

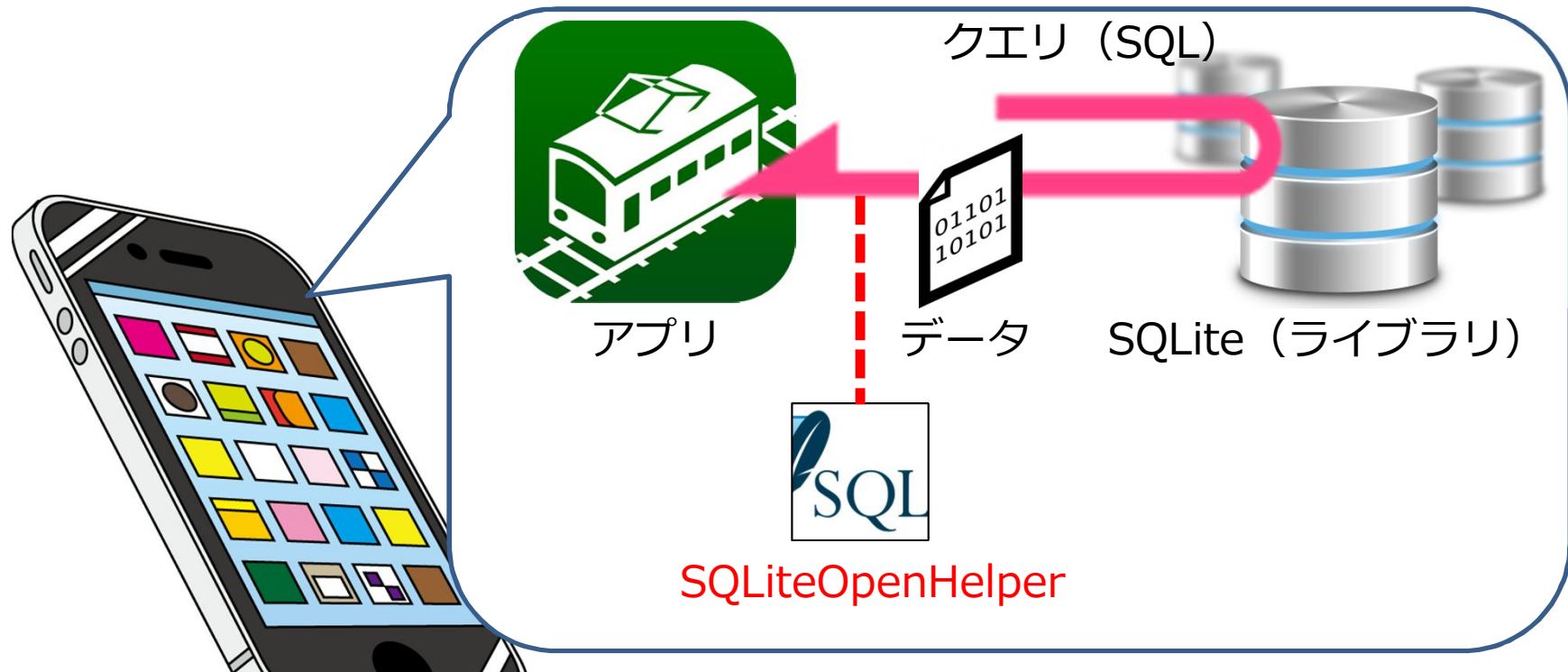
内部DBを使おう 【続編 1 : SELECT】



v.1.0.0

by masatokg

アプリからSQLiteにアクセス



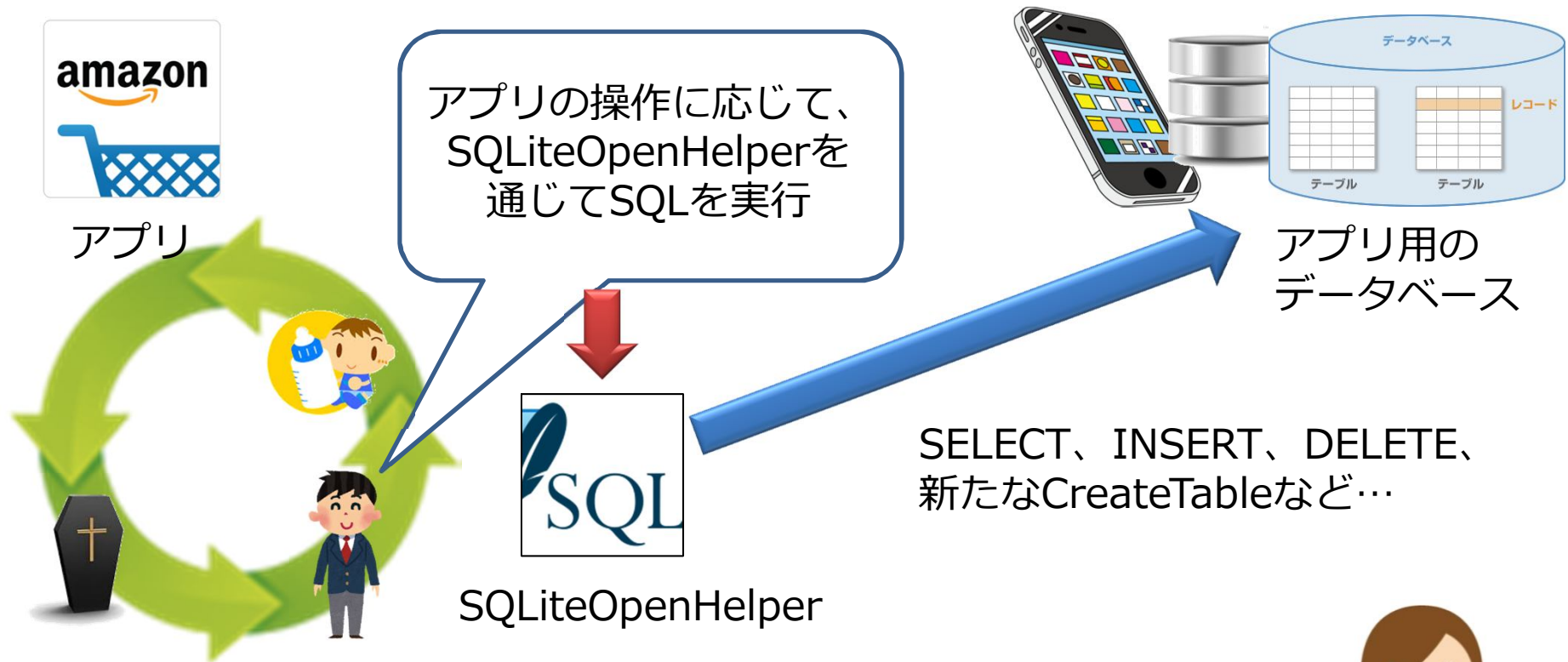
引き続き、実際に「SQLiteOpenHelper」を使ったSQLiteアクセスを実装していきます。

今回は、いよいよリスト一覧のSELECTとリストを選択してからのDELETEを実装します。まず今回は、表示／再表示でのSELECT読み込みを作っていきます。





アプリからDB内の表を操作する



これまで見てきたように、アプリ内のボタンクリック等のいろいろな操作に応じて、SELECTやINSERT、DELETEなどの処理をSQLを、SQLiteOpenHelperを通じて実行し、表を操作していきます。



一言を登録、表示は完成



前回までで、「今日の一言」をDBに登録、一言表示画面で登録した一言をランダムに表示する処理が完成したところで、続きを実装していきます。



登録された一言をリスト表示、選択削除



今度は、画面遷移後、登録済みの一言を一覧表示、そのうちの一つを選択して削除ボタンを押下するとデータベースのHitokotoテーブルから該当データを削除する、という処理を実装します。



MySQLiteOpenHelperを利用する

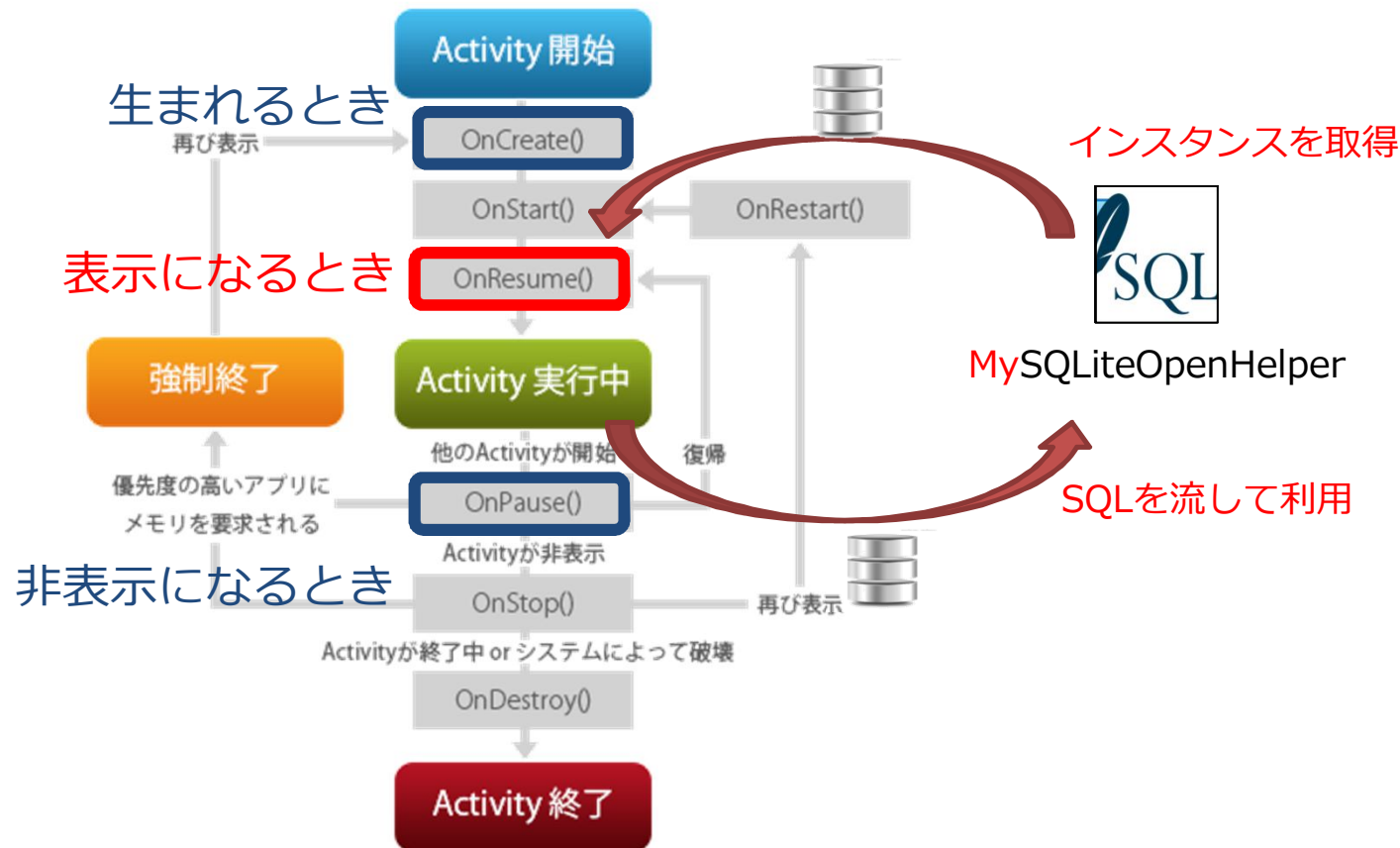


MySQLiteOpenHelper

今回も、「SQLiteOpenHelper」の機能を受け継いだ継承クラス「My SQLiteOpenHelper」のインスタンスを、各画面（Activity）の中で利用していきます。まずは、表示／再表示時のSELECTを実装していきます。



SQLiteDatabaseインスタンスを取得



おさらいですが、画面（Activity）からSQLiteデータベースを操作するには、まずOnResumeライフサイクルイベントにて、作成したMySQLOpenHelperを利用してSQLiteDatabaseインスタンスを取得します。ActivityのソースコードにOnResumeイベントハンドラをまだ実装していない場合は、ソースメニューから選択して追加します。



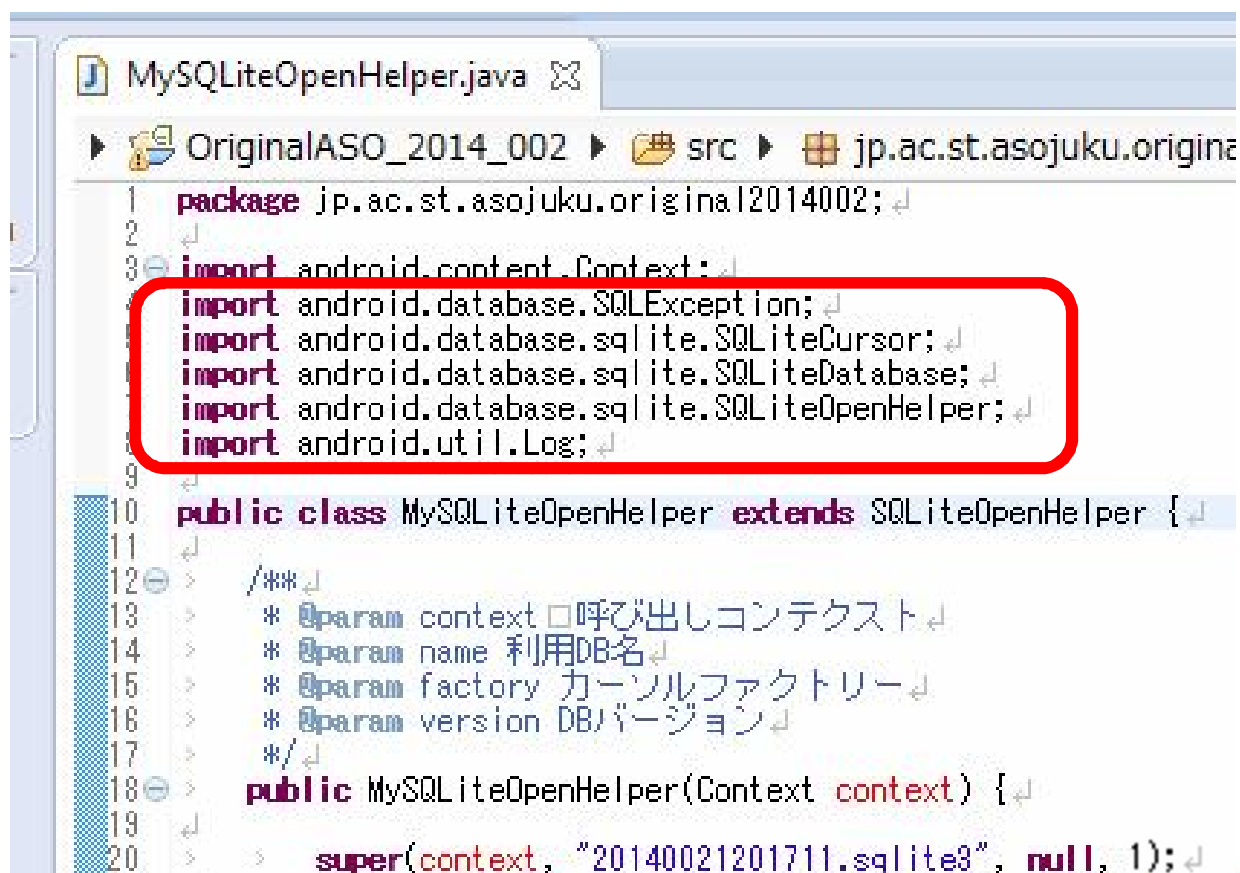
SQLiteOpenHelperの処理

SQLiteOpenHelper

SQLiteを利用

SELECTとDELETEの機能を追加するため、DBを操作するSQLiteOpenHelperの処理から実装します。
SQLiteOpenHelper.javaを追加編集します。

必要な部品クラスをimport



```
MySQLiteOpenHelper.java
OriginalASO_2014_002 > src > jp.ac.st.asojuku.origina
1 package jp.ac.st.asojuku.original2014002;
2
3 import android.content.Context;
4 import android.database.SQLException;
5 import android.database.sqlite.SQLiteDatabase;
6 import android.database.sqlite.SQLiteOpenHelper;
7 import android.util.Log;
8
9
10 public class MySQLiteOpenHelper extends SQLiteOpenHelper {
11
12     /**
13      * @param context 呼び出しコンテキスト
14      * @param name 利用DB名
15      * @param factory カーソルファクトリー
16      * @param version DBバージョン
17      */
18     public MySQLiteOpenHelper(Context context) {
19
20         super(context, "20140021201711.sqlite3", null, 1);
```

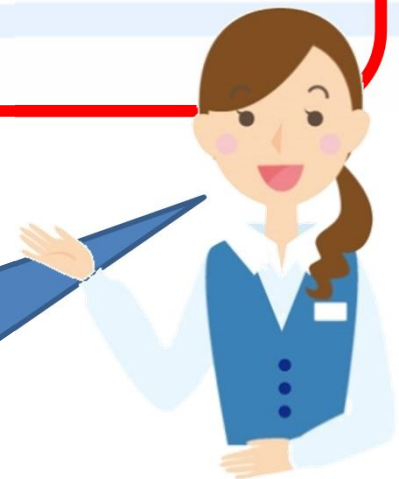
前回と同様、DB操作に必要な部品クラスをimportしていない場合は、importします。



テーブル作成処理を確認

```
17 > */  
18 > public MySQLiteOpenHelper(Context context) {  
19 >   
20 >     super(context, "20140021201711.sqlite3", null, 1);  
21 >   
22 > }  
23 >   
24 > @Override  
25 > public void onCreate(SQLiteDatabase db) {  
26 >     // TODO 自動生成されたメソッド・スタブ  
27 >     db.execSQL("CREATE TABLE IF NOT EXISTS "  
28 >     >     >     "Hitokoto ( _id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL , phrase TEXT )");  
29 > }  
30 >   
31 > @Override  
32 > public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
33 >     db.execSQL("drop table Hitokoto;");  
34 >     onCreate(db);  
35 > }  
36 >
```

前回、Create Table文のSQLをOnCreate時に実行する処理を実装しているはずですが。今回はこのテーブルにDELETEや、SELECTを実行するため、実装済みであることを確認します。



SELECT処理用のメソッドを作成

```
87 > /**  
88 > * Hitokotoテーブルからデータをすべて取得  
89 > * @param SQLiteDatabase SELECTアクセスするDBのインスタンス変数  
90 > * @return 取得したデータの塊の表（導出表）のレコードをポイントするカーソル  
91 > */  
92 >   
93 > public SQLiteCursor selectHitokotoList(SQLiteDatabase db){  
94 >   
95 >     SQLiteCursor cursor = null;  
96 >   
97 >     String sqlstr = " SELECT _id, phrase FROM Hitokoto ORDER BY _id; ";  
98 >     try {  
99 >         //トランザクション開始  
100 >         cursor = (SQLiteCursor)db.rawQuery(sqlstr, null);  
101 >         if(cursor.getCount()!=0){  
102 >             //カーソル開始位置を先頭にする  
103 >             cursor.moveToFirst();  
104 >         }  
105 >         // cursorは呼び出し元へ返すからここではcloseしない  
106 >         // cursor.close();  
107 >     } catch (SQLException e) {  
108 >         Log.e("ERROR", e.toString());  
109 >     } finally {  
110 >     }  
111 >   
112 >     }  
113 >     return cursor;  
114 > }  
115 > }
```

同様に、引数で渡されたDBに、SELECT文を実行して結果を受け取るメソッドを書きます。SELECTのように結果データを受け取るSQLは、SQLiteDatabase#rawQuery()メソッドで実行します。結果は、いったんcursorという形で受け取ってから、中身を取り出します。このCursorを、今回は呼び出し元に戻り値としてリターンします。書き込み場所は、selectRandomHitokotoメソッドの次あたりがよいでしょう。



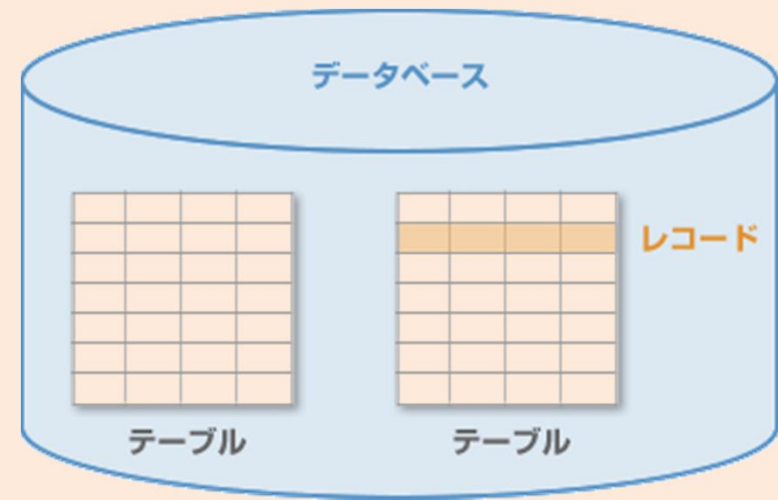
カーソル (Cursor) とは



SQL

cursor

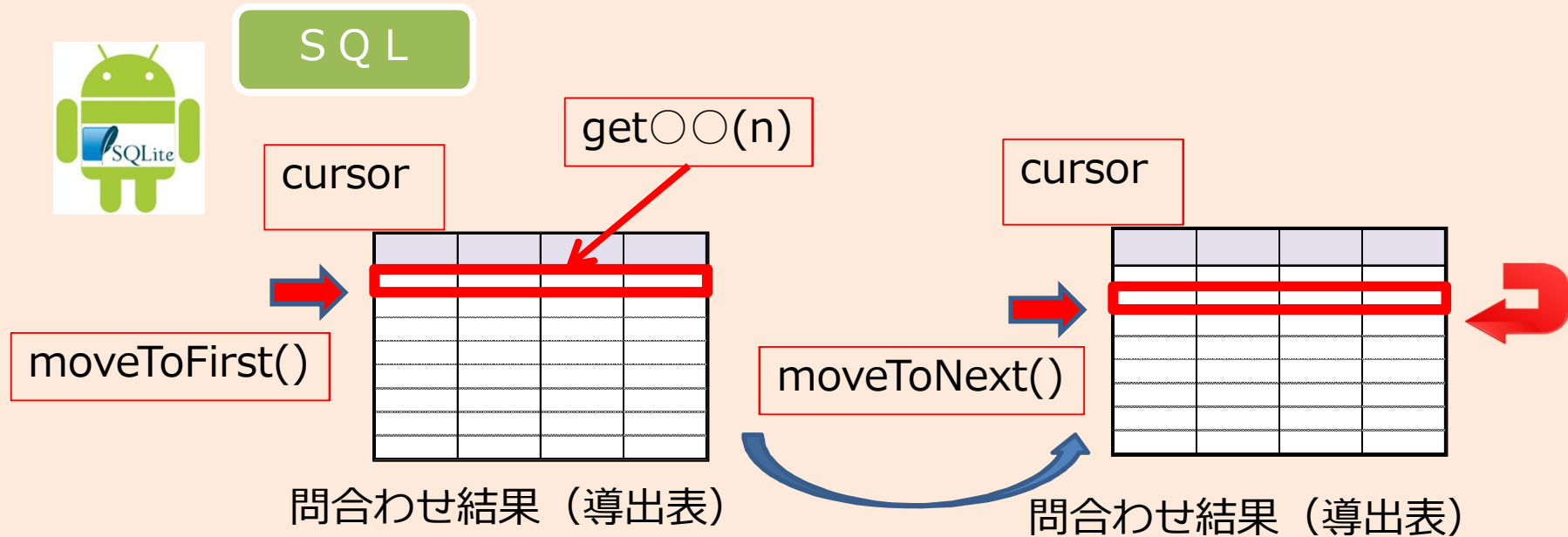
問い合わせ結果 (導出表)



カーソルとは、RDBからのSELECT結果のデータの塊 (導出表) 上のデータを指し示すポインタにあたるオブジェクト変数です。



カーソルは導出表のレコードを指し示す



カーソルとは、通常、導出表の中の1レコード分のデータを指し示し、前回のSELECT文で実施したように、`get○○(n)`の形で操作してそのレコード内のカラムを指定、値を取り出します。最初に`Cursor#moveToFirst()`を実行して先頭レコードを指し示します。先頭レコードからデータを取り出したら、`Cursor#moveToNext()`を実行して、カーソルが次のレコードデータを指し示すようにします。これを導出表のレコード数分ループしながら、繰り返し実行することで、全てのデータが取り出せます。



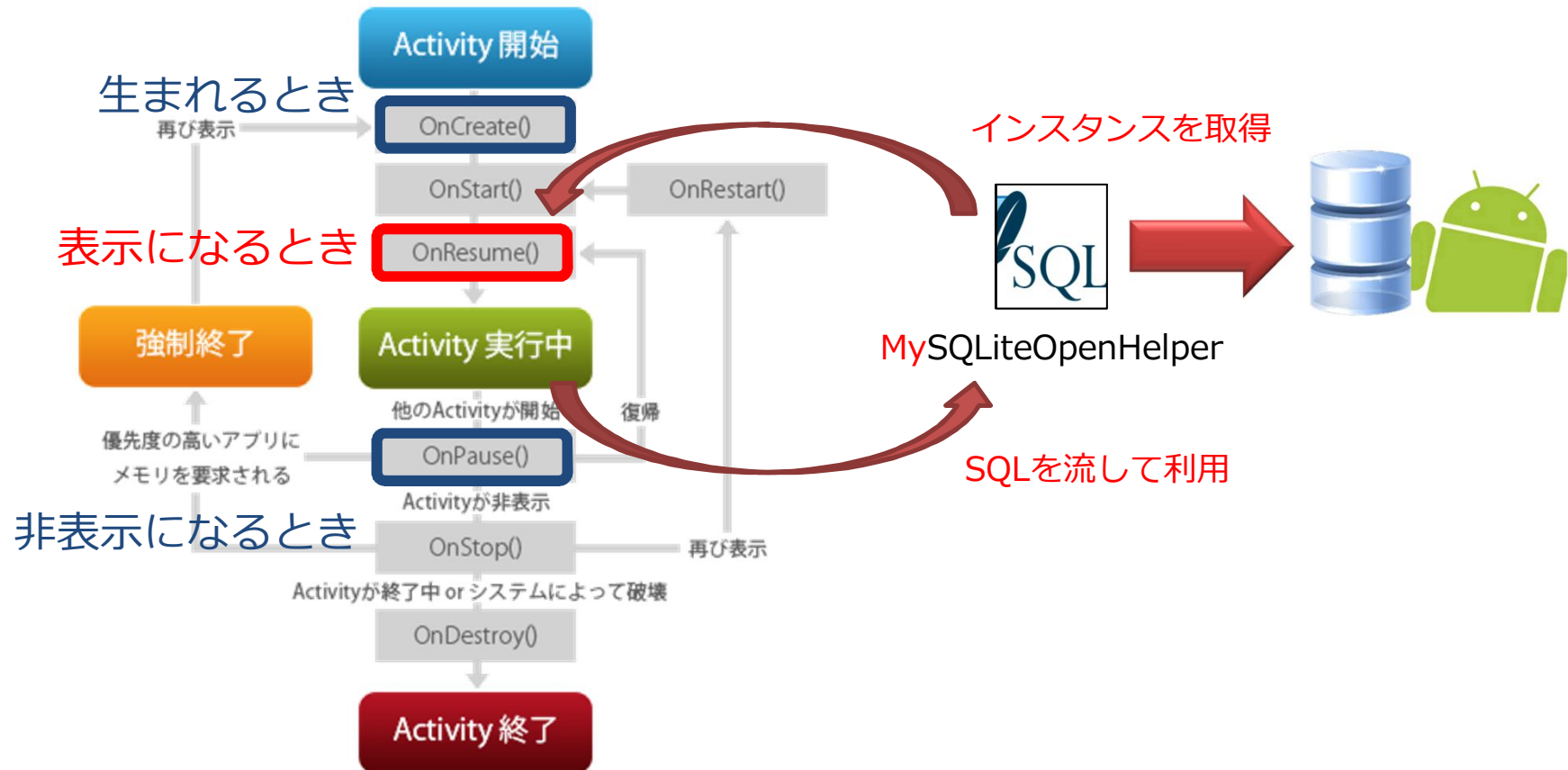
DELETE処理用のメソッドを作成

```
117 /**  
118 > * Hitokoto表から引数 (id) で指定した値とカラム「_id」の値が等しいレコードを削除  
119 > * @param SQLiteDatabase DELETEアクセスするDBのインスタンス変数  
120 > * @param id カラム「_id」と比較するために指定する削除条件の値  
121 > */  
122 > public void deleteHitokoto(SQLiteDatabase db, int id){  
123 > >  
124 > > String sqlstr = " DELETE FROM Hitokoto where _id = " + id + " ";  
125 > > try {  
126 > > > //トランザクション開始  
127 > > > db.beginTransaction();  
128 > > > db.execSQL(sqlstr);  
129 > > > //トランザクション成功  
130 > > > db.setTransactionSuccessful();  
131 > > } catch (SQLException e) {  
132 > > > Log.e("ERROR", e.toString());  
133 > > } finally {  
134 > > > //トランザクション終了  
135 > > > db.endTransaction();  
136 > > }  
137 }  
138 }
```

次に、DBへDELETE文を実行するメソッドを書きます。引数には、実行先のSQLiteDatabaseを操作するためのインスタンス変数と、削除条件の「_id」カラムの値を渡すことにします。SQL文を組立て、SQLiteDatabase#execSQL()メソッドで実行します。万一のエラーのために、トランザクション処理と、try&catchでエラー処理を実施するようにしましょう。書き込み場所は、先ほどのselectHitokotoListメソッドの次あたりがよいでしょう。



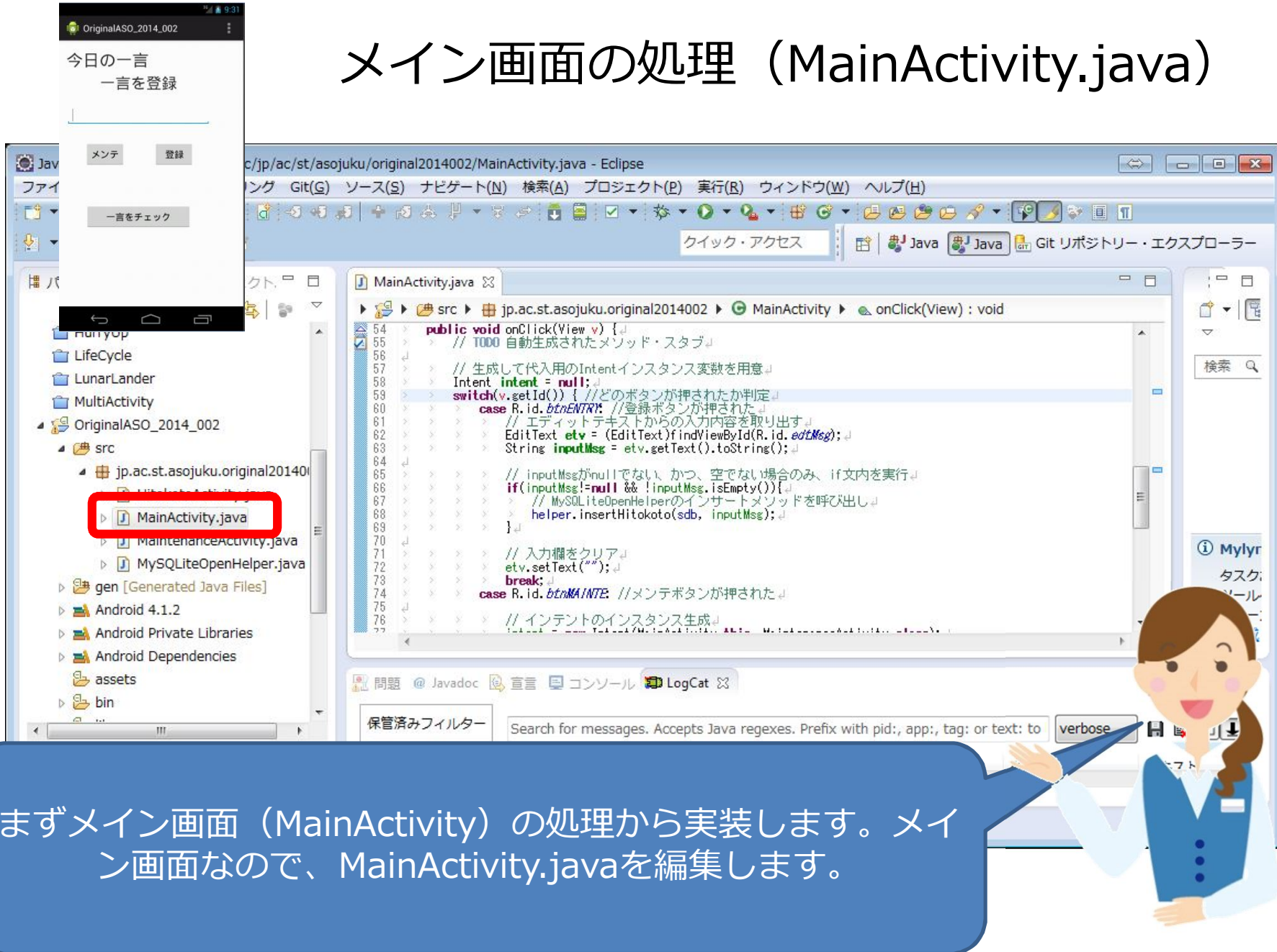
SQLiteOpenHelperDatabaseを呼び出して使う



こうして前回同様、MySQLiteHelperに書き込んだDELETE、SELECT用のメソッドを、各画面のActivityから呼び出すことで、DBを操作していきます。次に、画面（Activity）側の呼び出し処理を書いていきます。



メイン画面の処理 (MainActivity.java)




The image displays the development environment for an Android application. On the left, a preview of the app's main screen is shown, featuring a title bar 'OriginalASO_2014_002', a text input field with the placeholder '今日の一言 一言を登録', and buttons for 'メンテ' (Maintenance), '登録' (Register), and '一言をチェック' (Check a sentence). The main part of the image is the Eclipse IDE window, showing the source code for MainActivity.java. The file explorer on the left highlights MainActivity.java within the package 'jp.ac.st.asojuku.original2014002'. The code editor shows the following Java code:

```
54 public void onClick(View v) {  
55     // TODO 自动生成されたメソッド・スタブ  
56  
57     // 生成して代入用のIntentインスタンス変数を用意  
58     Intent intent = null;  
59     switch(v.getId()) { //どのボタンが押されたか判定  
60         case R.id.btnENTRY: //登録ボタンが押された  
61             // エディットテキストからの入力内容を取り出す  
62             EditText etv = (EditText)findViewById(R.id.edtMsg);  
63             String inputMsg = etv.getText().toString();  
64  
65             // inputMsgがnullでない、かつ、空でない場合のみ、if文内を実行  
66             if(inputMsg!=null && !inputMsg.isEmpty()){  
67                 // MySQLiteOpenHelperのインサートメソッドを呼び出し  
68                 helper.insertHitokoto(sdb, inputMsg);  
69             }  
70  
71             // 入力欄をクリア  
72             etv.setText("");  
73             break;  
74         case R.id.btnMAINT: //メンテボタンが押された  
75  
76             // インテントのインスタンス生成  
77     }
```

A blue callout box at the bottom contains the following text:

まずメイン画面（MainActivity）の処理から実装します。メイン画面なので、MainActivity.javaを編集します。



必要な部品クラスをimport

```
3 import android.app.Activity;
4 import android.content.Intent;
5 import android.database.sqlite.SQLiteDatabase;
6 import android.database.sqlite.SQLiteException;
7 import android.os.Bundle;
8 import android.view.Menu;
9 import android.view.View;
10 import android.widget.Button;
11 import android.widget.EditText;
12
13 public class MainActivity extends Activity implements View.OnClickListener {
14
15     > SQLiteDatabase sdb = null;
16     > MySQLiteOpenHelper helper = null;
```

DB操作に必要な部品クラスをimportしていない場合は、importします。



DB操作に必要なクラスをインスタンス変数として用意

```
12  ↵  
13  public class MainActivity extends Activity implements View.OnClickListener {  
14  ↵  
15  > SQLiteDatabase sdb = null;  
16  > MySQLiteOpenHelper helper = null;  
17  ↵  
18  @Override  
19  > protected void onCreate(Bundle savedInstanceState) {  
20  > > super.onCreate(savedInstanceState);  
21  > > setContentView(R.layout.activity_main);  
22  > }  
23  ↵
```

DB操作に必要な以下のクラスをインスタンス変数として操作するために、クラスフィールド変数（どのメソッドの枠からも超えた、クラス全体で使える変数）として宣言します。



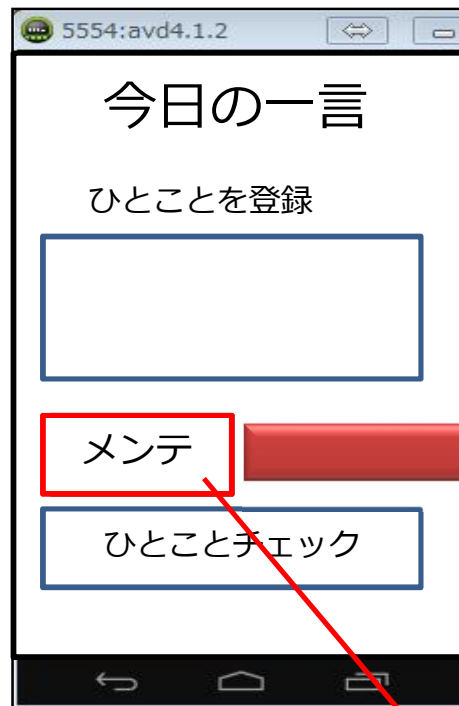
画面（Activity）にてMySQLiteOpenHelperのインスタンスを生成、オープンしたSQLiteDatabaseを取り出す

```
24 > override
25 > protected void onResume() {
26 >     // TODO 自動生成されたメソッド・スタブ
27 >     super.onResume();
28 >
29 >     // 登録ボタン変数にリスナーを登録する
30 >     Button btnENTRY = (Button)findViewById(R.id.btnENTRY);
31 >     btnENTRY.setOnClickListener(this);
32 >
33 >     // メンテボタン変数にリスナーを登録する
34 >     Button btnMAINT = (Button)findViewById(R.id.btnMAINT);
35 >     btnMAINT.setOnClickListener(this);
36 >
37 >     // 一言チェックボタン変数にリスナーを登録する
38 >     Button btnCHECK = (Button)findViewById(R.id.btnCHECK);
39 >     btnCHECK.setOnClickListener(this);
40 >
41 >     // フラグのフィールド変数がNULLなら、データベース空間オープン
42 >     if(sdb == null) {
43 >         helper = new MySQLiteOpenHelper(getApplicationContext());
44 >     }
45 >     try{
46 >         sdb = helper.getWritableDatabase();
47 >     } catch(SQLiteException e){
48 >         // 異常終了
49 >         return;
50 >     }
51 > }
52 >
```

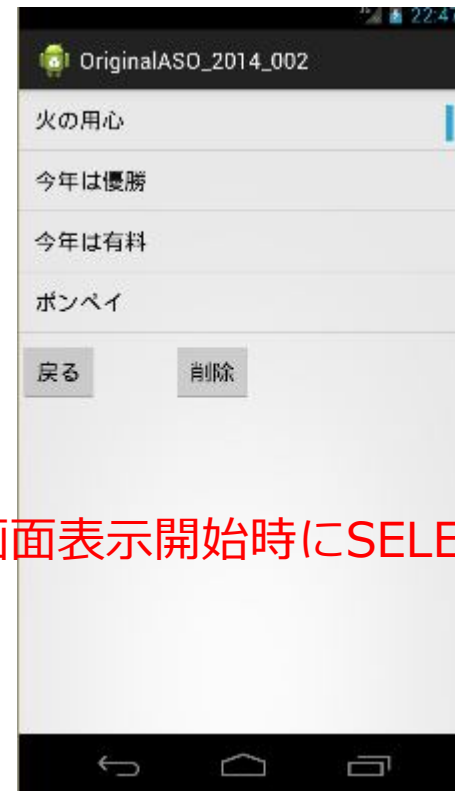
前回、OnResumeイベントハンドラの中に、MySQLiteOpenHelperのインスタンスを取得する処理を追加していることを確認します。



OnClickイベントハンドラ内に画面遷移処理を書き込む



OnClick()



画面表示開始時にSELECT

OnClickの中でメンテナンスボタンに対応するcase文処理の中で、メンテナンス画面への画面遷移を実行します。今回は、SELECT結果が複数行になって多いので、呼ばれた画面側でSQLを実行して表示しましょう。



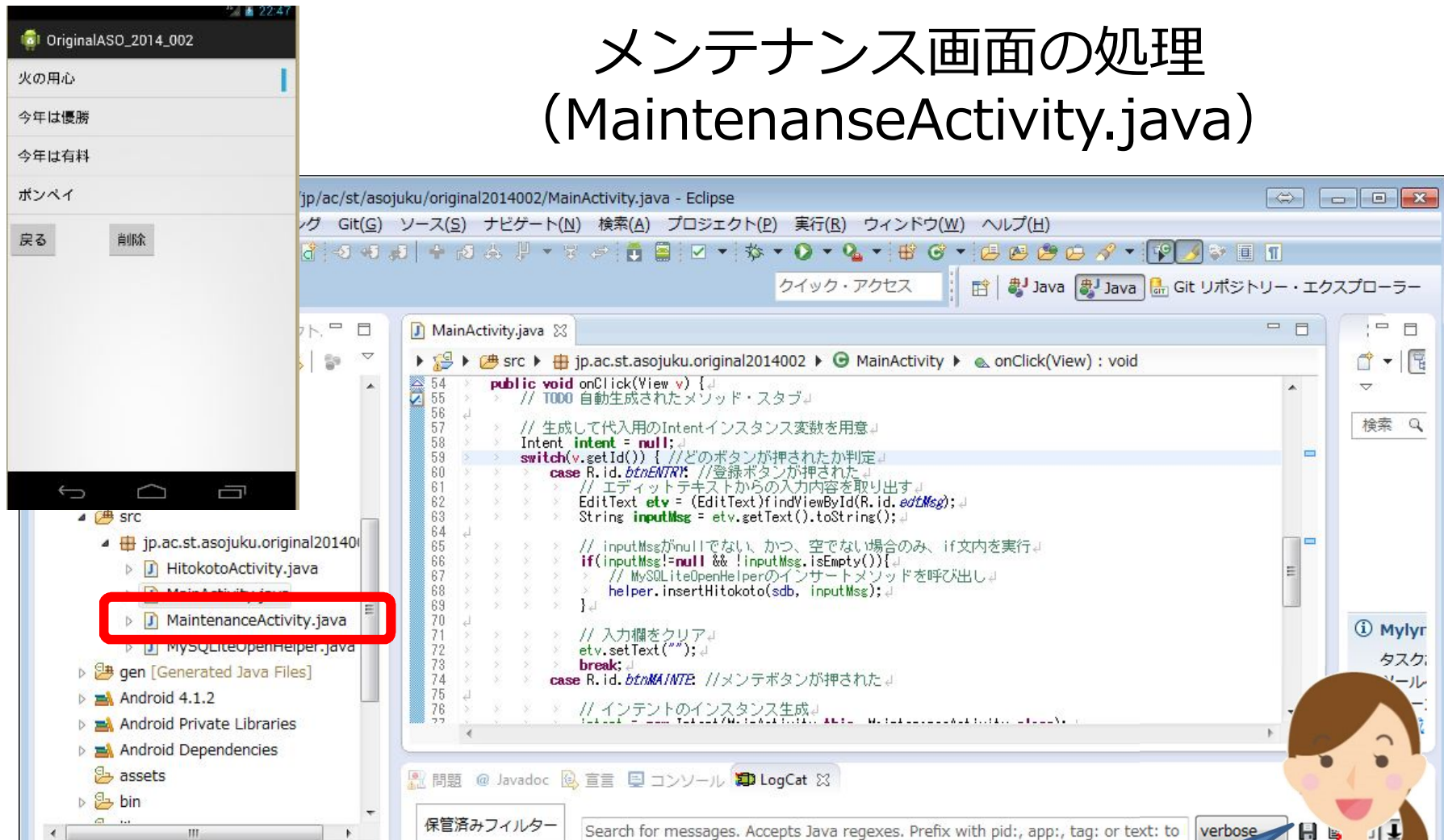
メンテボタン処理の部分にて、メンテナンス画面へ画面遷移

```
10 < < < // 生成して代入用のIntentインスタンス変数を用意<
11 < < < Intent intent = null;<
12 < < < switch(v.getId()) { //どのボタンが押されたか判定<
13 < < < < case R.id.btnENTRY: //登録ボタンが押された<
14 < < < < < // エディットテキストからの入力内容を取り出す<
15 < < < < < EditText etv = (EditText)findViewById(R.id.edtMsg);<
16 < < < < < String inputMsg = etv.getText().toString();<
17 < < < < < // inputMsgがnullでない、かつ、空でない場合のみ、if文内を実行<
18 < < < < < if(inputMsg!=null && !inputMsg.isEmpty()){<
19 < < < < < < // MySQLiteOpenHelperのインサートメソッドを呼び出し<
20 < < < < < < helper.insertHitokoto(sdb, inputMsg);<
21 < < < < < }<
22 < < < < < // 入力欄をクリア<
23 < < < < < etv.setText("");<
24 < < < < < break;<
25 < < < < < case R.id.btnMAINT: //メンテボタンが押された<
26 < < < < < < // インテントのインスタンス生成<
27 < < < < < < intent = new Intent(MainActivity.this, MaintenanceActivity.class);<
28 < < < < < < // 次画面のアクティビティ起動<
29 < < < < < < startActivity(intent);<
30 < < < < < < break;<
31 < < < < < case R.id.btnEXIT: // 終了ボタンが押された<
32 < < < < < <
```

onClickイベント反応処理のメンテボタン部分にて、（仮にVIEW部品IDをbtnMAINTとしたとします）メンテナンス画面（MaintenanceActivity）への画面遷移処理を書き込みます



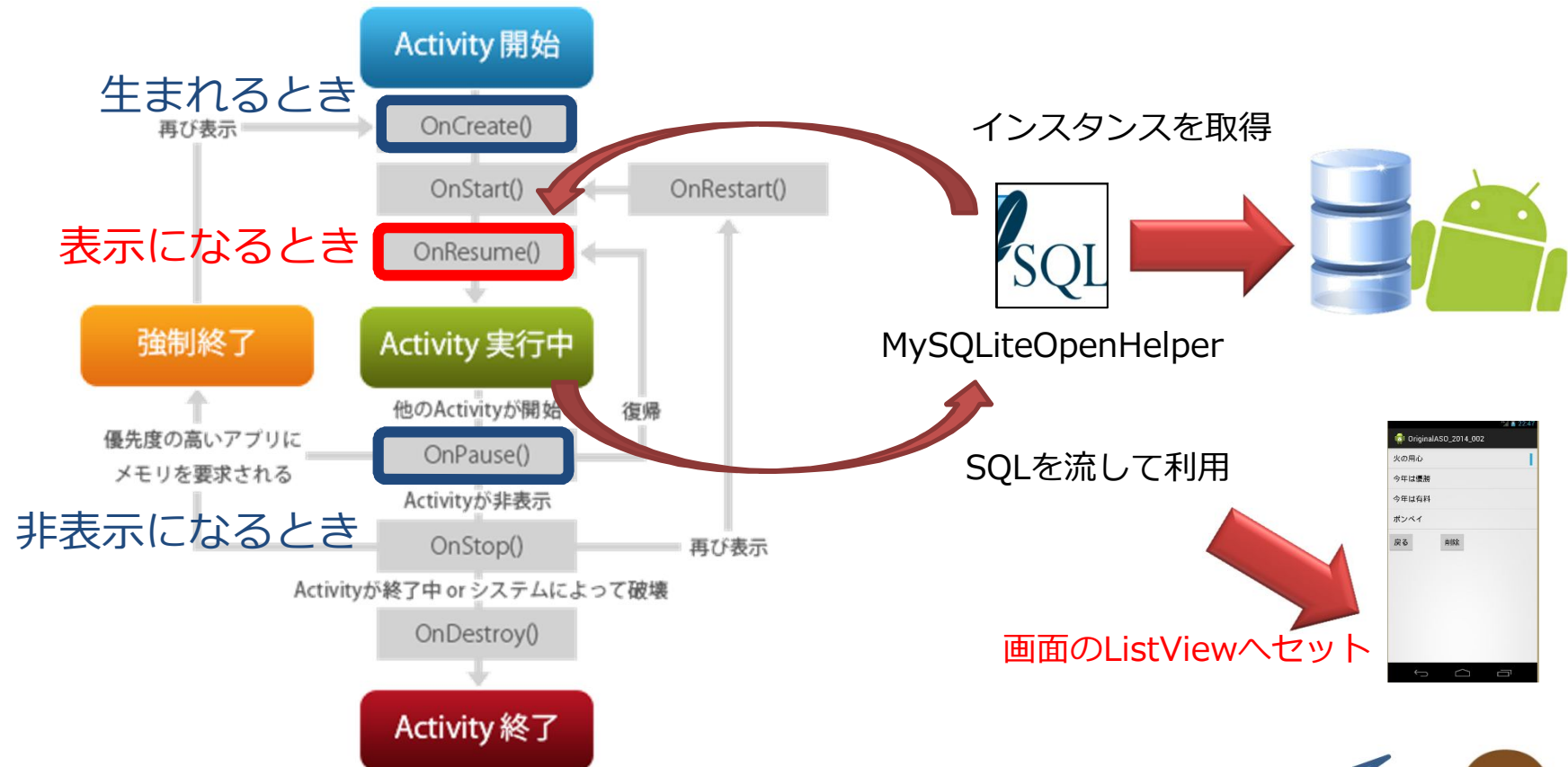
メンテナンス画面の処理 (MaintenanceActivity.java)



メイン画面 (MainActivity) 側の呼び出し処理は完成したので、今度はメンテナンス画面 (HitokotoActivity) への画面遷移後の処理を実装します



画面表示のタイミングでSELECTを実行



まず、メンテナンス画面表示のタイミングで、SELECT 処理を呼び出し、ListViewに表示します。



必要な部品クラスをimport

```
MaintenanceActivity.java  MainActivity.java  MySQLiteOpenHelper.java
OriginalASO_2014_002  src  jp.ac.st.asojuku.original2014002  G
1 package jp.ac.st.asojuku.original2014002;
2
3 import android.app.Activity;
4 import android.database.sqlite.SQLiteDatabase;
5 import android.database.sqlite.SQLiteCursor;
6 import android.database.sqlite.SQLiteOpenHelper;
7 import android.os.Bundle;
8 import android.util.Log;
9 import android.view.View;
10 import android.widget.AdapterView;
11 import android.widget.Button;
12 import android.widget.ListView;
13 import android.widget.SimpleCursorAdapter;
14 import android.widget.Toast;
15
16 /**
17  * @author masatoki
18  *
19  */
20 public class MaintenanceActivity extends Activity implements View.OnClickListener
21 {
22     SQLiteDatabase db = null;
```

※ミニメッセージ（トースト）表示に必要なクラス

DB操作に必要な部品クラスをimportしていない場合は、importします。



メンテナンス画面のソースコード (MaintenanceActivity.java)

```
16 /**  
17  * @author masatoki  
18  *  
19  */  
20 public class MaintenanceActivity extends Activity implements View.OnClickListener, AdapterView  
21 {  
22     > // SQLiteデータベース空間を操作するインスタンス変数を宣言  
23     > SQLiteDatabase sdb = null;  
24     > // MySQLiteOpenHelperを操作するインスタンス変数を宣言  
25     > MySQLiteOpenHelper helper = null;  
26     <  
27     > // リストにて選択したHitokotoテーブルのレコードの「_id」カラム値を保持する変数の宣言  
28     > int selectedID = -1;  
29     > // リストにて選択した行番号を保持する変数の宣言  
30     > int lastPosition = -1;  
31     <  
32     > @Override  
33     > protected void onCreate(Bundle savedInstanceState) {  
34     >     > // TODO 自動生成されたメソッド・スタブ  
35     > }
```

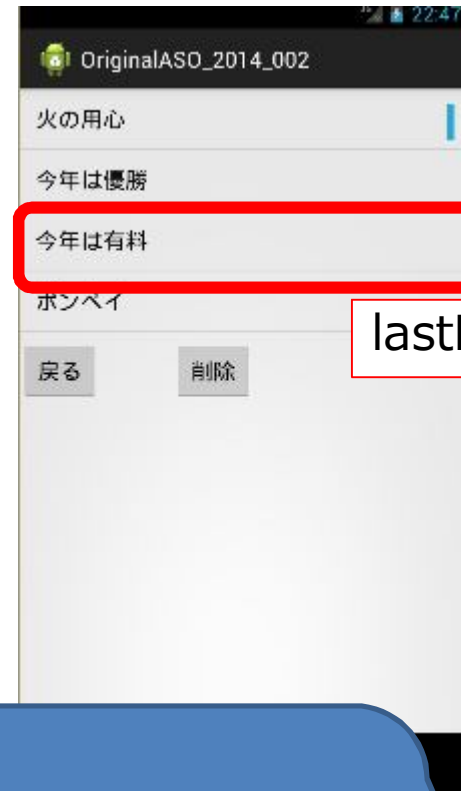
メンテナンス画面（MaintenanceActivity）全体で使用する変数
（クラスフィールド変数）を宣言して用意します。



selectedIDとlastPosition

Hitokoto表

_id	pharase
selectedID	



lastPosition

先ほど宣言した変数のうち、「selectedID」はListviewの選択行にセットされたHitokoto表のレコードデータのうち「_id」カラムの値を、「lastPosition」はListviewでの選択した行番号を、それぞれセットしておく変数です。



Listviewでは、 setOnItemClickListenerが反応する

```
8 import android.util.Log;
9 import android.view.View;
10 import android.widget.AdapterView;
11 import android.widget.Button;
12 import android.widget.ListView;
13 import android.widget.SimpleCursorAdapter;
14 import android.widget.Toast;
15
16 /**
17  * @author masatoks
18  *
19  */
20 public class MaintenanceActivity extends Activity implements
21     View.OnClickListener, AdapterView.OnItemClickListener {
22
23     // SQLiteデータベース空間を操作するインスタンス変数を宣言
24     SQLiteDatabase sdb = null;
25     // MySQLiteOpenHelperを操作するインスタンス変数を宣言
26     MySQLiteOpenHelper helper = null;
27
28     // リストにて選択したHitokotoテーブルのレコードの「_id」カラム値
29     int selectedID = -1;
30     // リストにて選択した行番号を保持する変数の宣言
31     int lastPosition = -1;
32
33     @Override
34     protected void onCreate(Bundle savedInstanceState) {
```

ところで、Listviewをクリックした時のイベントには、onClickListnerではなく、onItemClickListenerが反応します。新たに、AdapterViewをimportして、OnItemClickListenerをimplementsします



メンテナンス画面のソースコード (MaintenanceActivity.java)

```
32 > @Override  
33 > protected void onCreate(Bundle savedInstanceState) {  
34 >     // TODO 自動生成されたメソッド・スタブ  
35 >     super.onCreate(savedInstanceState);  
36 >     setContentView(R.layout.maintenance_activity);  
37 > }  
38  
39 > @Override  
40 > protected void onResume() {  
41 >     // TODO 自動生成されたメソッド・スタブ  
42 >     super.onResume();  
43  
44 >     // 各view部品を操作するidを取得  
45 >     Button btnDelete = (Button)findViewById(R.id.btnDelete);  
46 >     Button btnMainte_Back = (Button)findViewById(R.id.btnMAINTE_BACK);  
47 >     ListView lstHitokoto = (ListView)findViewById(R.id.lvHITOKOTO);  
48  
49 >     // 各ButtonにOnClickListenerをセット  
50 >     btnDelete.setOnClickListener(this);  
51 >     btnMainte_Back.setOnClickListener(this);  
52  
53 >     // ListViewにItemClickListenerをセット  
54 >     lstHitokoto.setOnItemClickListener(this);  
55  
56 >     // ListViewにDBの値をセット  
57 >     this.setDBValueToList(lstHitokoto);  
58  
59 > }
```

画面表示時に行う処理は、これまでのように、onResume()メソッド内に書き込みます。「戻る」「削除」ボタン、一覧表示のListviewの操作用変数をidを基に取得して、それぞれイベントリスナーをセットします。また、エラーにならないように、空のonClickメソッド、onItemClickメソッドをメニューのソース→メソッドのオーバーライド／実装で作成しておきます。



DBのSELECT処理実行を呼び出す

```
32 > @Override  
33 > protected void onCreate(Bundle savedInstanceState) {  
34 >     // TODO 自動生成されたメソッド・スタブ  
35 >     super.onCreate(savedInstanceState);  
36 >     setContentView(R.layout.maintenance_activity);  
37 > }  
38  
39 > @Override  
40 > protected void onResume() {  
41 >     // TODO 自動生成されたメソッド・スタブ  
42 >     super.onResume();  
43  
44 >     // 各view部品を操作するidを取得  
45 >     Button btnDelete = (Button)findViewById(R.id.btnDelete);  
46 >     Button btnMainte_Back = (Button)findViewById(R.id.btnMAINTE_BACK);  
47 >     ListView lstHitokoto = (ListView)findViewById(R.id.LvHITOKOTO);  
48  
49 >     // 各ButtonにOnClickListenerをセット  
50 >     btnDelete.setOnClickListener(this);  
51 >     btnMainte_Back.setOnClickListener(this);  
52  
53 >     // ListViewにOnItemClickListenerをセット  
54 >     lstHitokoto.setOnItemClickListener(this);  
55  
56 >     // ListViewにDBの値をセット  
57 >     this.setDBValuetToList(lstHitokoto);  
58  
59 > }
```

説明してきたように、この表示／再表示（onResume）のタイミングで、DBからSELECT→Listviewへ格納実行を呼び出します。長くなると読みづらくなるので、1つのprivateなメソッドにまとめて、「setDBValuetToList」と名前をつけ、ここでは呼び出すだけにしましょう。



呼び出すsetDBValuetoListメソッドを作る

```
17 > /**  
18 >  * 引数のListViewにDBのデータをセット  
19 >  * @param lstHitokoto 対象となるListView  
20 >  */  
21 > private void setDBValuetoList(ListView lstHitokoto){  
22 >   
23 >     SQLiteCursor cursor = null;  
24 >   
25 >     // クラスのフィールド変数がNULLなら、データベース空間オープン  
26 >     if(sdb == null) {  
27 >         helper = new MySQLiteOpenHelper(getApplicationContext());  
28 >     }  
29 >     try{  
30 >         sdb = helper.getWritableDatabase();  
31 >     }catch(SQLiteException e){  
32 >         // 異常終了  
33 >         Log.e("ERROR", e.toString());  
34 >     }  
35 >     // MySQLiteOpenHelperにSELECT文を実行させて結果のカーソルを受け取る  
36 >     cursor = this.helper.selectHitokotoList(sdb);  
37 >   
38 >     // dblayout: ListViewにさらにレイアウトを指定するもの  
39 >     int db_layout = android.R.layout.simple_list_item_activated_1;  
40 >     // from: カーソルからListviewに指定するカラムの値を指定するもの  
41 >     String[] from = {"phrase"};  
42 >     // to: Listviewの中に指定したdb_layoutに配置する、各行のview部品のid  
43 >     int[] to = new int[] {android.R.id.text1};  
44 >   
45 >     // ListViewにセットするアダプターを生成  
46 >     // カーソルをもとに、fromの列から、toのViewへ値のマッピングが行なわれる。  
47 >     SimpleCursorAdapter adapter =  
48 >         new SimpleCursorAdapter(this, db_layout, cursor, from, to, 0);  
49 >   
50 >     // アダプターを設定します  
51 >     lstHitokoto.setAdapter(adapter);  
52 > }
```

先ほど呼び出す形にした「setDBValuetoList」を作ります。
まずデータベースのオープン処理を書き込みます。



呼び出すsetDBValuetolistメソッドを作る

```
16 <
17 < /**
18 <  * 引数のListViewにDBのデータをセット
19 <  * @param lstHitokoto 対象となるListView
20 <  */
21 < private void setDBValuetolist(ListView lstHitokoto){
22 <
23 <     SQLiteCursor cursor = null;
24 <
25 <     // クラスのフィールド変数がNULLなら、データベース空間オープン
26 <     if(sdb == null) {
27 <         helper = new MySQLiteOpenHelper(getApplicationContext());
28 <     }
29 <     try{
30 <         sdb = helper.getWritableDatabase();
31 <     }catch(SQLiteException e){
32 <         // 異常終了
33 <         Log.e("ERROR", e.toString());
34 <     }
35 <     // MySQLiteOpenHelperにSELECT文を実行させて結果のカーソルを受け取る
36 <     cursor = this.helper.selectHitokotoList(sdb);
37 <
38 <     // dblayout: ListViewにさらにレイアウトを指定するもの
39 <     int db_layout = android.R.layout.simple_list_item_activated_1;
40 <     // from: カーソルからListviewに指定するカラムの値を指定するもの
41 <     String[] from = {"phrase"};
42 <     // to: Listviewの中に指定したdb_layoutに配置する、各行のview部品のid
43 <     int[] to = new int[] {android.R.id.text1};
44 <
45 <     // ListViewにセットするアダプターを生成
46 <     // カーソルをもとに、fromの列から、toのViewへ値のマッピングが行なわれる。
47 <     SimpleCursorAdapter adapter =
48 <         new SimpleCursorAdapter(this, db_layout, cursor, from, to, 0);
49 <
50 <     // アダプターを設定します
51 <     lstHitokoto.setAdapter(adapter);
52 < }
```

次に、DBからレコードを指し示すカーソル（curosor）を取得、Listviewへ設定する処理を書きます。



アダプターとは

```
// MySQLiteOpenHelperにSELECT文を実行させて結果のカーソルを受け取る↓  
cursor = this.helper.selectHitokotoList(sdb);↓  
  
// dblayout: ListViewにさらにレイアウトを指定するもの↓  
int db_layout = android.R.layout.simple_list_item_activated_1;↓  
// from: カーソルからListviewに指定するカラムの値を指定するもの↓  
String[] from = {"phrase"};↓  
// to: Listviewの中に指定したdb_layoutに配置する、各行のview部品のid↓  
int[] to = new int[]{android.R.id.text1};↓  
  
// ListViewにセットするアダプターを生成↓  
// カーソルをもとに、fromの列から、toのViewへ値のマッピングが行なわれる。↓  
SimpleCursorAdapter adapter =  
    new SimpleCursorAdapter(this, db_layout, cursor, from, to, 0);↓  
  
// アダプターを設定します↓  
listHitokoto.setAdapter(adapter);↓
```

先ほどの上図のようなソースコードの中で、「アダプター (Adapter)」というものを使用しています。アダプターとは、何でしょうか。



アダプターはデータの塊をリスト部品に紐付ける



アダプターとは、データの塊と、リストなどの部品とを、紐付けることができる、便利なクラスです



SimpleCursorAdapter



Hitokoto表

_id	pharase

ここで使用したSimpleCursorAdapterは、中でも、データの塊を、ループを繰り返さずに一度にごっそりリストにセットできる、便利なアダプタです。



SimpleCursorAdapter の作り方

```
// ListViewにセットするアダプターを生成↓  
// カーソルをもとに、fromの列から、toのViewへ値のマッピングが行なわれる。↓  
SimpleCursorAdapter adapter =  
    new SimpleCursorAdapter(this, db_layout, cursor, from, to, 0);  
  
// アダプターを設定します↓  
lstHitokoto.setAdapter(adapter);
```

```
public SimpleCursorAdapter (Context context, int layout, Cursor c, String[] from, int[] to, int flags)
```

SimpleCursorAdapterをnewするコンストラクタの第1引数 contextには、画面やアプリなどのContextを指定します。第2引数 layoutはListviewに使用するレイアウトファイルのIDを指定します。第3引数 cにはリストに表示するデータの塊のカーソルを指定します。第4引数 fromはカーソルで指定したデータの塊（テーブル）から、表示したいカラム名をString配列で指定します。第5引数 toは第4引数 fromで指定したカラムの値を表示するレイアウトファイル内のViewのIdをint配列で指定します。



登録された一言をリスト表示、完成



これで、登録したデータを一言表示画面で一
覧表示させることまで完成したはずです。実
行して確認してみてください。



次はdelete文



次は、いよいよ最後に残った、一覧からの選択、DELETE文実行に入っていきたいと思います。



githubにアップロードして提出

Androidプロジェクトをgithubに
アップロード



v.1.0.0

どこでも再開できるように、できあがったところまで、まめにGithubにアップロードしておいてください。



内部DBを使おう 【続編 1 : SELECT】



おしまい

by masatokg