MENU & DATABASE

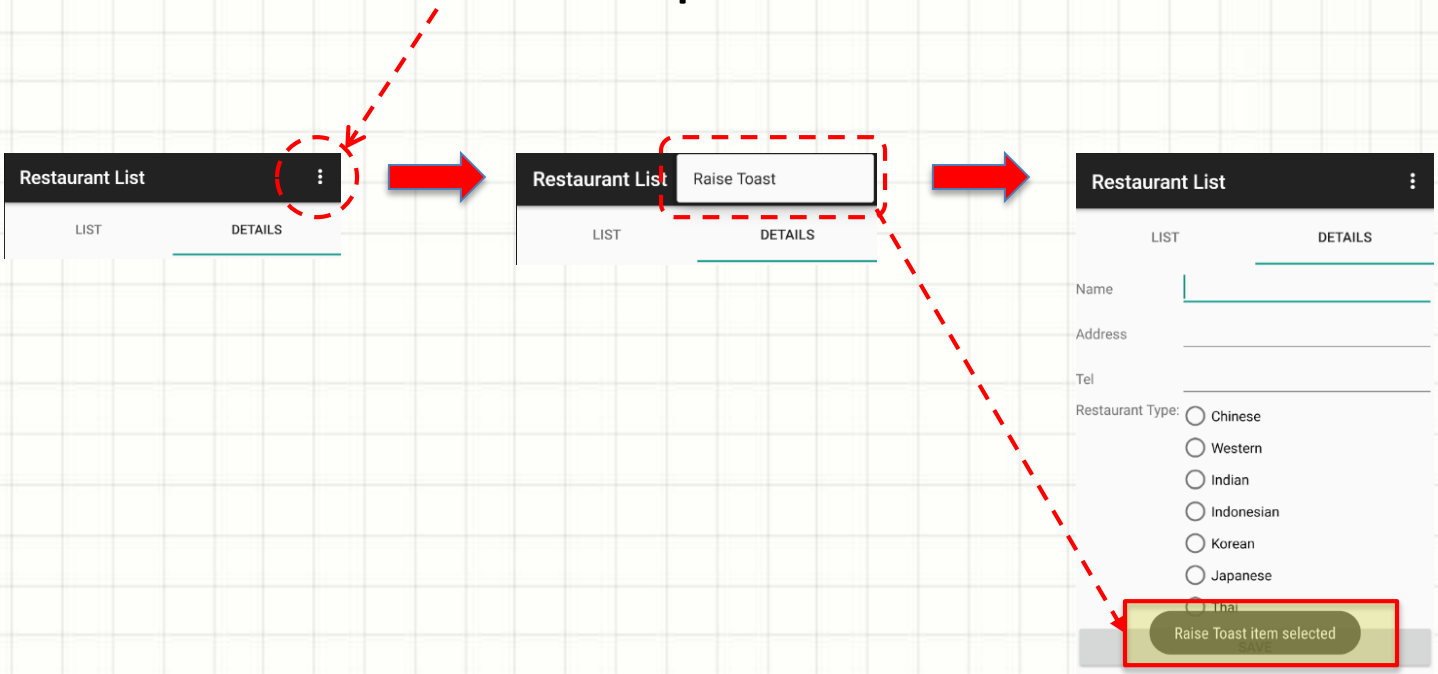
Today's Overview

1. Menu
2. Database

Menu

Menu

- In first part of Practical 3 exercise, you will learn
  - to **detect** a MENU option **item selection**



Menu

- Let's check what do we need to modify from the previous exercise to incorporate the new MENU display?
  - Model - Any change in Data Model?
  - View - Do you need to modify any of the user interface view?
  - Controller - Do you need to tell the Controller to do any thing new?

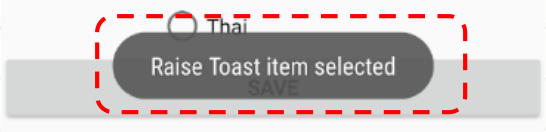
Menu

- Model - NO
- View - YES

A MENU has been added with label ‘Raise Toast’

- Controller – YES

The layout file named ‘*option.xml*’ will be created. We will need to update the Controller to link to the new file



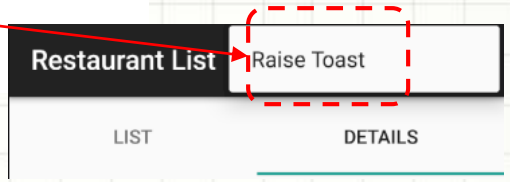
UI View - Menu

View

Menu

- When MENU button on the device is pressed, an option menu will be shown at the bottom of the UI View.
- An item with id *raise_toast* is added to the MENU

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <menu xmlns:android="http://schemas.android.com/apk/res/android">
3
4       <item android:title="Raise Toast"
5           android:id="@+id/raise_toast" />
6   </menu>
```

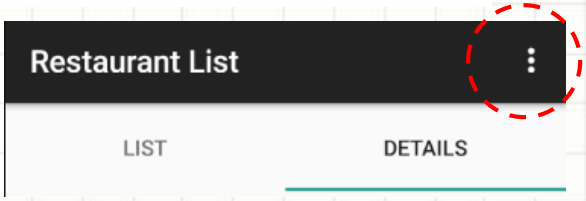


Controller – Menu

Option Menu

- When the MENU button is pressed, by default, the Controller will trigger the onCreateOptionsMenu(Menu) method and inflate the layout defined by *option.xml* for current UI View

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.option, menu);
    return super.onCreateOptionsMenu(menu);
}
```



UI View – Menu Item Detection

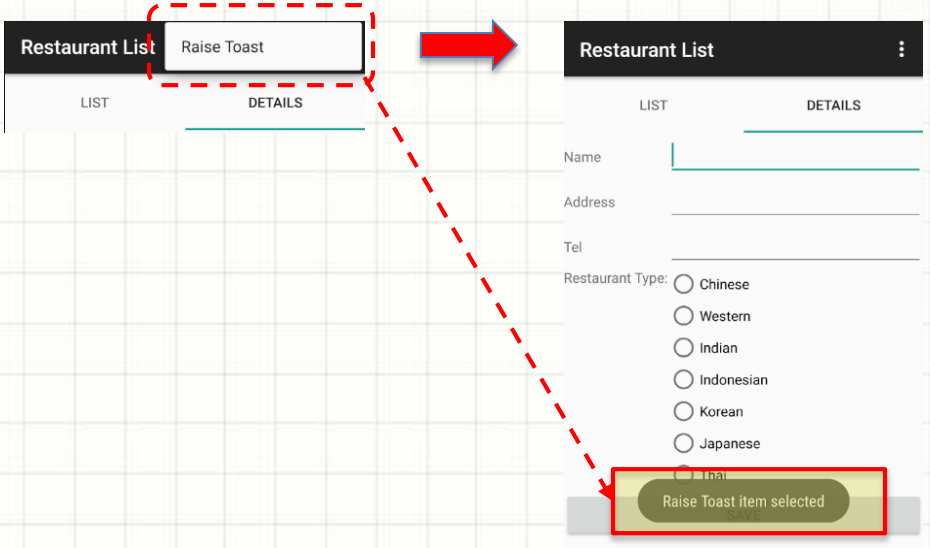
MENU

- Let's check what do we need to modify from the previous exercise to detect MENU item selection?
  - Model - Any change in Data Model?
  - View - Do you need to modify any of the user interface view?
  - Controller - Do you need to tell the Controller to do any thing new?

MENU

- Model - NO
- View - NO
- Controller – YES

onOptionsItemSelected(MenuItem item) method is added to the Controller to handle the option item select event



MENU

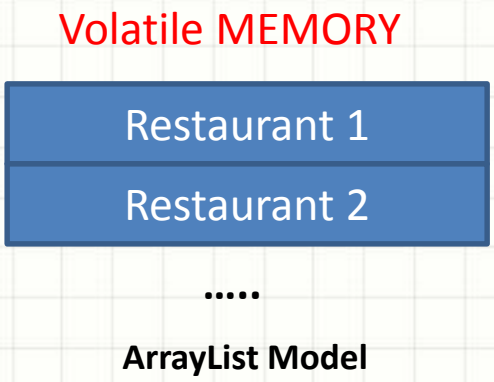
- When an option item is selected, the Controller will capture the selection event and trigger the onOptionItemsSelected(MenuItem) method to take action. In this case, a Toast message box is used to acknowledge the event triggered

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case (R.id.raise_toast):
            Toast.makeText(this, "Raise Toast item selected", Toast.LENGTH_LONG).show();
            break;
    }
    return super.onOptionsItemSelected(item);
}
```

DATABASE MODEL

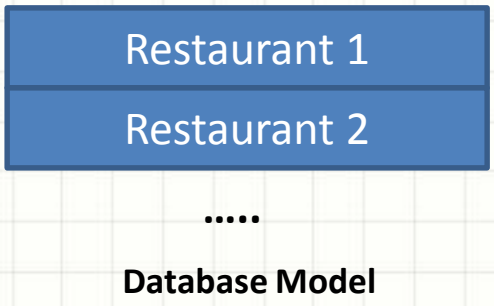
Database

- The restaurant Model created so far to hold the restaurant data is temporary! When user presses the BACK button and launch the Restaurant List app again, the data in restaurants list entered will be gone. This is because the Model uses memory to store its information



Database

- What happen if the restaurant data needs to stay and non-volatile?
- In second part of Practical 3 exercise, a non-volatile data Model (Database) is used to replace the volatile data Model (ArrayList)
- The restaurant data is saved as a local file using SQLite database



Database

- Let's check what do we need to modify from the previous exercise to use SQLite database?
  - Model - Any change in Data Model?
  - View - Do you need to modify any of the user interface view?
  - Controller - Do you need to tell the Controller to do any thing new?

Database

- Model - YES

  Due to a total changed in Data Model, the *ArrayList* is replaced by *restaurantlist.db* database with *restaurants_table* as Data Model. The *Restaurant.java* file is deleted and replaced by *RestaurantHelper.java* (a sub-class of SQLiteOpenHelper)

Database

- View - NO
- Controller - YES

When using memory as a temporary data storage, *ArrayList* and *ArrayAdapter* are used.

When data is stored as a record file in *restaurants_table*, *Cursor* and *CursorAdapter* will be used for handling database Model and *ListView*

Model - Database Model

Model

- The *RestaurantHelper.java* "helper" (SQLiteOpenHelper sub-class) is created to have access to database Model

```
class RestaurantHelper extends SQLiteOpenHelper {
```

- It provides methods for creating and opening database Model, and inserting and reading data from table Model

Model

RestaurantHelper

- A table model named *restaurants_table* is created with the necessary data elements structure

```
@Override
public void onCreate(SQLiteDatabase db) {
    // Will be called once when the database is not created
    db.execSQL("CREATE TABLE restaurants_table (_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "restaurantName TEXT, restaurantAddress TEXT, restaurantTel TEXT, restaurantType TEXT);");
}
```

- Table model data structure

Field Name	Type	Key	
_id	INTEGER	PRIMARY	Unique ID for each record
restaurantName	TEXT		
restaurantAddress	TEXT		
restaurantTel	TEXT		
restaurantType	TEXT		

Model

RestaurantHelper

- SQLiteOpenHelper provides methods getReadableDatabase() and getWritableDatabase() to get access to SQLiteDatabase objects; either in read or write mode

```
/* Read all records from restaurants_table */
public Cursor getAll() {
    return (getReadableDatabase().rawQuery(
            "SELECT _id, restaurantName, restaurantAddress, restaurantTel, restaurantType " +
                    "FROM restaurants_table ORDER BY restaurantName", null));
}
/* Write a record into restaurants_table */
public void insert(String restaurantName, String restaurantAddress, String restaurantTel,
                   String restaurantType) {
    ContentValues cv = new ContentValues();

    cv.put("restaurantName", restaurantName);
    cv.put("restaurantAddress", restaurantAddress);
    cv.put("restaurantTel", restaurantTel);
    cv.put("restaurantType", restaurantType);

    getWritableDatabase().insert("restaurants_table", "restaurantName", cv);
}
```

Controller – Setting up

Controller

- The Adapter to handle Database Model and *ListView* is replaced by *CursorAdapter*
- *CursorAdapter* creates a *View* for every needed row in a Cursor i.e. row in a table Model
- A *CursorAdapter* does not use getView(), but rather newView() and bindView() methods

Controller

CursorAdapter

- The newView() method handles inflating new row in UI View

```
@Override
public View newView(Context ctxt, Cursor c, ViewGroup parent) {
    LayoutInflater inflater = getLayoutInflater();
    View row = inflater.inflate(R.layout.row, parent, false);
    RestaurantHolder holder = new RestaurantHolder(row);

    row.setTag(holder);
    return (row);
}
```

- The bindView() handles recycled row disappears from the UI View

```
@Override
public void bindView(View row, Context ctxt, Cursor c) {
    RestaurantHolder holder = (RestaurantHolder) row.getTag();

    holder.populateFrom(c, helper);
}
```

Controller

- Instead of using *ArrayList* Model to hold the table Model, it is changed to *Cursor*
- The *getAll()* method will return the Cursor (rows of data from table Model)

```
private Cursor model = null;
```

```
helper = new RestaurantHelper(this);
list = (ListView) findViewById(R.id.restaurants);
model = helper.getAll();
adapter = new RestaurantAdapter(model);
list.setAdapter(adapter);
```

Controller

- When saving the restaurant to table Model using insert() method, the *Cursor* and the *CursorAdapter* will not realise that the database contents have been changed. The *swapCursor()* method for *CursorAdapter* will the View with new *Cursor*

```
//Insert record into SQLite table
helper.insert(nameStr, addrStr, telStr, restType);

model = helper.getAll();     //Reload the content of Cursor (record list)
adapter.swapCursor(model);   //Update List View with new record list
```

END