ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**

Edición 2020 - Materia: **Aplicaciones Android**

TEMA: Persistencia

Consideraciones

- Estas transparencias **no** tienen el objetivo de suplir las clases.
- Por tanto, serán **complementadas** con ejemplos, códigos, profundizaciones y comentarios por parte del docente.
- El **orden** de dictado de estos temas está sujeto a la consideración del docente.

Referencias

- Documentación para developers:
  - *http://developer.android.com/*
- Opciones de almacenamiento:
  - *https://developer.android.com/guide/topics/data/data-storage.html*
- Cómo guardar conjuntos clave - valor:
  - *https://developer.android.com/training/basics/data-storage/shared-preferences.html*
- Cómo guardar archivos:
  - *https://developer.android.com/training/basics/data-storage/files.html*
- Ubicación de instalación de la app:
  - *https://developer.android.com/guide/topics/data/install-location.html*
- Cómo guardar datos en bases de datos SQL:
  - *https://developer.android.com/training/basics/data-storage/databases.html*

Agenda

- Persistencia
- SharedPreferences
- Almacenamiento Interno
- Almacenamiento Externo
- SQLite

Persistencia

- Alternativas para persistir datos en el dispositivo Android
  - **SharedPreferences**:
    - Los datos son privados.
    - Parejas de clave - valor.
    - Tipos primitivos de datos.
  - Almacenamiento **interno**:
    - Los datos son privados.
    - Memoria interna del dispositivo.
  - Almacenamiento **externo**:
    - Los datos son públicos:
    - Almacenamiento externo del dispositivo (si tiene).
  - **SQLite**
    - Los datos son privados.
    - Datos estructurados.

SharedPreferences

- Tipos de datos soportados: booleans, floats, ints, longs, y strings
- La información es persistente entre sesiones de usuario. Incluso si la aplicación deja de ejecutarse.
- Se utiliza la clase **SharedPreferences** como framework para almacenar / obtener la información.
- **getSharedPreferences(String name, int** mode): si se necesitan almacenamientos diferentes para distintos propósitos.
- **getPreferences(int mode**): si la aplicación utiliza un único almacenamiento.
    - *MODE_PRIVATE, MODE_WORLD_READABLE,*
    - *MODE_WORLD_WRITEABLE, MODE_MULTI_PROCESS*

Almacenamiento Interno (1)

- Utilizado para almacenar archivos en el dispositivo.
- Por defecto, los archivos almacenados por una aplicación no son visibles por otras.
- Los archivos son removidos al desinstalar la aplicación.
- Métodos del API:
  - **openFileOutput**(): abre un archivo y retorna un *FileOutputStream* para su escritura.
  - **write**(): escribe al archivo.
  - **openFileInput**(): abre un archivo y retorna un *FileInputStream* para su lectura.
  - **read**(): lee del archivo.

Almacenamiento Interno (2)

- Métodos del API (cont.):
  - **close**(): cierra el archivo. Modos: *MODE_PRIVATE, MODE_APPEND, MODE_WORLD_READABLE, MODE_WORLD_WRITEABLE*
  - **getFilesDir**(): Obtiene la ruta absoluta del directorio donde los archivos para una aplicación dada son almacenados.
  - **getDir**(): Crea (o abre si existe) un directorio en la memoria interna.
  - **deleteFile**(): Borra archivo almacenado en la memoria interna.
  - **fileList**(): Retorna lista de archivos almacenados por la aplicación.

Almacenamiento Externo (1)

- Utilizado para almacenar archivos en el almacenamiento externo del dispositivo.
- Los archivos almacenados por una aplicación son visibles por otras. Incluso pueden ser vistos por cualquier persona al montar el teléfono como Almacenamiento Masivo USB en un PC convencional.
- No esta siempre disponible. Monta como Almacenamiento Masivo. Se extrae.
- Los archivos son removidos durante la desinstalación: sólo los que se hayan almacenado obteniendo el directorio destino mediante **getExternalFilesDir**().

Almacenamiento Externo (2)

- La aplicación (apk) puede ser movida al almacenamiento externo:
  - Es posible ofrecer al usuario la alternativa de almacenar la aplicación en almacenamiento externo.
  - Opcional (por el usuario).
  - Atributo installLocation del MANIFEST: *android:installLocation=["auto" | "internalOnly" | "preferExternal"]*
  - La información de la aplicación aún se persiste en almacenamiento interno.
- Permisos en **MANIFEST**:
  - *READ_EXTERNAL_STORAGE*: leer del almacenamiento externo
  - *WRITE_EXTERNAL_STORAGE*: escribir en el almacenamiento externo

Almacenamiento Externo (3)

- Por defecto todas las aplicaciones pueden leer.
  - Probablemente cambie en futuras versiones de Android.
  - Se recomienda pedir explícitamente el permiso en caso de querer leer del almacenamiento externo.
- Para escribir siempre hay que pedir explícitamente el permiso.
- Si se pide *WRITE_EXTERNAL_STORAGE*, se obtiene también *READ_EXTERNAL_STORAGE*. Se asume que quien escribe en el almacenamiento, va a querer leer en algún momento
- Como el almacenamiento puede estar disponible o no (teléfono conectado a USB, almacenamiento removido) es necesario controlar el estado previo al leer / escribir en él.

Almacenamiento Externo (4)

- Utilizar Environment:
  - *Environment.getExternalStorageState()*
  - *Environment.MEDIA_MOUNTED*
  - */Environment.MEDIA_MOUNTED_READ_ONLY*
- Si bien el almacenamiento externo es inherentemente publico, es posible indicar tipos de archivos “Públicos” o “Privados”.
- **Públicos**:
  - Environment.getExternalStoragePublicDirectory()
  - Están disponible a otras aplicaciones.
  - Cuando se desinstala la aplicación quedan en el almacenamiento.

Almacenamiento Externo (5)

- **Privados**:
  - **Environment.getExternalStorageDirectory**()
  - No están disponibles a otras aplicaciones.
  - No deberían exponer información ni funcionalidad.
  - Cuando se desinstala la aplicación son removidos.
  - Típicamente: archivos de log, descargas de internet, etc.
- Compatibilidad v4.3 y anterior – Biblioteca de Soporte
  - **ContextCompat.getExternalFilesDir**()
  - **ContextCompat.getExternalCachesDir**()

Almacenamiento Externo (6)

- **ContextWrapper.getExternalFilesDir(Environment.<field>)**
  - *DIRECTORY_ALARMS*: archivos de audio para ofrecer durante creación de alarmas
  - *DIRECTORY_DCIM*: ubicación de imágenes y videos cuando se monta el dispositivo como cámara
  - *DIRECTORY_DOCUMENTS*: documentos creados por el usuario
  - *DIRECTORY_DOWNLOADS*: ubicación estándar para las descargas del usuario
  - *DIRECTORY_MOVIES*: ubicación estándar de películas del usuario

Almacenamiento Externo (7)

- **ContextWrapper.getExternalFilesDir(Environment.<field>)**
  - *DIRECTORY_MUSIC*: música del usuario que le aparecerá disponible para reproducción
  - *DIRECTORY_NOTIFICATIONS*: distintos tipos de audios (archivos) disponibles para las notificaciones (eventos) del teléfono
  - *DIRECTORY_PICTURES*: ubicación estándar para fotos del usuario
  - *DIRECTORY_PODCASTS*: archivos de audio para ofrecer en la lista de Podcasts
  - *DIRECTORY_RINGTONES*: archivos de audio para ofrecer en la lista de timbres

Almacenamiento Externo (8)

- **Environment.getExternalStorageState**() --> **String** (estado)
  - *MEDIA_BAD_REMOVAL*: el dispositivo fue mal removido.
  - *MEDIA_CHECKING*: el dispositivo está disponible pero siendo validado por el sistema.
  - *MEDIA_MOUNTED*: el dispositivo está montado con acceso de lectura / escritura.
  - *MEDIA_MOUNTED_READ_ONLY*: el dispositivo está montado con acceso de sólo lectura.
  - *MEDIA_NOFS*: el almacenamiento está presente pero en blanco o con un sistema de archivos no soportado.

Almacenamiento Externo (9)

- **Environment.getExternalStorageState**() --> **String** (estado)
  - *MEDIA_REMOVED*: el almacenamiento no está disponible.
  - *MEDIA_SHARED*: el dispositivo de almacenamiento está disponible, pero está siendo accedido vía Almacenamiento Masivo USB.
  - *MEDIA_UNKNOWN*: estado desconocido.
  - *MEDIA_UNMOUNTABLE*: el almacenamiento está presente pero no puede ser montado.
  - *MEDIA_UNMOUNTED*: el almacenamiento está presente pero no montado (aún).

SQLite (1)

- Android ofrece soporte competo para bases de datos SQLite (www.sqlite.com).
  - Auto contenida
  - Mínima (zero) configuración
  - Implementa lenguaje SQL estándar
  - Soporta transacciones
- Las bases de datos creadas por una aplicación son accesibles por cualquier actividad de la aplicación.
  - No son visibles por otras aplicaciones.
  - Son implementadas en el sistema de archivos. Se utilizan los mismos mecanismos para aislar de otras aplicaciones.

SQLite (2)

- ***SQLiteOpenHelper***
  - Set de utilerías para el manejo de bases de datos.
  - Se recomienda extenderla y sobrescribir al menos **onCreate**().

- Para acceder a la base de datos:
  - **getWritableDatabase**() o **getReadableDatabase**() sobre el ***SQLiteOpenHelper*** (específico).
  - Luego se obtiene un ***SQLiteDabatase*** sobre el que se ejecutan las consultas.

SQLite (3)

- Para ejecutar consultas sobre la base de datos:
  - **SQLiteDababase.query**()
  - **SQLiteQueryBuilder** útil cuando se necesitan consultas complejas, con 'aliases', unions, distincts, etc.
  - **Cursor**
    - Mecanismo provisto para navegar sobre columnas y filas.
    - Contiene punteros a las filas resultado de una consulta.
    - ***Cursor.getColumnIndex(String)*** o ***Cursor.getColumnIndexOrThrow(String)*** para obtener el índice de una columna de la tabla.

SQLite (4)

- SQLiteOpenHelper específico

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
                "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
                KEY_WORD + " TEXT, " +
                KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}
```

SQLite (5)

- *ContentValues* e insert en la base de datos:

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
         FeedEntry.TABLE_NAME,
         FeedEntry.COLUMN_NAME_NULLABLE,
         values);
```

SQLite (6)

- Lectura de registros y navegación (Cursor):

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ...
    };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedEntry.TABLE_NAME,  // The table to query
    projection,                               // The columns to return
    selection,                                // The columns for the WHERE clause
    selectionArgs,                            // The values for the WHERE clause
    null,                                     // don't group the rows
    null,                                     // don't filter by row groups
    sortOrder                                 // The sort order
    );
```

SQLite (7)

- Remover registros por una condición (LIKE):

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

SQLite (8)

- Actualizar registros en base a una condición (LIKE):

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

SQLite (9)

- Es posible acceder a la base de datos por consola.
- **ADB** (Android Debug Bridge):
- **Utilería sqlite** (otra: shell, screenrecord, ls, logcat, network tools, etc.)

```
adb -s emulator-5554 shell
# sqlite3 /data/data/com.example.google.rss.rssexample/databases/rssitems.db
SQLite version 3.3.12
Enter ".help" for instructions
.... enter commands, then quit...
sqlite> .exit
```