SQLite and App

Dr. Abraham (Abey) Campbell

Sqlite

- In Version 2, we perform SQL operations.
- We create a database.
- We create a table inside that database.
- We insert data in that table.

SQLite – A Brief Introduction

SQLite

- Public-domain, lightweight RDBMS, follows nearly entire SQL-92
- Designed by D. Richard Hipp, 2000
- Main features:
  - Serverless
  - Zero configuration
  - Cross-platform
  - Small Runtime Footprint
  - Highly Reliable
  - Self-contained

SQLite and Mobile Apps

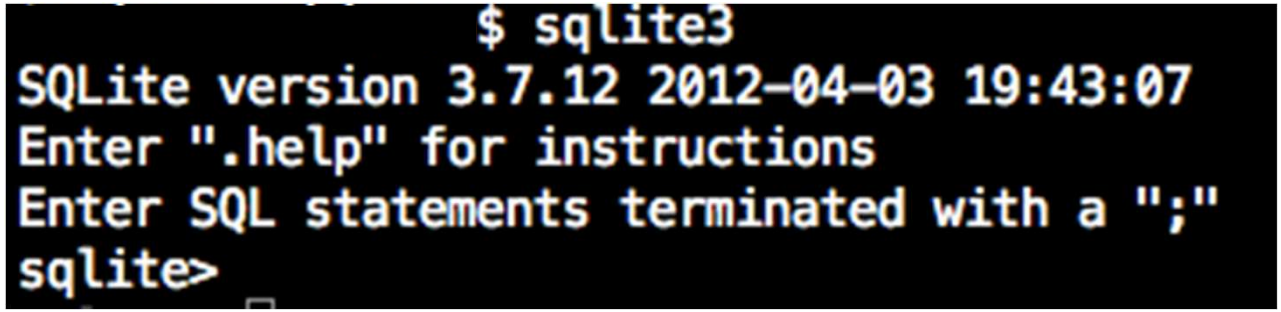
- “All” mobile applications use SQLite:
  - Social Media: Whatsapp, Skype,
  - Web browser: Mozilla Firefox, Chrome…
  - …

SQLite

- Command line interface
- “dot” commands:
  - used for management
  - Named as they start with a dot:
    - .tables
    - .schema
    - .output
    - .quit

Using SQLite

- Launch SQLite: **sqlite3**



- Create a SQLite file: **sqlite3 <filename>**

Interacting with SQLite Database

- Open a SQLite file: sqlite3 <filename>

```
C:\PythonExamples>sqlite3 messages.db
SQLite version 3.8.4.1 2014-03-11 15:27:36
Enter ".help" for usage hints.
sqlite> .tables
messages
sqlite> .schema messages
CREATE TABLE messages (id int auto_increment primary key, sender varchar(10), receiver varchar(10),
time datatime, content text);
sqlite>
```

- List names of tables: **.tables**
- Show the structure of a table:

  .schema <table name>

Creating Tables

- Example:
  - Lecturer table:

ID	First_Name	Surname	Position

```
sqlite> CREATE TABLE Lecturer
   ...> (
   ...>     ID          INTEGER,
   ...>     First_Name  VARCHAR(20),
   ...>     Surname     VARCHAR(20),
   ...>     Position    VARCHAR(30)
   ...> );
sqlite> .tables
Lecturer
```

Inserting information

- Example:

ID	First_Name	Surname	Position
1	Joe	Carthy	College Principal
2	Pavel	Gladyshev	MSC Director
3	Fergus	Toolan	Lecturer

```
sqlite> INSERT INTO Lecturer VALUES (1, 'Joe', 'Carthy', 'College Principal');
sqlite> INSERT INTO Lecturer VALUES (2, 'Pavel', 'Gladyshev', 'MSC Director');
sqlite> INSERT INTO Lecturer VALUES (3, 'Fergus', 'Toolan', 'Lecturer');
```

```
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
```

Datatype Overview

- Datatypes in SQLite can be broadly divided into five categories:
  - INTEGER
  - TEXT (CHAR, VARCHAR, TEXT, …)
  - NONE
  - REAL (REAL, DOUBLE, FLOAT, …)
  - NUMERIC (BOOLEAN, DATETIME, NUMERIC, …)

Using SELECT

(1)

```
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
sqlite>
```

(2)

```
sqlite> SELECT First_Name FROM Lecturer;
Joe
Pavel
Fergus
sqlite>
```

(3)

```
sqlite> SELECT First_Name, Position FROM Lecturer;
Joe|College Principal
Pavel|MSC Director
Fergus|Lecturer
sqlite>
```

WHERE Clause example

(1)
```
sqlite> SELECT * FROM Lecturer WHERE ID = 2;
2|Pavel|Gladyshev|MSC Director
sqlite>
```

(2)
```
sqlite> SELECT First_Name,Surname,Position FROM Lecturer WHERE ID = 2;
Pavel|Gladyshev|MSC Director
sqlite>
```

(3)
```
sqlite> SELECT First_Name,Surname,Position
   ...> FROM   Lecturer
   ...> WHERE  ID = 2;
Pavel|Gladyshev|MSC Director
sqlite>
```

ORDER BY example

```
sqlite> SELECT * FROM Lecturer
   ...> ORDER BY First_Name DESC;
2|Pavel|Gladyshev|MSC Director
1|Joe|Carthy|College Principal
3|Fergus|Toolan|Lecturer
sqlite>
sqlite> SELECT * FROM Lecturer
   ...> ORDER BY First_Name ASC;
3|Fergus|Toolan|Lecturer
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
```

Example of DELETE

```
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
4|John|Smith|Senior Lecturer
5|Jane|Smith|Officer
6|Joe|Bloggs|Lecturer
7|James|Bloggs|Officer
sqlite>
sqlite> DELETE FROM Lecturer
   ...> WHERE Surname='Bloggs';
sqlite>
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
4|John|Smith|Senior Lecturer
5|Jane|Smith|Officer
```

(I)

```
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
4|John|Smith|Senior Lecturer
5|Jane|Smith|Officer
sqlite>
sqlite> DELETE FROM Lecturer
   ...> WHERE Surname='Smith' AND First_Name='John';
sqlite>
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
5|Jane|Smith|Officer
```

(II)

```
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
5|Jane|Smith|Officer
sqlite>
sqlite> DELETE FROM Lecturer;
sqlite>
sqlite> SELECT * FROM Lecturer;
sqlite>
```

(III)

Example of UPDATE

```
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
4|John|Smith|Senior Lecturer
5|Jane|Smith|Officer
6|Joe|Bloggs|Lecturer
sqlite>
sqlite> UPDATE Lecturer
   ...> SET Position='Senior Lecturer'
   ...> WHERE Surname='Bloggs';
sqlite>
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal
2|Pavel|Gladyshev|MSC Director
3|Fergus|Toolan|Lecturer
4|John|Smith|Senior Lecturer
5|Jane|Smith|Officer
6|Joe|Bloggs|Senior Lecturer
```

Example of ALTER Command

```
sqlite> ALTER TABLE Lecturer
   ...> ADD COLUMN Office TEXT;
sqlite> SELECT * FROM Lecturer;
1|Joe|Carthy|College Principal|
2|Pavel|Gladyshev|MSC Director|
3|Fergus|Toolan|Lecturer|
```

```
sqlite> .schema
CREATE TABLE Lecturer
(
   ID            INTEGER,
   First_Name    TEXT,
   Surname       TEXT,
   Position      TEXT
   Office TEXT);
```

```
sqlite> .tables
Lecturer   course     lecturers
sqlite>
sqlite> ALTER TABLE Lecturer
   ...> RENAME TO CSILecturer;
sqlite>
sqlite> .tables
CSILecturer   course       lecturers
```

Auto Increment

```
sqlite> CREATE TABLE csicourse
   ...> (
   ...>     CID            INTEGER PRIMARY KEY AUTOINCREMENT,
   ...>     Name           TEXT,
   ...>     lecturer       INTEGER
   ...> );
```

```
sqlite> INSERT INTO csicourse VALUES (NULL,'Project Management',2);
sqlite> INSERT INTO csicourse VALUES (NULL,'Digital Forensic Analysis',1);
```

```
sqlite> SELECT * FROM csicourse;
1|Project Management|2
2|Digital Forensic Analysis|1
```

Examples of JOIN

```
sqlite> SELECT lecturers.name, course.name
   ...> FROM lecturers, course
   ...> WHERE lecturers.ID = course.lecturer;
Fergus Toolan|Digital Forensic Analysis
Fergus Toolan|Discrete Mathematics
Joe Carthy|Information Retrieval
Joe Carthy|Project Management
```

```
sqlite> INSERT INTO lecturers VALUES (3,'Pavel Gladyshev');
sqlite> SELECT lecturers.Name, course.Name
   ...> FROM lecturers
   ...> LEFT JOIN course ON
   ...> lecturers.ID = course.lecturer;
Fergus Toolan|Digital Forensic Analysis
Fergus Toolan|Discrete Mathematics
Joe Carthy|Information Retrieval
Joe Carthy|Project Management
Pavel Gladyshev|
```

Sqlite – Candy App

- In Version 2, we perform SQL operations.
- We create a database.
- We create a table inside that database.
- We insert data in that table.

Candy Table (1 of 2)

- We keep it simple and store the following data:
  - Candy id, candy name, and candy price
  - ➔ 3 columns (int auto increment, string, double or float)

Candy Table (2 of 2)

Possible data in the candy table:

Id	Name	Price
1	Chocolate fudge	1.99
2	Walnut chocolate	2.99
3	Hazelnut chocolate	3.49

Candy Class

- As part of the Model, we write a Candy class to encapsulate a Candy.
- Three instance variables:
  - id, an int
  - name, a String
  - price, a double
- Constructor, accessors, mutators, toString.
- See Example

```java
public class Candy {
  private int id;
  private String name;
  private double price;

  public Candy( int newId, String newName, double newPrice ) {
    setId( newId );
    setName( newName );
    setPrice( newPrice );
  }

  public void setId( int newId ) {
    id = newId;
  }

  public void setName( String newName ) {
    name = newName;
  }

  public void setPrice( double newPrice ) {
    if( newPrice >= 0.0 )
      price = newPrice;
  }

  public int getId( ) {
    return id;
  }

  public String getName( ) {
    return name;
  }

  public double getPrice( ) {
    return price;
  }

  public String toString( ) {
    return id + "; " + name + "; " + price;
  }
}
```

Sqlite Related Classes

- SQLiteOpenHelper: we extend this abstract class to manage a database along with its version.
- We must override its onCreate and onUpgrade methods.

DatabaseManager

- As part of our Model, we code the DatabaseManager class.
- It contains methods to perform SQL operations related to the Candy class.

SQLiteOpenHelper

```
public class DatabaseManager extends SQLiteOpenHelper
    {
    private static final String  DATABASE_NAME =
          "candyDB";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_CANDY = "candy";
    // other constants for column names
```

onCreate and onUpgrade

- Inside onCreate: create the candy table
- Inside onUpgrade: drop the candy table and call onCreate to recreate it

Sqlite Related Classes

- SQLiteDatabase: includes methods to execute SQL statements: insert, delete, update, select
- Both onCreate and onUpgrade include an SQLiteDatabase parameter.

SQLiteDatabase: execSql Method

void execSql(String sql)

- Executes an SQL query that does not return data (for example, create, insert, delete, update – not select)

onCreate Method

```
// create candy table
public void onCreate( SQLiteDatabase db ) {
    // build sql create statement
    String sqlCreate = "create table " +  TABLE_CANDY +
        "( " + ID;
    sqlCreate += " integer primary key autoincrement, " +
        NAME;
    sqlCreate += " text, " + PRICE + " real )";
    db.execSQL( sqlCreate );
 }
```

onUpgrade Method

```
// drop candy table, recreate it
public void onUpgrade( SQLiteDatabase db,
        int oldVersion, int newVersion ) {
    // Drop old table if it exists
    db.execSQL( "drop table if exists " +
                TABLE_CANDY );
    // Re-create table(s)
    onCreate( db );
}
```

DatabaseManager Methods (1 of 2)

Inside DatabaseManager, we also provide methods to:

- Insert a candy
- Delete a candy based on its id
- Update a candy (name and price, based on id)

DatabaseManager Methods (2 of 2)

We add more methods to:

- Select a candy based on its id
- Select all the candies

Insert Method (1 of 2)

- One parameter, a Candy reference (to a Candy object that we insert in the candy table)
- Get a reference to our SQLiteDatabase
- Define the insert SQL String
- Execute the insert SQL statement

Insert Method (2 of 2)

```
public void insert( Candy candy ) {
   // insert candy
}
```

Accessing the Database

- We use the getWritableDatabase method of the SQLiteOpenHelper class.

  SQLiteDatabase getWritableDatabase( )

- ➔ It creates or opens a database that will be used for reading and writing.
- ➔ The first time this method is called, onCreate, onUpgrade, and onOpen are automatically called.

Insert Method (1 of 5)

```
public void insert( Candy candy ) {
  // Database Manager extends SQLiteOpenHelper
  // Thus, this "is a" SQLiteOpenHelper
  SQLiteDatabase db = this.getWritableDatabase( );
  // construct insert SQL String
  // execute the SQL query
  // close db
}
```

Insert Method (2 of 5)

- To execute the SQL insert, we call the execSQL method with db (the SQLiteDatabase reference), passing the appropriate SQL String.

Insert Method (3 of 5)

```
public void insert( Candy candy ) {
  SQLiteDatabase db = this.getWritableDatabase( );
  String sqlInsert = "…";
  db.execSQL( sqlInsert );
  db.close( );
}
```

Insert Method (4 of 5)

- The candy table has three columns (id, name, price).
- The type of the first column (id) is int auto-increment.
- ➔ We use null.
- We use the name and price of the candy parameter for the second and third columns.

Insert Method (5 of 5)

```
public void insert( Candy candy ) {
    SQLiteDatabase db = this.getWritableDatabase( );
    String sqlInsert = "insert into " +
        TABLE_CANDY;
    sqlInsert += " values( null, '" + candy.getName( );
    sqlInsert += "', '" + candy.getPrice( ) + "' )";
    db.execSQL( sqlInsert );
    db.close( );
}
```

Update and Delete (1 of 2)

- The updateById and deleteById methods are similar to the insert method.
- updateById takes three parameters: an int for the id, a String for the updated name, and a double for the updated price.
- deleteById takes one parameter, an int representing a Candy id.

Update and Delete (2 of 2)

- Similar to the insert method:
  - Get a reference to the SQLiteDatabase
  - Construct the SQL query as a String
  - Execute it

More Methods

- We provide methods to select a candy based on its id and to select all the candies from the candy table.

selectById Method (1 of 2)

```
public Candy selectById( int id ) {
  // select the row in the candy table
  // whose id value is id
  // return a reference to the Candy object
  // stored in that row
}
```

selectById Method (2 of 2)

- We need to execute an SQL select query.
- Since we select based on the value of the primary key (id) of the table, the select query will return a table of 0 or 1 row (a select query returns a table).
- If it returns a table of 1 row, we need to access the columns of that row.

rawQuery Method

- The rawQuery method executes an SQL select query.

public Cursor rawQuery(String sql, String [ ] selectionArgs)

- If sql String contains ? placeholders, then we provide values in selectionArgs, otherwise selectionArgs is null.

Sqlite Related Classes/Interfaces

- The Cursor interface encapsulates a table returned by a select statement (a select statement returns a table).
- It provides methods to loop through the rows and columns of that table.

Cursor Interface (1 of 2)

- Move this Cursor to first row:

  boolean moveToFirst( )
- Move this Cursor to the next row:

  boolean moveToNext( )
- These methods return false if there is not another row to process.

Cursor Interface (2 of 2)

- To retrieve the value of a column of the current row, we use the methods:

  DataType getDataType(int columnIndex)

- For example:

  int getInt(int columnIndex)

  double getDouble(int columnIndex)

  String getString(int columnIndex)

  …

selectById Method (1 of 6)

```
SQLiteDatabase db = this.getWritableDatabase( );
// construct sqlQuery, a select query
// call rawQuery to execute the select query
Cursor cursor = db.rawQuery( sqlQuery, null );
// process the results
// build a Candy object, then return it
```

selectById Method (2 of 6)

```
// construct sqlQuery, a select query
String sqlQuery = "select * from " + TABLE_CANDY;
sqlQuery += " where " + ID + " = " + id;
```

selectById Method (3 of 6)

```
// process the result of the query
Candy candy = null;
if( cursor.moveToFirst( ) )
    candy = new Candy(
        Integer.parseInt( cursor.getString( 0 ) ),
                          cursor.getString( 1 ),
                          cursor.getDouble( 2 ) );
return candy;
```

selectById Method (4 of 6)

- select * from candy
- Build and return an ArrayList of Candy objects
- Loop through all the rows using moveToNext method

selectById Method (5 of 6)

```
public ArrayList<Candy> selectAll( ) {
    // select all the rows in the candy table
    // return an ArrayList of Candy objects
}
```

selectById Method (6 of 6)

```
ArrayList<Candy> candies = new ArrayList<Candy>( );
while( cursor.moveToNext( ) ) {
    Candy currentCandy = new Candy( Integer.parseInt(
                cursor.getString( 0 ) ),
                cursor.getString( 1 ), cursor.getDouble( 2 ) );
    candies.add( currentCandy );
}
db.close( );
return candies;
```

InsertActivity Class (1 of 2)

- Now that our Model is finalized, we add code to the InsertActivity class (its insert method) to actually add a candy to the database.
- We include a DatabaseManager instance variable.
- We instantiate it inside the onCreate method.

InsertActivity Class (2 of 2)

- Inside the insert(View) method, we call the insert method of the DatabaseManager class.
- We show a Toast to confirm that the candy was added.

Toast (1 of 4)

- A Toast is a small visual element that is displayed for a short amount of time.
- It automatically disappears after that time.
- The Toast class is in the android.widget package.

Toast (2 of 4)

- To create a Toast, use one of the static makeText methods:

public static Toast makeText(Context context, CharSequence text, int duration)

Toast (3 of 4)

- context ➔ *this* (an Activity "is a" Context)
- text ➔ what we want to display (a String "is a" CharSequence)
- duration: number of seconds (use the LENGTH_SHORT (3 seconds ?) or LENGTH_LONG (5 seconds ?) constants of the Toast class)

Toast (4 of 4)

- The makeText method creates a Toast but it does not show it.
- We call the show instance method to show it.
- public void show( )

```
Toast.makeText( this, "Candy added",
      Toast.LENGTH_LONG ).show( );
```

Testing the App (1 of 2)

- When we run the app, enter some data and click on the ADD icon, the Toast message appears.
- If we want to check that a new row is added to the candy table, we can call the selectAll method of the DatabaseManager class and loop through the resulting ArrayList of Candy objects.

Testing the App (2 of 2)

- We can write these statements at the end of the insert method and check the output in Logcat:

```
ArrayList<Candy> candies = dbManager.selectAll( );
for( Candy candy : candies )
   Log.w( "MainActivity", "candy = " + candy.toString( ) );
```

Deleting a Candy

- In Version 3, we enable the user to delete a candy from the database.
- We create and code the DeleteActivity class and provide a View for it.
- ➔ DeleteActivity.java

Update the Menu Selection

```
public boolean onOptionsItemSelected( MenuItem item ) {
    switch ( item.getItemId( ) ) {
      …
    case R.id.action_delete:
        Intent deleteIntent = new Intent(
         this, DeleteActivity.class );
        this.startActivity( deleteIntent );
        …
```

Deleting a Candy (1 of 4)

- We add an activity element to the AndroidManifest.xml file for DeleteActivity.

Deleting a Candy (2 of 4)

- We provide a list of radio buttons, one per candy.
- The user selects one to delete the corresponding candy.
- We do not know in advance how many candies there are in the database.
- ➔ We need to build the GUI by code.

Deleting a Candy (3 of 4)

- The list of radio buttons may be too big for the screen.
- ➔ We place it inside a ScrollView.

Deleting a Candy (4 of 4)

- We could go back to the previous activity (the main screen) when the user selects a candy to delete.
- But maybe the user wants to delete more than one candy ... so we stay on this screen.
- … and provide a "back" button that takes the user back to the main screen.
- We place the ScrollView and the "Back" button inside a Relative Layout.

Deleting a Candy: GUI

Deleting a Candy

- We code a separate method, updateView, to code the GUI.
- When the user selects a candy to delete, not only do we delete it from the database, but we also refresh the current screen by calling updateView.

updateView Method (1 of 4)

- We call selectAll from the DatabaseManager class.

```
ArrayList<Candy> candies =
  dbManager.selectAll( );
```

updateView Method (2 of 4)

- We loop through the results and build the radio buttons:

```
for ( Candy candy : candies ) {
   RadioButton rb = new RadioButton( this );
   …
```

updateView Method (3 of 4)

- We set the id of each radio button to the id of the candy.
- We set the text of each radio button to a String representation of the candy by calling the toString method of the Candy class.

updateView Method (4 of 4)

- We loop through the results and build the radio buttons:

```
rb.setId( candy.getId( ) );
rb.setText( candy.toString( ) );
// add rb to a RadioGroup group
// in order to make them mutually exclusive
group.addView( rb );
```

Event Handling (1 of 4)

- We need to set up event handling.
- We code a private class that implements the RadioGroup.OnCheckedChangeListener interface.
- That class overrides the onCheckedChanged method.

Event Handling (2 of 4)

- The onCheckedChanged method includes a parameter that represents the id of the radio button selected.
- … which is the id of the candy that we need to delete.

Event Handling (3 of 4)

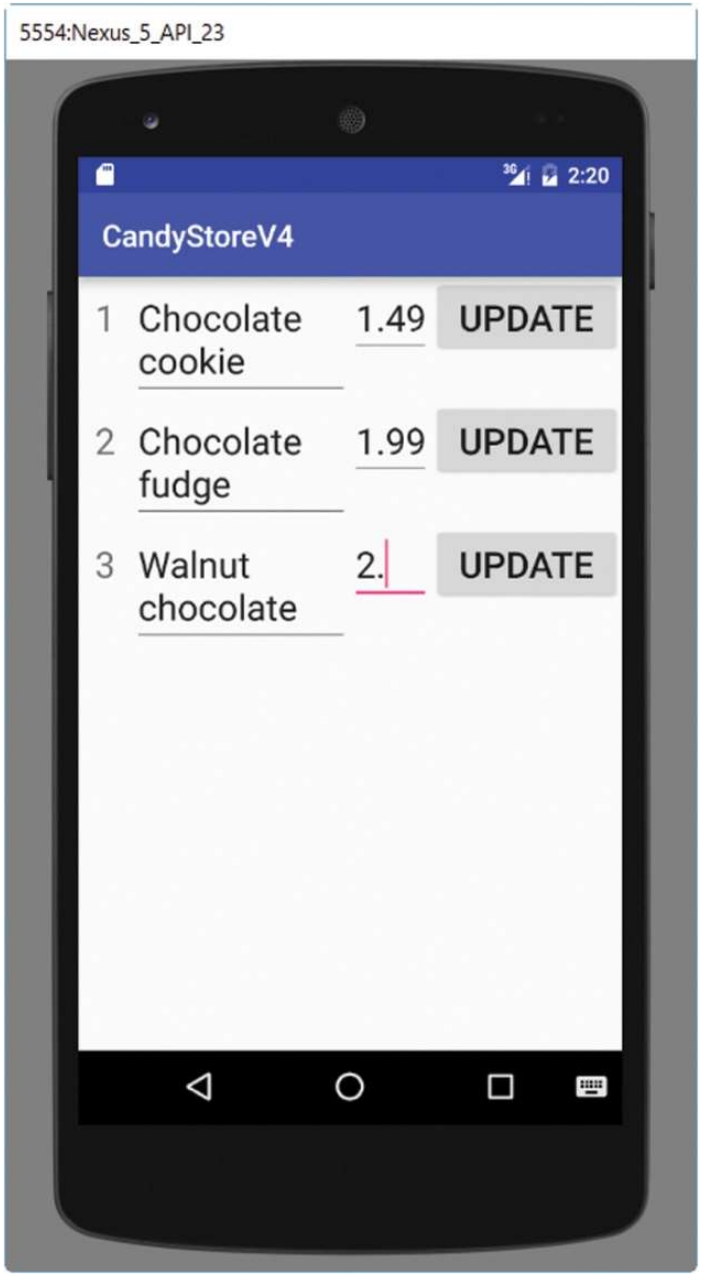
```
public void onCheckedChanged(
        RadioGroup group,
        int checkedId ) {
   // delete candy from database
   dbManager.deleteById( checkedId );
   …
```

Event Handling (4 of 4)

```
// confirm to the user
Toast.makeText( DeleteActivity.this, "Candy
    deleted", Toast.LENGTH_SHORT ).show( );
// update screen (the list of candies has changed)
updateView( );
```

Updating a Candy (1 of 2)

- In Version 4, we enable the user to update a candy: change its name or its price.
- We create and code the UpdateActivity class ➔ UpdateActivity.java
- The View is also dynamic ➔ we build it by code.
- We update the AndroidManifest.xml file.


5554:Nexus_5_API_23
2:20
CandyStoreV4
1 Chocolate cookie
1.49
UPDATE
2 Chocolate fudge
1.99
UPDATE
3 Walnut chocolate
2.
UPDATE

Updating a Candy (2 of 2)

- The list of candies to display for potential editing may be too big for the screen.
- ➔ We place it inside a ScrollView.

Update the Menu Selection

```
public boolean onOptionsItemSelected( MenuItem item ) {
    switch ( item.getItemId( ) ) {
      …
    case R.id.action_delete:
        Intent updateIntent = new Intent(
         this, UpdateActivity.class );
        this.startActivity( updateIntent );
        …
```

Updating a Candy

- We add an activity element to the AndroidManifest.xml file for UpdateActivity.
- Inside UpdateActivity, we need a DatabaseManager instance variable to perform database operations.

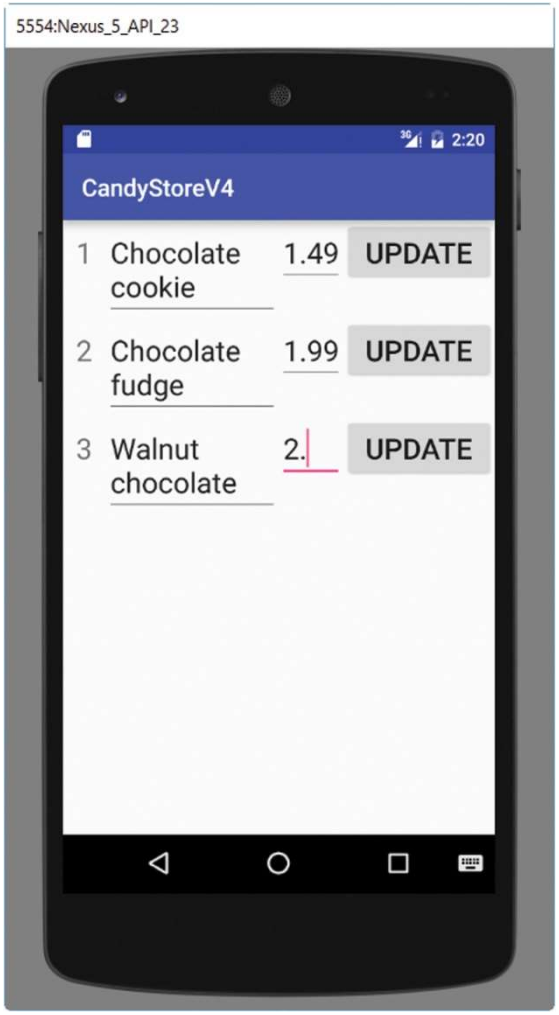
Updating a Candy—GUI (1 of 3)

- Four elements per Candy:
  - One TextView (id)
  - One EditText (decription)
  - One EditText (price)
  - One Button ➔ commit the update
- The user can update description and price.

Updating a Candy—GUI (2 of 3)

- As in the delete activity, we could provide a back button to go back to the main activity.
- However, there is no need for it: the user can use the device's back button, which pops the current activity off the stack.

Updating a Candy—GUI (3 of 3)



Updating a Candy

- As in the delete activity, we code a separate method, updateView, to code the GUI.
- When the user edits a candy and commits the update, not only do we update it in the database, but we also refresh the current screen by calling updateView.

updateView Method (1 of 14)

- We call selectAll from the DatabaseManager class:

```
ArrayList<Candy> candies =
        dbManager.selectAll( );
```

updateView Method (2 of 14)

- We place everything in a GridLayout, itself in a ScrollView.
- The GridLayout has four columns (one TextView, two EditTexts , one Button), and as many rows as there are candies.

updateView Method (3 of 14)

```
GridLayout grid = new GridLayout( this );
grid.setRowCount( candies.size( ) );
grid.setColumnCount( 4 );
```

updateView Method (4 of 14)

- We create three arrays: one for the TextViews, one for the Buttons, and one two-dimensional array for the EditTexts.

updateView Method (5 of 14)

- We loop through all the candies to build the grid.
- We need to retrieve the width of the screen to size the width of the four components properly.

updateView Method (6 of 14)

```
Point size = new Point( );
getDefaultManager( ).getDefaultDisplay( )
   .getSize( size );
// Capture the width of screen
int width = size.x
```

updateView Method (7 of 14)

- We set the text inside the TextView of each row to the id of the candy:

```
ids[i] = new TextView( this );
ids[i].setText( "" + candy.getId( ) );
```

updateView Method (8 of 14)

- We set the text inside the EditTexts of each row to the name and price of the candy

```
namesAndPrices[i][0] = new EditText( this );
namesAndPrices[i][1] = new EditText( this );
namesAndPrices[i][0]
   .setText( candy.getName( ) );
namesAndPrices[i][1]
   .setText( "" + candy.getPrice( ) );
```

updateView Method (9 of 14)

- We give a unique id to each EditText:

```
namesAndPrices[i][0]
    .setId( 10 * candy.getId( ) );
namesAndPrices[i][1]
    .setId( 10 * candy.getId( ) + 1 );
```

updateView Method (10 of 14)

- We set the text inside the Button of each row to UPDATE, and give each button the id of the candy (so we can match the candy to update with the button clicked):

```
buttons[i] = new Button( this );
buttons[i].setText( "Update" );
buttons[i].setId( candy.getId( ) );
```

updateView Method (11 of 14)

- We set up event handling for each button:

  buttons[i].setOnClickListener(bh);

- bh is an object of class ButtonHandler
- We need to code that class.

updateView Method (12 of 14)

- We add the four components to the current row of the GridLayout with the following space allocation:
  - Id: 10%
  - Name: 50%
  - Price: 15%
  - Button: 25%

updateView Method (13 of 14)

- We add the components

```
// width is retrieved dynamically
// current TextView, 10% of width, minimal height
grid.addView( ids[i], width / 10,
   ViewGroup.LayoutParams.WRAP_CONTENT );
```

- Similarly, we add the EditTexts (two per row) and the Button (one per row).

updateView Method (14 of 14)

- Outside the loop:
- We add the GridLayout to the ScrollView

  scrollView.addView(grid);
- We set the ScrollView as the content View for this activity:

  setContentView(scrollView);

Update: Event Handling (1 of 8)

```
private class ButtonHandler implements
                    View.OnClickListener {
    public void onClick( View v ) {
        …
```

Update: Event Handling (2 of 8)

- Inside the onClick method of ButtonHandler:
  - We retrieve the id of the candy that is being updated.
  - We retrieve its edited name and price.
  - We update the database.
  - We show a Toast for feedback.
  - We update the view.

Update: Event Handling (3 of 8)

- We first retrieve the id of the candy that is being updated.
- The View parameter v is a reference to the button that was clicked.
- Earlier, we gave each button an id, the id of the candy for that row.

```
int candyId = v.getId( );
```

Update: Event Handling (4 of 8)

- We then get a reference to the two EditTexts:

```
nameET = ( EditText )
                findViewById( 10 * candyId );
priceET = ( EditText )
                findViewById( 10 * candyId +
1 );
```

Update: Event Handling (5 of 8)

- Then we retrieve the candy's edited name and price:

```
String name = nameET.getText( ).toString( );
String priceString =
    priceET.getText( ).toString( );
// we need to convert priceString to a double
```

Update: Event Handling (6 of 8)

- Next, we update the database.
- Inside try block:

```
Double price = Double.parseDouble( priceString );
dbManager.updateById( candyId, name, price );
```

Update: Event Handling (7 of 8)

- We show a Toast for feedback:

```
Toast.makeText( UpdateActivity.this, "Candy
   updated", Toast.LENGTH_SHORT )
    .show( );
```

Update: Event Handling (8 of 8)

- Finally, we update the view

updateView( );

Cash Register (1 of 3)

- In Version 5, we enable the user to use a cash register on the first screen.
- As the user clicks on candies, we show a running total in a Toast.

Cash Register (2 of 3)

- We display a grid of buttons, one per candy.
- When the user clicks on a button, we update the total and show it in a Toast.

Cash Register (3 of 3)

- Thus, when the user clicks on a button, we need to know what candy is selected.
- An easy way to do that is to extend the Button class and create a CandyButton class (see Example 5.23).
- A CandyButton is a Button that has a Candy instance variable.
- We provide a getPrice method for convenience.

CandyButton Class

```
// import statements

public class CandyButton extends Button {
    private Candy candy;
     public CandyButton( Context context, Candy newCandy
) {
     super( context );
     candy = newCandy;
  }
   public double getPrice( ) {
    return candy.getPrice( );
  }
}
```

Cash Register

- We create a new icon (ic_reset.png), and place it in the drawable directory.
- We add an item in the menu.
- We run the cash register inside MainActivity.
- When the user select the icon for the cash register (action_reset id), we reset the total to 0.

menu_main.xml (1 of 2)

- Since we made our own icon and place it in the drawable directory, we use the syntax:

  android:icon="@drawable/name_of_icon "

  android:icon="@drawable/ic_reset"

menu_main.xml (2 of 2)

```
<menu …>
  <item android:id="@+id/action_reset"
        android:title="@string/reset"
        android:icon="@drawable/ic_reset"
        app:showAsAction="ifRoom"/>

  <item android:id="@+id/action_add"
        …
```

Cash Register—MainActivity (1 of 2)

- Inside onOptionsItemSelected method:

```
case R.id.action_reset:
    total = 0.0;
return true;
```

- Note that we stay in this activity (MainActivity).

Cash Register—MainActivity (2 of 2)

- The updateView method displays the GUI.
- One Button for each candy, two per row inside a GridLayout.
- The GridLayout is inside a ScrollView (we do not know in advance how many buttons we have).

ScrollView

- An easy way to provide a ScrollView is to replace the RelativeLayout inside content_main.xml with a ScrollView element.
- We give it an id so that we can retrieve it inside the MainActivity class.
- We also eliminate the padding inside the ScrollView.

content_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    …
    android:id="@+id/scrollView">
</ScrollView>
```

Cash RegisterMainActivity

- We include a DatabaseManager instance variable to perform database operations:

```
private DatabaseManager dbManager;
```

Cash Register—updateView (1 of 4)

ArrayList<Candy> candies =
dbManager.selectAll( );

- The GridLayout has two columns:

  and (candies.size( ) + 1) / 2 rows

- If there are 6 candies ➔ 7 / 2 = 3 rows
- If there are 7 candies ➔ 8 / 2 = 4 rows

Cash Register—updateView (2 of 4)

- We retrieve the width of screen; the button width is the width of screen divided by 2 (there are two buttons per row).
- We create an array of candyButtons:

```
CandyButton [ ] buttons = new
CandyButton[candies.size( )];
```

- For each candy, we add a CandyButton to the GridLayout.

Cash Register—updateView (3 of 4)

```
buttons[i] = new CandyButton( this, candy );
buttons[i].setText( candy.getName( ) + "\n" +
   candy.getPrice( ) );
```

- We set up event handling

```
buttons[i].setOnClickListener( bh );
```

- bh is an object of type ButtonHandler (which implements View.OnClickListener)

Cash Register—updateView (4 of 4)

- We add each button to GridLayout.
- The width of each button is buttonWidth.
- Its height is minimal height.

```
grid.addView( buttons[i], buttonWidth,
   GridLayout.LayoutParams.WRAP_CONTENT );
```

updateView Method

- Outside the loop:
  - We add the GridLayout to the ScrollView.

    scrollView.addView(grid);
- We set the ScrollView as the content View for this activity.

    setContentView(scrollView);

Cash Register: Event Handling (1 of 4)

- Inside the onClick method of ButtonHandler:
  - We retrieve the price of the candy that is purchased.
  - We add it to the total.
  - We show a Toast for feedback.

Cash Register: Event Handling (2 of 4)

- We retrieve the price of the candy that is purchased.
- The View parameter of onClick, v, is a reference to the button that was clicked, which is a CandyButton.

Cash Register: Event Handling (3 of 4)

- We cast v to a CandyButton.

  (CandyButton) v

- We add the price of the candy selected to the total.

  total += ((CandyButton) v).getPrice( );

Cash Register: Event Handling (4 of 4)

- We format total:

```
String pay =
    NumberFormat.getCurrencyInstance( )
                .format( total );
```

- We show a Toast for feedback:

```
Toast.makeText( MainActivity.this, pay,
  Toast.LENGTH_SHORT ).show( );
```

Summary

- Menu, menu items (text, icon)
- SQLite
- Toast
- Building a GUI programmatically
- ScrollView

Reference

- Books
  - Android in Action
  - Learn Android App Development
  - Android Studio Essentials
  - Android Programming for Beginners
  - Teach Yourself Android Application Developments
- Websites
  - Android Development
- Video
  - Youtube