Android – Eine Einführung

Einstellungen, SQLite, Dateisystem & ContentProvider

Andreas Wilhelm

CSC Computer-Schulung & Consulting GmbH

Contents

Einstellungen

Contents

Überblick

- Anpassung von Verhalten und Aussehen der Applikation durch den Anwender
- Erstellung von Oberflächen mit Android *Preference*-API
- Deklaration von Einstellungen mit XML
- Zugriff auf Einstellungen mit *SharedPreferences*
- Darstellung mit PreferenceActivity bzw. -Fragment
- Ableiten eigener Einstellungen

PreferenceActivity

Bis Android 3.0 wurden Einstellungen über eine von *PreferenceActivity* abgeleitete Activity angezeigt. Seit Android 3.0 verwendet man vorzugsweise *PreferenceFragments*.

Boolean	Float	Int
Long	String	String Set

Tabelle: Typen von Einstellungen

Überblick

- Deklaration in XML-Datei (*res/xml/*)
- Alternativ können Einstellungen auch zur Laufzeit hinzugefügt werden
- Wurzelknoten muss ein *PreferenceScreen* sein
- Unterhalb des Wurzelnkotens Verschachtelung von *Preference* Objekten

Klasse	Beschreibung
CheckBoxPreference	Checkbox-Eintrag – Einstellungen mit boolschem Wert
EditTextPreference	Einstellungen als Text
ListPreference	Auswahlliste mit nur einer wählbaren Option
MultiSelectListPreference	Auswahlliste mit mehreren wählbaren Optionen
SwitchPreference	Speichern eines boolschen Werts
RingtonePreference	Auswahl eines der auf dem Gerät verfügbaren Klingeltöne
DialogPreference	Dialog über den Einstellungen vorgenommen werden können
PreferenceCategory	Gruppiert Einstellungen unter gemeinsamen Abschnitt
PreferenceScreen	Gruppiert Einstellungen unter gemeinsamen Menüpunkt

Attribute

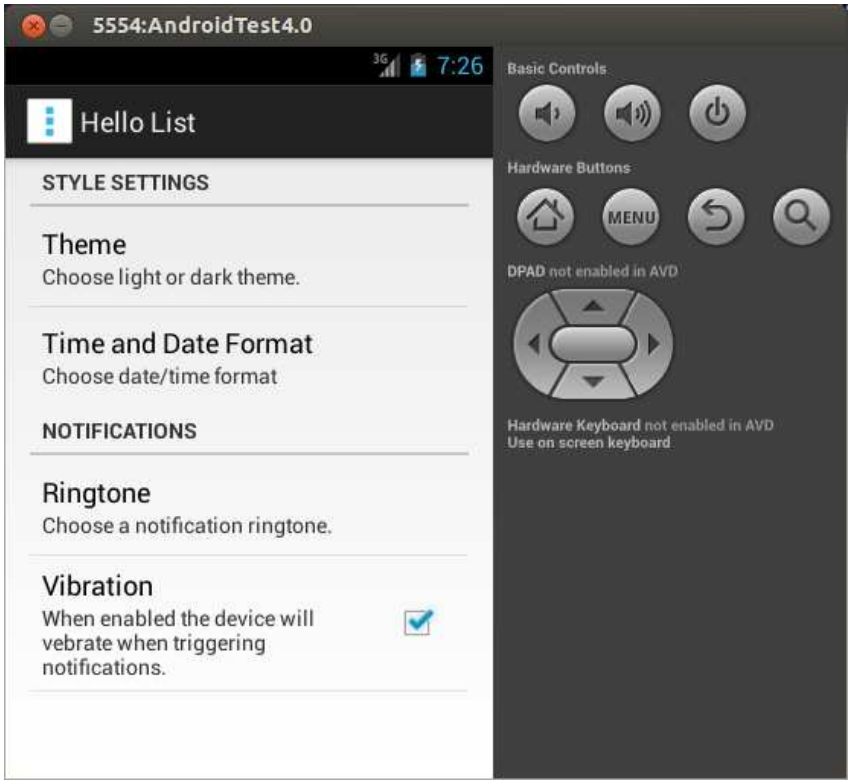
Attribut	Beschreibung
android:key	Schlüssel über den Einstellung erreicht werden kann. Anders als sonst üblich wird hier keine Android-ID sondern eine Zeichenkette verwendet. Elemente, die nicht vom Typ *PreferenceCategory* der *PreferenceScreen* sind oder ein Intent oder ein Fragment starten, müssen diesen Wert setzen.
android:title	Der Titel der Einstellung
android:summary	Ein kurzer Text mit Zusatzinformationen
android:defaultValue	Der voreingestellte Standard-Wert dieser Einstellung
android:fragment	Deklariert ein anzuzeigendes Fragment

Beispiel

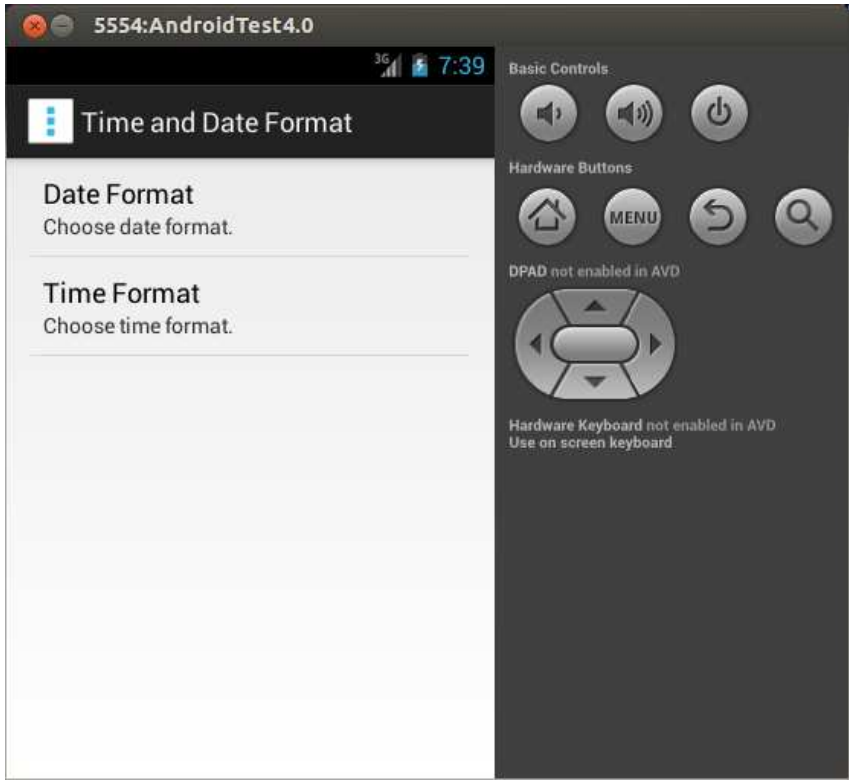
```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >

    ...

    <PreferenceScreen
        android:summary="@string/lblSettingsDatetimeSummary"
        android:title="@string/lblSettingsDatetime" >
        <ListPreference
            android:dialogTitle="@string/lblSettingsDateTitle"
            android:key="dateFormat"
            android:summary="@string/lblSettingsDateSummary"
            android:title="@string/lblSettingsDateTitle" />

        ...
    </PreferenceScreen>

    <PreferenceCategory android:title="@string/lblSettingsNotifications" >
        <RingtonePreference
            android:key="notificationRingtone"
            android:ringtoneType="notification"
            android:summary="@string/lblSettingsNotificationRingtoneSummary"
            android:title="@string/lblSettingsNotificationRingtone" />

        <CheckBoxPreference
            android:defaultValue="false"
            android:key="notificationVibrate"
            android:summary="@string/lblSettingsNotificationVibrateSummary"
            android:title="@string/lblSettingsNotificationVibrate" />
    </PreferenceCategory>
</PreferenceScreen>
```

Listing : Deklaration der Einstellungen

Screenshots



(a) Startansicht der Einstellungen



(b) Der Datetime PreferenceScreen

Abbildung: Die Einstellungen

Überblick

- Direkte Verwendung von *Preference*
- Deklaration eines impliziten Intents

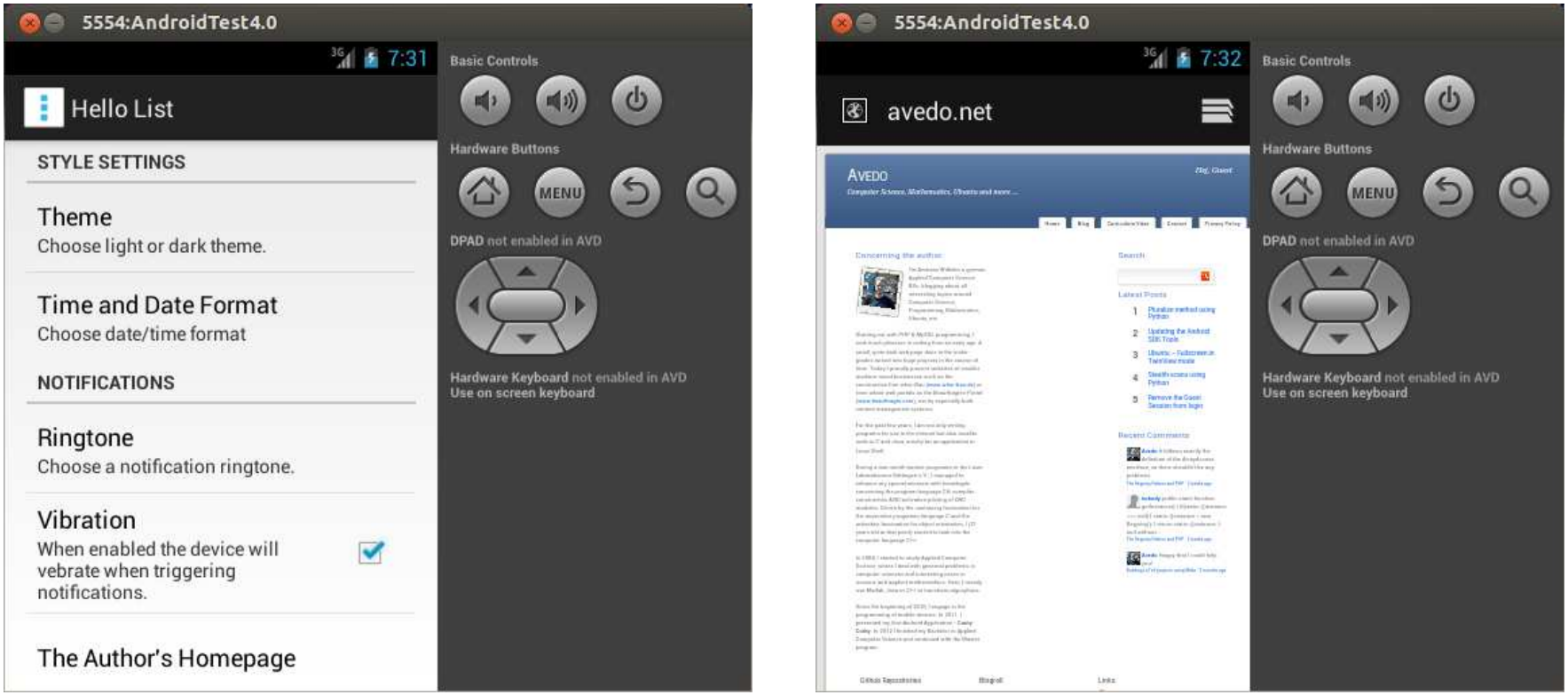
Attribut	Beschreibung
android:action	Die Aktion, die das Intent auslösen soll
android:data	Daten, die über das Intent weitergegeben werden sollen
android:mimeType	Der zu verwendende MIME Typ
android:targetClass	Der Name der Zielklasse (der Activity)
android:targetPackage	Der Paket der Zielklasse (der Activity)

Implementierung

```
...

<Preference android:title="@string/prefs_web_page" >
    <intent android:action="android.intent.action.VIEW"
            android:data="http://www.example.com" />
</Preference>

...
```

Listing : Einstellungen und Intents

Screenshots



(a) Der Intent-Eintrag in den Einstellungen

(b) Die Seite im Browser

Abbildung: Einstellungen und Intents

PreferenceActivity

- Bis Android 3.0 Einsatz von *PreferenceActivity*
- Laden der Einstellungen in *onCreate()*
- Zuweisen eines Layouts nicht nötig
- Einstellungen werden automatisch gespeichert
- Dynamisches Laden mit *findPreference()*

```
public class SettingsActivity extends PreferenceActivity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.preferences);
   }
}
```

Listing : Implementierung einer PreferenceActivity

PreferenceFragment

- Seit Android 3.0 Einsatz von *PreferenceFragments*
- Implementierung wie bei *PreferenceActivity*

```
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Listing : Implementierung eines PreferenceFragments

Laden der Fragments

```
public class SettingsActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      getFragmentManager().beginTransaction()
         .replace(android.R.id.content, new SettingsFragment())
         .commit();
   }
}
```

Listing : Einbinden des PreferenceFragments

Überblick

- Setzen von Standard-Werten für *SharedPreferences*
- Zuweisung über Attribut *android:defaultValue*
- Initialisierung mit *setDefaultValues()* in allen *onCreate()* Methoden von Activities, die als Einstiegspunkte dienen

```
PreferenceManager.setDefaultValues(this, R.xml.preferences, false);
```

Listing : Initialisierung der SharedPreferences

Die Methode setDefaultValues()

Die Methode *setDefaultValues()* nimmt drei Argumente entgegen. Die ersten beiden sind der Kontext und die Deklaration der Einstellungen, die in den SharedPreferences initialisiert werden sollen. Der dritte Wert legt allerdings fest, ob die bisherigen Benutzereinstellungen überschrieben werden sollen oder nicht. Steht dieser Wert auf *true* werden alle bisherigen Einstellungen des Benutzers gelöscht und mit den Standard-Einstellungen überschrieben.

Überblick

- Umfangreiche Einstellungen recht unübersichtlich
- Bisher Lösung durch verschachtelte *PreferenceScreens*
- Seit Android 3.0 Lösung über *Header*
- Vorteil: Header erzeugen auf größeren Bildschirmen ein Zwei-Spalten-Layout

```
<preference-headers
    xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="math.elearning.hello.list.GeneralSettings"
        android:summary="@string/lblSettingsGeneralSummary"
        android:title="@string/lblSettingsGeneral" />
    <header
        android:fragment="math.elearning.hello.list.DatetimeSettings"
        android:summary="@string/lblSettingsDatetimeSummary"
        android:title="@string/lblSettingsDatetime" />
</preference-headers>
```

Listing : Einstellungen kategorisieren

Laden der Header

```
public class SettingsActivity extends PreferenceActivity {
   final static String ACTION_PREFS_GENERAL = "math.elearning.hello.list.GeneralSettings";
   final static String ACTION_PREFS_DATETIME = "math.elearning.hello.list.DatetimeSettings";

   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);

       String action = getIntent().getAction();
       if (action != null && action.equals(ACTION_PREFS_GENERAL)) {
           addPreferencesFromResource(R.xml.preferences_general);
       } else if(action != null && action.equals(ACTION_PREFS_DATETIME)) {
           addPreferencesFromResource(R.xml.preferences_datetime);
       } else if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
           // Load the legacy preferences headers.
           addPreferencesFromResource(R.xml.preference_headers_legacy);
       }
   }

   @Override
   public void onBuildHeaders(List<Header> target) {
      loadHeadersFromResource(R.xml.preference_headers, target);
   }
}
```

Listing : Initialisierung der SharedPreferences

Hinweis

Kompatibilität

Da Header erst in Android 3.0 eingeführt wurden, muss man eine gesonderte Behandlung für ältere Geräte einfügen. Anhand der Version können unterschiedliche Deklarationen der Einstellungen geladen werden (siehe Listing). Die Methode *onBuildHeaders()* wird dann in älteren Versionen einfach ignoriert.

Generische Fragments

- Bisher Deklaration einzelner Fragments
- Implementierung eines generischen Fragments
- Übergabe von Inhalten mit *<extras>*-Element

```
<preference-headers
    xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="math.elearning.hello.list.SettingsFragment"
        android:summary="@string/lblSettingsGeneralSummary"
        android:title="@string/lblSettingsGeneral">
          <extra android:name="settings" android:value="general" />
     </header>
    <header
        android:fragment="math.elearning.hello.list.SettingsFragment"
        android:summary="@string/lblSettingsDatetimeSummary"
        android:title="@string/lblSettingsDatetime">
          <extra android:name="settings" android:value="datetime" />
     </header>
</preference-headers>
```

Listing : Einstellungen kategorisieren

Behandlung von generischen Fragments

```
public static class SettingsFragment extends PreferenceFragment {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      String settings = getArguments().getString("settings");

      if ("general".equals(settings)) {
         addPreferencesFromResource(R.xml.preferences_general);
      } else if ("datetime".equals(settings)) {
         addPreferencesFromResource(R.xml.preferences_datetime);
      }
   }
}
```

Listing : Laden der Preferences

Überblick

- Einstellungen einer Applikation werden in einer Datei gespeichert
- Zugriff über Klasse *SharedPreferences*
- Verwaltung durch *PreferenceManager*
- Laden mit Methode *getDefaultSharedPreferences()*

```
// Load the shared preferemces ...
SharedPreferences sharedPref = PreferenceManager.getDefaultSharedPreferences(this);

// ... and fetch a single entry.
String dateFormat = sharedPref.getString(SettingsActivity.DATE_FORMAT, "");
```

Listing : Laden der Einstellungen

Aktualisierungen

- Änderungen in Einstellungen nicht sofort berücksichtigt
- Implementierung eines Listeners

```
public class UbuntuReleaseList extends ListActivity
        implements OnSharedPreferenceChangeListener {

   public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {
      if (key.equals(KEY_PREF_DATETIME)) {
         // Force an update.
      }
   }

   @Override
   protected void onResume() {
      super.onResume();
      getPreferenceScreen().getSharedPreferences()
         .registerOnSharedPreferenceChangeListener(this);
   }

   @Override
   protected void onPause() {
      super.onPause();
      getPreferenceScreen().getSharedPreferences()
         .unregisterOnSharedPreferenceChangeListener(this);
   }
}
```

Listing : Reagieren auf Änderungen an den Einstellungen

Überblick

- Grundlegende Einstellungswidgets in Android enthalten
- Weitere Widgets können selbst implementiert werden
- Beispiel: NumberPicker
- Typischerweise Ableitung von *DialogPreference*
- Bei Ableitung von *Preference* Implementierung von *onClick()* Methode

NumberPickerPreference

```
public class NumberPickerPreference extends DialogPreference {
   public NumberPickerPreference(Context context, AttributeSet attrs) {
      super(context, attrs);

      setDialogLayoutResource(R.layout.numberpicker_dialog);
      setPositiveButtonText(android.R.string.ok);
      setNegativeButtonText(android.R.string.cancel);

      setDialogIcon(null);
   }

   ...
}
```

Listing : Konstruktor des Einstellungselements

Speichern der Einstellung

- Speichern der Einstellungen mit *persist*()* Methoden
- Automatisches Übertragen in *SharedPreferences*

```
public class NumberPickerPreference extends DialogPreference {
   private Integer value;

   ...

   @Override
   protected void onDialogClosed(boolean positiveResult) {
      if (positiveResult) {
         persistInt(value);
      }
   }
}
```

Listing : Speichern der Einstellungen

Laden der Einstellung

- Initialisierung mit Methode *onSetInitialValue()*
- Laden von Einstellung mit Methode *getPersisted*()*

```
public class NumberPickerPreference extends DialogPreference {
   private static int DEFAULT_VALUE = 0;

   ...

   @Override
   protected void onSetInitialValue(boolean restorePersistedValue, Object defaultValue) {
      if (restorePersistedValue) {
         value = this.getPersistedInt(DEFAULT_VALUE);
      } else {
         value = (Integer) defaultValue;
         persistInt(value);
      }
   }
}
```

Listing : Laden der Einstellungen

Standard-Wert

Man kann den an die Methode *onSetInitialValue()* übergebenen Standard-Wert nicht verwenden, denn falls das Flag *restorePersistedValue* gesetzt wurde, ist dieser Wert automatisch *null*.

Setzen des Standard-Werts

- Implementierung des Setzens mit *android:defaultValue* über Methode *onGetDefaultValue()*
- Auch wenn Methode Standard-Wert übergeben bekommt muss eigener Wert bereitgestellt werden

```
public class NumberPickerPreference extends DialogPreference {
   ...

   @Override
5  protected Object onGetDefaultValue(TypedArray a, int index) {
      return a.getInteger(index, DEFAULT_VALUE);
   }
}
```

Listing : Laden der Standard-Einstellungen

Speichern des aktuellen Status

- Einstellungen gehen beim Neustart (Rotation des Geräts) verloren
- Methode *onSaveInstanceState()* speichert aktuellen Status
- Methode *onRestoreInstanceState()* lädt aktuellen Status

```
public class NumberPickerPreference extends DialogPreference {
   ...

   @Override
   protected Parcelable onSaveInstanceState() {
      final Parcelable superState = super.onSaveInstanceState();

      // Skip if preference is not persisted.
      if (isPersistent()) return superState;

      // Otherwise setup the instance of the custom NumberPickerState.
      myState.value = value;

      return new NumberPickerState(superState);
   }

   @Override
   protected void onRestoreInstanceState(Parcelable state) {
      // Load state of superclass if state was not saved.
      if (state == null || !state.getClass().equals(SavedState.class)) {
         super.onRestoreInstanceState(state);
         return;
      }

      // Pass the state to the super class and update the local value.
      NumberPickerState myState = (NumberPickerState) state;
      super.onRestoreInstanceState(myState.getSuperState());
      value = myState.value;
   }
}
```

Den Status verwalten

- Implementierung durch Ableitung von *Preference.BaseSavedState*
- Überschreiben einiger Methoden
- Implementierung eines Creators

Standard-Wert

Ein Creator ist eine Klasse, die das *Creator* Interface implementiert. Sie müssen in einer Klasse, die das Parcelable Interface implementiert, als öffentlich zugreifbares Feld *CREATOR* hinterlegt werden, dass dazu verwendet wird Instanzen dieser Klasse von einem *Parcel* zu erzeugen.

Eine eigene Status Klasse

```
private static class NumberPickerState extends BaseSavedState {
   int value;

   public NumberPickerState(Parcelable superState) {
      super(superState);
   }

   public NumberPickerState(Parcel source) {
      super(source);
      value = source.readInt();
   }

   @Override
   public void writeToParcel(Parcel dest, int flags) {
      super.writeToParcel(dest, flags);
      dest.writeInt(value);
   }

   // Standard creator object.
   public static final Parcelable.Creator<NumberPickerState> CREATOR =
         new Parcelable.Creator<NumberPickerState>() {

      public NumberPickerState createFromParcel(Parcel in) {
         return new NumberPickerState(in);
      }

      public NumberPickerState[] newArray(int size) {
         return new NumberPickerState[size];
      }
   };
}
```

Listing : Das NumberPicker Status-Objekt

Interner Speicher

Contents

Allgemeines

- Dateien im internen Speicher meistens nur von erstellenden Applikation nutzbar
- Dateien werden beim Deinstallieren der Applikation gelöscht
- Schreiben und Lesen von Dateien über Java-Streams
- Öffnen der Streams über Kontext der Applikation (*openFileOutput()* und *openFileInput()*)

Datei schreiben

Beim Öffnen des Ausgabe-Streams mit *openFileOutput()* muss ein Datei-Modus angegeben werden:

Attribut	Beschreibung
MODE_PRIVATE	Erstellt Datei oder ersetzt existierende Datei, die nur für Applikation zugreifbar ist
MODE_APPEND	Fügt Text am Ende einer existierenden Datei ein
MODE_WORLD_READABLE	Erlaubt allen Applikationen das Lesen der Datei
MODE_WORLD_WRITEABLE	Erlaubt allen Applikationen das Schreiben der Datei

```
// Prepare the data, ...
String data = "This is a tasty test!";

// ... open the data stream ...
FileOutputStream fos = openFileOutput("test.txt", Context.MODE_PRIVATE);

// ... and write out the data.
fos.write(data.getBytes());

// Finally close the data stream.
fos.close();
```

Listing : Eine interne Datei beschreiben

Datei lesen

- Dateien werden mit *FileInputStreams* gelesen
- Einziger Parameter Name der Datei
- Blockweise Verarbeitung

```
// Prepare read buffer and the result string, ...
byte[] buffer = new byte[4096];
String data = "";

// ... open the data stream ...
FileInputStream fis = openFileInput("test.txt");

// ... and read in all data.
int block = in.read(buffer, 0, buffer.length);

while(block >= 0) {
   data += new String(buffer);
   block = in.read(buffer, 0, buffer.length);
}

// Finally close the data stream.
fis.close();
```

Listing : Eine interne Datei lesen

Statische Text-Ressourcen

Größere Texte, wie eine Dokumentation oder Hilfe, können als Ressourcen hinterlegt werden. Dazu können beliebige Dateien im Ordner *res/raw/* gespeichert und mit *Resources.openRawResource()*, unter Angabe der ID *R.raw.<filename>*, geladen werden.

Cache-Dateien

- Normaler Datei-Zugriff
- Methode *getCacheDir()* liefert Zielordner von Cache-Dateien
- System kann Cache-Dateien jeder Zeit löschen
- Cache-Dateien werden bei Deinstallation der Applikation gelöscht

Erweiterte Datei-Zugriffe

Methode	Beschreibung
getFilesDir()	Liefert absoluten Pfad zum internen Dateisystem
getDir()	Öffnet oder erstellt ein Verzeichnis im internen Speicherbereich
deleteFile()	Löscht eine Datei aus dem internen Speicher
fileList()	Gibt *File*-Array der aktuell durch die Applikation im internen Speicher abgelegten Dateien zurück

File-Methoden

Zudem stehen natürlich die Methoden der *File*-Klasse, wie *isDirectory()* und *canRead()* zur Verfügung.

Externer Speicher

Contents

Allgemeines

- Externer Speicher vom Gerät abhängig
- Kann als interner und SD-Speicher vorliegen
- Bei SD-Speicher muss unbedingt Verfügbarkeit geprüft werden
- Verwaltet privaten und öffentlichen Speicher
- Zugriff über *getExternalFilesDir()* oder *getExternalStoragePublicDirectory()*
- Zugriff über selbst erzeugten Input- und OutputStream

```
// Setup the available and writeable flag ...
boolean extAvailable = false;
boolean extWriteable = false;

// ... and fetch the state of the external storage.
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // The external storage is available and writeable.
    extAvailable = extWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // The external storage is available but only readable.
    extAvailable = true;
    extWriteable = false;
} else {
    // The external storage is not available.
    extAvailable = extWriteable = false;
}
```

Listing : Externen Speicher überprüfen

Privater Speicher

- Enthält nur Dateien der Applikation
- Wird beim Deinstallieren gelöscht
- *getExternalFilesDir()* erwartet Typ des zu öffnenden Verzeichnisses
- Typ ist Datei abhängig, Beispiel *Context.DIRECTORY_MUSIC*
- Dateien können so von Android-Media-Scanner gefunden werden
- Android-Media-Scanner stellt Dateien anderen Applikationen zur Verfügung
- Datei *.nomedia* im Wurzelverzeichnis verhindert Durchsuchen durch Android-Media-Scanner
- *getExternalFilesDir()* mit Parameter *null* liefert Wurzelverzeichnis

Öffentlicher Speicher

- Enthält öffentliche Dateien der Applikation
- Dateien sind für alle Applikationen les- und schreibbar
- Dateien bleiben auch nach Deinstallation gespeichert
- *getExternalStoragePublicDirectory()* erwartet Typ des zu öffneden Verzeichnisses
- Typ ist Datei abhängig, Beispiel *Context.DIRECTORY_MUSIC*

Ordner	Dateitypen
Music/	Verschiedene Musik-Dateien.
Podcasts/	Podcasts.
Ringtones/	Musik und Sound-Dateien, die als Klingeltöne verwendet werden.
Alarms/	Signaltöne, die für Alarme verwendet werden.
Notifications/	Signaltöne, die für Notifikationen verwendet werden.
Pictures/	Alle Bilder, die nicht mit der Kamera aufgezeichnet wurden.
Movies/	Alle Videos, die nicht mit der Kamera aufgezeichnet wurden.
Download/	Dateien beliebigen Typs, die heruntergeladen wurden.

Cache-Speicher

- Zugriff mit *getExternalCacheDir()*
- Kann vom System jeder Zeit gelöscht werden
- Wird bei Deinstallation gelöscht

Cache verwalten

Im Regelfall ist der externe Cache dem internen vorzuziehen, da einige Endgeräte, wie beispielsweise das *HTC Desire* nur einen sehr kleinen internen Speicher mitbringen. Speichermangel, der zum Löschen der Applikation durch den Benutzer führen könnte, wird so vermieden.

SQLite

Contents

Allgemeines

- Standard Datenbank-System unter Android
- Unterstützt alle gängigen Features von relationalen Datenbanken
- *SQL*-Syntax, *Prepared Statement* und *Transactions*
- Infos unter www.sqlite.org
- Zugriff unter Android nur durch erstellende Applikation
- Erstellen einer Datenbank optimaler Weise mit *SQLiteOpenHelper*
- Methode *onCreate()* erstellt Datenbank
- Methode *onUpgrade()* verwaltet Änderungen an Datenbank-Struktur

Implementierung

```
public class UbuntuReleasesHelper extends SQLiteOpenHelper {
   public static final String DATABASE_NAME = "ubuntu.releases";
   public static final int DATABASE_VERSION = 2;
   public static final int FIRST_UPDATE_VERSION = 2;
   private Context context;

   public SqliteDatabaseHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
      this.context = context;
   }

   @Override
   public void onCreate(SQLiteDatabase sqlite) {
      sqlite.execSQL(
         "CREATE TABLE IF NOT EXISTS releases ("
         + "_ID INTEGER PRIMARY KEY ASC, "
         + "name VARCHAR(160), version VARCHAR(10),"
         + "releasedAt INTEGER, supportedUntil INTEGER,"
         + "description TEXT, logo VARCHAR(160) NOT NULL DEFAULT 'ubuntu.jpg'"
         + ");"
      );
   }

   @Override
   public void onUpgrade(SQLiteDatabase sqlite, int oldVersion, int newVersion) {
      if(newVersion == FIRST_UPDATE_VERSION) {
         // Add a new column in database version 2.
         sqlite.execSQL(
            "ALTER TABLE releases ADD COLUMN "
            + "logo VARCHAR(160) NOT NULL DEFAULT 'ubuntu.jpg';"
         );
      }
   }
}
```

Listing : Den SQLiteOpenHelper nutzen

Datenbank-Zugriff

- Zugriff mit *SQLiteDatabase*-Objekt
- Lesender Zugriff mit *getReadableDatabase()* anfordern
- Schreibender Zugriff mit *getWriteableDatabase()* anfordern

SQLite-Tabellen

Android stellt keine Anforderungen an eine SQLite-Tabelle, es empfiehlt sich jedoch in jeder Tabelle eine ID zu hinterlegen. *ContentProvider*, die auch anderen Programmen den Zugriff auf die in dieser Datenbank hinterlegten Daten ermöglichen, fordern beispielsweise einen solchen Schlüssel mit dem Namen *_ID*.

Asynchroner Zugriff

Datenbank-Anfragen sind sehr langsam, da ein Zugriff auf das Dateisystem nötig ist. Es empfiehlt sich seine Anfragen asynchron zur Applikation, also in einem *Thread* oder *AsyncTask* auszuführen.

Anfragen absetzen

- Direktes Ausführen einer Anfrage mit *execSQL()*
- Methoden *insert()*, *update()* und *delete()* zum Einfügen, Ändern und Löschen von Datensätzen
- *query()* und *rawQuery()* um Abfragen zu senden
- Ergebnisse von Anfragen werden als *Cursor*-Objekte zurückgeliefert

Cursor

- *Cursor* zeigt auf eine Zeile des Ergebnisses (nicht alle Zeilen werden in Speicher geladen)
- *moveToFirst()* schiebt Cursor an erste Position
- *moveToNext()* ermöglicht durchlaufen des Ergebnisses
- Auf Ende mit Rückgabewert von *moveToNext()* oder direkt mit *isAfterLast()* prüfen
- Cursor stellt typenbasierte *get*()* Methoden zur Verfügung
- Beispiel: *getString(columnIndex)*
- Spalten-Index kann mit *getColumnIndexOrThrow()* erfragt werden
- Cursor immer mit *close()* schließen

Implementierung

```
try {
   // Instance the SqliteDatabaseHelper and fetch SQLite database handle.
   UbuntuReleasesHelper dbHelper = new UbuntuReleasesHelper(this);
   SQLiteDatabase sqlite = dbHelper.getWritableDatabase();

   // Declare the sql statement and execute the query.
   String sql = "SELECT * FROM releases;";
   Cursor sqlCursor = this.sqlite.rawQuery(sql, null);

   // Check if query has a result.
   if(sqlCursor != null) {
      // Try to move to the first entry.
      if(sqlCursor.moveToFirst()) {
         // Get the column indices and loop over all results.
         int nameIndex = sqlCursor.getColumnIndexOrThrow("name");
         int versionIndex = sqlCursor.getColumnIndexOrThrow("version");
         int releaseIndex = sqlCursor.getColumnIndexOrThrow("releasedAt");

         do {
            // Fetch the column values and do something with them.
            String name = sqlCursor.getString(nameIndex);
            String version = sqlCursor.getString(versionIndex);
            Long release = sqlCursor.getLong(releaseIndex);
         } while(sqlCursor.moveToNext());
      }
   }
} catch(Exception e) {
   e.printStackTrace();
} finally {
   if(sqlite != null)
      sqlite.close();
   if(sqlCursor != null)
      sqlCursor.close();
}
```

Listing : SQLiteDatabase nutzen

Anmerkung

sqlite3 Werkzeug

Das Android SDK stellt das Werkzeug *sqlite3* zur Verfügung, dass dazu verwendet werden kann die Datenbanken direkt zu verwalten, wodurch Datenbank-Zugriffe besser analysiert und Probleme schneller gefunden werden können.

ContentProvider

Contents

Allgemeines

- Verwaltet den Zugriff auf zentral verfügbare Daten
- Meistens Teil einer Applikation die selbst Zugriff auf diese Daten ermöglicht
- Stellt anderen Applikationen einen oder mehrere Datenstämme (Tabellen) zur Verfügung
- Kontrolliert den Zugriff
- Beispiel: *DictionaryProvider* & *ContactProvider*
- Zugriff über *ContentResolver*-Objekt
- Daten werden über *URI* identifiziert
- *URI* setzt sich aus symbolischen Namen des Providers und einer Tabelle zusammen

ContentResolver

- Zugriffsmethoden (*CRUD*-Methoden) zum Erstellen, Lesen, Ändern und Löschen der Daten
- Anfrage über *query()* Methode
- Starke Ähnlichkeit zur SQL-Syntax
- Liefert wie *SQLite*-Bibliothek einen Cursor

Parameter	Dateitypen
URI	Identifiziert den ContentProvider und die durch ihn verwaltete Tabelle
Spalten	Enthält angeforderte Spaltennamen
Auswahl Statement	Deklariert einen SQL-*WHERE* ähnliches Statement
Auswahl Argumente	Liefert die zur Auswahl benötigten Argumente
Sortierung	Legt eine Sortierung des Ergebnisses fest

Beispiel

```
Cursor cur = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,
    columns,
    selectionClause,
    selectionArgs,
    sorting);
```

Listing : Einen ContentResolver nutzen

Implementierung

- Rechte werden im Android-Manifest gesetzt
- Beispiel: *android.permission.READ_USER_DICTIONARY*
- Einbinden mit <*uses-permission*>-Element

```
SELECT _ID, word FROM words WHERE word = <userinput> ORDER BY word ASC;
```

Listing : Die SQL-Abfrage

Implementierung II

```
// Declare the columns that should be selected, ...
String[] columns = {
    UserDictionary.Words._ID,
    UserDictionary.Words.WORD
};

// ... setup the selection clause ...
String selectionClause = null;

// ... and initialize the selection arguments.
String[] selectionArgs = {""};

// Declare the sort order ...
String sortOrder = UserDictionary.Words.WORD + " ASC";

// ... and fetch the userinput from a EditText field.
searchString = searchTxt.getText().toString();

// An empty input selects all rows of the table.
if (TextUtils.isEmpty(searchString)) {
    // Nothing to do here. Everything should be selected.
} else {
    // Setup the selection clause ...
    selectionClause = UserDictionary.Words.WORD + " = ?";

    // ... and place the user input in the selection arguments array.
    selectionArgs[0] = searchString;
}

// Finally query the word cursor.
wordCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI, columns,
    selectionClause, selectionArgs, sortOrder);
```

Listing : Anfrage an einen ContentProvider

Ergebnisse Durchlaufen

```
if (null == wordCursor) {
   // If an error occurred some providers will return null.
} else if (wordCursor.getCount() < 1) {
   // The query didn't match any data.
} else {
   // Try to move to the first entry.
   if(wordCursor.moveToFirst()) {
      // Get the column indices, ...
      int idIndex = wordCursor.getColumnIndexOrThrow(UserDictionary.Words._ID);
      int wordIndex = wordCursor.getColumnIndexOrThrow(UserDictionary.Words.WORD);

      // ... and loop over all results.
      do {
         // Fetch the column values ...
         String id = wordCursor.getString(idIndex);
         String word = wordCursor.getString(wordIndex);

         // ... and do something with them.
      } while(wordCursor.moveToNext());
   }
}
```

Listing : Manuelles Durchlaufen des Cursors

Anzeigen von Ergebnissen

```
// Declare the column to select from the cursor ...
String[] columns = {UserDictionary.Words.WORD};

// ... and the views that should be used to display the data.
int[] viewIds = { R.id.dictWord};

// Finally setup a new SimpleCursorAdapter ...
SimpleCursorAdapter wordAdapter = new SimpleCursorAdapter(
    getApplicationContext(), R.layout.wordlistrow,
    wordCursor, columns, viewIds, 0);

// ... and assign this adapter to the ListView.
wordList.setAdapter(wordAdapter);
```

Listing : Verwendung des SimpleCursorAdapters

Daten einfügen

```
// Setup an uri that receives the result of the insertion.
Uri insertedUri;

// Setup the object of values to be inserted ...
ContentValues insertValues = new ContentValues();
insertValues.put(UserDictionary.Words.APP_ID, "example.user");
insertValues.put(UserDictionary.Words.LOCALE, "en_US");
insertValues.put(UserDictionary.Words.WORD, "insert");
insertValues.put(UserDictionary.Words.FREQUENCY, "100");

// ... and run the insert query.
insertedUri = getContentResolver().insert(
    UserDictionary.Word.CONTENT_URI, insertValues);
```

Listing : Einfügen eines neuen Datensatzes

Ergebnis-ID

Möchte man diese ID aus der URI extrahieren, so kann man dazu *ContentUris.parseId()* unter Angabe der URI aufrufen.

Daten ändern

```
// Setup the selection clause and arguments ...
String selectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";
String[] selectionArgs = {"en_%"};

// ... and initialize the number of affected rows.
int affectedRows = 0;

// Setup the object of values to be updated ...
ContentValues updateValues = new ContentValues();
updateValues.put(UserDictionary.Words.LOCALE, "en_US");
updateValues.putNull(UserDictionary.Words.FREQUENCY);

// ... and run the update query.
affectedRows = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI, updateValues,
    selectionClause, selectionArgs);
```

Listing : Ändern von Datensätzen

Daten löschen

```
// Setup the selection clause and arguments ...
String selectionClause = UserDictionary.Words.LOCALE + " LIKE ?";
String[] selectionArgs = {"en_%"};

// ... and initialize the number of affected rows.
int affectedRows = 0;

// Finally run the delete query.
affectedRows = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI, selectionClause, selectionArgs);
```

Listing : Löschen von Datensätzen

Zugriff über ID

```
// Setup the lookup URI ...
Uri lookUp = ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI, 13);

// ... and initialize the number of affected rows.
int affectedRows = 0;

// Finally run the delete query.
affectedRows = getContentResolver().delete(
   lookUp, null, new String[] {""});
```

Listing : Datensatz über URI ansprechen

Asynchrone Anfragen

Alle Beispiele führen die Anfragen im Main-Thread aus. Da der Zugriff über einen ContentResolver sehr langsam ist, ist es empfehlenswert Anfragen asynchron, also in einem *Thread* oder *AsyncTask*, auszuführen.

Allgemeines

- Ableiten einer Klasse von abstrakter Klasse *ContentProvider*
- Registrierung des ContentProviders im Manifest
- Festlegen eines symbolischen Provider-Namens
- Hinterlegen der Spaltennamen
- Deklaration der Möglichen URIs
- Alternativ können Konstanten auch in einer Contracts-Klasse implementiert werden
- Hinterlegen der benötigten Zugriffsrechte

Beispiel

Möglicher Provider-Name:
net.avedo.ubuntu.releases.<provider>
Sich daraus ergebene URIs:

- content://net.avedo.ubuntu.releases.<provider>/<file>
- content://net.avedo.ubuntu.releases.<provider>/<table>
- content://net.avedo.ubuntu.releases.<provider>/<table>/<id>

Basis-Methoden

- Erwartete Funktionen *onCreate()*, *insert()*, *update()*, *delete()*, *query()* und *getType()*
- *UnsupportedOperationException* sollte beispielsweise schreibender Zugriff unterbunden werden
- Alle Methoden außer *onCreate()* können aus verschiedenen Threads aufgerufen werden
  $\rightarrow$ Implementierung Thread-Safe
- Keine längeren Operationen in *onCreate()*
- Nur bekannte Exceptions, wie *UnsupportedOperationException*, *NullPointerException* oder *IllegalArgumentException* verwenden
- *getType()* liefert MIME Typ (*text/html* oder *image/jpeg*)
- Alternative: ”Entwickler-deklarierter” (vendor-specific) MIME-Typ

Entwickler-deklarierte Typen haben die Formen:

1. vnd.android.cursor.item/vnd.<provider>/<table>
2. vnd.android.cursor.dir/vnd.<provider>/<table>

Verarbeitung der URIs

- Verwendung von *UriMatcher* zur Wildcard-Analyse
  - `*` Sucht einen String aus beliebigen Zeichen und beliebiger Länge.
  - `#` Sucht einen numerischen Wert beliebiger Länge.
- Hinzufügen einer URI mit *UriMatcher.match(uri)*
- Beispiel: Unterscheidung zwischen Tabellen- und Datensatz-URI

Android-Manifest

- Eintrag mit <*provider*>-Element
- Deklaration der Zugriffsrechte mit <*permission*>-Elemente
- Erlauben eines temporären Zugriffs mit *android:grantUriPermssions true*
- Alternative: Gesteuerte Vergabe von temporärem Zugriff über *<grant-uri-permission>*

Attribut	Beschreibung
android:authorities	Symbolischer Name des Providers, der diesen innerhalb des Systems eindeutig identifiziert
android:name	Name der ContentProvider Klasse
android:icon	Icon für den Provider
android:label	Kurze Beschreibung des Providers

Beispiel

```
<manifest>
   <permission android:name="net.avedo.ubuntu.releases.READ" />
   <permission android:name="net.avedo.ubuntu.releases.WRITE" />

   ...

   <application>
      <provider
         android:name="net.avedo.ubuntu.releases.UbuntuOneProvider"
         android:authorities="net.avedo.ubuntu.releases.ubuntu_one_provider"
         android:label="@string/provider_description"
         android:icon="@drawable/ubuntu_logo"
         android:readPermission="net.avedo.ubuntu.releases.READ"
         android:writePermission="net.avedo.ubuntu.releases.WRITE"
         android:grantUriPermssions="false">
         <grant-uri-permission
            android:path="net.avedo.ubuntu.releases.ubuntu_one_provider/documents/ubuntu/" />
         <grant-uri-permission
            android:pathPattern="/images/*/ubuntu/" />
         <grant-uri-permission
            android:pathPrefix="net.avedo.ubuntu.releases.ubuntu_one_provider/images/icons" />
      </provider>
   </application>
</manifest>
```

Listing : Provider und das Manifest