Mobile Apps with Android

▼ The future says:

I don't know what cute means but everyone says, I am cute!

I have more processing power in my hand than my dad had in his computing school! And if it ain't mobile, it's not interesting!

▼ Android Apps Setup & Review

- Delivery, Assessment and Resources

  What, How & When (See Assesment brief and Module guide)
- Java Review

  ▼ Java Language and Memory Management

  Java review and a practical look at the *stack* and *heap*.
- Use git, create branches, merge and commit regulalry to a remote repository. Lab examples are on GitHub: https://github.com/ebbi

- Build and Run the default "Hello World!" example in Android Studio using *Android 8.1 (Oreo) SDK and a Virtual Device, targetting Android 8.1 (Google Play)*

▼ Activity Manager and Activity and Intent Components

- No default `main` method, the *ActvityManager* in the Android OS, launches `Activity` classes.
- `Activity` and `Intent` must communicate via the *`ActvityManager`* in the Android OS and not within the App address space
- The pattern with the OS, `ActvityManager` allows for an `Activity` in one App to work with an `Activity` in another App.
- Before starting an Activity, the Activity Manager:
  - Check the package manifest for a class declaration with the same name, If found start activity else raise `ActivityNotFoundException`
  - The Activity Manager uses `Intent` objects to start activities and pass data between activities

Intent, Extras, and Hashes; a little like a constructor or session data but through the OS

▼ Activity and Intent Sequence Diagram - *Todo App*


activity intent

▼ Activity and *Inflating* View objects

- An `Activity` *should be* a controller class responsible for user interaction on it's associated UI.
- UI is composed of *Widgets* as building blocks and include layouts, text input controls, checkboxes, buttons and so on.
- Every widget is an instance of the `view` class or one of it subclsses, such as `ViewGroup, TextView` and `Button`

Definning Views and *inflating* View objects

- Widgets exist in a hierarchy and are defined as a "well-formed" XML tree data structure.
- Example Widgets:

```
sample code for view
```

▼ `ViewGroup, View`, and `View` attributes

- The default root element is a `LinearLayout`, a single column which inherits from a subclass of `View`, named `ViewGroup`. Others include, `RelativeLayout,`

`FrameLayout` and `TableLayout`. These can be nested as parent branches in the XML definition.

- Each widget has a set of attributes for its gemotry, position and content, for example:

- ▼ Example Widget attributes
  - `android:layout_width` and `android:layout_height` are common to most elements. The value is often, `match_content` expand to content size and `match_parent` fit the parent size.
  - `android:text` applies to elements such as `TextView` and `Button`

- All *literal strings* in the XML definition but as a resource. For example:

  `android:text="@string/todo_list_view"`

  is a literal value defined in the default `res/values/strings.xml` file as:

  ```
  <string name="todo_list_view">Todo list!
  </string>
  ```

  (*All XML definition with a `resources` root element stored in any file in the `res/values/` folder can be used with the above syntax.*)

- Android generates a *resource id* for the entire layout and for every widget occurance with the `android:id`. For example, `android:id="@+id/buttonNext"`.

- ▼ View Objects from XML layout definitions
  - Android generates a *resource id* for the entire layout and for every widget occurance with the `android:id`.

    For example, `android:id="@+id/buttonNext"`.
  - Android Asset Packaging Tool (aapt) dynamically generates an `R.java` class (see the generated java code folder for and example R.java class).
  - The generated `R` class has convenience methods to access the widget id by it's defined name in the XML definition
  - The Android Asset Packaging Tool (aapt) compiles layout file resources which are packaged into .apk files.
  - `setContentView(Resouce)` uses the `LayoutInflator` class to instantiate each of the *View* objects defined in the layout file.

- Getting a reference to the the widget is achieved by a call to `findViewById()`

  ```
  mButton = (Button) findViewById(R.id.buttonNext)
  ```

  (The build process has generated a Java static class, `R` which has convenience methods to specify the widget id by it's defined name in the XML definition).

▼ Events & Listeners

Androids maintains an event queue and implements event listeners to capture events and fire corresponding event handlers.

- Objects that respond to *events* are called *listeners*. Listeners fire events that call corresponding event handlers.
- For example, on a UI *click* event, the listener, `View.onClickListener` fires an event that results in the corresponding event handler, `onClick` call back method being called.

▼ A consistant approach to Listeners

There are various ways of registering an event listener with an event handler and the following are some options.

▼ Listener defined as an annonymous inner class

Advantage of having access private Activity data.

Each view object with a seperate listener.

```
buttonNext.setOnClickListener(new
View.OnClickListener(){
  @Override
  public void onClick(View v) {
    // do something when the button is clicked
  }
});
```

▼ Implement the `interface` in the class definition

```
public class ActivityMain extends Activity
    implements View.OnClickListener {
  protected void onCreate(Bundle savedValues)
{
    …
    Button button =
(Button)findViewById(R.id.next);
    button.setOnClickListener(this);
  }

  @Override
  public void onClick(View view) {
    // do something when the button is clicked
  {
  …
}
```

▼ Anonymous implementation of the `interface` for any view objects

```
/* Create an anonymous implementation of
OnClickListener
for all clickable view objects */

private View.OnClickListener mTodoListener =
    new View.OnClickListener() {

  public void onClick(View v) {
    // get the clicked object and do something
    switch (v.getId() {
    case R.id.checkBoxIsComplete:

    default:
        break;
    }
  }
};
```

And the usage would be:

```
CheckBox checkboxIsComplete =
(CheckBox)findViewById(R.id.checkBoxIsComplete)


checkboxIsComplete.setOnClickListener(mTodoList
```

▼ Define the listener in the XML view definition.

Please avoid this for MVC SoC!

```
/*
listener implementation with the method name
defined in the view definition
*/
android:onClick="onCheckboxIsCompleteClick"
```

▼ Intent

▼ A messaging object to request an action from another app component.

Typical *explicit* intents include the following parameters:

- *action* - the first parameter is an explicit component name; without a component name the intent becomes *implicit*
- *data* - the data to operate on such as a `uri` to a database record.
- *extras* - a `Bundle` of any additional information.

Intens provide different constructors for their multipurpose use.

▼ Where should an intent be defined?

- An *explicit* `Activity` may be called from any number of other activities, so the data needed should be defined in the `Activity` that uses it.
- Typically implemented with a `static` method recieving the data and returning the `intent` object

For example, the following recieves the `todoIndex` and returns an `intent` object.

```
public static Intent newIntent(
  Context packageContext, int todoIndex){

Intent intent = new Intent(
  packageContext, TodoDetailActivity.class);

intent.putExtra(TODO_INDEX,todoIndex);

return intent;

}
```

Note the `intent.putExtra(…)`, a map data structure with key, value pairs.

The calling activity calls the static `newIntent` method of the `TodoDetailActivity`:

```
Intent intent =

TodoDetailActivity.newIntent(TodoActivity.this,
```

```
mTodoIndex);
```

▼ Getting the `Intent` result back

Activity implement a method for when there is data to be passed back to the calling `Activity`

```
startActivityForResult(Intent intent, int
requestCode);
```

requestCode, is an integer identifying each child activity; for single calls it is set to a constant

▼ Typical usage:

```
@Override
public void onClick(View v) {
  Intent intent =
TodoDetailActivity.newIntent(
  TodoActivity.this, mTodoIndex);
  startActivityForResult(intent,
IS_SUCCESS);
}
```

▼ Setting a result

`Activity` implement two method for setting results

```
setResult(int resultCode);

setResult(int resultCode, Intent intent);
```

Typically, resultCode, is `Activity.RESULT_OK` or `Activity.RESULT_CANCELED`

useful for navigation logic.

for larger number of attribute, *extras* can be added to the `intent` object.

▼ Typical usage:

```
Intent intent = new Intent();
intent.putExtra(IS_TODO_COMPLETE, isChecked);
setResult(RESULT_OK, intent);
```

▼ Handling the result

Override the callback method `onActivityResult()` to retrieve any set result.

▼ onActivityResult

```
@Override
protected void onActivityResult(
      int requestCode, int resultCode, Intent
intent) {
  if (intent != null) {
    // data in intent from child activity
    boolean isTodoComplete =
    intent.getBooleanExtra(IS_TODO_COMPLETE,
false);
    …
  }
}
```

▼ Fragments

"A `Fragment` represents a behavior or a portion of user interface in a `FragmentActivity`. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities."

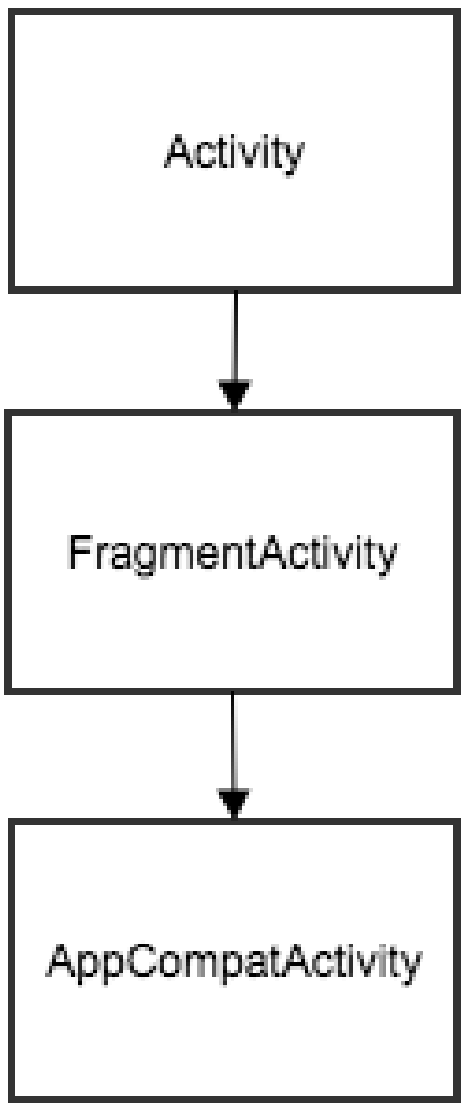
developer.android.com

- The original Android architecture required the controller code for User Interation to reside in `Activity` classes.
- New devices forced a change in this architecture and the tight coupling was decoupled by moving the UI to

`Fragment`

▼ Native Android or Support Libraries?

- Support libraries were created to accommodate for older versions of Android. Google provided `support-v4` which includes fragment support `android.support.v4.app.Fragment`
- The single support library is now a group of libraries including: `support-v7, appcompat-v7, recyclerview-v7` and many more
- *Avoid the native Android OS implemetations and use the Support library versions. Many more releases and easier to update.*



- An `Activity` has one (or more) *placeholder* views for one (or more) `fragments` to be loaded into dynamically.

  ▼ Todo with fragments

Todo mvc object diagram

- An Activity's view is a container `FrameLayout` and remains the same throughout the lifecycle

  ▼ TodoActivity layout

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

    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </FrameLayout>
  ```

- add the `fragments` to the activity's *layout* - avoid coupling!
- TodoFragment is a controller object that interacts with the view and the model.

  ▼ TodoFragment

```
public class TodoFragment extends
Fragment {
  private Todo mTodo;
  private CheckBox mCheckBoxIsComplete;

  @Override
  public void onCreate(Bundle
savedInstanceState){
    super.onCreate(savedInstanceState);
    mTodo = new Todo();
  }

  @Override
  public View onCreateView(LayoutInflater
inflater,
      ViewGroup container, Bundle
savedInstanceState) {

    View view = inflater.inflate(
      R.layout.fragment_todo, container,
false);
    mCheckBoxIsComplete = (CheckBox)

view.findViewById(R.id.todo_complete);


mCheckBoxIsComplete.setOnCheckedChangeList

        new OnCheckedChangeListener() {

      @Override
```

```
        public void onCheckedChanged(
          CompoundButton buttonView,
boolean isChecked) {
          mTodo.setComplete(isChecked);
        }

    });

    return view;
  }
}
```

- Note, `Activity.onCreate` had `protected` scope, `Fragment.onCreate` is `public`, this allows any activity to host the fragment and makes it reusable
- Similar to Activity, a Fragment has a Bundle to save state.
- The previous Activity `setContentView()` is no longer called in the `onCreate` method, instead the `onCreateView()` explicitly *inflates* the fragment's view by a call to `inflater.inflate(R.layout.fragment_todo, container, false)`. The third parameter `false` indicates, it should not be added to the

parent view as it will be done in the host Activity.

- Referencing a view object is now with an `int` reference;

  `view.findViewById(R.id.todo_complete)`

- reuse the `fragments` in any activity's *code*

▼ TodoActivity

```
public class TodoActivity extends
AppCompatActivity {
  @Override
  protected void onCreate(Bundle
savedInstanceState) {
    super.onCreate(savedInstanceState);

setContentView(R.layout.activity_fragment)

    FragmentManager fm =
getSupportFragmentManager();
     Fragment fragment =

fm.findFragmentById(R.id.fragment_containe

    if (fragment == null){
      Fragment todoFragment = new
TodoFragment();
      fm.beginTransaction()
        .add(R.id.fragment_container,
todoFragment)
        .commit();
    }
  }
…
```

- The `FragmentManger` keeps a *Back* stack of `FragmentTransactions` that can be navigated and a list of `Fragments`.

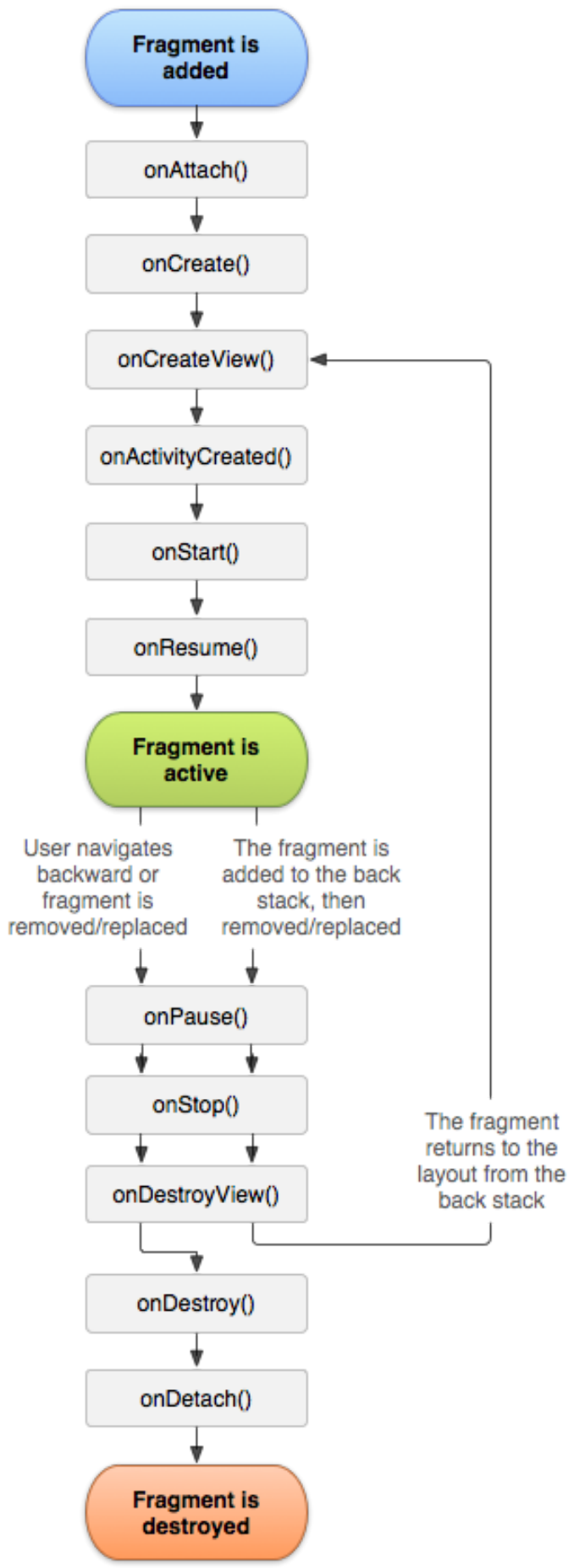
- `FragmentTransactions` are used to `add, remove, attach, detach` or `replace` fragments.
- The `FragmentManager.beginTransaction` creates and returns an instance of `FragmentTransaction` and not `void`, hence it can be chained together.
- The `.add` method has two parameters, the container view ID and a `fragment`. The ID serves two purposes:
  - it identifies where in the Activity's view the `fragment` should appear.
  - it is a unique identifier for the `fragment` in the `FragmentManager` list

Why check for `null`? The `fragment` may already be in the in the `FragmentManager` list due to events such as destoyed on rotation or to reclaim memory.

What happens if the `Activity` has resumed and a `Fragment` is added? The `FragmentManager` calls all the necessary methods to synchronise with

the `Activity` state.

For example, adding a fragment to a resumed activity results in the fragment getting calls to `onAttach(Context)`, `onCreate(Bundle)`, `onCreateView(…)`, `onActivityCreated(Bundle)`, `onStart()` and `onResume`.



- The `FragmentManager` calls the Lifecycle methods of the `Fragments` in its list.
- When a `fragment` is added to the `FragmentManager` list leads to the `onAttach(Context)`, `onCreate(Bundle)`, and

`onCreateView(…)` lifecycle methods being called.

- When the `.add` mehod adds a `fragment` to the `FragmentManager` it leads to the `onAttach(Context), onCreate(Bundle)`, and `onCreateView(…)` lifecycle methods being called.
    - The model should return the resultant data to the fragment controller; a simple implemetation is a singleton

        ▼ TodoModel

```
public class TodoDS {
  private static TodoDS sTodoDS;
  private List<Todo> mTodoList;

  public static TodoDS get(Context
context) {
    if (sTodoDS == null) {
      sTodoDS = new TodoDS(context);
    }
    return sTodoDS;
  }

  private TodoDS(Context context){
    mTodoList = new ArrayList<>();
    // simulate some data for testing
    for (int i=0; i < 3; i++){
      Todo todo = new Todo();
      todo.setTitle("Todo title " + i);
      todo.setIsComplete(false);
      mTodoList.add(todo);
    }
  }

  public Todo getTodo(UUID todoId) {
    for (Todo todo : mTodos) {
      if (todo.getId().equals(todoId)){
          return todo;
      }
    }
    return null;
  }
```

```
  public List<Todo> getTodos() {
    return mTodoList;
  }
}
```

And the `Todo` class

```
public class Todo {
  private UUID mId;
  private String mTitle;
  private boolean mIsComplete;

  public Todo() {
      mId = UUID.randomUUID();
      mDate = new Date();
  }
  public UUID getId() {
      return mId;
  }
  public void setId(UUID id) {
      mId = id;
  }
  public String getTitle() {
      return mTitle;
  }
  public void setTitle(String title) {
      mTitle = title;
  }
}
```

▼ Fragment Arguments

See the `TodoListApp` example for sample code.

Please avoid  Fragment Arguments using the `getActivity` method to access the parent Activity's intent data directly. This is simple code but loses the encapsulation of the fragment and it will not be reusable.

Instead use the Bundle object and a static method to bundle any arguments and set it as arguments attached to the fragment and return the fragment. The frgment is now available accross Activities as each passes its own bundle data attributes.

▼ Fragment Arguments example

```
In TodoFragment
public static TodoFragment newInstance(UUID
todoId) {
    Bundle args = new Bundle();
    args.putSerializable(ARG_TODO_ID, todoId);

    TodoFragment fragment = new TodoFragment();
    fragment.setArguments(args);
    return fragment;
}
```

```
Other Activity such as TodoActivity can now call
the static method

protected Fragment createFragment(){
    UUID todoId = (UUID) getIntent()
      .getSerializableExtra(EXTRA_TODO_ID);
    return TodoFragment.newInstance(todoId);
}

The same pattern was applied to a newIntent where
todoId is added:

In TodoListFragment
public void onClick(View view) {
  Intent intent = TodoActivity
    .newIntent(getActivity(), mTodo.getId());
  startActivity(intent);
}

and in TodoActivity:
```

```
public static Intent newIntent(
    Context packageContext, UUID todoId) {
  Intent intent = new Intent(packageContext,
TodoActivity.class);
  intent.putExtra(EXTRA_TODO_ID, todoId);
  return intent;
}
```

▼ Getting results with fragments

- Fragments equivalent to `startActivityForResult()` is `Fragment.startActivityForResult()`
- Instead of overriding `Activity.onActivityResult()` overide `Fragment.Activity.onActivityResult()`
- Fragments do not have results, there is no `Fragment` setResult, only Activity has `Activity.setResult()`
- Instead Fragmnets calls a method on the host activity to return a vlaue,

  `getActivity().setResult(Activity.RESULT_OK, null);`

▼ RecyclerView

- For scrolling list of elements use RecyclerView
- Recycles (reuses) `view` objects to fill a screen

- `RecyclerView` relies on an `Adapter` with a typical sequence of calls:
  a. `getItemCount()`
  b. create a new `viewHolder` with a call to the adapter's `onCreateViewHolder()`
  c. Adpter looks up model data and fills the list item's `viewHolder view`
  d. `RecyclerView` places the list item on the screen
  e. Once enough `viewHolder`'s have been created to fill the screen, they are reused

▼ `RecyclerView` & Todo fragments

▼ Create a `RecyclerView` in the `onCreateView` method

```
In TodoListFragment:
private RecyclerView mTodoRecyclerView;

mTodoRecyclerView = (RecyclerView)

view.findViewById(R.id.todo_recycler_view);
// it will crash without a LayoutManager
mTodoRecyclerView.setLayoutManager(
    new LinearLayoutManager(getActivity()) );
```

▼ Just like `Fragments`, `RecyclerView` has its own `view` hierarchy

In fragment_todo_list.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- A RecyclerView with some commonly used
attributes -->
<android.support.v7.widget.RecyclerView

xmlns:android="http://schemas.android.com/apk/r

    android:id="@+id/todo_recycler_view"
    android:scrollbars="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

▼ Define the `ViewHolder` to inflate and fill the layout

```
In TodoListFragment:

public class TodoHolder extends
RecyclerView.ViewHolder
  implements View.OnClickListener {

  public TodoHolder(LayoutInflater inflater,
ViewGroup parent) {
      super(inflater.inflate(
          R.layout.list_item_todo, parent,
false));
  }
}
```

▼ Create the `Adapter` and override three methods

In TodoListFragment:

```
public class TodoAdapter extends

RecyclerView.Adapter<TodoListFragment.TodoHolde
 {

  private List<Todo> mTodos;

  public TodoAdapter(List<Todo> todos) {
      mTodos = todos;
  }

  @Override
  public TodoListFragment.TodoHolder
onCreateViewHolder(
        ViewGroup parent, int viewType) {
    LayoutInflater layoutInflater =
        LayoutInflater.from(getActivity());

    return new TodoHolder(layoutInflater,
parent);
  }

  @Override
  public void onBindViewHolder(
      TodoHolder holder, int position) {
    Todo todo = mTodos.get(position);
    holder.bind(todo);
  }
```

```
    @Override
    public int getItemCount() {
        return mTodos.size();
    }

}
```

▼ Bind List Items

Seperating creation and binding allows views to be (Recycled) reused

- The binding starts with the views in the `ViewHolder` constructor
- `ViewHolder` relies on a `bind(data)` method to set the values of views it holds.

Lab test is on the material above this line.

See Github TodoListApp_2 for sample code for FileProvider, Implicit intent (camera) and SQLite

▼ SQLite

- SQLite, open source flat file relational DB
- Ideal for embedded applications; No DBMS or scalability

- SQLite included in Android standard library
- Android helper classes to open/read/write in device's *sandbox*
- ▼ Schema, 3rd NF

```
package database;
import java.util.Date;
import java.util.UUID;

public class TodoDbSchema {
    public static final class TodoTable {
        public static final String NAME =
"todos";
        public static final class Cols {
            public static final String UUID =
"uuid";
            public static final  String TITLE =
"title";
            public static final  String DETAIL
= "detail";
            public static final  String DATE =
"date";
            public static final  String
IS_COMPLETE = "isComplete";
        }
    }
}
```

```
/* Columns can be refered to in a Java safe way
*/
TodoDbSchema.Cols.TITLE
```

- ▼ Building a DB steps:
  - (!Exist DB) Create DB (and Seed data)
  - Else open DB (and check version)

    ▼ Android `SQLiteOpenHelper` class handles building a DB

```
package database;
import android.content.Context;
import
android.database.sqlite.SQLiteDatabase;
import
android.database.sqlite.SQLiteOpenHelper;
import database.TodoDbSchema.TodoTable;

public class TodoBaseHelper extends
SQLiteOpenHelper {
  private static final int VERSION = 1;
  private static final String DATABASE_NAME
= "todo.db";

  public TodoBaseHelper(Context context) {
      super(context, DATABASE_NAME, null,
VERSION);
  }
  @Override
  public void onCreate(SQLiteDatabase db) {
      db.execSQL("create table " +
TodoTable.NAME + "(" +
                          TodoTable.Cols.UUID +
", " +
                          TodoTable.Cols.TITLE
+ ", " +
                          TodoTable.Cols.DETAIL
+ ", " +
                          TodoTable.Cols.DATE +
", " +
```

```
TodoTable.Cols.IS_COMPLETE + ")"
        );
  }
  @Override
  public void onUpgrade(SQLiteDatabase db,
    int oldVersion, int newVersion) {
  }
}
```

▼ Example Model using `SQLiteOpenHelper` to create a DB

```
public class TodoModel {
  private static TodoModel sTodoModel;
  private static Context mContext;
  private SQLiteDatabase mDatabase;

  public static TodoModel get(Context
context) {
    mContext =
context.getApplicationContext();
    if (sTodoModel == null) {
      sTodoModel = new
TodoModel(context);
    }
    return sTodoModel;
  }

  private TodoModel(Context context){
    mContext =
context.getApplicationContext();
    mDatabase = new
TodoBaseHelper(mContext)
            .getWritableDatabase();
    }
    /* insert seed test data */
  }
…
```

▼ CRUD Operations

- ▼ CRUD uses `ContentValues` class to store *key/value* maps

```
/* Model static method for
ContentValues */
private static ContentValues
getContentValues(Todo todo) {
  ContentValues contentValues = new
ContentValues();

contentValues.put(TodoDbSchema.TodoTabl

    todo.getId().toString());

contentValues.put(TodoDbSchema.TodoTabl

    todo.getTitle());

contentValues.put(TodoDbSchema.TodoTabl

    todo.getDetail());

contentValues.put(TodoDbSchema.TodoTabl

    todo.getDate().getTime());

contentValues.put(TodoDbSchema.TodoTabl

    todo.isComplete()==1 ? 1 : 0);

  return contentValues;
}
```

- ▼ Create or Write to the DB

```
public void addTodo(Todo todo){
  ContentValues contentValues =
getContentValues(todo);
  /* contentValues = null raises an
exception except
    when 2nd parameter is null in
which case a new row is inserted */

mDatabase.insert(TodoDbSchema.TodoTable
 null, contentValues);
}
```

- ▼ Update a record

```
public void updateTodo(Todo todo){
  String uuidString =
todo.getId().toString();
  ContentValues contentValues =
getContentValues(todo);
  /* stop sql injection, pass
uuidString to new String
  so, it is treated as string rather
than code */

mDatabase.update(TodoDbSchema.TodoTable
 contentValues,
    TodoDbSchema.TodoTable.Cols.UUID +
" = ?",
    new String[] { uuidString });
}
```

- ▼ Read a record
  - `SQLiteDatabase.query()` has many *overloads* corresponding to a SQL query `SELECT columns FROM Table WHERE wherArgs GROUPBY, HAVING, ORDERBY, LIMIT .`
  - `SQLiteDatabase.query()` returns a `cursor` object

    ```
    Cursor cursor =
    ```

```
mDatabase.query(TodoDbSchema.TodoTable
.NAME, … )
```

- `Cursor` interface provides random read-write access to the result set returned by a database query.

```
cursor.getColumnIndex(TodoDbSchema.Tod
oTable.Cols.TITLE));
```

- DRY by using `CursorWrapper` to subclass `Cursor`

  (CursorWrapper delegates all calls to the actual cursor object. The primary use for this class is to extend a cursor while overriding only a subset of its methods.)

```
public class TodoCursorWrapper
extends CursorWrapper {

  public TodoCursorWrapper(Cursor
cursor){
      super(cursor);
  }

  public Todo getTodo() {
    String uuidString =
getString(getColumnIndex(TodoDbSchema

    String title =
getString(getColumnIndex(TodoDbSchema

    String detail =
getString(getColumnIndex(TodoDbSchema

    Long date =
getLong(getColumnIndex(TodoDbSchema.T

    int isComplete =
getInt(getColumnIndex(TodoDbSchema.To



    Todo todo = new
Todo(UUID.fromString(uuidString));
    todo.setTitle(title);
    todo.setDetail(detail);
    todo.setDate(new Date(date));
    todo.setComplete(isComplete);
```

```
    return todo;
  }
}
```

- Generic `select` (Read) using

  `TodoCursorWrapper`

  ```
  private TodoCursorWrapper
  queryTodoList(
      String whereClause, String[]
  whereArgs) {

    Cursor cursor = mDatabase.query(
      TodoDbSchema.TodoTable.NAME,
  null, whereClause, whereArgs, …
    );

    return new
  TodoCursorWrapper(cursor);
  }
  ```

- ▶ Read a Todo using

  `TodoCursorWrapper`

- ▶ Read a list of Todos using

  `TodoCursorWrapper`

- ▼ Debugging
  - Changes in DDL should lead to the `SQLiteOpenHelper` change the version number and update the tables in the `onUpgrade` method
  - Or, destroy the database by deleting the app on the device and start again.

    (For a new app, `SQLiteOpenHelper.onCreate()` is called and a new database instance is created).
  - Use a breakpoint and examine the detail of `SQL` statement for correct syntax.

▼ FileProvider

- `FileProvider` has directory and file handling stored in a private space within the app.
- The authority is a location files are saved to and is defined in the `AndroidManifest` file

```
<provider

android:authorities="com.example.todolistapp.filepro



android:name="android.support.v4.content.FileProvide

  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data

android:name="android.support.FILE_PROVIDER_PATHS"
    android:resource="@xml/files" />
</provider>
```

- `android:exported="false"` only this app (and any it gives permission to) have access to this `provider`. This app has `authority` to this location.
- The `FileProvider` knows the location of files with the `resource="@xml/files"` declaration:

```
<paths>
  <files-path
    name="todo_photos"
    path=".">
  </files-path>
</paths>
```

- Note the `meta` tag in the manifest file for the `FileProvider`
- `FileProvider.getUriForFile()` translates the local filepath to a `uri` that other apps, such as a camera app, can see.
- To grant permission to write to an `activity`:

```
getActivity().grantUriPermission(
  activity.activityInfo.packageName,
  uri, Intent.FLAG_GRANT_WRITE_URI_PERMISSION );
```

- To access a `uri`:

```
FileProvider.getUriForFile(getActivity(),
  "com.example.todolistapp.fileprovider",
mPhotoFile);
```

- To revoke permission:

```
getActivity().revokeUriPermission(
  uri, Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
```

▼ Implicit Intent using a Camera App

▼ QA

In Android, what is an Activity?

a. A single screen the user sees on the device at any one time
b. A message sent among the major classes
c. A component that runs in the background without any interface
d. Context refering to the application environment

▼ Answer!

Process of elimination?!

An Activity can be thought of as corresponding to what?

a. An Android project
b. A Java Class
c. A method call
d. An object field

▼ Answer!

Process of elimination again?!

Lifecycles and Best Practice

> The android.arch.lifecycle package provides classes and interfaces that let you build lifecycle-aware components — which are components that can automatically adjust their behavior based on the current lifecycle of an activity or fragment.
>
> *developer.android.com*

- Keep UI controllers (activities and fragments) as lean as possible and delegate to `ViewModel` and use `LiveData` to reflect the changes back to the views
- It is the UI's controller that should update views as data changes
- Data logic should be encapsulated in the `ViewModel` which sits between the UI controller and the model; *the `ViewModel` should call other components to fetch data and not be tightly coupled to the data source*

Android Platform Architecture

activity intent

Android Platform Architecture


System Apps
Dialer
Email
Calendar
Camera
. . .
Java API Framework
Content Providers
View System
Managers
Activity
Location
Package
Notification
Resource
Telephony
Window
Native C/C++ Libraries
Webkit
OpenMAX AL
Libc
Media Framework
OpenGL ES
. . .
Android Runtime
Android Runtime (ART)
Core Libraries
Hardware Abstraction Layer (HAL)
Audio
Bluetooth
Camera
Sensors
. . .
Linux Kernel
Drivers
Audio
Binder (IPC)
Display
Keypad
Bluetooth
Camera
Shared Memory
USB
WIFI
Power Management

Java

Look up the following in the Java documentation: package, import, public, protected, private, class, extends, @override, void, method, final, casting, Arrays, new, instance variable, events, listeners

JVM

An abstract (virtual) computing machine to run a Java program.

Address Space, Memory Management and Garbage collection

Class loader

The *.class* files are in machine independant *bytecode* format and are read into memory by:

- the *App class loader* for the classes in the App *CLASSPATH*
- *Extension Class loader* for the extension classes (jre/lib/ext) imported into the App and
- the *Bootstrap class loader* (rt.jar)

Link

stage starts to process the *bytecode* in the class files in three stages:

- **Verify** the bytcode loaded
- **Prepare** allocate memory for class scope `static` variables
- **Resolve** all symbolic references to actual addresses.

- **Initialise** set initialised constants, execute static block.

Memory

- **Metaspace** Class data, static variables
- **Heap** objects, instance variables, arrays
- **PC Register** Program Counter per thread
- **Java Stacks** Stack frames per method call per thread.
- **Native method stack**

Memory management

[A practical look at the *stack* and *heap*.]

Address Space, Memory Management and Garbage collection

PC Registers	**Java Stacks** (Primitive data types and methods)	**Heap** (Shared by all threads)	**Method area** (Shared by all threads)	**Native Method Stacks** (Shared by all threads)
thread 1	e.g. MyClass myObject = new MyClass() third_method() second_method() int j first_method() int i	int m (assume int m is a class variable defined in MyClass	class data class data	

Execution engine

JRE has:

- An interperter and a Just In Time (JIT) compiler that interfaces with Native method interface and libraries to execute the compiled code.

- It also does garbage collection of objects that have gone out of scope.
- Android execution of mobile apps is slightly different in that the bytcode is compiled. Android 4.4 introduced **ART** Android Runtime environment which compiles the bytcode to machine code before the App is installed.