Mobile Applications Development

07.1 – Data Management and Persistency

Objectives

- To learn what are the mechanisms provided by Android for data persistency.
- Data consistency
- To learn how to share data with the mobile applications.

Saving data

- Persistent or Non-Persistent
- In computer science, persistence refers to the characteristic of state that outlives the process that created it..
  - E.g. Data that can be retrieved after the device's power cycle.
    - Date, time, settings

Saving data – Non-Persistent

- Making use of Application Class

```
class SalesApplication : Application() {

}
```

- Update AndroidManifest.xml

Package name

File name

```
<application
    android:name=".lab07.SalesApplication"
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="SalesTracker Complete"
    android:supportsRtl="true"
    android:theme="@style/AppTheme" >
    <activity android:name=".MainActivity"...>
```

Saving data – Non-Persistent

- Creating / modifying data

Holds a set of data

Adds / removes data

Method to retrieve all data

```
class SalesApplication : Application() {

    var salesArray: ArrayList<Sale>? = null

    var sdf: SimpleDateFormat? = null
    private val ourInstance = SalesApplication()

    fun getInstance(): SalesApplication {...}

    init {...}

    fun getDateFormat(): SimpleDateFormat? {...}

    fun addSales(newSale: Sale) {...}

    fun removeSales(index: Int) {...}

    fun getArray() : ArrayList<Sale>?{...}
}
```

Saving data – Non-Persistent

- Usage of application class

```
class SalesApplication : Application() {

    var salesArray: ArrayList<Sale>? = null

    var sdf: SimpleDateFormat? = null

    private val ourInstance = SalesApplication()

    fun getInstance(): SalesApplication {
        return ourInstance
    }
```

Creates a static instance of class

Method to return the instance of the class

- In Activity (E.g. AddSalesActivity.java)

Get static instance and assign to “sa”

```
class AddSales : AppCompatActivity() {

    var sa : SalesApplication = SalesApplication().getInstance()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_add_sales)
```

```
s = Sale(name,cost,sellprice,dateSale,staffDisc,qty)

addSales(s)
```

Make use of “sa” to add sales.
i.e. adding “s” to the array list in the SalesApplication

Persistent data management techniques

- Files
- Shared Preferences
- SQLite Database
- Content Providers

Files

- Applications are allow to create their data files on device storage when permission is granted.
- Files are can be stored in application directory or on external storage (e.g. SD card).
- File read and write is done using the usual File I/O libraries (e.g. java.io).

Persistent UI State

- Activities offer “onSaveInstanceState”.
- Designed specifically to persist the UI state.
- Works like the Shared Preference.
- Offers a Bundle parameter that represents a key / vale map of primitive types.
- E.g.
  - public void onCreate (Bundle savedInstanceState) { }

Bundle

- Used to record the values.
- Needed for an activity to provide an identical UI following unexpected restarts.
- Provide a consistent UI for the user.

Saving UI State

- Make use of bundle that is being passed in as a parameter in
  - onCreate()
  - onRestoreInstanceState()

Saving UI State - Example

```
override fun onSaveInstanceState(outState: Bundle?) {

    super.onSaveInstanceState(outState)

    outState?.putString("TEXTVIEW_STATE_KEY",myTextView.text.toString())

    super.onSaveInstanceState(outState)

}
```

Extracting UI State - Example

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    var text =" ";

    if(savedInstanceState != null &&
            savedInstanceState.containsKey("TEXTVIEW_STATE_KEY"))
    {
        text = savedInstanceState.getString("TEXTVIEW_STATE_KEY")
    }


}
```

Extracting UI State - Example

```
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    super.onRestoreInstanceState(savedInstanceState)

    var text = " "
    if(savedInstanceState != null )
    {
        if(savedInstanceState.containsKey("TEXTVIEW_STATE_KEY"))
        {
            text = savedInstanceState.getString("TEXTVIEW_STATE_KEY")
            myTextView.setText(text);
        }
    }


}
```

Database

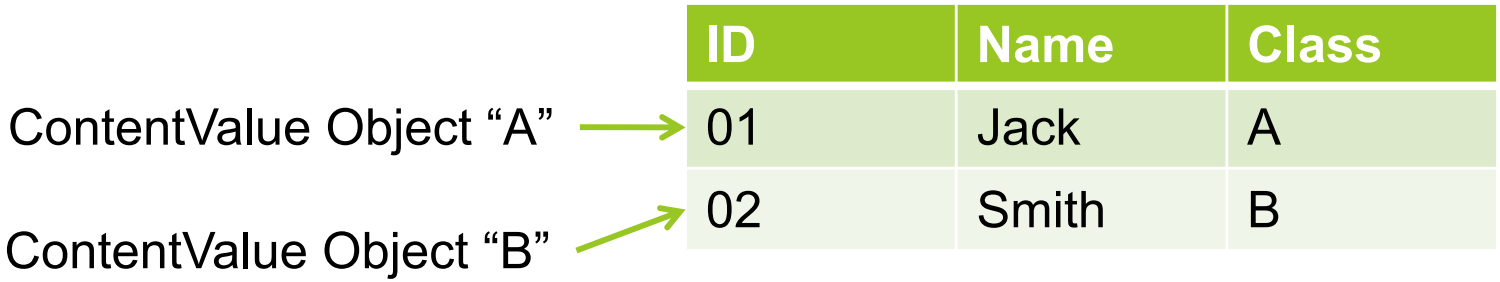
- SQLite
  - Open source
  - Standards-compliant
  - Lightweight
  - Single-tier
- Uses SQLiteOpenHelper to get database instance.
- Makes use of ContentValues and Cursor objects.

SQLiteOpenHelper

- A helper class that manages database creation and versioning.
- Wraps up the best practice for databases
  - Hides the logic used to decide if a database needs to be created or upgraded before it's opened
- Calls getReadableDatabase or getWriteableDatabase to open and return a readable / writable instance of the database

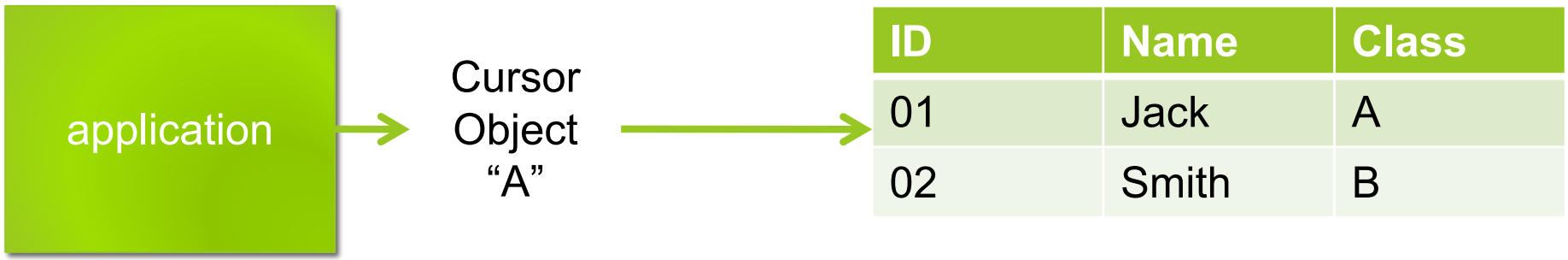
ContentValues

- Used to insert new rows into database and content providers.
- Each ContentValue object represents a single row.



Cursor

- Used to navigate query results.
- Row by row.
- Instead of extracting and returning a copy of the result values, it is just a “pointer”.



ID	Name	Class
01	Jack	A
02	Smith	B

SQLiteHelper - Example

- Creating and opening database

```
private val DATABASE_NAME = "test.db"
private val DATABASE_TABLE = "myTestDb”
protected val DATABASE_CREATE = “create table ” + DATABASE_TABLE +
            “( _id integer primary key autoincrement , ” + “column_one text not null;”) ;

private var dbHelper: MyDBOpenHelper? = null
init {
    dbHelper = MyDBOpenHelper(c, DATABASE_NAME, DATABASE_VERSION)
}
fun open() {
    try {
        _db = dbHelper?.getWritableDatabase()
    } catch (e: SQLiteException) {
        _db = dbHelper?.getReadableDatabase()
    }

}
```

SQLiteHelper - Example

```
inner class MyDBOpenHelper(c: Context, db_name : String, ver_no : Int ):
SQLiteOpenHelper(c, db_name, null, ver_no){


    override fun onCreate(db: SQLiteDatabase?) {
        db!!.execSQL(DATABASE_CREATE)
        Log.w(MYDBADAPTER_LOG_CAT, "HELPER : DB $DATABASE_TABLE created!")

    }

    override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {

    }
}
```

Database Management

- Query

```
// Return all rows for columns ENTRY_NAME and ENTRY_TEL
// (String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)
c = _db?.query(DATABASE_TABLE, arrayOf(KEY_ID, ENTRY_NAME, ENTRY_TEL), null, null, null, null, null)
// Return all columns for rows where column ENTRY_NAME equals a set value
// and the rows are ordered by ENTRY_NAME in descending order
var requiredValue = "John"
var where = ENTRY_NAME + "="+requiredValue
var order = ENTRY_NAME + "DESC";

c = _db?.query(DATABASE_TABLE, null, where, null, null, null, order);
```

Database Management

- Extracting data with cursor

```
c = _db?.query(DATABASE_TABLE, arrayOf(KEY_ID, ENTRY_NAME, ENTRY_TEL),
null, null, null, null, null)
var RESULT_COLUMN = 2
var myTotalFloat = 0;

//Make sure there is at least 1 row
if (myCursor != null && myCursor!!.count > 0) {

    myCursor!!.moveToFirst()    //iterate over each cursor
    do
    {
        var myValue = myCursor.getFloat(2);
        myTotalFloat += myValue;
    }
    while(myCursor.movetoNext());
}
```

Database Management

- Adding rows
  - val newValues = ContentValues();
  - newValues.put(COLUMN_NAME, newValue);
  - myDB?.insert(DATABASE_TABLE, null, newValues);
- Updating rows
  - val updatedValues = ContentValues();
  - updatedValues.put(COLUMN_NAME, newValue);
  - String where = KEY_ID + "= " + rowId;
  - myDB?.update(DATABASE_TABLE, updatedValues, where , null);
- Deleting rows
  - myDB!!.delete(DATABASE_TABLE, KEY_ID + " = " + rowId, null);

Summary

- Depending on the requirements, different data persistency methods are used.
  - E.g. Simple user preference settings: SharedPreferences
  - E.g. More complex data storage: SQLite Database
- PreferenceActivity are commonly used to create application settings screen.
- The state of the UI can be saved and restore using the onSaveInstanceState parameter.
- The use of ContentProvider and ContentResolver to share data between applications.