# Tutorial

# Programmierung
# einer
# Android-Applikation
# Notizzettel

# Teil 2

Autor: Oliver Matle
Datum: April 2014, Version 1.0

# Inhaltsverzeichnis

Kapitel 1 – Einleitung.....3
Fachliche Beschreibung.....3
Kapitel 2 – Die Oberfläche wird schöner.....4
Kapitel 3 – Daten in einer Datei speichern.....10
Die Klasse Memo.....10
Die Klasse DataController.....11
Kapitel 4 – Der List-Adapter.....17
Kapitel 5 – Erweiterung der Activity-Controller.....20
Erweiterung der Start-Activity.....20
Erweiterung der ShowMemosActivity.....21
Erweiterung der ShowMemoActivity.....22
Erweiterung der CreateMemoActivity.....22
Kapitel 6 – Übungsaufgaben.....23
Lösungen Aufgabe 1.....23
Lösungen Aufgabe 2.....23

# Kapitel 1 – Einleitung

In diesem Tutorial wird die Programmierung einer Android-App beschrieben, mit der man persönliche Notizen verwalten kann.

In diesem zweiten Teil des Tutorials erweitern wir die Funktionalität der App, so dass auch Notizen erstellt und gespeichert werden können. Die Oberfläche wir etwas schöner gestaltet. Die Notizen lassen sich jetzt in einer Liste darstellen. Dazu erstellen wir einen Adapter.

Der erste Teil dieses Tutorials ist Voraussetzung zum Verständnis, weil darauf aufgebaut wird.

**Diese Tutorials sind für den privaten Gebrauch kostenlos, unterliegen aber dem Urheberrecht. Für Schäden kann keine Haftung übernommen werden. Die Nutzung der Programme erfolgt auf eigene Gefahr. Die Programme wurden getestet. Eine Garantie, dass die Programme immer und überall ohne Fehler laufen, kann nicht gewährleistet werden. Alle genannten Namen sind Eigentum der jeweiligen Rechteinhaber.**

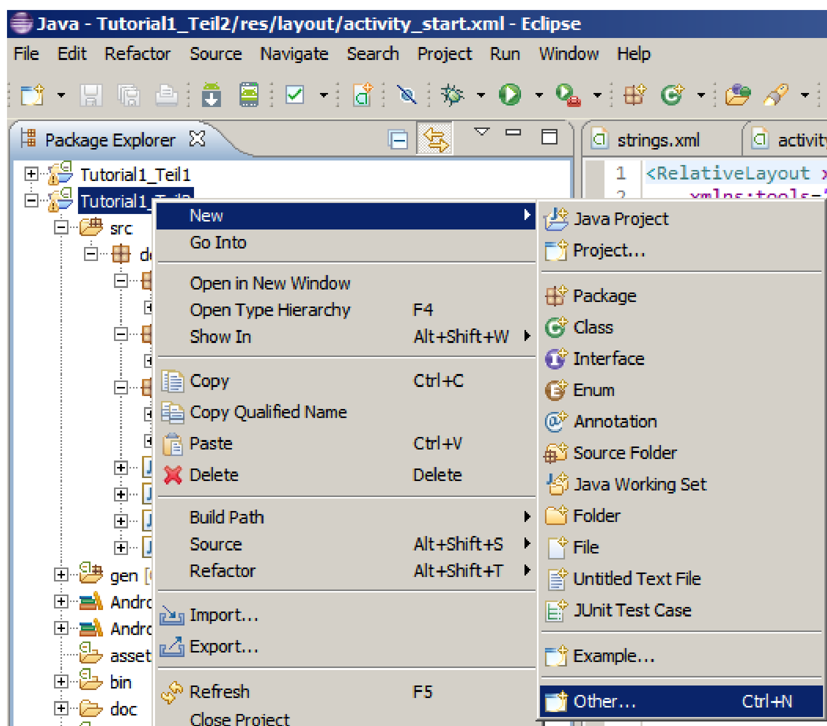
## *Fachliche Beschreibung*

Eine Notiz besteht aus einem Thema und einem Notizinhalt, also dem eigentlichen Text. Notizen können neu angelegt, gelöscht oder geändert werden. Eine Liste zeigt alle Notizenthemen an. In einer Detailansicht kann man den Notizeninhalt lesen, ändern oder die komplette Notiz löschen.

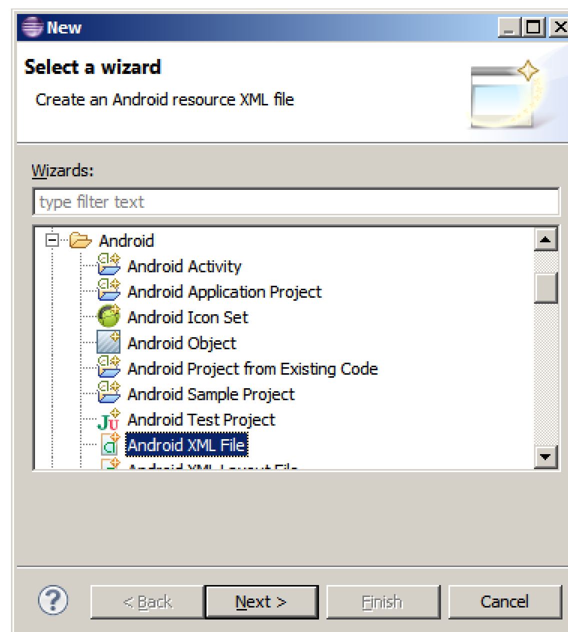
# Kapitel 2 – Die Oberfläche wird schöner

In diesem Kapitel möchte ich zeigen, wie man mit wenigem Aufwand ein etwas schöneres Layout gestalten kann.

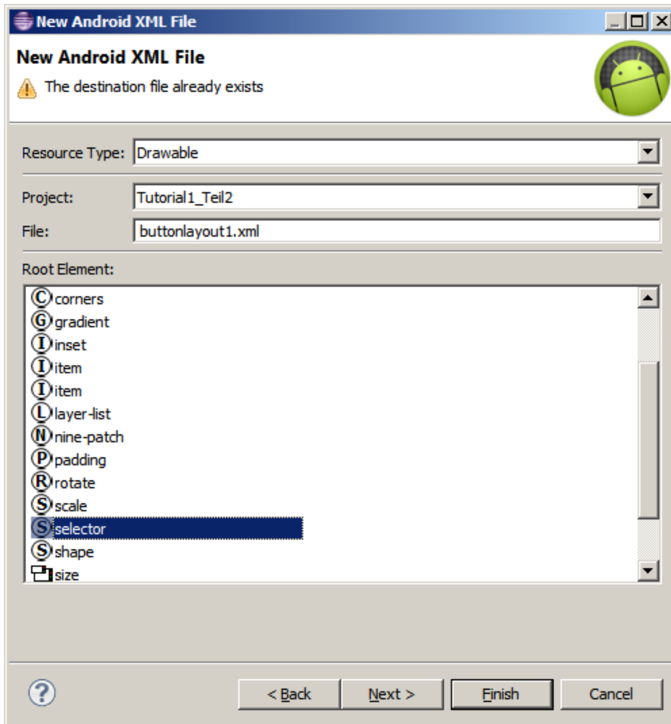
Zunächst möchte ich die Buttons etwas bunter gestalten. Dazu legen wir ein neues Layout an. Wir erstellen im Menü New/Other ein neues XML-File.



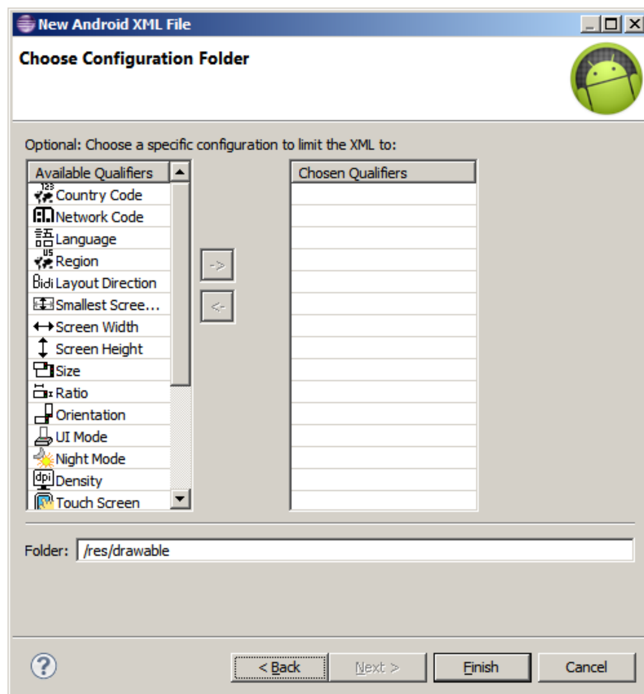

Dies findet man im Zweig Android und dort unter Android XML-File.



Bei Resource Typ wählt man Drawable aus und als Dateinamen verwenden wir buttonlayout1.xml.




Als Root-Element wählen wir selector aus.

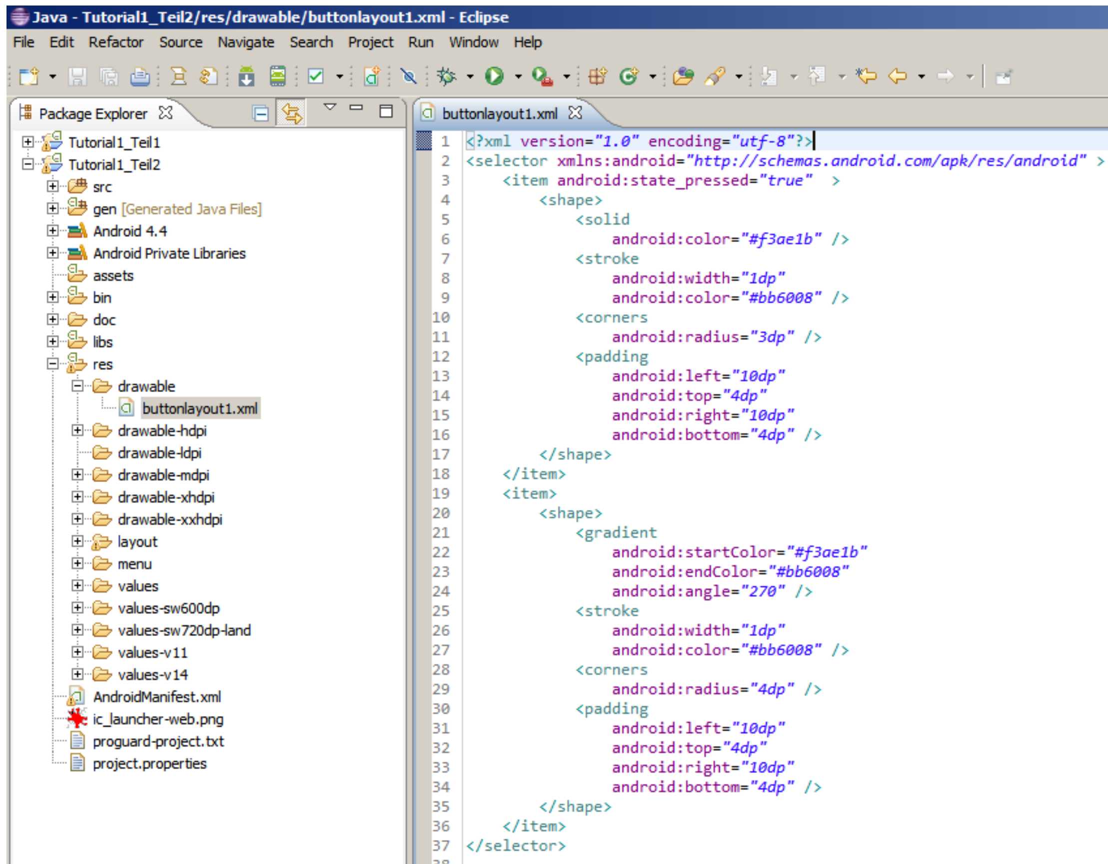



In den folgenden Dialogen ändern wir nichts und klicken auf Next und schließlich auf Finish.

In die nun erstellte Datei fügen wir folgenden Inhalt ein.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:state_pressed="true"  >
        <shape>
            <solid
                android:color="#f3ae1b" />
            <stroke
                android:width="1dp"
                android:color="#bb6008" />
            <corners
                android:radius="3dp" />
            <padding
                android:left="10dp"
                android:top="4dp"
                android:right="10dp"
                android:bottom="4dp" />
        </shape>
    </item>
    <item>
        <shape>
            <gradient
                android:startColor="#f3ae1b"
                android:endColor="#bb6008"
                android:angle="270" />
            <stroke
                android:width="1dp"
                android:color="#bb6008" />
            <corners
                android:radius="4dp" />
            <padding
                android:left="10dp"
                android:top="4dp"
                android:right="10dp"
                android:bottom="4dp" />
        </shape>
    </item>
</selector>
```

Die Datei wurde im Zweig **res/drawable** gespeichert.



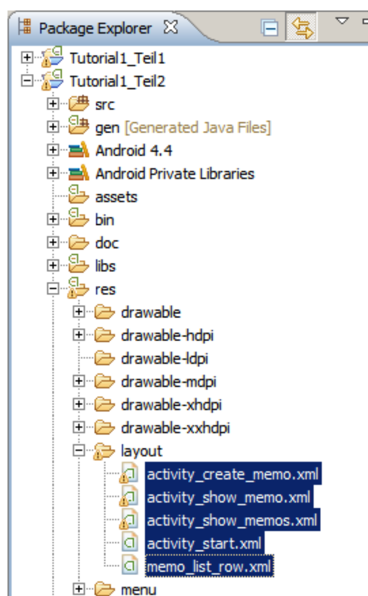

Als nächstes fügen wir in der Datei **res/values/strings.xml** noch folgende Zeilen ein. Damit definieren wir die Farben *color_beautifulgreen, color_beautifullightgreen, color_beautifuldarkgreen* und *color_white* als hexadezimale Werte an zentraler Stelle und müssen bei Bedarf nur hier etwas ändern.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">Notizenverwaltung</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="label">label</string>
    <string name="title_activity_show_memos">Notizen-Liste</string>
    <string name="title_activity_create_memo">Notiz anlegen</string>
    <string name="title_activity_show_memo">Notiz-Inhalt</string>

    <color name="color_beautifulgreen">#40700C</color>
      <color name="color_beautifullightgreen">#7CD11F</color>
      <color name="color_beautifuldarkgreen">#274508</color>
      <color name="color_white">#FFFFFF</color>

</resources>
```

Jetzt ändern wir das Layout unserer Buttons in den Views. Dazu ändern wir in allen Actvity-Dateien den Parameter `android:background` in `@drawable/buttonlayout1`.

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="70dp"
    android:layout_alignLeft="@+id/button2"
    android:layout_below="@+id/button2"
    android:layout_marginTop="16dp"
    android:background="@drawable/buttonlayout1"
    android:text="@string/label" />

<Button
    android:id="@+id/button2"
    android:layout_width="match_parent"
    android:layout_height="70dp"
    android:layout_alignLeft="@+id/textView1"
    android:layout_below="@+id/textView1"
    android:layout_marginTop="34dp"
    android:background="@drawable/buttonlayout1"
    android:text="@string/label" />
```

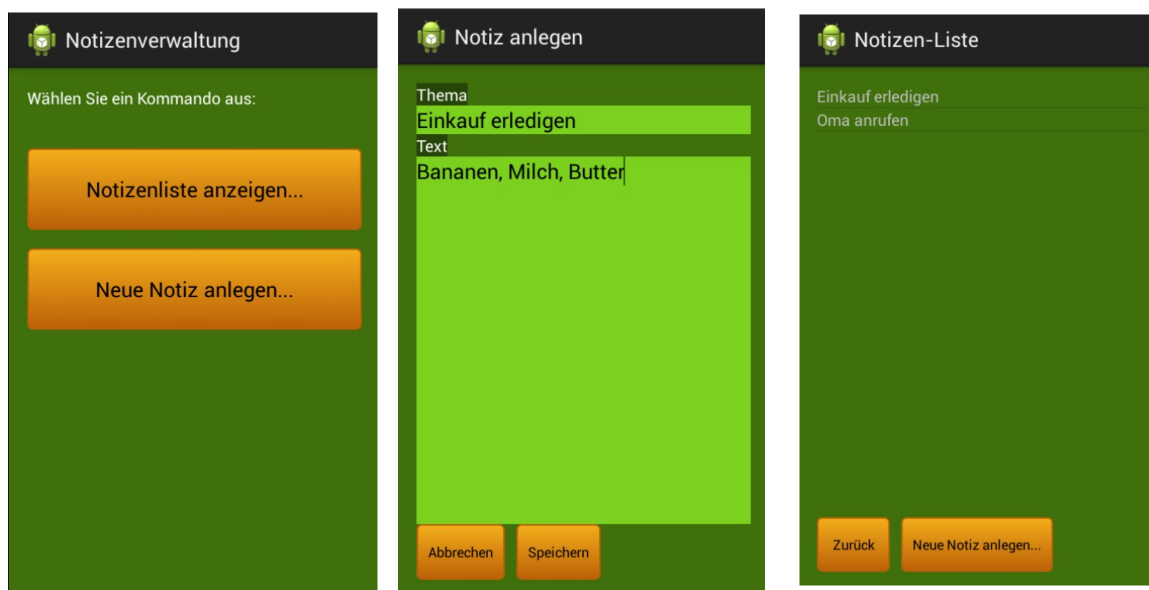
Diese Änderungen führen wir nun in allen restlichen Actvity-Dateien **activity_create_memo.xml**, **activity_show_memo.xml**, **activity_show_memos.xml** und **activity_start.xml** durch.



Danach ändern wir die Hintergrundfarben diverser Flächen in die definierten Farben aus der Datei strings.xml. Auch dies führen wir in allen Activity-Dateien durch.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:background="@color/color_beautifulgreen"
    tools:context=".ShowMemoActivity" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:background="@color/color_beautifullightgreen">

          <TextView
              android:id="@+id/textView1"
              android:background="@color/color_beautifuldarkgreen"
              android:textColor="@color/color_white"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:text="@string/hello_world" />
```

Wenn wir das alles geschafft haben, sieht die Notizen-App schon etwas ansehlicher aus.




Damit beenden wir das Kapitel und kümmern uns um das Thema Speicherung der Daten.

# Kapitel 3 – Daten in einer Datei speichern

Die Notizen sollen in einer Datei gespeichert werden und beim Start der App geladen und angezeigt werden. Wir müssen uns Gedanken zu folgenden Fragen machen.

- Format der Datei
- Dateinamen
- Speicherort
- Behandlung von Sonderfällen

Die Lösung der drei ersten Punkte ist einfach. Die Daten werden im CSV-Format in der Datei notizen.txt gespeichert. Der Ort der Dateiablage soll hier nur insofern interessieren, als dass er in einem abgesicherten App-Bereich liegt und dort keine andere App Zugang hat. Das Format ist wie folgt definiert. Die Datenfelder werden durch Komma getrennt und mit Doppelhochkommas (Gänsefüschen) eingeschlossen. Beispiel:

```
“Einkauf erledigen“,“Bananen, Milch, Butter“
“Oma anrufen“,“Oma hat um einen Rückruf ab 16:00 Uhr gebeten.“
```

Dabei gibt es folgende Besonderheiten zu beachten, die wir hier nur kurz erwähnen, aber später lösen müssen.

- Der erste Start der App: Datei existiert noch nicht und/oder sie ist leer
- Der Anwender gibt im Notizentext ein Komma ein. Dass dies zu einem Problem werden kann, sehen wir später

Nachdem das Dateiformat geklärt ist, erstellen wir im folgenden alle nötigen Java-Klassen.

## *Die Klasse Memo*

Zunächst erstellen wir eine Java-Klasse, die die Daten aufnimmt. Sie enthält einen String **theme** und einen String **memoText** und die Getter/Setter-Methoden. Wir erstellen ein Package **de.matletarium.android.tutorial1.bo**

```
package de.matletarium.android.tutorial1.bo;

public class Memo {
	private String theme=null;
	private String memoText=null;

	public Memo(String theme, String memoText){
		this.theme=theme;
		this.memoText=memoText;
	}
	public Memo(){
	}
	public String getTheme() {
		return theme;
	}
	public void setTheme(String theme) {
		this.theme = theme;
	}
	public String getText() {
		return memoText;
	}
```

```
	public void setText(String text) {
		this.memoText = text;
	}
}
```

## *Die Klasse DataController*

Dieser Controller hat folgende Aufgaben

- Laden und Speichern der Daten in die Datei
- Eine neue Notiz hinzufügen, löschen, ändern
- Eine Liste aller Notizen liefern
- kleine Utility-Methoden anbieten

Beginnen wir mit dem Initialisierungsteil.

```
public class DataController {

	private static DataController aDataController=null;
	private static final String TAG=DataController.class.getSimpleName();

	private ArrayList<Memo> memoList=null;
	private File appHomeDir=null;
	private String dataFile="memos.txt";

	private Memo curSelectedMemo=null;

	private DataController(){
	}

	/**
	 * Singleton
	 * @return
	 */
	public static DataController getInstance(){
		if(aDataController==null)
			aDataController=new DataController();
		return aDataController;
	}
```

In Zeile 3 definieren wir einen String namens **TAG**, mit dem wir später in die Logging-Konsole schreiben. Doch dazu später mehr. Anschließend definieren wir einige Variablen.

- memoList: Datenstruktur Liste mit unseren Memos
- appHomeDir: Pfad zu unserem privaten Verzeichnis, in das wir die Datei abspeichern
- dataFile: Name der Datei, in die wir die Memos abspeichern
- curSelectedMemo: das aktuell ausgewählte Memo

Als nächstes definieren wir den Konstruktor der Klasse privat, erstellen eine statische Methode getInstance und erzeugen somit das Entwurfsmuster eines Singelton. Der Controller wird in mehreren Activity-Controllern benötigt, jeoch nicht als eigenständige Instanz.

Dann erstellen wir die Methoden zum Setzen des AppHomeDir-Verzeichnis sowie zur Verwaltung der Memo-Liste (add, delete, getList, getCount).

```
	/**
	 * Setzt das Verzeichnis, in dem die App Dateien ablegen kann.
	 * @param aFile
	 */
	public void setAppHomeDir(File aFile){
		this.appHomeDir=aFile;
	}

	/**
	 * Fügt ein neues Memo der Liste hinzu.
	 * @param theme
	 * @param text
	 */
	public void addMemo(String theme, String text){
		Memo aMemo=new Memo(theme,text);
		memoList.add(aMemo);
	}

	/**
	 * Löscht das übergebene Memo aus der Liste
	 * @param aMemo
	 */
	public void deleteMemo(Memo aMemo){
		memoList.remove(aMemo);
	}

	/**
	 * Liefert die Liste aller Memos.
	 * @return
	 */
	public ArrayList<Memo> getMemoList(){
		return memoList;
	}

	/**
	 * Liefert die Anzahl der Memos in der Liste
	 * @return
	 */
	public int getMemoCount(){
		if(memoList==null)
			return 0;
		else
			return memoList.size();
	}
```

Später, wenn wir in der Activity Memo-Liste ein Memo auswählen, wechseln wir zur Ansicht Memo-Details. Damit wir in dieser Activity die Daten des Memos anzeigen können, merken wir sie uns. Dazu benötigen wir die beiden folgenden Methoden.

```
	/**
	 * @return the curSelectedMemo
	 */
	public Memo getCurSelectedMemo() {
		return curSelectedMemo;
	}

	/**
	 * @param curSelectedMemo the curSelectedMemo to set
	 */
	public void setCurSelectedMemo(Memo curSelectedMemo) {
		this.curSelectedMemo = curSelectedMemo;
	}
```

Die Methode zum Lesen der Daten aus der Datei ist unspektakulär. Einzig die Hilfsmethode **cleanCVSString** bedarf einer besonderen Erklärung.

```
	/**
	 * Liest aus einer Datei alle Memos
	 * @throws ApplicationException
	 */
	public void readMemoFile() throws ApplicationException{
		File memoFile=new File(appHomeDir,dataFile);
		String zeile=null;
		int zeilenCounter=0;
		String [] data;
		Memo aMemo=null;

		memoList=new ArrayList<Memo>(50);
		if(!memoFile.exists())
			return;
		try {
			BufferedReader in = new BufferedReader( new FileReader(memoFile) );
			while(true){
				zeile=in.readLine();
				if(zeile==null) break;//Dateiende erreicht
				zeilenCounter++;
				try{
					Log.d(TAG, "Zeile1"+zeile);
					data=cleanCVSString(zeile);
					Log.d(TAG, "Zeile2"+zeile);
					aMemo=new Memo();
					aMemo.setTheme(data[0].substring(1,
						data[0].length()-1));
					aMemo.setText(data[1].substring(1,
						data[1].length()-1));
					memoList.add(aMemo);
```

```
                        }
                      catch(ArrayIndexOutOfBoundsException e){
                        Log.e(TAG,
            "Daten konnten nicht gelesen werden. Int. Fehler:"+e.getMessage());
                        }catch(StringIndexOutOfBoundsException e){
                               Log.e(TAG,
             "Daten konnten nicht gelesen werden. Int. Fehler:"+e.getMessage());
                        }
                 }
                 in.close();
                 Log.d(TAG, "gelesene Memos:"+zeilenCounter);

          } catch (FileNotFoundException e) {
                 throw new ApplicationException("Daten konnten nicht gelesen werden.
Int. Fehler:"+e.getMessage());
          } catch (IOException e) {
                 throw new ApplicationException("Daten konnten nicht gelesen werden.
Int. Fehler:"+e.getMessage());
          }

      }
```

Zunächst zur Methode readMemoFile. Wir erstellen eine leere Liste mit einem Anfangsplatz für 50 Einträgen. Dann bilden wir aus AppHomeDir und Dateiname den Pfad zur Datei. Wenn diese nicht existiert, müssen wir auch nichts einlesen und verlassen die Read-Methode wieder. Wenn sie existiert, lesen wir die Datei zeilenweise und speichern die Daten in ein neu zu erzeugendes Memo-Objekt, welches anschließend in die Liste abgelegt wird.

Ein Problem beim Format CSV entsteht, wenn dort auch Zeichen erlaubt sind, die gleichzeitig zur Trennung der einzelnen Datenfelder verwendet werden. An einem Beispiel soll dies verdeutlicht werden. Die Datenfelder werden durch das Komma getrennt. In der folgenden Zeilen wurden jetzt nicht zwei Datenfelder gelesen werden, sondern vier. Denn zwischen Banane, Milch und Butter befindet sich ebenfalls ein Komma.

```
“Einkauf erledigen“,“Bananen, Milch, Butter“
```

Dieser Sonderfall wird in der Hilfsmethode **cleanCVSString** behandelt. Ein Feld ist immer dann korrekt gelesen, wenn auch vorne und hinten ein Doppelhochkomma vorhanden ist. Ist dies nicht der Fall, befinden wir uns noch innerhalb des Token und lesen das nächste Stückchen. Dabei behandeln wir vier Fälle

- Fall 1 “Daten“: Im Normalfall sind keine Kommas in den Datenfeldern
- Fall 2 “Daten,: Es befindet sich ein Komma in den Daten, jedoch kein abschließendes Doppelhochkomma.
- Fall 3 ‚Daten“: Es befindet sich ein Komma in den Daten, jedoch kein beginnendes Doppelhochkomma.
- Fall 4 ‚Daten,: Wie Fall 2 und 3. Hier fehlen Doppelhochkommas vorne und hinten.

```
	/**
	 * Bereinigt eine gelesene Zeile, damit die CSV-Spalten richtig
	   verarbeitet werden
	 * und auch Kommas möglich sind.
	 * @param str Zeile der Art "ab,c,r,o","D"EF"","xyz"
	 * @return String-Array mit korrekt getrennten Tokens
	 */
	public String[] cleanCVSString(String str){
		String fieldDelimiter="\"";
		String delimiter=",";
		String[] result;
		String[] tmp;
		String stack=null;
		int resultTokenindex=0;
		tmp=str.split(delimiter);
		result=new String[tmp.length];
		for(int tmpIndex=0;tmpIndex<tmp.length;tmpIndex++){


if(tmp[tmpIndex].startsWith(fieldDelimiter)&&tmp[tmpIndex].endsWith(fieldDelimiter))
{// Fall 1: "Token"
					result[resultTokenindex]=tmp[tmpIndex];
					resultTokenindex++;
				}
				if(tmp[tmpIndex].startsWith(fieldDelimiter)&&!
tmp[tmpIndex].endsWith(fieldDelimiter)){ // Fall 2: "Token
					stack=tmp[tmpIndex];
					stack=stack+delimiter;
				}
				if(!
tmp[tmpIndex].startsWith(fieldDelimiter)&&tmp[tmpIndex].endsWith(fieldDelimiter)){
// Fall 3: Token"
					result[resultTokenindex]=stack+tmp[tmpIndex];
					stack=null;
					resultTokenindex++;
				}
				if(!tmp[tmpIndex].startsWith(fieldDelimiter)&&!
tmp[tmpIndex].endsWith(fieldDelimiter)){  // Fall 4:  token
					stack=stack+tmp[tmpIndex];
					stack=stack+delimiter;
				}
		}
		tmp=null;
		return result;
	}
```

Was jetzt noch bleibt ist die Speichern-Methode. Hierin iterieren wir durch die Memo-Liste und schreiben die Datensätze formatgetreu in die Datei.

```
	/**
	 * Speichert die Memos in eine Datei
	 * @throws ApplicationException
	 */
	public void saveMemoFile() throws ApplicationException {
		File memoFile=new File(appHomeDir,dataFile);
		int zeilenCounter=memoList.size();
		Memo aMemo=null;

		try {
			BufferedWriter out = new BufferedWriter( new 
FileWriter(memoFile) );
			for(int index=0;index<zeilenCounter;index++){
				aMemo=memoList.get(index);
				out.write("\"");
				out.write(aMemo.getTheme());
				out.write("\",\"");
				out.write(aMemo.getText());
				out.write("\"\n");
			}
			out.close();
			Log.d(TAG, "geschriebene Memos:"+zeilenCounter);
		}  catch (FileNotFoundException e) {
			throw new ApplicationException("Daten konnten nicht geschrieben 
werden. Int. Fehler:"+e.getMessage());
		} catch (IOException e) {
			throw new ApplicationException("Daten konnten nicht geschrieben 
werden. Int. Fehler:"+e.getMessage());
		}
	}
```

Die Exception-Klasse, mit der wir alle anderen Exceptions kapseln, ist wie folgt definiert.

```
public class ApplicationException extends RuntimeException {

	private String mesg=null;
	public ApplicationException(String mesg){
		this.mesg=mesg;
	}

	/**
	 * @return the mesg
	 */
	public String getMesg() {
		return mesg;
	}
}
```

Die Verwendung des DataControllers erfolg später, wenn wir uns mit den Activity-Controllern beschäftigen.

# Kapitel 4 – Der List-Adapter

In diesem Kapitel geht es darum, die Daten einer Liste mit der Darstellung der Android-Oberfläche zu verbinden. Die ListView in der ShowMemoActivity muss mit einem ArrayAdapter gefüllt werden.

Wir nennen unsere Klasse natürlich nach unserem Thema Memo und erben von ArrayAdapter. Darin defnieren wir unsere Datenstruktur als ArrayList mit dem Datentyp Memo. Außerdem brauchen wir den Context und die Resource.

```
public class MemolistAdapter extends ArrayAdapter<Memo> {

	private ArrayList<Memo> data=null;
	private Context context;
	private int resource;

```

Dem Konstruktor werden wir, wie wir später noch sehen werden,  den Context, die Resource und vor allem unsere Datenliste mitgeben.

```
public MemolistAdapter(Context context, int resource, ArrayList<Memo> memoList) {
		super(context, resource, memoList);
		this.data=memoList;
		this.context = context;
		this.resource=resource;
}
```

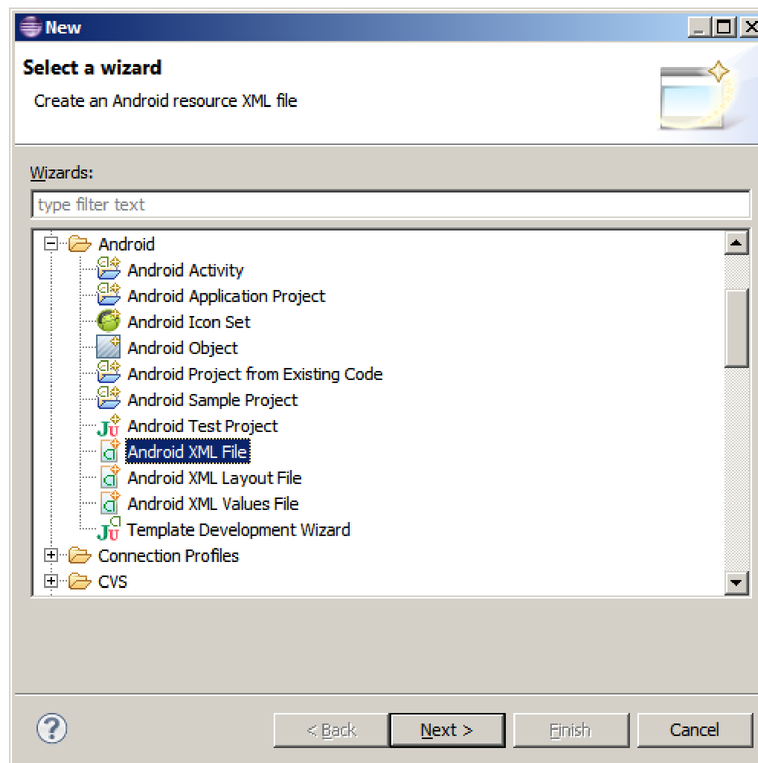
Wenn dann später Android die Daten verlangt, um sie in der View darzustellen, wird immer die folgende Methode **getView** aufgerufen. Wichtig ist der Wert **position**, denn damit greifen wir in der Datenliste auf den richtigen Datensatz zu. Da der Datensatz aus der Klasse Memo besteht, hat er auch die Methode **getTheme()**. Diesen String liefern wir an die **textView1**.

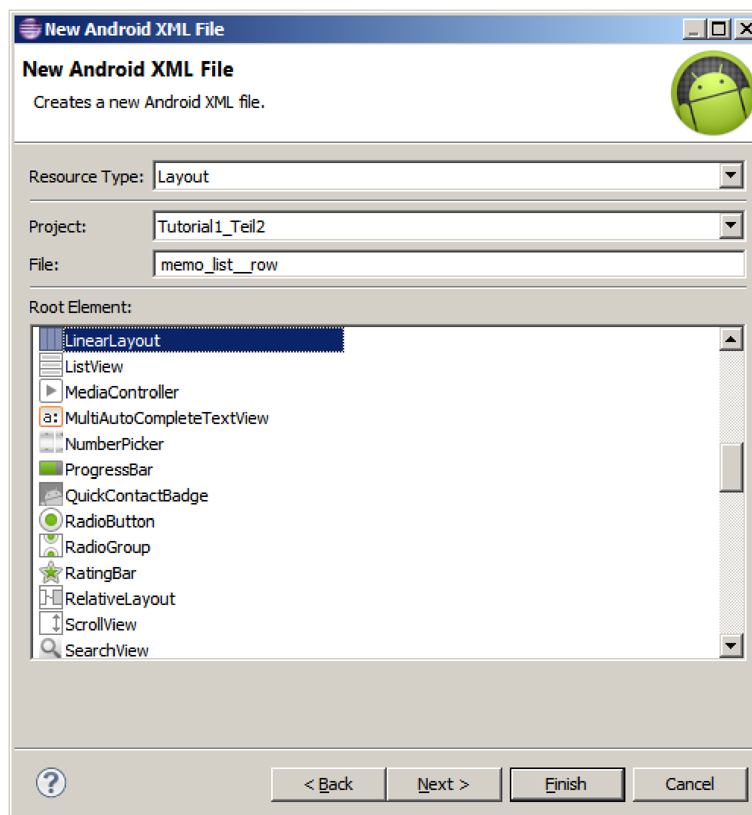
```
	@Override
	public View getView(int position, View convertView, ViewGroup parent) {
		if (convertView == null) {
			LayoutInflater inflater =
(LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
		 convertView = inflater.inflate(resource, parent,false);
		}
		TextView textView1= (TextView)convertView.findViewById(R.id.textView1);
		textView1.setText(data.get(position).getTheme());

		return convertView;
	}

```

Wichtig ist noch die folgende Methode, die die Anzahl der Datensätze zurückliefert.

```
public int getCount() {
		return data.size();
}
```

Um die Zeilen einer Liste schöner zu gestalten, benötigen wir auch dazu ein Layout.
Dazu erstellen wir eine neue XML-Datei namens **memo_list_row.xml** mit dem Root-Element **LinearLayout**.



In die erzeugte Datei fügen wir folgende XML-Zeilen ein.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

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

</LinearLayout>
```

Wie wir nun dieses Layout einem Eintrag einer Liste zuweisen, sehen wir später.

# Kapitel 5 – Erweiterung der Activity-Controller

Jetzt sind alle Vorraussetzungen geschaffen, damit Daten gespeichert und geladen werden können. Nun bauen wir die restlichen Controller so um, dass sie diese Funktionen auch benutzen können. Die folgenden Erweiterungen beinhalten nicht die Code-Teile aus dem Tutorial-Teil-1, sondern nur die Kommandos und Variablen, die zusätzlich eingefügt werden müssen.

## *Erweiterung der Start-Activity*

Fangen wir wieder mit der Start-Activity an. Zunächst besorgen wir uns eine Instanz des DataControllers. Zu Debug-Zwecken defineiren wir auch wie bereits weiter oben beschrieben ein TAG, welches wir beim Logging verwenden.

```
public class StartActivity extends Activity {

	DataController aDataController=DataController.getInstance();
	private static final String TAG=StartActivity.class.getSimpleName();
```

Jetzt fügen wir den Codeteil ein, der die Daten aus der Datei liest. Das macht der DataController für uns. Vorher sagen wir dem Controller den Pfad zu unserem Home-Verzeichnis, welches wir mit der Methode getFilesDir() aus unserer Activity herausfinden können.

```
protected void onCreate(Bundle savedInstanceState) {

		// 4. Daten laden
		aDataController.setAppHomeDir(getFilesDir());
		aDataController.readMemoFile();
```

Zuletzt müsen wir noch auf das Ende der App reagieren. Wenn der Anwender die App schließt, sollen auch die Daten in die Datei gespeichert werden.

```
	@Override
	protected void onStop() {
		super.onStop();
		Log.d(TAG, "onStop");
		aDataController.saveMemoFile();
	}
```

### Erweiterung der ShowMemosActivity

Diese View zeigt eine Liste aller Memos an. Hier kommt jetzt die Klasse MemolistAdapter zum Einsatz. Ebenso benötigen wir den DataController. Wir wir später noch sehen werden, implementieren wir auch  OnItemClickListener.

```
public class ShowMemosActivity extends Activity implements OnItemClickListener {

	private MemolistAdapter aMemolistAdapter=null;
	private DataController aDataController=DataController.getInstance();
	private static final String TAG=ShowMemosActivity.class.getSimpleName();
```

Zunächst warten wir, bis die Activity angezeigt wird. Dabei wird **onStart()** aufgerufen. Hierin rufen wir **refreshData()** auf.

```
	@Override
	protected void onStart() {
		super.onStart();
		Log.d(TAG, "onStart");
		refreshData();
		aDataController.setCurSelectedMemo(null);
	}
```

In **refreshData()** erzeugen wir eine Instanz des  MemolistAdapter und geben ihm neben dem Context und der Resource auch eine Liste mit Daten mit, die uns der DataController liefert. Diesen Adapter verbinden wir per **setAdapter()** mit der ListView.
Damit wir mitbekommen, wenn der Anwender einen Listeneintrag antippt, geben wir der ListView per **setOnItemClickListener()** auch einen Listener mit.

```
	private void refreshData(){
		aMemolistAdapter = new MemolistAdapter(getApplicationContext(),
					R.layout.memo_list_row,aDataController.getMemoList());
		aListView.setAdapter(aMemolistAdapter);
		aListView.setOnItemClickListener(this);
	}
```

Dem Konstruktor des MemoListAdapters geben wir auch das Layout **memo_list_row** der Listenzeile mit, die wir oben definiert haben. Wenn der Anwender jetzt einen Eintrag auswählt, wird durch Android die Methode **onItemClick** aufgerufen.

```
@Override
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
	Intent i= new Intent(ShowMemosActivity.this,ShowMemoActivity.class);
	aDataController.setCurSelectedMemo((Memo)aListView.getItemAtPosition(position));
	startActivity(i);
}
```

Darin merken wir uns den aktuel ausgewählten Listeneintrag im DataController und starten anschließend die ShowMemoActivity.

## *Erweiterung der ShowMemoActivity*

Die Erweiterungen in diesem ViewController sind klein.

```
public class ShowMemoActivity extends Activity {

      private DataController aDataController=DataController.getInstance();
      private static final String TAG=ShowMemoActivity.class.getSimpleName();
```

Wenn die Activity aufgerufen wird, rufen wir **refreshData()** auf.

```
@Override
protected void onStart() {
      super.onStart();
      Log.d(TAG, "onStart");
      refreshData();
}
```

Dort prüfen wir, ob ein Datensatz ausgewählt wurde. Die erfahren wir vom DataController. Wenn dem so ist, stellen wir die Daten in den jeweiligen Feldern der View dar.

```
private void refreshData(){
   if(aDataController.getCurSelectedMemo()!=null){
      textViewTheme.setText(aDataController.getCurSelectedMemo().getTheme());
      textViewMemotext.setText(aDataController.getCurSelectedMemo().getText());
   }
}
```

## *Erweiterung der CreateMemoActivity*

Diese Activity verwenden wir für zwei unterschiedliche Aufgaben, einmal zum erfassen einer neuen Notiz, zum anderen um eine bestehende Notiz zu ändern.

```
public class CreateMemoActivity extends Activity {

      DataController aDataController=DataController.getInstance();
      private boolean changeMode=false;
```

# Kapitel 6 – Übungsaufgaben

Hier definiere ich einige Übungsaufgaben, die man jetzt lösen kann, um das Gelernte zu vertiefen.

Aufgabe 1:

Das Aussehen einer Zeile in der Memoliste soll noch verschönert werden. Das Ziel ist ein gelber Hintergrund pro Zeile mit dunkelgrüner Schrift. Somit sind die Zeilen direkt erkennbar und der Anwender erkennt wegen der identischen Farben zu den Buttons, dass man durch einen Klick auf den Eintrag eine weitere Funkton aufrufen kann.

Aufgabe 2: Innerhalb der Activity Memo-Liste soll die Anzahl der Einträge angezeigt werden.

## *Lösungen Aufgabe 1*

Zuerst ist eine Farbe zu definieren. Dies erfolgt in der Datei res/values/strings.xml. Dazu fügen wir zum Beispiel folgende Zeile hinzu.

```
<color name="color_beautifulyellow">#f3ae1b</color>
```

Dann fügen wir die Farbdefinition dem Row-Layout hinzu. Wir fügen folgende Zeilen in die Datei **memo_list_row.xml**

```
<TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@color/color_beautifulyellow"
        android:textColor="@color/color_beautifuldarkgreen"
        android:text="TextView" />
```

Damit ändern wir den Hintergrund in gelb und die Textfarbe in dunkelgrün.

## *Lösungen Aufgabe 2*

Wir zeigen die Anzahl im Titel der View und ändern im Controller **ShowMemosActivitiy** die Methode **refreshData()** wie folgt

```
private void refreshData(){
      String oldTitle=null;
      aMemolistAdapter = new
              MemolistAdapter(getApplicationContext(),R.layout.memo_list_row,
                              aDataController.getMemoList());
      aListView.setAdapter(aMemolistAdapter);
      aListView.setOnItemClickListener(this);
      oldTitle=getTitle().toString();
      setTitle(oldTitle+"  ("+aDataController.getMemoCount()+")");
}
```

Zuerst lesen wir den Titel der View, fügen diesem die Anzahl der Datensätze hinzu, die uns der DataController liefert und schreiben den Titel zurück in die View.