

## Inhaltsverzeichnis

1	Allgemeines .....	2
2	Das Layout Grundproblem .....	2
3	Alternative Layoutvorgaben .....	3
4	XML als Layoutdefinition .....	4
4.1	Allgemeine XML Layoutangaben .....	4
4.1.1	Padding / Margin .....	5
4.1.2	Height / Width .....	6
4.1.3	Gravity .....	6
4.2	View Elemente .....	7
4.2.1	TextView .....	7
4.2.2	EditText .....	8
4.2.3	ImageView .....	9
4.2.4	Button .....	10
4.2.5	Selector .....	11
4.3	Layouts .....	12
4.3.1	FrameLayout.....	12
4.3.2	LinearLayout .....	13
4.3.3	RelativeLayout.....	15
4.3.4	TableLayout.....	17
4.3.5	GridLayout .....	20
4.3.6	Nested Layouts .....	23
4.3.7	Scrollbars.....	25
5	Layout mit Java .....	26
5.1	Die id als eindeutige Adresse .....	26
5.2	Identifikation und Anpassung der Elemente.....	26
5.3	Erzeugung eines View-Objektes in Java .....	27
5.4	Layout Inflater – der elegante Weg .....	29
6	Lizenz .....	32
7	Haftung.....	32

## 1 Allgemeines

Dieses Dokument behandelt den Umgang mit Layouts. Hierbei werden die grundlegenden Elemente besprochen, sowie einfache Anwendungen realisiert. Am Ende dieses Kursteils sollten Sie in der Lage sein, einfache Applikationen selbstständig zu realisieren. Hierzu gibt es auch eine entsprechende Aufgabenstellung im Dokument „[AnPr\\_Android\\_Course\\_1\\_3\\_Task](#)“. Fragments, Adapter Layout und das Zoomen ist nicht Teil dieser Stufe, diese Elemente kommt bei Stufe 2 (grün) zur Sprache.

Voraussetzung für diesen Kursteil ist, dass Sie den Vorgängerkursteil ([AnPr\\_Android\\_Course\\_1\\_1\\_Setup](#)) durchgearbeitet und verstanden haben.

Zu jedem Kapitel finden Sie unter YouTube ein Lehrvideo, welches die Inhalte darstellt. Details hierzu entnehmen Sie bitte meiner Webpage ([www.codeconcert.de](http://www.codeconcert.de)). Wenn für die Beispiele Grafiken o.Ä. verwendet werden, finden Sie diese Ressourcen ebenfalls in meiner Homepage (für diese Stufe ist dies [Dice1.zip](#)).

Grundsätzlich gilt zu sagen, dass dieses Dokument nicht alles abdecken kann, was Android an Optionen bietet. Vielmehr versuche ich mich auf die Themen zu konzentrieren, welche den Anfang Ihrer Android Karriere möglichst einfach gestalten. Auch möchte ich darauf hinweisen, dass ich trotzdem versucht habe die Layoutinfos recht umfangreich zu beschreiben. Dies liegt daran, dass Sie die Layouts nur dann sinnvoll nutzen können, wenn Sie die einzelnen Themen auch wirklich verstanden haben.

## 2 Das Layout Grundproblem

Im Gegensatz zu Desktop Applikationen werden die Bildschirme von mobilen Geräten meist immer voll ausgenutzt. Dadurch hängt die darstellbare App – Fläche sehr stark vom genutzten Gerät ab. Da wir aber verständlicherweise versuchen, den Nutzerkreis unserer Programme möglichst breit zu halten, sollten wir versuchen, unsere Programme möglichst flexibel zu gestalten. Android verfolgt hier das Konzept, die Logikimplementierung von den Layout Informationen zu trennen, wobei diese zusätzlich noch für verschiedene Bildschirmssituationen unterschiedlich vorgehalten werden können.



So kann bspw. eine Applikation sowohl für ein kleines Device, als auch für ein großes Device optimiert werden, ohne den Logikcode anpassen zu müssen. Weiterhin gibt es noch die Möglichkeit, dass vertikale und horizontale Ausrichtungen unterschieden werden müssen. Schließlich gibt es noch Tablettegeräte, bei denen bspw. sequenzielle Abläufe parallelisiert werden können. Bspw. können Menüdialoge bei einem Handheld Gerät nur über den gesamten Bildschirm aufgebaut werden, bei Tablets ist der Platz ausreichend groß, um dieses Menü neben dem Hauptfenster zu platzieren.

Wie wir aus dem vorigen Kapitel „Setup“ gelernt haben, gibt es derzeit vier verschiedene Bildschirmkategorien, small, medium, large und extra-large. Für jede dieser Kategorien kann man theoretisch eine Layoutvorgabe hinterlegen. Diese Details werden aber in den folgenden Kapiteln noch näher beleuchtet.

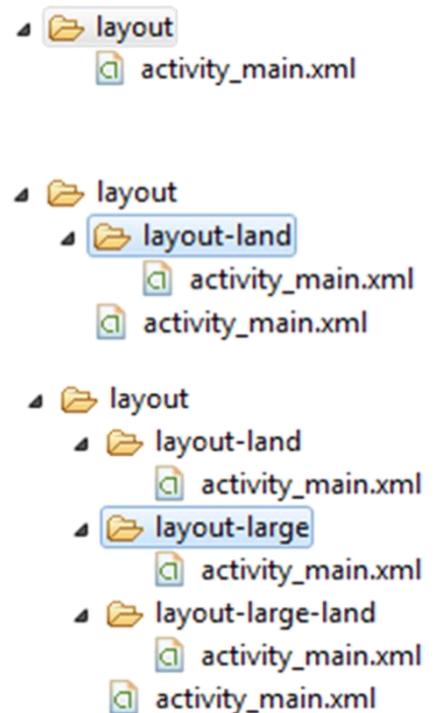
### 3 Alternative Layoutvorgaben

Die Basis ist immer auf die Medium Density Kategorie ausgelegt, bei vertikaler Gerätehaltung (also Portrait). Sollte keine spezifischere Layoutvorgabe gefunden werden, so wird immer diese genommen. Sie befindet sich im Rootordner **layout**.

Als nächstes können wir die Layoutvorgaben für die horizontale Gerätehaltung (Landscape) eintragen. Hierzu wird ein Unterordner angelegt, welcher **layout-land** genannt wird. „land“ ist der sogenannte „Qualifier“ für die horizontale Ausrichtung. Auch dort wird nun ein entsprechendes **activity\_main.xml** File abgelegt, jedoch mit entsprechenden geänderten Vorgaben.

Sollten nun für bspw. eine höhere Auflösung andere Einstellungen vonnöten sein (sagen wir mal „large“), indem wir den Qualifier für große Bildschirmgrößen anhängen. Wenn wir für diese Bildschirmgrößen noch zusätzlich die horizontalen Layoutvorgaben machen wollen, dann ergänzen wir den Ordner layout-large mit dem Qualifier „land“.

Folgende Qualifier existieren<sup>1</sup>:



Bildschirm-Charakteristik	Qualifier	Beschreibung – Ressourcen für:
Größe	small	Kleine Bildschirmgrößen (ca. 426dp x 320dp)
	normal	Normale Bildschirmgrößen (mind. 470dp x 320dp)
	large	Große Bildschirmgrößen (mind. 640dp x 480dp)
	xlarge	Sehr große Bildschirmgrößen (mind. 960dp x 720dp)
Pixeldichte	ldpi	Low density Bildschirme (~120dpi)
	mdpi	Medium density Bildschirme (~120dpi)
	hdpi	High density Bildschirme (~120dpi)
	xhdpi	Extra high density Bildschirme (~120dpi)
	nodpi	Alle Densities. Dies sind von der Pixeldichte unabhängige Ressourcen. Das System wird diese Ressourcen nicht skalieren – unabhängig von der aktuellen Bildschirm Pixeldichte.
	tvdpi	Bildschirme irgendwo zwischen mdpi und hdpi (ca. 213dpi). Dies wird nicht als primäre density Gruppe angesehen. Sie ist hauptsächlich für Fernsehgeräte vorgesehen – die meisten Apps werden diese Gruppe somit nicht benötigen. Mdpi und hdpi sollte ausreichend sein. Wenn Sie tvdpi Ressourcen vorhalten wollen, sollten Sie sie auf etwa das 1,33 fache von mdpi dimensionieren. Bspw. sollte ein 100px x 100px Bild für mdpi Bildschirme bei tvdpi auf 133px x 133px skaliert werden.
Orientierung	land	Bildschirme in horizontaler Ausrichtung (breites Seitenverhältnis).
	port	Bildschirme in vertikaler Ausrichtung (hohes Seitenverhältnis).
	long	Bildschirme, welche ein bedeutend höheres oder breiteres Seitenverhältnis aufweisen (entsprechend vertikaler bzw. horizontaler Ausrichtung) als die standard Konfiguration.
	notlong	Bildschirme, die ein eher „gestauchtes“ Seitenverhältnis aufweisen (bspw. 4:3).

<sup>1</sup> (Quelle: [http://developer.android.com/guide/practices/screens\\_support.html](http://developer.android.com/guide/practices/screens_support.html))

Darüber hinaus gibt es seit Android 3.2 eine neue Gruppe von Qualifiern, welche eine explizite Größenangabe (in dp) bezüglich des Bildschirms erlaubt. Bspw. erlaubt der Qualifier sw600dp eine Angabe, dass der Bildschirm mindestens 600dp breit sein muss, egal, ob er horizontal oder vertikal gehalten wird (wobei die Zahl 600 von Ihnen als Entwickler definiert wird). Ein Wert von w1024dp steht für eine Bildschirmbreite von mindestens 1024 dp und entsprechend h1024dp für eine Mindesthöhe von 1024 dp. Details hierzu siehe: [http://developer.android.com/guide/practices/screens\\_support.html#NewQualifiers](http://developer.android.com/guide/practices/screens_support.html#NewQualifiers)

Basierend auf diesen Angaben versucht Android, die am besten passende Konfiguration auszuwählen. Unter folgendem Link wird der Algorithmus für Interessierte genauer erklärt:

<http://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch>

Weiterhin sein noch erwähnt, dass Sie im Manifest.xml festlegen können, ob das Layout beim Schwenken der Geräteorientierung sich nicht ändert. Hierfür gibt es das Attribut screenOrientation:

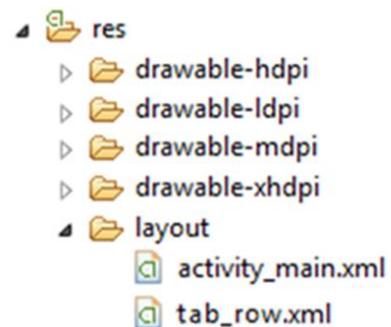
```
<activity
    android:name="com.aicher.dicer.MainActivity"
    android:label="@string/app_name"
    android:screenOrientation="portrait" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Die hier gezeigte Einstellung belässt die Orientierung auf „portrait“, also Hochkant. Mit dem Wert „landscape“ würde die Darstellung immer im Querformat erfolgen. Weitere Werte zum screenOrientation Attribut finden Sie unter:

<http://developer.android.com/reference/android/R.attr.html#screenOrientation>

## 4 XML als Layoutdefinition

Die Designer von Android haben XML als Beschreibungssprache der Layouts gewählt. Dies wurde wohl aufgrund der großen Akzeptanz und Verbreitung von XML so gewählt und auch, weil XML einfach parsebar und trotzdem von Menschen lesbar ist. Layoutinfos in XML Form finden sich im Regelfall im **layout** Ordner, oder einem Unterordner (es gibt zwar ein paar Ausnahmen, davon aber später).



Hier finden wir vor allem die Layoutinfos für die einzelnen Activities. Aber auch andere Layoutdaten können hier abgelegt werden. Bspw. können Elemente dynamisch im Java Code erzeugt werden. Jede Layoutangabe kann somit über geeignete Settermethoden gesetzt werden.

Da dies aber sehr mühsam ist, können auch Layoutfragmente in ein XML File eingetragen werden (bspw. einer Tabellenzeile) und diese dann anschließend mit einem sog. Layout Inflater instanziiert werden, je nachdem wie viele Tabellenzeilen man im Programm braucht. Auf diese Technik wird später nochmal eingegangen.

Innerhalb der XML Files werden die Layoutangaben als Attribute der XML Tags festgelegt. Hier gilt aber zu beachten, dass nicht jedes Element jede Layoutangabe unterstützt. Dies wird in den folgenden Kapiteln nun näher beschrieben.

### 4.1 Allgemeine XML Layoutangaben

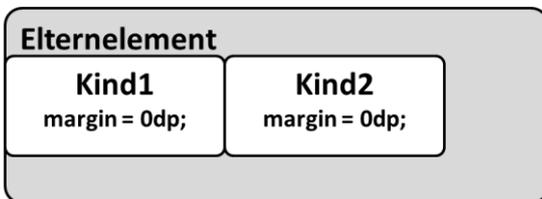
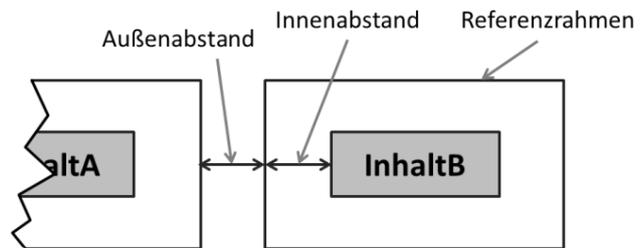
Ähnlich, wie man es bei HTML kennt, werden in Android Abstands- und Größenangaben gemacht, nur hier werden alle Daten in einem XML File abgelegt, welches einem eigenen Android Namespace gehorcht. Jede Activity besitzt ein zugrundeliegendes Layout, welches das Wurzelement des XML Files darstellt. Innerhalb dieses Wurzelements können nun weitere Layouts, oder View Elemente eingefügt werden. Die Layout Elemente werden in der Android Literatur auch als „ViewGroup“ bezeichnet.

ViewGroups können beliebig tief verschachtelt werden, doch wird davor gewarnt, dass eine zu tiefe Verschachtelung Performanceeinbußen mit sich bringen kann. Insofern sollte man immer versuchen, den Verschachtelungsbaum möglichst flach zu halten und die vorhandenen Android Layoutformate optimal einzusetzen.

Die einzelnen Elemente können nun wiederum per XML Attribute vom Layoutverhalten her angepasst werden. Ein einfaches Beispiel ist die Breiten- und Höhenangabe. Im Folgenden werden nun die wesentlichen Punkte durchgearbeitet, welche für Layouts von Interesse sind.

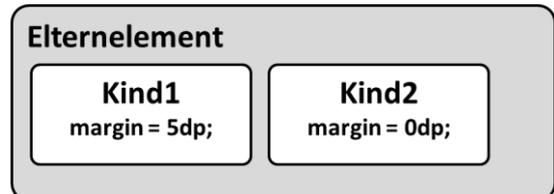
### 4.1.1 Padding / Margin

Wie HTML verfolgt Android auch ein sogenanntes Box Modell. Hierbei werden die Inhalte in einer Box gesehen und die Innen- bzw. Außenabstände als Padding und Margin bezeichnet. Bei der Nutzung dieser beiden Eigenschaften ist es wichtig den Unterschied zu kennen und das Verhalten vorherzusehen (und nicht etwa nach dem „Try and Error“ Verfahren vorzugehen...).

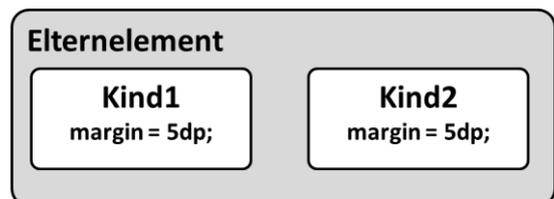


Zuerst wollen wir uns das Verhalten von „margin“ ansehen. Margin ist der Außenabstand eines Elements. Bevor wir uns mit dem Verhalten der Eigenschaft beschäftigen, werfen wir jedoch einen Blick auf die Ausgangssituation. Wir setzen den Margin Wert beider Elemente auf 0, so dass sie sich berühren. Nun setzen wir bei dem ersten Kindelement die Margin –

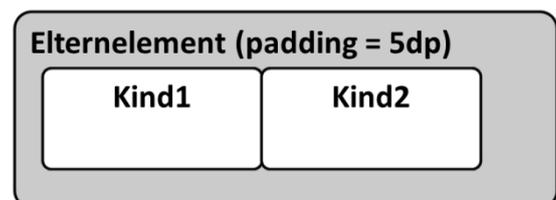
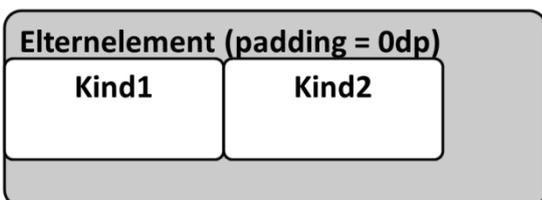
also den Außenabstand auf 5 dp (wir erinnern uns – dp ist ein density independent pixel). Wie wir sehen, wandert das Kind1 wegen der Abstandsangabe nach unten und nach rechts. Aber auch das Kind2 wandert nach rechts, da der Abstand natürlich auch auf der rechten Seite von Kind1 wirkt.



Wenn wir nun Kind2 auch auf 5dp setzen, so bewegen sich die beiden Kindelemente auf 10dp auseinander, da sie ja nun beide Abstände halten müssen.



Bei Padding (also Innenabstand) ist nun der Wert des Elternelements von Bedeutung. Wenn beim Elternelement der Paddingwert auf 0dp gesetzt wird, liegen die beiden Kindelemente (unter der Voraussetzung, dass die Kind Marginwerte ebenfalls auf 0dp liegen) direkt am Rand des Elternelements. Wird nun der Wert auf 5dp geändert, so wandern die beiden Kindelemente entsprechend 5dp vom Rand weg, berühren sich jedoch gegenseitig noch.



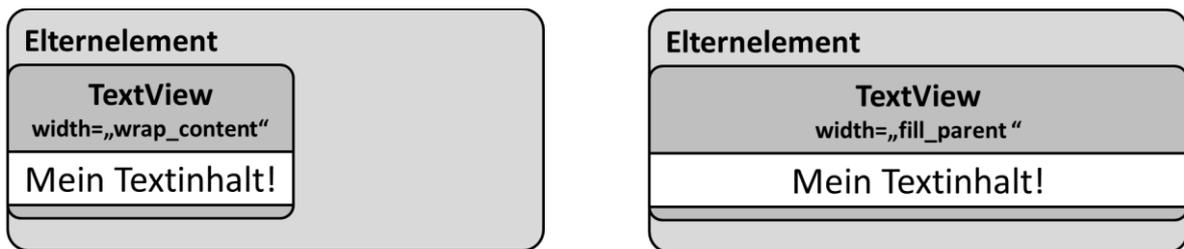
Die Angaben der Marginwerte erfolgen in px (pixels), dp (density-independent pixels), sp (scaled pixels basierend auf der Schriftgröße), in (Zoll) und in mm (Millimeter). Es wird gemeinhin empfohlen, dp zu verwenden.

Hier nochmal eine Aufstellung der möglichen layout\_margin Werte:

Eigenschaft:	Bedeutung:
android:layout_margin	Außenabstand nach allen 4 Richtungen.
android:layout_marginTop	Außenabstand nach oben.
android:layout_marginBottom	Außenabstand nach unten.
android:layout_marginLeft	Außenabstand nach links.
android:layout_marginRight	Außenabstand nach rechts.
android:padding	Innenabstand nach allen 4 Richtungen.
android:paddingTop	Innenabstand nach oben.
android:paddingBottom	Innenabstand nach unten.
android:paddingLeft	Innenabstand nach links.
android:paddingRight	Innenabstand nach rechts.

#### 4.1.2 Height / Width

Die Höhen und Breitenangaben können, wie auch die Margin/Padding Angaben in absoluten Größenangaben erfolgen. Jedoch wird hiervon abgeraten. Vielmehr sollte man die beiden Konstanten „match\_parent“ und „wrap\_content“ verwenden. „match\_parent“ dient dazu, die Größe des Elements auf die Größe des Elternelements auszuweiten. „wrap\_content“ bewirkt, dass die Größe des Elements auf den Inhalt (bspw. dem angezeigten Text) reduziert wird.



Eine weitere wichtige Eigenschaft im Zusammenhang mit der Breiten- bzw. Höhenangabe ist die „weight“ Eigenschaft. Diese greift vor allem beim Linear Layout und wird dort näher behandelt. Soviel sei hier gesagt – wenn mit dem „weight“ Attribut gearbeitet wird, sollte die „width“ Eigenschaft auf „0dp“ gesetzt werden.

Eigenschaft:	Bedeutung:
android:layout_width	Breite des Views.
android:layout_height	Höhe des Views.

Wichtig zu wissen ist, dass die beiden Werte zwingend für jedes Element anzugeben sind – Eclipse würde ansonsten bereits im Designprozess einen Fehler melden.

#### 4.1.3 Gravity

Gravity sorgt bei vielen für die meiste Frustration. Grundsätzlich gilt, dass die Gravity Eigenschaft nicht für das RelativeLayout gedacht ist. Wenn also die Gravity Eigenschaften ausprobiert werden, immer LinearLayout oder FrameLayout verwenden, wobei layout\_gravity auch bei GridLayout einzusetzen ist.

Für die Gravity gibt es zwei verschiedene Attribute:

**gravity:** Angabe im Elternelement, wie seine Kindelemente innerhalb des Elternelements ausgerichtet werden sollen.

**layout\_gravity:** Angabe im Kindelement, wie es innerhalb seines Elternelements ausgerichtet werden soll.

Die möglichen Werte sind für beide Attribute identisch. Hier die wichtigsten:

Eigenschaft:	Bedeutung:
top	Ausrichtung oben.
bottom	Ausrichtung unten.
left	Ausrichtung links.
right	Ausrichtung rechts.
center_vertical	Ausrichtung mittig in vertikaler Richtung.
fill_vertical	Objekt wächst bis zur umgebenden Containergröße in vertikaler Richtung.
center_horizontal	Ausrichtung mittig in horizontaler Richtung.
fill_horizontal	Objekt wächst bis zur umgebenden Containergröße in horizontaler Richtung.
center	Zentrierung horizontal und vertikal.
fill	Objekt wächst bis zur umgebenden Containergröße.

Wenn Sie nun zwei verschiedene Layoutsettings machen möchten, dann können Sie diese auch per ODER Verknüpfung verbinden. Bspw. wird folgendes Attribut:

```
android:layout_gravity="top|center_horizontal"
```

das Element sowohl oben, als auch horizontal zentriert ausrichten.

## 4.2 View Elemente

Als nächsten Punkt sehen wir uns an, welche Elemente (oder besser Views) wir in der Activity positionieren können. Hierbei werde ich mich auf die elementarsten Views konzentrieren. Für weitere Views sollten Sie entweder die Android Guide konsultieren (<http://developer.android.com/guide/topics/ui/index.html>) oder einfach bei Eclipse ein wenig durch die Toolbox stöbern.

### 4.2.1 TextView

Der auf den ersten Blick denkbar einfachste View entpuppt sich bei näherem Hinsehen als durchaus vielseitig. Doch eins nach dem anderen... Zuallererst ist der TextView für das Anzeigen von einfachem Text gedacht:

```
<TextView
android:id="@+id/textView1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/hello_world" />
```

Beachten Sie bitte, dass die `layout_width` und `layout_height` Angaben auf jeden Fall gemacht werden müssen. Optional ist die `id`, welche aber immer empfohlen wird, da wir den TextView im Java Code im Regelfall wiederfinden wollen. Der eigentliche Text kann als statischer String eingegeben werden, jedoch ist auch hier dringend anzuraten, über die **strings.xml** Datei die Werte festzulegen.

Nun können noch diverse Stilmittel eingesetzt werden. Hierzu zählen die Textgröße, welche entweder über `textAppearance` oder über `textSize` angegeben wird, Schriftfarbe und die Hintergrundfarbe:

```
android:textAppearance="?android:attr/textAppearanceMedium"
android:textColor="#0000ff"
android:background="#ffff00"
```

Die Schriftart wird über das Attribut `typeface` gesetzt. Da das Gesamtlayout über Themes gesetzt wird (bei uns war dies in der ersten App „Holo Light with dark Action Bar“) werden die Schriftarten nicht direkt eingestellt, sondern als Kategorie. Folgende sollten genutzt werden:

typeface:	Bedeutung:
DEFAULT	Standardschriftart, wie es im System festgelegt wurde.
DEFAULT_BOLD	Standardschriftart in fett, wie es im System festgelegt wurde.
MONOSPACE	Schriftart mit festen Abständen.
SANS_SERIF	Schriftart ohne Serifen.
SERIF	Schriftart mit Serifen.

Es ist auch möglich, im Rahmen des Textviews Grafiken (im Android Jargon „drawables“) einzufügen. Hierzu wird das Attribut „drawable???“, wobei ??? für Top, Bottom, Left oder Right steht, je nachdem wo das Bild eingefügt werden soll. Der Abstand zwischen dem Bild und dem Text wird über „drawablePadding“ gesetzt. Um den Text vertikal mit dem Bild zu zentrieren, wird über das gravity Attribut das Element zentriert. Wie man mit „drawables“ umgeht können Sie im Kapitel „ImageView“ nochmal vertiefen.

```
android:drawableLeft="@drawable/symb_player_a_1"
android:drawablePadding="10dp"
android:gravity="center"
```

Um nun innerhalb des Java Codes den Wert eines TextView Elements ändern zu können, müssen wir eine Referenz auf dieses Element ermitteln und die entsprechenden Methoden aufrufen. Der folgende Code verändert den angezeigten Wert eines Elements:

```
TextView myTextView = (TextView) findViewById(R.id.textView1);
myTextView.setText("neuer Text");
```

Der Code ermittelt zuerst den TextView, indem ein allgemeiner View über die ID gesucht wird und über einen Typecast zu einem TextView Objekt überführt wird. Danach sind die Methoden nutzbar – in diesem Beispiel wird der Text mit einer Stringkonstante neu gesetzt. In realen Apps sollte dies jedoch eine Stringkonstante aus **strings.xml** sein. Die API Beschreibungen unter: <http://developer.android.com/reference/android/widget/TextView.html> zeigen alle weiteren Methoden auf. Programmatisch kann der TextView noch erweitert werden, er kann als Texteditor fungieren etc. wobei diese Erweiterungen im EditText Element bereits umgesetzt wurden.

#### 4.2.2 EditText

Um Texte eingeben zu können, nutzen wir üblicherweise den EditText View. Dieser View stellt im Großen und Ganzen einen vollständigen Text Editor dar, der entsprechend der Layoutvorgaben konfiguriert werden kann.

```
<EditText
android:id="@+id/editText1"
android:layout_width="fill_parent"
android:layout_height="wrap_content"/>
```

Wie bei einem normalen TextView wird eine ID festgelegt und die Abmessungen vorgegeben. Wenn nun ein initialer Hinweistext vorgegeben werden soll, dann erfolgt dies mit dem „hint“ Attribut:

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

Hier wird eine Stringkonstante „editTextHint“ aus der **strings.xml** verwendet und eingetragen. Dieser Wert wird im Eingabefeld angezeigt und sobald der User eine Texteingabe macht, entfernt. Nachdem EditText eine Ableitung von TextView ist, können die Stilangaben (Schriftart, Schriftfarbe etc.) genauso vorgenommen werden, wie beim TextView.

Ein wichtiges Attribut des EditText Views ist der inputType. Hier werden sämtliche Verhaltensweisen des Elements festgelegt.

```
android:inputType="textNoSuggestions"
```

Hier die wichtigsten Werte, welche als Flag ein und ausgeschaltet werden (true, false):

<b>inputType:</b>	<b>Bedeutung:</b>
text	Standard Text.
textNoSuggestions	Automatische Textvorschlagfunktion deaktivieren.
textPassword	Versteckte Passworteingabe.
number	Es werden nur Nummern akzeptiert.
textMultiLine	Mehrere Zeilen können editiert werden.
textAutoCorrect	Autokorrektur

Eine vollständige Liste finden Sie unter:

<http://developer.android.com/reference/android/R.attr.html#inputType>

Kombinationen verschiedener Werte können mit einem ODER (also dem | Sybmol) verknüpft werden. Die Anzahl der Zeilen (sichtbar und erlaubt) wird über folgende Attribute gesetzt:

```
android:lines="4"
android:maxLines="8"
```

Weitere hilfreiche Attribute sind die Scrollbars:

```
android:scrollbars="vertical"
android:scrollHorizontally="true"
```

Die Einstellung scrollbars="vertical" bewirkt, dass vertikale Scrollbars angezeigt werden. Mit dem Wert „horizontal“ würde eine horizontale Scrollbar erzeugt und mit „horizontal|vertical“ entsprechend beide. scrollHorizontally="true" bewirkt, dass der Text über die Viewgrenzen hinaus editiert werden und mit horizontalem Scrollen hin- und hergeschoben werden kann.

Den Text greift man programmatisch wieder über das Objekt ab:

```
EditText myEditText = (EditText) findViewById(R.id.editText1);
String sValue = myEditText.getText().toString();
```

### 4.2.3 ImageView

Um Bilder in unser Layout zu bekommen, bedienen wir uns dem ImageView. Dieses ruft aus unserer Ressourcenbibliothek die sogenannten „drawables“ auf und fügt sie entsprechend der Layoutsettings (ldpi, mdpi, ldpi, xldpi) auf dem Screen ein.

Vom Aufbau her ist der ImageView mit der einfachste. Wie bei allen anderen Views müssen hier die Breite und Höhe gesetzt werden, wobei hier in aller Regel „wrap\_content“ eingesetzt wird.

```
<ImageView
android:id="@+id/imageView0"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:contentDescription="@string/desc_dice"
android:src="@drawable/dice_6" />
```

Das contentDescription Attribut sollte ein String mit einer kurzen Beschreibung des angezeigten Bildes beinhalten. Hintergrund ist, dass es dem Betriebssystem somit ermöglicht wird, sehbehinderten Usern einen Hinweis auf das gerade angezeigte Bild (vor allem bei icons) zu geben. Das src (Source) Attribut gibt lediglich an, welche drawable Ressource für das Bild verwendet werden soll.

Alle anderen Attribute sind weitestgehend selbsterklärend. Eine Besonderheit eines ImageViews ist noch zu nennen. ImageViews können clickable gesetzt (also anklickbar) und dieses Event programmatisch ausgewertet werden. Dies ist für Spiele sehr nützlich. Hierauf wird jedoch später erst eingegangen.

Um nun ein Bild zu verändern, muss man wieder das Objekt identifizieren:

```
ImageView myImageView = (ImageView) findViewById(R.id.imageView0);
String sValue = myImageView.setImageResource(R.drawable.dice_1);
```

Mit setImageResource wird das Bild neu gesetzt. Hier wird ebenfalls über die Ressourcen ID agiert.

*Hinweis: die Bilder finden Sie auf meiner Webseite [www.codeconcert.de](http://www.codeconcert.de) als **Dice1.zip**.*

#### 4.2.4 Button

Der einfache Button ist zwar für „state of the art“ Programme nicht mehr zeitgemäß, für kurze Prüfprogramme jedoch durchaus erwähnenswert.

```
<Button
  android:id="@+id/button1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:onClick="myClickMethod"
  android:text="@string/btnCapture" />
```

Wie bei allen anderen Views müssen die Breiten- und Höhenangaben gemacht werden. Eine ID ist ebenfalls sinnvoll. onClick erwartet die Angabe einer Methode, welche beim Klicken aufgerufen werden soll. Diese muss in der entsprechenden Javaklasse stehen und darf nur die View als Methodenparameter haben:

```
public void myClickMethod(View view) {
    // eigener Code
}
```

Der Methodename kann von Ihnen beliebig gewählt werden.

Es können auch Bilder eingebunden werden, wobei Sie hier padding auf 0dp setzen sollten:

```
<Button
  android:id="@+id/button1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:padding="0dp"
  android:onClick="myClickMethod"
  android:drawableTop="@drawable/dice_1"
/>
```

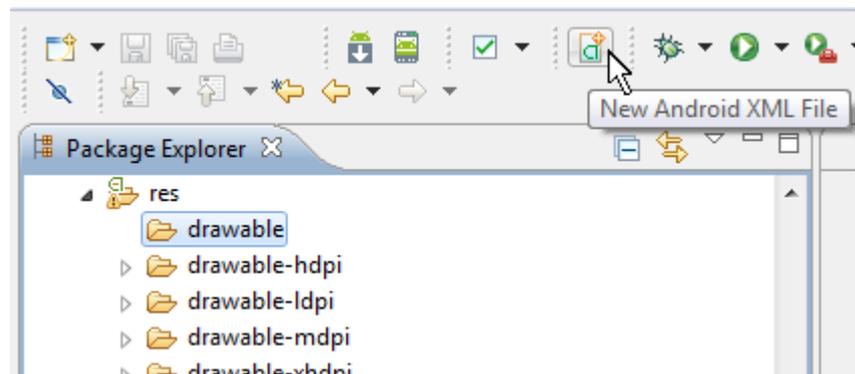
Das Image wird mit drawableTop eingebunden, wobei Sie auch die gleichen Möglichkeiten haben, wie beim TextView – vor allem wenn Sie ein Bild und parallel dazu noch die Beschriftung anzeigen lassen möchten.

Alternativ geht hier auch der ImageButton (siehe hierzu nächstes Kapitel).

## 4.2.5 Selector

Wenn Sie einen Button realisieren wollen, bei dem in den verschiedenen Situationen (gedrückt, inaktiv etc.) verschiedene Bilder angezeigt werden sollen, dann bieten sich zwei Möglichkeiten an. Entweder, Sie tauschen bei jeder Aktion das Bild über Java aus, oder sie bedienen sich eines Selectors, welcher aus Performance-sicht die bessere Lösung ist. Hier ein Beispiel:

Zuerst erstellen wir uns ein neues XML File im **res/drawable** Ordner (sollte der noch nicht existieren, müssen Sie ihn mit der rechten Maustaste und „Neu-> Ordner“ erstellen), um den Selector zu definieren, indem wir auf den Android XML File Button klicken. Wir benennen wir File als **my\_own\_button.xml** und wählen „selector“ aus.



Danach schließen wir den Wizzard durch einen Klick auf „Finish“.

Nun tragen wir für die drei verschiedenen Situationen „gedrückt“, „inaktiv“ und „normal“ die Bilder ein, welche den Button darstellen sollen (auf „focused“ verzichten wir an dieser Stelle):

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
  <item
    android:drawable="@drawable/btn_play_p"
    android:state_pressed="true"/>
  <item
    android:drawable="@drawable/btn_play_i"
    android:state_enabled="false"/>
  <item
    android:drawable="@drawable/btn_play_a"/>
</selector>
```

Wichtig ist hier, dass der allgemeinste Zustand (also der Normalzustand) ganz unten eingetragen wird, da Android von oben nach unten nach dem ersten Match die entsprechende Ressource verwenden wird.

*Hinweis: die Bilder finden Sie auf meiner Webseite [www.codeconcert.de](http://www.codeconcert.de) als **Dice1.zip**.*

Nun erstellen Sie einen Button (dies kann zwar ein normaler Button sein – da wir aber rein mit Bildern arbeiten nutzen wir hier einen ImageButton) und tragen anstatt einer normalen Bildressource, den von Ihnen gerade erstellten Selector ein:

```
<ImageButton
  android:id="@+id/button1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:onClick="myClickMethod"
  android:padding="0dp"
  android:src="@drawable/my_own_button"
/>
```

Wenn Sie die App testen werden Sie sehen, dass der Button beim Anklicken das Design ändert. Wenn Sie nun die Grafik für einen inaktiven Button testen wollen, so müssen wir in der Java Klasse diesen deaktivieren. Für Testzwecke ist es vorerst ausreichend, dies irgendwo im Code durchzuführen. Wir können das kurzfristig in der Methode „myClickMethod“ testen:

```
public void myClickMethod(View view) {
    view.setEnabled(false);
}
```

Wenn Sie den Code nun nochmal testen, sehen Sie alle drei Grafikwerte – zuerst normal, anschließend gedrückt und am Schluss inaktiv.

## 4.3 Layouts

Layouts sind Anordnungsregeln von Inhalten – also Views. Da mehrere Layouts in einer Activity eingesetzt werden können – auch ineinander verschachtelt – spricht die Literatur mitunter auch von ViewGroups. Wenn man sich die Klassenvererbungsstruktur ansieht erkennt man auch, dass die Layouts „lediglich“ Subklassen von ViewGroups darstellen. Dieses Kapitel beschäftigt sich nun primär mit den XML Definitionen von Layouts. In einem späteren Kapitel werden wir uns die Möglichkeiten ansehen, wie man Layouts dynamisch mit Hilfe von Java Code erzeugt oder anpasst. Weiterhin ist hier anzumerken, dass wir uns hier nur mit den wesentlichen Elementen der Layouts auseinandersetzen. Bei spezielleren Problemen muss man sich durch die Online Dokumentation wälzen.

### 4.3.1 FrameLayout

Das wohl simpelste Layout ist das FrameLayout. Es hat fast keinerlei Funktionalität und ist somit nur für ganz bestimmte Situationen nützlich. Grundsätzlich ist es jedoch aufgrund eben dieser Eigenschaft aus Performancegesichtspunkten sehr effizient.

FrameLayouts können zwar mehrere Kindelemente anzeigen, sind jedoch vom Prinzip her auf nur ein Kindelement ausgelegt. Hat man zwei Kindelemente, so werden sie übereinandergeschichtet; der erste View wird also vom zweiten verdeckt. Mit Hilfe der `layout_gravity` oder den `margins` kann man die Elemente zwar ansatzweise positionieren, vor dem Hintergrund der unvorhersehbaren Bildschirmgrößen ist das aber nur ein Randeffekt.

Genutzt werden die FrameLayouts hauptsächlich als Platzhalter für andere Layouts, welche dynamisch in das FrameLayout positioniert werden.

Folgender Syntax liegt den FrameLayouts zugrunde:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</FrameLayout>
```

Mit Hilfe der Attribute `foreground` und `foregroundGravity` können noch Drawables eingesetzt und ausgerichtet werden, oder auch die Ausfüllfarbe des Layouts.

Wenn man nun Views positionieren möchte, werden diese zwischen den beiden FrameLayout Tags platziert:

```
<ImageView
    android:id="@+id/imageView_dice0"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:contentDescription="@string/desc_dice"
    android:src="@drawable/dice_6"
/>
```

Mit `margin` kann man das Element vom Rand entfernen:

```
android:margin="10dp"
```

Die `layout_gravity` Eigenschaft kann verwendet werden, um das Element in eine bestimmte Richtung zu zentrieren oder positionieren.

```
android:layout_gravity="center"
```

Wenn Sie ein zweites Element einfügen sehen Sie, dass das im XML nachfolgende Element über dem voranstehenden liegen wird – es also verdeckt.

```
<ImageView
  android:id="@+id/imageView_dice1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:contentDescription="@string/desc_dice"
  android:src="@drawable/dice_1"
/>
```

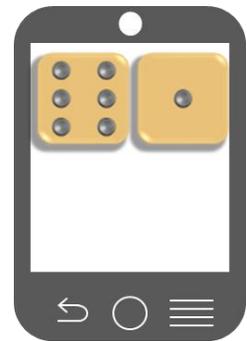
Weitere Details zum `FrameLayout` finden Sie unter:

<http://developer.android.com/reference/android/widget/FrameLayout.html>

### 4.3.2 LinearLayout

Das `LinearLayout` ist für einfache Strukturen das Layout der Wahl. Es positioniert alle Elemente in einer Linie. Der Parameter „orientation“ gibt an, ob die Elemente waagrecht (`android:orientation="horizontal"`) oder senkrecht (`android:orientation="vertical"`) angeordnet werden sollen. Hier ein Beispiel für eine horizontale Ausrichtung:

```
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">
<ImageView
  android:id="@+id/imageView_dice0"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:src="@drawable/dice_6" />
<ImageView
  android:id="@+id/imageView_dice1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:src="@drawable/dice_1" />
</LinearLayout>
```

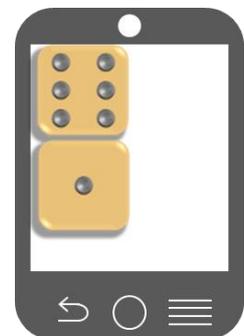


Achten Sie darauf, dass die Angabe „horizontal“ nichts mit der Geräteausrichtung zu tun hat, sondern ausschließlich mit der Ausrichtung der Elementanordnung. Wenn Sie nun den orientation - Parameter wie folgt ändern:

```
android:orientation="vertical"
```

werden sich die Grafiken nun senkrecht anordnen

Sie können beliebig viele Elemente nebeneinander anordnen. Achten Sie aber darauf, dass der Platz begrenzt ist und wenn die Summe der Abmessungen der Elemente über die des verfügbaren Bildschirms hinausragen, kann es dazu führen, dass einige Elemente kleiner dargestellt werden, oder gar komplett verschwinden.



Bei dem oberen Beispiel haben wir Grafiken verwendet, welche aufgrund Ihrer Pixelanzahl eine fest definierte Größe haben. Nun gibt es Views, welche keine solche fest definierte Größe haben. Beispielfhaft sei hier ein EditText View genannt. Die Frage ist nun, wie kann ich die Ausmaße dieses Elements basierend auf dem noch vorhandenen Platz festlegen, so dass der Bildschirm optimal ausgenutzt wird?

Hier hilft das sogenannte „weight“ Attribut. Hiermit kann man den einzelnen Elementen einen Anteil des Bildschirms zuordnen. Folgendes Beispiel soll dies verdeutlichen:

```
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <ImageView
        android:id="@+id/imageView_dice0"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/dice_6" />
    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:text="text"/>
    <EditText
        android:id="@+id/editText2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:text="text"/>
</LinearLayout>
```



*Anmerkung: Der Textinhalt „text“ wird nur zur besseren Sichtbarkeit eingefügt. In normalen Eingabefeldern würde man eher keinen Inhalt vorbelegen. Um die Sichtbarkeit für dieses Beispiel noch besser darzustellen, können verschiedene Hintergrundfarben für die beiden EditText Elemente vorgegeben werden.*

Wie sie sehen, werden die Elemente via wrap\_content minimiert dargestellt. Wenn wir nun aber die beiden EditText Elemente so dimensionieren wollen, dass sie den Bildschirm voll ausfüllen sollen, müssen wir ein neues Attribut bei allen View-Elementen einfügen:

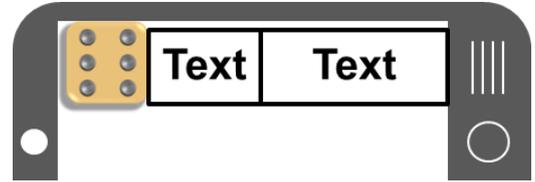
```
android:layout_weight="0"
```

Das layout\_weight Attribut sorgt dafür, dass alle Elemente einen Anteil der Breite entsprechend des layout\_weight Wertes zugesprochen bekommen.

Folgende Werte sollen gesetzt werden:

Element:	layout_weight	Ergebnis:
imageView_dice0	0	Breite wird auf vergleichbar wie bei wrap_content gesetzt.
editText1	1	(Rest)Breite wird auf $1 / (0+1+2) = 33,3\%$ gesetzt.
editText2	2	(Rest)Breite wird auf $2 / (0+1+2) = 66,6\%$ gesetzt.

Da die Breite nun vom Weight Attribut bestimmt wird, muss der Wert von **layout\_width** nun auf „0dp“ gesetzt werden, damit dieser Wert nicht mit der Weight Berechnung kollidiert. Vom Ergebnis her können Sie nun sehen, dass der Würfel auf „wrap\_content“ gesetzt wurde und die beiden Textfelder den Restbereich voll ausfüllen.



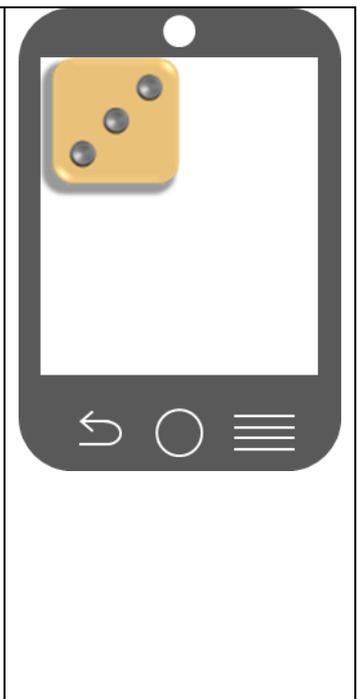
Weitere Details zum LinearLayout finden Sie unter:

<http://developer.android.com/reference/android/widget/LinearLayout.html>

### 4.3.3 RelativeLayout

Prinzipiell kann man mit verschachtelten LinearLayouts fast alles erschlagen. Das Problem daran ist jedoch, dass unter einer zu tiefen Verschachtelung die Performance leidet. Die beste Lösung für „kompliziertere“ Anordnungen ist daher das RelativeLayout. Es zeichnet sich dadurch aus, dass die einzelnen Elemente relativ zu anderen Elementen oder zum Parentelement, also dem eigentlichen Layout ausgerichtet werden. Folgende Konfigurationsinformationen sollen nun die Basis für die folgenden Beispiele sein:

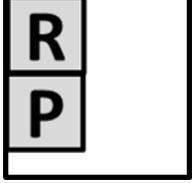
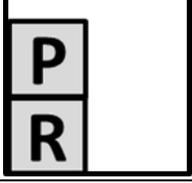
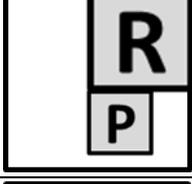
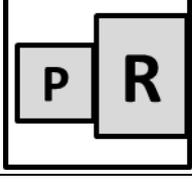
```
<RelativeLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
<ImageView
    android:id="@+id/imageView_dice1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/dice_1" />
<ImageView
    android:id="@+id/imageView_dice2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/dice_2" />
<ImageView
    android:id="@+id/imageView_dice3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/dice_3" />
</RelativeLayout>
```

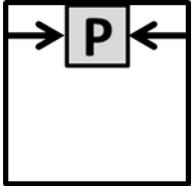
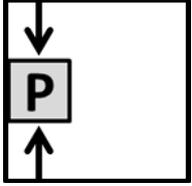
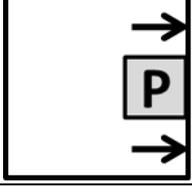
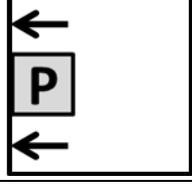
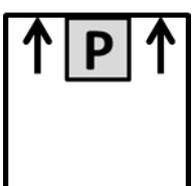
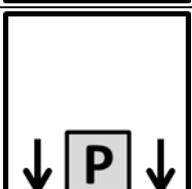


Wie Sie sehen, ist nur der dritte Würfel sichtbar. Dies liegt daran, dass keine Relationsangaben bei den einzelnen Views gemacht wurden. Somit werden alle Elemente links oben angeordnet und zwar übereinander (in der sog. Z-Achse).

Die folgende Tabelle soll nun darüber Aufschluss geben, wie die einzelnen Elemente nun positioniert werden. Hierbei wird folgende Symbolik verwendet:

Das Referenzelement wird vom zu positionierenden Element referenziert. Insofern wird das Referenzelement mit dem Buchstaben **R** gekennzeichnet, das zu positionierende Element mit **P**. Um die Eigenheiten der verschiedenen Attribute kenntlich zu machen, werden die Elemente mitunter in verschiedener Größe angezeigt. Der Rahmen um die beiden Elemente ist immer das Parent Element – somit das eigentliche Layout.

Attribut von P:	Wirkung:	Beispiel:
layout_toRightOf("@id/R")	Das Element P wird rechts vom Element R positioniert. Hat gleiches Verhalten wie layout_toEndOf – soweit ich das beurteilen kann.	
layout_toLeftOf("@id/R")	Das Element P wird links vom Element R positioniert. Dies ist jedoch nur dann sinnvoll, wenn das Element R nicht am linken Rand befindet! Hat gleiches Verhalten wie layout_toStartOf, soweit ich das beurteilen kann.	
layout_below("@id/R")	Das Element P wird unter dem Element R positioniert.	
layout_above("@id/R")	Das Element P wird über dem Element R positioniert. Dies ist jedoch nur dann sinnvoll, wenn das Element R nicht am oberen Rand befindet!	
layout_alignRight("@id/R")	Das Element P wird am rechten Rand des Elements R ausgerichtet.	
layout_alignLeft("@id/R")	Das Element P wird am linken Rand des Elements R ausgerichtet.	
layout_alignTop("@id/R")	Das Element P wird am oberen Rand des Elements R ausgerichtet.	
layout_alignBottom("@id/R")	Das Element P wird am unteren Rand des Elements R ausgerichtet.	
layout_alignBaseline("@id/R")	Die untere Seite des Inhalts von Element P (also Element ohne Padding) wird an der unteren Seite des Inhalts von Element R ausgerichtet. Bei Textinhalten würde also die Schrift unten gleich ausgerichtet sein.	

Attribut von P:	Wirkung:	Beispiel:
<code>layout_centerInParent("true")</code>	Das Element P wird horizontal und vertikal im Elternelement zentriert.	
<code>layout_centerHorizontal("true")</code>	Das Element P wird horizontal im Elternelement zentriert.	
<code>layout_centerVertical("true")</code>	Das Element P wird vertikal im Elternelement zentriert.	
<code>layout_alignParentRight("true")</code>	Das Element P wird am rechten Rand des Elternelements ausgerichtet.	
<code>layout_alignParentLeft("true")</code>	Das Element P wird am linken Rand des Elternelements ausgerichtet.	
<code>layout_alignParentTop("true")</code>	Das Element P wird am oberen Rand des Elternelements ausgerichtet.	
<code>layout_alignParentBottom("true")</code>	Das Element P wird am unteren Rand des Elternelements ausgerichtet.	

Weitere Attribute (wie bspw. `layout_gravity`) können zwar auch gesetzt werden, wobei davon abzuraten ist, da die oben genannten Positionierungsangaben effizienter sind.

Weitere Details zum `RelativeLayout` finden Sie unter:

<http://developer.android.com/reference/android/widget/RelativeLayout.html>

#### 4.3.4 `TableLayout`

Ein weiteres, recht nützliches Layout ist das `TableLayout`. Wie bei HTML Tables kann man hier die Elemente in Zeilen und Spalten anordnen. Im Gegensatz zu HTML jedoch gibt es nur Tags, welche die Zeile markieren – die Datenelemente (bei HTML wäre das `<td>`) gibt es hier nicht. Jedes Element wird automatisch als ein Spaltenelement gesehen. Dadurch wird die Anzahl der existierenden Spalten über die Zeile definiert, welche die maximale Anzahl der Elemente aufweist.

Weiterhin ist zu bemerken, dass die einzelnen Spalten eine Positionsnummer haben, wobei die linke Spalte mit der Nummer 0 beginnt.

```
<TableLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <TableRow
    android:id="@+id/tableRow1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
      android:id="@+id/imageView_dice1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/dice_6"/>
    <ImageView
      android:id="@+id/imageView_btn1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/btn_stop_a"/>
  </TableRow>
  <TableRow
    android:id="@+id/tableRow2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
      android:id="@+id/imageView_btn2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/btn_start_a"/>
    <ImageView
      android:id="@+id/imageView_dice2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/dice_1"/>
  </TableRow>
</TableLayout>
```



Gehen wir für die folgenden Erklärungen davon aus, dass wir folgende Aufteilung haben:

Hello World	1	Ich
Hello Jupiter	22	Du
Hello Saturn		Er
Hello Merkur		Sie

Zu beachten sind hier die letzten beiden Zeilen. Die Zeile „Hello Saturn“ beinhaltet nur zwei Spalten – da die Spalten 0 und 1 miteinander verbunden sind. Dies geschieht mit folgendem Code:

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hello Saturn"
  android:layout_span="2"/>
```

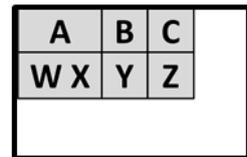
Das layout\_span Attribut legt fest, über wie viele Spalten sich das jeweilige Element erstrecken soll.

Die Zeile „Hello Merkur“ hat ebenfalls nur zwei Spalten, wobei nur die beiden äußeren belegt sind und die mittlere ausgelassen wurde. Dies wird dadurch ermöglicht, dass man explizit angeben kann, welche Spalten genutzt werden können:

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hello Merkur"
  android:layout_column="0"/>
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Sie"
  android:layout_column="2"/>
```

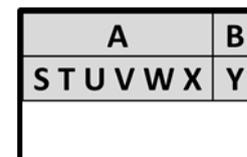
Bei dem layout\_column Attribut ist zu beachten, dass die Nummerierung bei 0 beginnt.

Für die folgenden Beispiele gehen wir von der rechts stehenden Grundstruktur aus. Im Folgenden werden wir im Tag „TableLayout“ zusätzliche Attribute einfügen, und das entsprechende Verhalten darstellen:



Attribut von TableLayout:	Wirkung:	Beispiel:
android:stretchColumns="*"	Alle Spalten werden gleichmäßig bis auf die Eltern-grenzen gestreckt.	
android:stretchColumns="0"	Nur die Spalte, wird gestreckt, welche in stretchColumns angegeben wurde (bzw. wenn man mehrere Spalten mit Komma trennt entsprechend „die Spalten“).	

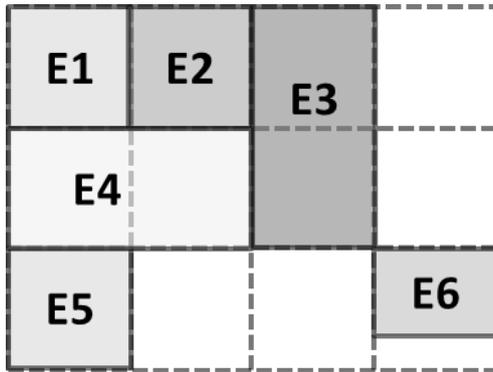
Für die folgenden Beispiele gehen wir von der rechts stehenden Grundstruktur aus. Beachten Sie, dass aufgrund der Textmenge in der Spalte 0 die rechte Spalte nicht mehr dargestellt werden kann. Im Folgenden werden wir im Tag „TableLayout“ zusätzliche Attribute einfügen, und das entsprechende Verhalten darstellen:



Attribut von TableLayout:	Wirkung:	Beispiel:
android:shrinkColumns="0"	In der angegebenen Spalte wird ein Zeilenumbruch durchgeführt, so dass alle Spalten im Elternelement Platz finden. Ein „*“ würde dazu führen, dass alle Spalten Zeilenumbrüche durchführen würden, wo es sinnvoll wäre.	
android:collapseColumns="0"	Die Angegebene Spalte (bzw. die Spalten) werden ausgeblendet.	

### 4.3.5 GridLayout

Mit Android 4.0 (API Level 14) wurde das GridLayout eingeführt, um noch mehr Flexibilität zu erlangen. Das GridLayout geht davon aus, dass ähnlich wie beim TableLayout der gesamte Screen in ein Raster von Zellen aufgeteilt wird, wobei jedes Element eine Row und eine Column (also Zeile und Spalte) aufweist.



Weiterhin können die Elemente sowohl mehrere Zeilen, als auch mehrere Spalten überdecken. In der folgenden Grafik sehen Sie ein Grid mit 4 Spalten und 3 Zeilen. E1 und E2 werden ganz normal eingefügt. E3 wird über zwei Zeilen positioniert (nicht gestreckt, sondern positioniert!). E4 über zwei Spalten. E5 wird wieder normal platziert und E6 (welches bspw. etwas kleiner ist als die anderen Elemente), dediziert auf die Zeile 2 und Spalte 3.

Ein weiterer Punkt im Rahmen des GridLayouts ist das „Space“ Element. Dieses ermöglicht es, leere Bereiche zu spezifizieren.

Aber gehen wir die einzelnen Punkte anhand eines Codebeispiels durch:

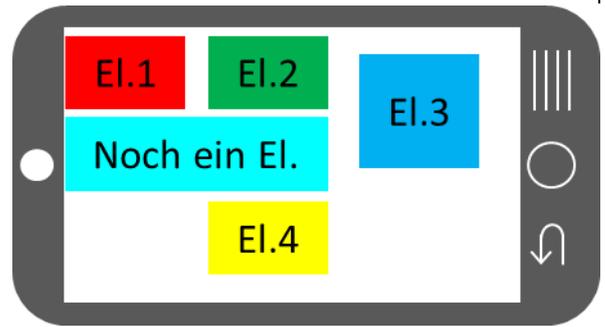
```
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/mainLinearLayout"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:alignmentMode="alignBounds"
  android:useDefaultMargins="true"
  android:columnCount="3">

  <TextView
    android:id="@+id/textView1"
    android:layout_gravity="left"
    android:background="#ff0000"
    android:padding="10dp"
    android:text="El.1"/>
  <TextView
    android:id="@+id/textView2"
    android:layout_gravity="right"
    android:background="#00ff00"
    android:padding="10dp"
    android:text="El.2"/>
  <TextView
    android:id="@+id/textView3"
    android:background="#0000ff"
    android:layout_rowSpan="2"
    android:padding="30dp"
    android:layout_gravity="center"
    android:text="El.3"/>
  <TextView
    android:id="@+id/textViewX"
    android:layout_gravity="center_horizontal"
    android:background="#00ffff"
    android:layout_columnSpan="2"
    android:padding="10dp"
    android:text="Noch ein El."/>
```

```

<TextView
  android:id="@+id/textView4"
  android:layout_gravity="top"
  android:layout_column="1"
  android:layout_row="2"
  android:background="#ffff00"
  android:padding="10dp"
  android:text="El.4"/>
<Space
  android:id="@id/space1"
  android:layout_row="2"
  android:layout_gravity="fill"/>
</GridLayout>

```



Das Ergebnis wird wie oben dargestellt aussehen (je nach Bildschirmgröße können die Abstände abweichen). Arbeiten wir die einzelnen Attribute durch. Beginnen wir mit dem GridLayout Attribut „alignmentMode“:

```
android:alignmentMode="alignBounds"
```

Es legt fest, ob die Ausrichtung auf die eigentliche Grenze des Elements (alignBounds), oder auf die Elementgrenze plus Margin (alignMargin) ausgerichtet ist. Wir erinnern uns, die Margin spezifiziert, wie groß der Abstand zwischen den einzelnen Elementen sein soll – man kann sich um das Element also eine unsichtbare Box mit dem Marginabstand vorstellen, nach der ausgerichtet wird.

```
android:useDefaultMargins="true"
```

Dieses Attribut setzt die Margins zwischen den einzelnen Elementen auf einen Wert auf Basis des aktuell eingestellten UI Styles. Wenn dieser Wert auf „true“ sitzt, dann muss alignmentMode auf „alignBounds“ stehen. Wenn er auf „false“ steht, dann schrumpfen die Margins auf 0, es sei denn es sind andere Werte in den einzelnen Kindelementen angegeben.

```
android:columnCount="3"
```

Hier wird die Anzahl der Spalten angegeben. Das ist deshalb notwendig, weil die einzelnen Elemente nicht zwingend auf eine dedizierte Spalte gesetzt werden müssen – in solch einem Fall platziert Android die Elemente nach einem speziellen Schema selbst (alle Spalten der ersten Zeile werden aufgefüllt, danach die Spalten der nächsten Reihe usw.). Im Zweifelsfall wird allerdings empfohlen, die Elemente per row/column Attribut zu positionieren (siehe weiter unten).

```

<TextView
  android:id="@+id/textView1"
  android:layout_gravity="left"
  android:background="#ff0000"
  android:padding="10dp"
  android:text="El.1"/>

```

Das erste Textelement wird, da es eben an erster Stelle steht, an der oberen linken Ecke platziert (also automatisch auf Zeile 0, Spalte 0). Mit Hilfe der layout\_gravity wird es an den linken Rand der Zelle gesetzt. Die Farbe, der Paddingwert und der Textinhalt dienen nur zur besseren Veranschaulichung für dieses Beispiel.

TextView2 wird per layout\_gravity an den rechten Rand der Zelle gesetzt. Dies hat dann einen sichtbaren Effekt, wenn die Zelle aufgrund des Inhalts einer anderen Zelle in der gleichen Spalte größer ist, als TextView2.

Beim TextView3 wird es nun spannend – hier wurde der Paddingwert nach Oben gedreht, damit wir ein im Vergleich zu TextView1 und TextView2 höheres Element erhalten. Normalerweise würde nun die erste Zeile von der Höhe her vergrößert, da wir aber mit `layout_rowSpan="2"` den TextView3 auf zwei Zeilen ausdehnen, muss die erste Zeile nicht nach unten ausgedehnt werden.

```
android:layout_rowSpan="2"  
android:padding="30dp"  
android:layout_gravity="center"
```

Das `layout_gravity="center"` Attribut richtet das Element mittig aus (sinnvoll für unser Beispiel allerdings nur in Verbindung mit "Space" – siehe weiter unten).

TextViewX wurde nun sehr viel größer gestaltet (durch einen entsprechend größeren Textinhalt). Mit einer Ausweitung der belegten Spalten, wird nun das Element in die ersten beiden Spalten platziert.

```
android:layout_columnSpan="2"
```

Das TextView4 Element wurde nun in die letzten Zeile gesetzt (dies wurde hier zwar mit `layout_row="2"` gemacht, würde aber aufgrund der Tatsache, dass TextViewX – also der Vorgänger – am rechten Rand liegt automatisch in die nächste Zeile rutschen). Danach wurde das Element auf die mittlere Spalte gesetzt und per `gravity` nach oben ausgerichtet.

```
android:layout_gravity="top"  
android:layout_column="1"  
android:layout_row="2"
```

Ein wichtiges neues Element ist das Space Element. Dieses sorgt dafür, dass leere Bereiche definiert werden können:

```
<Space  
    android:id="@id/space1"  
    android:layout_row="2"  
    android:layout_gravity="fill"/>
```

Hier wird ein Space Element in die Zeile 2 platziert und mit dem Attribut `gravity` auf "fill" gesetzt; es füllt also den gesamten noch zur Verfügung stehenden Bereich aus. Dies ist vor allem deshalb eingebaut worden, damit die unterste Zeile bis nach unten „wächst“. Hätten wir dieses Setting nicht eingestellt, dann würde `textView3` aufgrund des `gravity="center"` settings nach unten rutschen – `center` würde hier bedeuten "zentriere das Element im Rahmen des zur Verfügung stehenden Raumes". Da aber nun die Zeile 2 bis nach unten geht, „nimmt“ sich `textView3` nur so viel Platz, wie vorhanden ist – und das ist entweder die Außengrenze des Textviews plus Margin, oder die Summe der beiden Zeilen 0 und 1 (je nachdem was größer ist).

Allgemein gilt zum GridLayout noch zu sagen, dass es vom Verhalten (und somit auch von der Nutzbarkeit der meisten Attribute) her „zwischen“ dem LinearLayout und dem TableLayout liegt – wobei das `weight` Attribut hier nicht funktioniert.

*Achtung: Wenn Sie das GridLayout verwenden, seien Sie sich darüber bewusst, dass Sie sich auf die API Version 14 oder höher festlegen (also ab Android 4.0).*

### 4.3.6 Nested Layouts

Es gibt Situationen, in denen die Möglichkeiten eines Layouts nicht ausreichend sind. In solchen Fällen kommen die verschachtelten Layouts ins Spiel. Layouts können nicht nur Views enthalten, sondern auch andere Layouts. Folgendes Beispiel soll dies verdeutlichen:

```
<LinearLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <ImageView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/dice_3" />
    <ImageView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/dice_2" />
  </LinearLayout>
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TableRow
      android:layout_width="wrap_content"
      android:layout_height="wrap_content">
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Kai:" />
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="12" />
    </TableRow>
    <TableRow
      android:layout_width="wrap_content"
      android:layout_height="wrap_content">
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Michaela:" />
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="8" />
    </TableRow>
  </TableLayout>
</LinearLayout>
```



Das Beispiel beinhaltet ein äußeres `LinearLayout`, welches den Gesamtrahmen bildet. Dies ist vertikal ausgerichtet, da es zuerst die beiden Würfel und anschließend die Tabelleninformation abbilden muss. Als erstes Kindelement fungiert ein weiteres `LinearLayout`, welches die beiden Würfel positioniert. Anschließend wird

ein weiteres Kindelement des äußeren LinearLayouts eingefügt, welches ein TableLayout mit zwei Zeilen a zwei Einträgen beinhaltet. Es werden also zwei Layouts ineinander verschachtelt.

Folgende Punkte sind zu beachten:

- Machen Sie sich vorher Gedanken, wie der Screen aufgeteilt werden soll.
- Zu tiefe Verschachtelungen beeinträchtigen die Performance.
- Prüfen Sie unbedingt, ob ein RelativeLayout nicht genauso funktionieren würde.

Die oben geforderte Anordnung können bspw. auch mit einem RelativeLayout erreicht werden:

```
<RelativeLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <ImageView
    android:id="@+id/imageView_dice0"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/dice_3" />
  <ImageView
    android:id="@+id/imageView_dice1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/imageView_dice0"
    android:src="@drawable/dice_2" />
  <TextView
    android:id="@+id/textView_name0"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/imageView_dice0"
    android:text="Kai:" />
  <TextView
    android:id="@+id/textView_number0"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/imageView_dice0"
    android:layout_toRightOf="@id/textView_name1"
    android:text="12" />
  <TextView
    android:id="@+id/textView_name1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/textView_name0"
    android:text="Michaela:" />
  <TextView
    android:id="@+id/textView_number1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/textView_name0"
    android:layout_alignLeft="@id/textView_number0"
    android:text="8" />
</RelativeLayout>
```

Grundsätzlich sollte eine möglichst flache Layoutstruktur angestrebt werden.

### 4.3.7 Scrollbars

Sollte der Platz trotz angepasster Views nicht ausreichend sein, so ist es oftmals notwendig Scrollbars einzufügen. Im einfachsten Fall wird lediglich eine View eingefügt, welche das Scrollen erlaubt. Hierbei ist zu beachten, dass das vertikale und horizontale Scrollen getrennt gehandhabt wird. Auch wichtig zu wissen ist, dass die Scrollviews nur ein einziges Kindelement akzeptieren. Folgendes Beispiel zeigt ein LinearLayout, welches in eine vertikale und horizontale Scrollviews eingebettet wurde:

```
<ScrollView xmlns:android=
  "http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/v_scroll"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical"
  tools:context=".MainActivity" >
  <HorizontalScrollView
    android:id="@+id/h_scroll"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal">
      ... Hier kommen die Inhalte
    </LinearLayout>
  </HorizontalScrollView>
</ScrollView>
```

In vielen Fällen wird nur eine vertikale Scrollbar benötigt. In solchen Fällen wird die eingebettete HorizontalScrollView einfach weggelassen und das innere Layout wird direkt in die ScrollView eingebettet.

Neben den offensichtlichen Funktionen – den Screen zu scrollen – kann man bei den Scrollviews noch viele andere Dinge einstellen. Hier eine kleine Auswahl der Möglichkeiten:

Attribut:	Wirkung:
scrollX	Offsetwert der Scrollbar in horizontaler – Richtung (in Pixel)
scrollY	Offsetwert der Scrollbar in vertikaler – Richtung (in Pixel)
scrollbars	Flag, ob Scrollbars angezeigt werden sollen (true) oder nicht (false), so dass nur der Bildschirm hin und her geschoben werden kann.
scrollbarStyle	Legt fest, wo der Scrollbar angelegt wird (bspw. über das Background Image, innerhalb der Padding etc.).
scrollbarThumbHorizontal	Referenz auf Drawable, welches festlegt, wie der horizontale Scrollbar Button aussehen soll.
scrollbarThumbVertical	Referenz auf Drawable, welches festlegt, wie der vertikale Scrollbar Button aussehen soll.
scrollbarAlwaysDrawHorizontalTrack	Flag, ob horizontale Scrollbar Track immer (true), oder nur beim Scrollen (false) angezeigt werden soll.
scrollbarAlwaysDrawVerticalTrack	Flag, ob vertikale Scrollbar Track immer (true), oder nur beim Scrollen (false) angezeigt werden soll.

Weitere Möglichkeiten können entweder über die Autovervollständigung von Eclipse, oder über folgende Referenzseite ersehen werden:

<http://developer.android.com/reference/android/view/View.html>

## 5 Layout mit Java

Layouts werden in aller Regel in den \*.xml Dateien im **layout** Ordner festgelegt. Trotzdem ist es oftmals notwendig, dynamisch auf die entsprechenden Einstellungen zuzugreifen. Hierzu dient die entsprechende Java API, welche hier in Grundzügen näher gebracht werden soll.

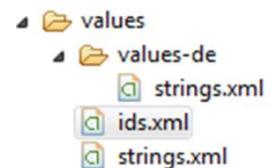
### 5.1 Die id als eindeutige Adresse

Wie Sie bei den vorausgegangenen Kapiteln gelernt haben, ermöglicht die XML Notation der einzelnen Elemente, eine eindeutige ID zu vergeben. Diese wird durch einen Namen identifiziert und innerhalb der App von Android als eindeutige Integer ID erzeugt. Folgender XML Code ist hierfür zuständig:

```
android:id="@+id/textView1"
```

Der Name „textView1“ ist frei vom User vergeben und kann (unter der Voraussetzung, dass er eindeutig ist und den üblichen Namenskonventionen folgt) beliebig sein. Android erzeugt auf dieser Basis beim Kompilieren nun eine Integerzahl, welche von diversen Java Funktionen als Referenzierungsmöglichkeit genutzt wird.

Sollten Sie eine View nicht über die \*.xml Dateien im **layout** Ordner, sondern dynamisch erzeugen wollen, dann bietet Android die Möglichkeit im **values** Ordner ein Ressourcenfile namens **ids.xml** zu erzeugen und mit folgendem Syntax eigene IDs zu erzeugen. In diesem File werden nun einfach die IDs als Namenwerte eingetragen:



```
<?xml version="1.0" encoding="UTF-8"?>
<resources>
    <item type="id" name="myDynElement1"/>
    <item type="id" name="myDynElement2"/>
</resources>
```

Diese können später den dynamisch erzeugten Elementen zugewiesen werden. Beispielhaften Code hierzu finden Sie in den folgenden Kapiteln.

### 5.2 Identifikation und Anpassung der Elemente

Gehen wir von folgendem XML Code aus:

```
<ImageView
    android:id="@+id/imgView_dice0"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/dice_1">
```

Es wird also eine ImageView erzeugt, mit der ID „imgView\_dice0“. Als Drawable Ressource wird der dice\_1 festgelegt. Wenn wir nun die Drawable Ressource austauschen möchten, benötigen wir eine Referenz auf das entsprechende Objekt. Folgender Code ermöglicht dies:

```
ImageView myImg = (ImageView) findViewById(R.id.imgView_dice0);
```

Es wird die Variable myImg vom Datentyp ImageView deklariert. Die Methode „findViewById“ liefert die entsprechende Referenz auf das gesuchte Objekt. Als Parameter wird die ID benötigt, wie sie in den Ressourcen festgelegt wurde (R.id.imgView\_dice0). Da die Methode nur Referenzen vom Datentyp View zurückgibt, muss ein entsprechender Typecast auf ImageView dafür sorgen, dass die Zuweisung zu unserer Variablen myImg möglich ist.

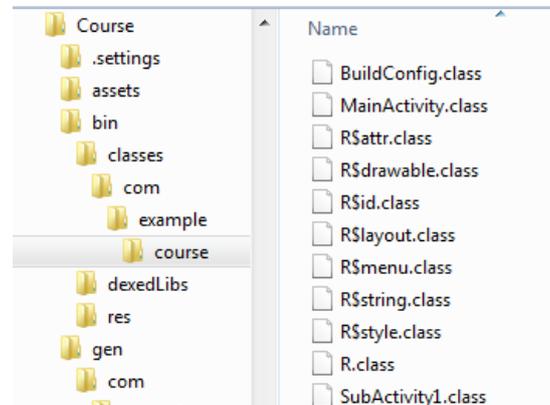
Nun haben wir die gesamte API von ImageView Objekten zur Verfügung. Eine kurze Recherche in der API Dokumentation liefert uns die Methode „setImageDrawable“. Hiermit können wir die Grafik austauschen.

Einziges Problem bleibt jetzt, die richtige Grafik zu identifizieren. Hierzu müssen wir eine Methode des Objektes Activity nutzen, welche uns einen Zugriff auf die Ressourcen ermöglicht: „getResources“. Diese Methode unserer Activity liefert uns eine Instanz auf die Ressourcen des Packages, welches wiederum einen Zugriff auf die Drawables ermöglicht:

```
myImg.setImageDrawable(getResources().getDrawable(R.drawable.dice_6));
```

Der obenstehende Code zeigt nun die komplette Sequenz. R.drawable.dice\_6 ist nun die eindeutige Adresse der Bilddatei **dice\_6.png** (im System wird dies als Integerwert abgelegt sein), mit der die Methode getDrawable die Drawable Ressource laden kann. Da diese in der Ressourceninstanz liegt, muss diese aus der Activity heraus mit getResources geholt werden. Nun kann die entsprechende Ressource in myImg eigesetzt werden.

Grundsätzlich ist zu sagen, dass sämtliche Konfigurationen am Ende in Code umgewandelt werden (müssen). Insofern ist es auch logisch, dass sämtliche, in XML existierenden Einstellungsmöglichkeiten auch programmatisch über Java APIs anpassbar sind. Erkennen kann man dies auch anhand der generierten \*.class Files auf dem Filesystem.



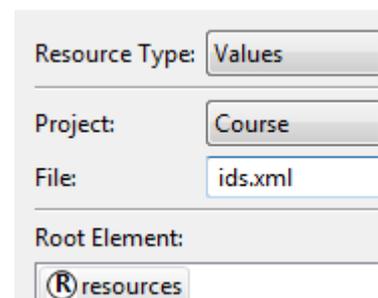
### 5.3 Erzeugung eines View-Objektes in Java

Im folgenden Kapitel werden wir nun ein komplettes Element (TextView) mit Hilfe von Java Code erstellen. Lediglich die ID und der Textinhalt werden über XML bereitgestellt. Zuerst erzeugen wir hierzu ein LinearLayout in unserer MainActivity:

```
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/mainLinLayout"
    android:orientation="vertical">
</LinearLayout>
```

Wichtig ist hier, dass wir dem Layout eine ID geben (mainLinLayout), um es später identifizieren zu können. Anschließend erzeugen wir eine weitere ID, um dem erzeugten TextView auch eine eindeutige ID zuweisen zu können. Hierzu wird in Eclipse unter **values** ein neues File (**ids.xml**) erstellt, sofern es noch nicht existiert. Klicken Sie hierzu in ihrem Android Projekt auf den Ordner **values**. Diesen finden Sie als Unterordner des Verzeichnisses **res**. Anschließend klicken Sie auf das  Symbol, um ein Android XML File zu erzeugen.

Nennen Sie es **ids.xml** und stellen Sie sicher, dass als Root Element „resources“ steht. Anschließend bestätigen Sie die Aktion mit „Finish“.



Nun sollten Sie unter dem **values** Ordner ein File namens **ids.xml** finden. Tragen Sie dort folgenden Code ein (sollten Sie bereits ein **ids.xml** File haben, so genügt es, lediglich den „item“ Tag zu ergänzen).

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item name="myDynID1" type="id"/>
</resources>
```

Ich habe mich für den (beliebigen) Namen „myDynID1“ entschieden, der später für das dynamisch erzeugte Element herangezogen wird.

Schließlich werden wir noch den String Inhalt in die **strings.xml** Datei einfügen, um den Android Designvorgaben entsprechend die Textflexibilität zu erhalten. Öffnen Sie die **strings.xml** (ebenfalls im **values** Ordner) und ergänzen Sie hier den folgenden Eintrag zwischen den `<resources>` und `</resources>` Tags:

```
<string name="myDynText">Hello, world!</string>
```

Der Name des Strings wurde von mir als „myDynText“ festgelegt – hier können Sie jedoch auch die Namen frei wählen. Nun müssen wir nur noch den Java Code ergänzen. Gehen Sie hierzu in die Klasse `MainActivity.java` im Ordner **src** (und hier in Ihrem Package Namen) und suchen Sie die Methode „onCreate“.

*Anmerkung: Die Aufrufstruktur innerhalb der Activities wird beim Grüngurt (also Stufe2) näher erläutert. Für uns ist es in diesem Augenblick nur wichtig zu verstehen, dass die Methode onCreate immer dann aufgerufen wird, wenn die Activity auf dem Screen aufgebaut wird.*

Am Ende der Methode onCreate wird nun folgender Code ergänzt:

```
TextView myTextView = new TextView(this);
myTextView.setText(getResources().getString(R.string.myDynText));
myTextView.setId(R.id.myDynID1);
myTextView.setLayoutParams(new LayoutParams
    (
        LayoutParams.MATCH_PARENT,
        LayoutParams.WRAP_CONTENT
    )
);

LinearLayout myLayout = (LinearLayout) findViewById(R.id.mainLinLayout);
myLayout.addView(myTextView);
```

Zuerst wird ein neues `TextView` Element erzeugt, wobei der Konstruktor eine Referenz auf die aktuelle Activity benötigt. Anschließend wird mit der Methode „setText“ der anzuzeigende Text gesetzt. Dieser wird mit `getResources()` aus der Ressourceninstanz geladen. Identifiziert wird die Stringressource über `R.string.myDynText`, also genau dem Namen, den wir in **strings.xml** festgelegt haben.

Anschließend wird mit `setId` dem `TextView` eine ID zugeordnet, welche über `R.id.myDynID1` gefunden wird. Hier benötigen wir keine `getResources()` Methode, da eine ID nur eine Zahl und keine Ressource ist.

Die nächsten Zeilen legen die grundlegenden Layout Parameter fest – hier entscheiden wir uns nur für die bei jedem View zwingend notwendigen Höhen- und Breitenangaben. Die Java API zeigt hier noch weitere Optionen auf, wobei für uns soll die hier gewählte Variante ausreichend sein.

Schließlich holen wir uns eine Referenz auf unser `LinearLayout` (wieder mit der Methode `findViewById`) und hängen den neu erzeugten `TextView` in das `LinearLayout` mit der Methode „addView“ hinein.

Wichtig zu verstehen ist, dass wir mit der hier gezeigten Methode alle Layoutangaben programmatisch durchführen müssen. Dies kann mitunter zu sehr aufwändigen Konstrukten führen – wenn wir an Schriftgrößen, Schriftarten, Hintergrund/Vordergrundfarben usw. denken, sollte dies klar sein.

## 5.4 Layout Inflater – der elegante Weg

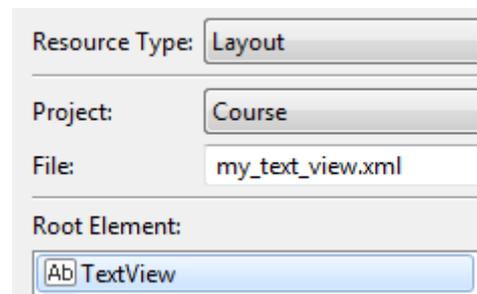
Um nun nicht jede einzelne Einstellung mühsam per Java einzutragen, haben sich die Android Entwickler etwas Besonderes einfallen lassen – den LayoutInflater. Dieser ermöglicht es, einzelne Layoutelemente als XML zu konfigurieren und mit einem einfachen Methodenaufruf zu instanziiieren. Dies ist vor allem dann sinnvoll, wenn man mehrere gleichartige Views erzeugen möchte – wie bspw. die Zeilen in einer Tabelle. Zum Verständnis werden wir wieder ein Beispiel ausprobieren. Wir verwenden das gleiche Layout wie in unserem vorigen Beispiel. Wir erzeugen nun noch zwei weitere Stringwerte und zwei weitere ID Werte:

```
<item name="myDynID2" type="id"/>
<item name="myDynID3" type="id"/>
```

Nun die Stringwerte:

```
<string name="myDynText2">Hello, jupiter!</string>
<string name="myDynText3">Hello, mars!</string>
```

Als nächstes erzeugen wir im Ordner **layout** ein neues Android XML File, indem wir auf das  Symbol klicken. Hier nennen wir das File **my\_text\_view.xml**, wählen als Root Element den TextView aus und klicken „Finish“.



Nun können wir in dem erzeugten File beliebige Einstellungen vornehmen. Wir ändern die beiden „fill\_parent“ Werte auf „wrap\_content“. Beispielhaft ändern wir nun auch die Farbe des Textes, indem wir folgendes Attribut einfügen:

```
android:textColor="#ff0000"
```

Nun werden wir mit Hilfe dieser Angaben zwei TextView Elemente erzeugen. Ergänzen Sie am Ende von onCreate in der MainActivity Klasse folgenden Code:

```
LayoutInflater inflater = getLayoutInflater();
TextView newView = (TextView) inflater.inflate(
    R.layout.my_text_view, null);
newView.setId(R.id.myDynID2);
newView.setText(getResources().getString(R.string.myDynText2));
myLayout.addView(newView);
newView = (TextView) inflater.inflate(R.layout.my_text_view, null);
newView.setId(R.id.myDynID3);
newView.setText(getResources().getString(R.string.myDynText3));
myLayout.addView(newView);
```

Zuerst wird der LayoutInflater über die getLayoutInflater Methode geholt. Alternativ kann dies auch mit folgender Methodik erfolgen:

```
LayoutInflater.from(getApplicationContext());
```

Als nächstes wird ein Textview erzeugt, welcher auf Basis der von uns erstellten **my\_text\_view.xml** Datei konstruiert wird. Wichtig ist, dass wir über R.layout.my\_text\_view unser Layout identifizieren (der Filename steht also für die Layout Kennung). Auch der zweite Parameter ist wichtig, den wir mit „null“ belegt haben. Hier kann man auch das Rotelement (also in unserem Beispiel das LinearLayout, in das der TextView eingehängt wird) einfügen, in das das neue Element eingehängt werden soll. Gibt man dies jedoch an, so ist der Rückgabewert dieser Methode das Rotelement und nicht das erzeugte Element (also in unserem Beispiel würde anstatt dem erzeugten TextView das LinearLayout zurückgegeben werden). Dies erschwert aber die anschließende Konfiguration des Elements.

Da wir jedoch „null“ angegeben haben, erhalten wir aus der inflate Methode eine Referenz auf das neu erzeugte TextView Element. Anhand dieser können wir nun den Text und die ID des Elements festlegen. In den darauf folgenden Zeilen des Beispielcodes sehen Sie, dass mit dem XML File nun ein weiteres Element erzeugt werden kann. Lediglich die ID und der Text werden mit anderen Werten bewirtschaftet.

Es können auch kompliziertere Konstrukte über einen LayoutInflater erzeugt werden. Hier ist jedoch zu beachten, dass Sie bereits im layout File temporäre IDs vergeben sollten, damit Sie bei der Konfiguration der Elemente diese wiederfinden können. Folgendes Beispiel soll dies erläutern – wir erzeugen ein layout File mit dem Namen **my\_lin\_example.xml** mit folgendem Inhalt:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content "
        android:textColor="#0000ff"
        android:id="@+id/firstText"/>
    <TextView
        android:layout_width="wrap_content "
        android:layout_height="wrap_content "
        android:textColor="#0000ff"
        android:id="@+id/secText"/>
</LinearLayout>
```

Wie Sie sehen, befinden sich in diesem File mehrere Entitäten, wobei die Kindelemente auch eine ID aufweisen. Diese benötigen wir, um später wieder Zugriff auf die Entitäten zu erhalten. Weiterhin benötigen wir nun wieder ID Werte, mit denen wir die existierenden überschreiben können:

```
<item name="myDynLinID1" type="id"/>
<item name="myDynTxtID11" type="id"/>
<item name="myDynTxtID12" type="id"/>
```

Nun können wir die gesamte Struktur über den LayoutInflater erzeugen. Die ursprünglichen IDs aus **my\_lin\_example.xml** werden nun übernommen. Wenn wir nun eventuell basierend darauf noch ein zweites Element aufbauen möchten, so müssen wir die IDs austauschen, da sonst das zweite Element die gleichen IDs aufweist, wie das erste.

```
LinearLayout myLinLayout =
    (LinearLayout) inflater.inflate(R.layout.my_lin_example, null);
myLinLayout.setId(R.id.myDynLinID1);
myLayout.addView(myLinLayout);
TextView myLinText1 = (TextView) findViewById(R.id.firstText);
TextView myLinText2 = (TextView) findViewById(R.id.secText);
myLinText1.setId(R.id.myDynTxtID11);
myLinText2.setId(R.id.myDynTxtID12);
myLinText1.setText(getResources().getString(R.string.myDynText2));
myLinText2.setText(getResources().getString(R.string.myDynText3));
```

Zuerst müssen wir also das my\_lin\_example erzeugen, indem wir den inflater die Referenz auf das Layout geben. Anschließend ändern wir die ID, so dass bei einer erneuten Erzeugung von my\_lin\_example keine ID Duplikate erhalten. Nun können wir das neue erzeugte LinearLayout in das äußere LinearLayout der MainActivity mit der addView Methode einfügen.

Erst jetzt ist es möglich, die beiden TextViews über die findViewById Methode zu referenzieren. Zu diesem Zeitpunkt ist dies aber nur über die IDs möglich, welche wir im **my\_lin\_example.xml** festgelegt haben.

Wenn die Referenz gefunden wurde, können die entsprechenden Anpassungen, wie IDs und Text setzen durchgeführt werden. Im Bildschirm können Sie sehen, dass die beiden TextViews nun nebeneinander angeordnet wurden.

## 6 Lizenz



Diese(s) Werk bzw. Inhalt von Maik Aicher ([www.codeconcert.de](http://www.codeconcert.de)) steht unter einer Creative Commons Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 3.0 Unported Lizenz.

The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

“Eclipse” and the Eclipse Logo are trademarks of Eclipse Foundation, Inc.

"Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners."

## 7 Haftung

Ich übernehme keinerlei Haftung für die Richtigkeit der hier gemachten Angaben. Sollten Fehler in dem Dokument enthalten sein, würde ich mich über eine kurze Info unter [maik.aicher@gmx.net](mailto:maik.aicher@gmx.net) freuen.