/Fehlersituation
15             // wuerden wir den Wert
16             // Activity.RESULT_CANCELED verwenden.
17             this.setResult(
18                 Activity.RESULT_OK, intent);
19
20             // Sub-Activity beenden und Ergebnis
21             // zurueckliefern.
22             this.finish();
23         } // end method reagiereAufEndeButton
24 } // end class NeueSubActivity
```

Listing 5.5: Sub-Activity generiert das Ergebnis für die übergeordnete Activity

### 5.2.3 Kommunikation zwischen Activities über Intents und Bundles

Im vorherigen Kapitel hat die übergeordnete Activity ausgewertet,

- welche Sub-Activity sich gerade beendet hat und ...
- ob diese sich erfolgreich beendet hat.

Die Sub-Activity hat zurückgemeldet,

- dass sie es war, die sich beendet hat, und
- ob sie sich erfolgreich beenden konnte.

Manchmal muss eine übergeordnete Activity aber der Sub-Activity noch weitere Informationen mitgeben, wie z. B. vom Nutzer eingegebene Werte. Ebenso muss die Sub-Activity häufig derartige Werte an ihren Aufrufer, die übergeordnete Activity, zurückmelden.

Damit eine übergeordnete Activity einer Sub-Activity Werte mitgeben kann, muss sie vor dem Start der Sub-Activity dem Intent diese Informationen mitgeben:

```
1 public class ParentActivity extends Activity {  
2     // wird in einer der Reaktionen  
3     // belegt und geht an die Sub-Activity  
4     private String messageToChild="are you ok?";  
5  
6     // kommt von der Sub-Activity zurück  
7     private String replyFromChild;  
8  
9     // onCreate() wird nicht gezeigt ...  
10    // onActivityResult() wird nicht gezeigt ...  
11  
12    // onClick()-Reaktionmethode z.B.  
13    // für Button  
14    public void ok(View button) {  
15        Intent intent =  
16            new Intent(ParentActivity.this,  
17                SubActivity.class);  
18  
19        intent.putExtra(  
20            "AdditionalInfo", messageToChild);  
21        this.startActivityForResult(  
22            intent, ColorActivity.COL_ACTIVITY);  
23    } // end method ok()  
24} // end class
```

Listing 5.6: Übergeordnete Activity gibt der Sub-Activity in ein Bundle gekapselte Informationen mit

Die Klasse *Intent* bietet hier verschiedene *putExtra()*-Methoden an, die alle nach dem folgenden Schema aufgebaut sind:

```
1 void putExtra(String key, DataType infoData);
```

Die Sub-Activity wertet diese Informationen in ihrer `onCreate()`-Methode aus. Sie benutzt hierzu ein Objekt der Klasse `Bundle`, welches die durchgereichten Informationen kapselt:

```
1 public class SubActivity extends Activity {
2
3     // Dieser String bekommt die Infos
4     // der uebergeordneten Activity
5     private String messageFromParent;
6
7     // Dieser String bekommt spaeter
8     // Infos, die an die uebergeordnete
9     // Activity zurueck geliefert werden
10    private String replyToParent="pretty fine!";
11
12    @Override
13    protected void onCreate(
14        Bundle savedInstanceState) {
15        super.onCreate(savedInstanceState);
16        setContentView(R.layout.activity_sub);
17        Bundle b = this.getIntent().getExtras();
18        messageFromParent =
19            b.getString("AdditionalInfo");
20        // do sth. with messageFromParent ...
21    } // end method onCreate()
22    // weitere Methoden ...
23
24 }
} // end of class
```

Listing 5.8: Sub-Activity wertet Bundle-Informationen der übergeordneten Activity aus

Wenn die Sub-Activity ihrer übergeordneten Activity neben dem Result zusätzliche Informationen mitgeben will, so geschieht das ähnlich, wie bei der übergeordneten Activity über ein Intent:

```
1 public class SubActivity extends Activity {
2
3     // Dieser String bekommt die Infos
4     // der uebergeordneten Activity
5     private String messageFromParent;
6
7     // Dieser String bekommt spaeter
8     // Infos, die an die uebergeordnete
9     // Activity zurueck geliefert werden
10    private String replyToParent="pretty fine!";
11
12    // onCreate() wird nicht mehr gezeigt ...
13
14    // Reaktionsmethode fuer Button Bye
15    public void bye(View button) {
16        Intent rueckmeldung = new Intent();
17        rueckmeldung.putExtra(
18            "ReplyFromChild", replyToParent);
19        this.setResult(
20            Activity.RESULT_OK, rueckmeldung);
21        this.finish();
22    }
23
24 } // end class
```

Listing 5.9: Sub-Activity gibt ihrer übergeordneten Activity Zusatz-Informationen über ein Intent mit

Die übergeordnete Activity schließlich wertet diese zurück gelieferte Information wieder über ein Bundle in ihrer *onActivityResult*-Methode aus:

```
1 public class ParentActivity extends Activity {
2     // wird in einer der Reaktionsmethoden
3     // belegt und geht an die Sub-Activity
4     private String messageToChild="are you ok?";
5
6     // kommt von der Sub-Activity zurueck
7     private String replyFromChild;
8
9     // onCreate() wird nicht gezeigt ...
10    // ok() wird nicht mehr gezeigt ...
11    @Override
12    protected void onActivityResult(
13        int requestCode,
14        int resultCode,
15        Intent data) {
16        // ererbte Methode aufrufen
17        super.onActivityResult(
18            requestCode, resultCode, data);
19
20        // result auswerten wie vorher ...
21
22        // uebergebener Intent enthaelt
23        // die Zusatzinfos der Sub-Activity
24        Bundle resultInfos=data.getExtras();
25        replyFromChild=
26            result.getString("ReplyFromChild");
27        // do sth. with replyFromChild ...
28
29    } // end method onActivityResult()
30 } // end class
```

Listing 5.10: Übergeordnete Activity werdet Bundle der Sub-Activity aus

## 5.3 Dateizugriff von Activities

Wir haben bislang gelernt: Dateizugriffe und Datenbankzugriffe werden in der GUI-Programmierung soweit wie möglich von der GUI-Implementierung getrennt. Dies gilt auch weiterhin. Allerdings gibt es für Android-Apps einige Regeln dahingehend, **wo** Daten auf dem Android-Dateisystem abgelegt werden.

### 5.3.1 Verzeichnisstruktur für Android-Apps

Anbei einige der wichtigsten Verzeichnisse auf Android-Systemen (nicht auf jedem Gerät gleich gemappt):

Wichtige Verzeichnisse im Android-Dateisystem	
Verzeichnis-Name	Erklärung
/data	Datenverzeichnis für Apps
/data/backup	Backup-Verzeichnis
/data/data/package.dieser.app	Lokales Datenverzeichnis für Apps, welche Daten permanent speichern.
/sdcard	Externer Speicher
/sdcard/DCIM	Verzeichnis für Kamerabilder
/sdcard/Pictures/Screenshots	Verzeichnis für Screenshots

Tabelle 5.1: Verzeichnisse im Android-Dateisystem

Der externe Speicher kann dabei durchaus fest verbaut sein. Er wird häufig auf das Verzeichnis /storage/emulated/0 abgebildet

### 5.3.2 Verzeichnis- und Datei-Zugriffsmethoden für Android-Apps

Diese Verzeichnisse werden i.d.R **nicht** fest im Java-Quellcode „verdrahtet“. Eine Android- Activity kann jedoch Abfragemethoden der Klasse *Environment* und eigene, ererbte Methoden nutzen, mit denen sie zur Laufzeit erfragt werden können:

**Abfragemethoden der Klasse *Environment* für systemweit bekannte Verzeichnisse**

Return-Wert	Methodename
public File	getDataDirectory() liefert das Data-Verzeichnis des Gerätes (meist /data/) als File-Objekt.
public File	getExternalStorageDirectory() liefert das externe Daten-Verzeichnis des Gerätes (meist /sdcard/ oder /sdcard0/ ) als File-Objekt.

Tabelle 5.2: Wichtigste Abfragemethoden der Klasse *Environment* für systemweit bekannte Verzeichnisse

**Abfragemethoden der Klasse *Activity* für systemweit bekannte Verzeichnisse**

---

Return-Wert	Methodename
public    FileInputStream	openFileInput(String fname) öffnet im Verzeichnis /data/data/package.dieser.app einen lesenden Stream für die Datei mit dem Namen fname
public    FileOutputStream	openFileOutput(String fname) öffnet im Verzeichnis /data/data/package.dieser.app einen schreibenden Stream für die Datei mit dem Namen fname
public    File	getFilesDir() liefert das lokale App-Datenverzeichnis (meist /data/data/package.dieser.app) zurück.
public    File	getFileStreamPath(String fname) liefert das lokale App-Datenverzeichnis (meist /data/data/package.dieser.app) für die Datei fname zurück.

---

Tabelle 5.3: Abfragemethoden der Klasse *Activity* für systemweit bekannte Verzeichnisse

### 5.3.3 Dateizugriffs-Berechtigungen in Manifest und Activity verankern

Wir verweisen hierzu auch auf Abschnitt ?? im Einführungskapitel.

Dateizugriffs-Berechtigungen müssen – wie alle benötigten Berechtigungen – im Manifest eingetragen werden. Im Falle von Lese-/Schreib-Berechtigungen auf den externen Speicher handelt es sich um die Permissions

- `android.permission.READ_EXTERNAL_STORAGE` und
- `android.permission.WRITE_EXTERNAL_STORAGE`.

Wichtig ist dabei, dass diese Privilegien **VOR** der eigentlichen App-Definition angegeben werden.

Daneben müssen diese Berechtigungen ab Android Version 6 (API-Level 23) in der Activity, die sie benötigt, abgefragt werden. Hierzu verweisen wir auf das Beispiel in Abschnitt ??.