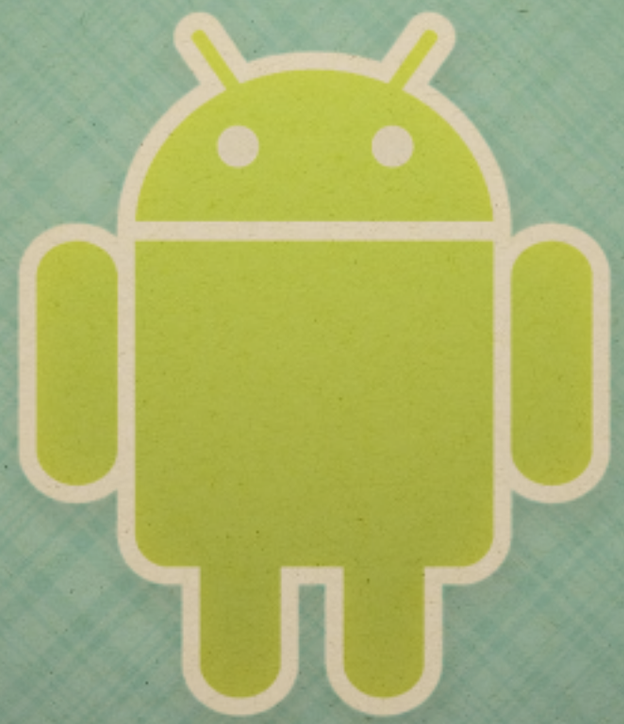
Android



Sesión 7

Qué vamos a ver hoy?

- Persistencia
  - A nivel de aplicación
  - A nivel de dispositivo
    - SharedPreferences
    - Almacenamiento interno y externo
    - SQLite

Persistencia

¿Qué significa la persistencia en móviles?:

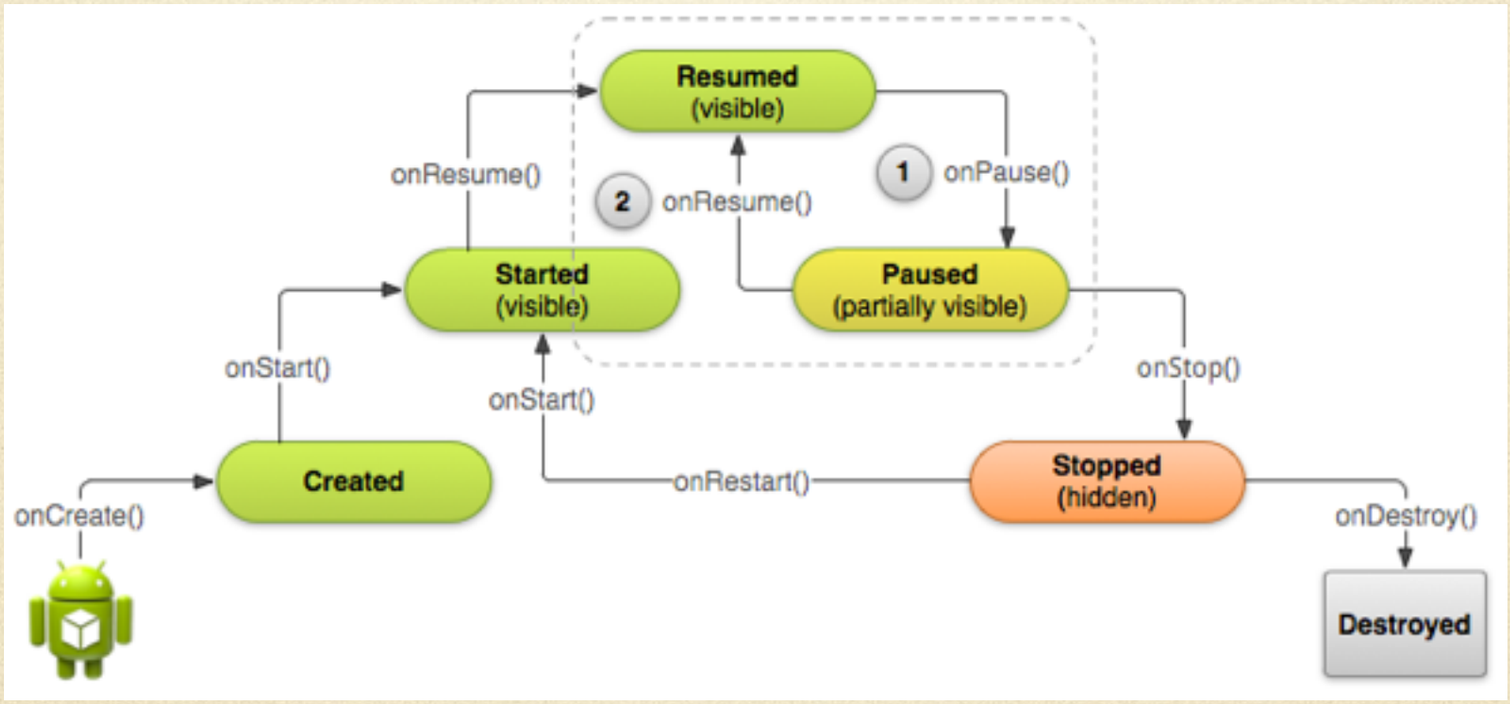
- Posibilidad de utilizar la memoria interna del dispositivo para almacenar datos
- Existen dos tipos:
  - A nivel de aplicación
  - A nivel del dispositivo

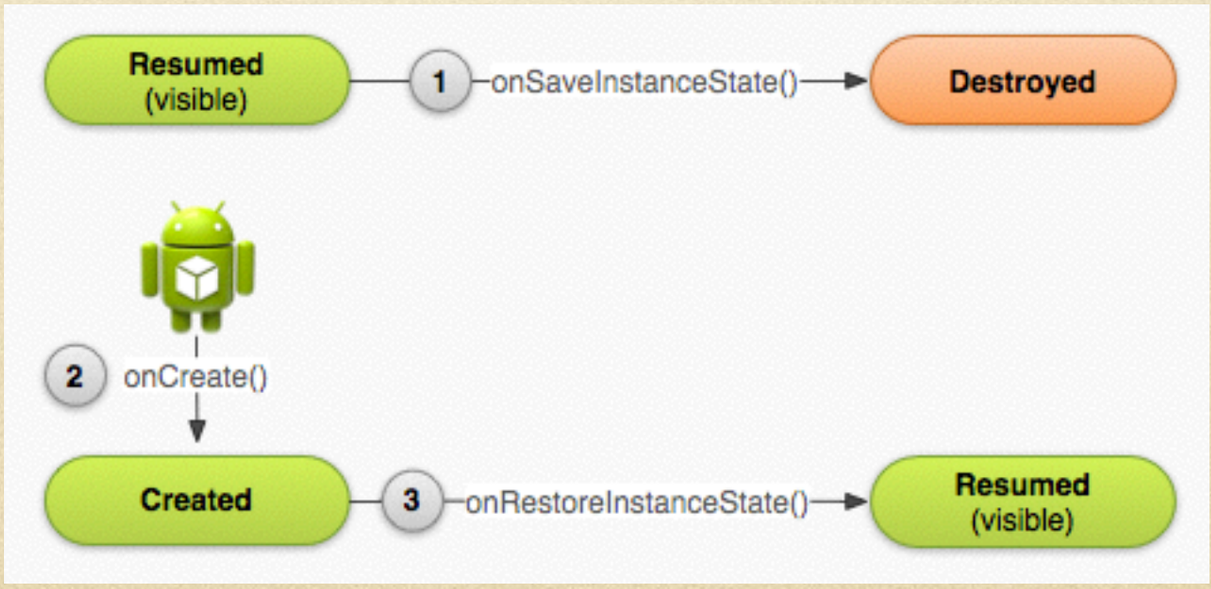
Persistencia a nivel de aplicación

- Debemos persistir ciertos datos durante el ciclo de vida de nuestra aplicación

- Nos sirva para poder persistir datos a pesar que nuestra aplicación haya entrado al background.

Persistencia a nivel de aplicación





onSaveInstanceState / onRestoreInstanceState

- `onSaveInstanceState` se llama cada vez que un Activity podría ser eliminada.
- Permite guardar datos importantes en la clase `Bundle`
- Siempre hay que llamar a `super.onSaveInstanceState`, dado que allí se guarda el estado de los Views
- Para restaurar los datos:
  - Hacerlo en `onCreate` (comprobar si es una restauración o no)
  - Hacerlo en `onRestoreInstanceState` (no es necesario comprobar)

Guardar el estado:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putString("nombreAnterior", nombreAnterior);
}
```

Restaurar:

En onCreate

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    if (savedInstanceState != null) {
        nombreAnterior = savedInstanceState.getString("nombreAnterior");
    }
```

En onRestoreInstanceState

```
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    nombreAnterior = savedInstanceState.getString("nombreAnterior");
}
```

Persistencia a nivel de dispositivo

- Cuatro diferentes maneras:
  - Shared Preferences
  - Almacenamiento interno
  - Almacenamiento externo
  - SQLite

SharedPreferences

- Nos permiten guardar y recuperar datos estilo clave-valor
- Solo se aceptan datos primitivos
- Estos datos se persistirán más allá del ciclo de vida de la aplicación
- Dos formas de utilizarla:
  - `getPreferences()`: datos exclusivos de un activity
  - `getSharedPrefences()`: datos independientes de los Activities (referenciados por nombre)

Para obtener preferences

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // ...

    SharedPreferences preferences = getPreferences(MODE_PRIVATE);
    String ultimoUsuario = preferences.getString("ultimoUsuario", "");
    eteUsuario.setText(ultimoUsuario);
}
```

Para guardar preferences

```
@Override
protected void onStop() {
    super.onStop();

    SharedPreferences preferences = getPreferences(MODE_PRIVATE);
    SharedPreferences.Editor editor = preferences.edit();
    editor.putString("ultimoUsuario", eteUsuario.getText().toString());
    editor.commit();
}
```

Almacenamiento interno

- Se guardan datos en la memoria interna del teléfono (diferente de la memoria SD)
- Los datos guardados son solo accesibles desde la misma aplicación (a no ser que se especifique lo contrario)
- Cuando se elimina una aplicación, estos datos también son eliminados
- Se maneja en forma de archivos

Almacenamiento interno - Lectura

- Llamar al método `openFileInput()`

```
FileInputStream inFile =
openFileInput("archivo")
```

- Usar el método `read()` para leer el contenido (como bytes)

```
inFile.read(bytes, offset, byte.length);
```

- Al finalizar, utilizar el método `close()` para cerrar el flujo.

Almacenamiento interno - Escritura

- Llamar al método `openFileOutput()`

```
FileOutputStream outFile =
openFileOutput("archivo", MODE_PRIVATE);
```

- Usar el método `write()` para escribir los datos

```
outFile.write(bytes, 0, bytes.length);
```

- Al terminar, se cierra el flujo utilizando el método `close()`

Modos de Operación

- `MODE_PRIVATE`: Se crea el archivo y lo hace privado solamente para la aplicación. Si existe, lo sobreescribe.
- `MODE_APPEND`: Si el archivo existe, añade los datos al final (sin sobreescribir)
- *Deprecated* a partir de la versión 4.2.2 (API level 17)
  - `MODE_WORLD_READABLE` (permite que otras aplicaciones puedan leer el archivo)
  - `MODE_WORLD_WRITEABLE` (permite que otras aplicaciones puedan modificar el archivo)

Archivos cache

- Son archivos temporales, que no se almacenan permanentemente.
- Con `getCacheDir()` podemos acceder al directorio donde se crean los archivos de cache.
- Cuando el equipo se encuentra bajo de almacenamiento, Android podrá eliminar estos archivos sin previo aviso.
  Esto no significa que no hay que hacerles seguimiento.

Otros métodos importantes

- `getFilesDir()`: Nos da el path absoluto del filesystem donde nuestros archivos son guardados
- `getDir()`: Crea (abre) un directorio dentro del espacio de memoria interna
- `deleteFile()`: Permite eliminar un archivos
- `fileList()`: Devuelve un arreglo con los archivos guardados por nuestra aplicación

Almacenamiento Externo

- Archivos que se guardan en un dispositivo de almacenamiento "externo" (que puede ser memoria interna no removible o memoria removible como un SD card)

- Estos archivos son `WORLD_READABLE` y pueden ser modificados por el usuario cuando activa el USB mass storage

- **Cuidado:** Al ser una almacenamiento externo, el usuario puede sacar el dispositivo, con lo cual el almacenamiento ya no estaría disponible
  Se debe verificar la disponibilidad antes de hacer la lectura

Pasos

- Indicar permisos para la aplicación
- Validar si hay un dispositivo de almacenamiento externo conectado al equipo
- Se verifica si el dispositivo esta montado solamente para lectura o también escritura
- Se procede con la lectura y escritura

Almacenamiento externo - Permisos

- Se necesitan los permisos
  `READ_EXTERNAL_STORAGE` (solo lectura) o
  `WRITE_EXTERNAL_STORAGE` (lectura y escritura)

  - A partir de la versión 4.4 (API level 18) no se requieren estos permisos para archivos privados

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18" />
    ...
</manifest>
```

Verificación de dispositivo

- Se debe verificar si el almacenamiento externo está montado y si es posible solamente la lectura o también la escritura

- Se emplea el método `getExternalStorageState()` y las constantes de la clase `Environment` para verificar el estado

Verificación del dispositivo

```java
/* Verifica si el almacenamiento externo está disponible para lectura y escritura */
public boolean almacenamientoExternoLecturaEscritura() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Verifica si el almacenamiento externo está disponible por lo menos para lectura */
public boolean almacenamientoExternoLectura() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Almacenamiento externo - Lectura / escritura

- Archivos públicos: `getExternalStoragePublicDirectory()`
  - Para contenido adquirido o generado por el usuario (fotos tomadas, música descargada, etc.)
- Archivos privados: `getExternalFilesDir()`
  - Para archivos privados de la aplicación (texturas gráficas, efectos de sonido)
  - Estos archivos se eliminan al desinstalar la aplicación

Almacenamiento externo - Lectura / escritura

```
public File obtenerDirectorioAlbumFotos(String albumName) {
    // Obtener el directorio donde se almacenan las fotos públicas del usuario.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "No se pudo crear el directorio");
    }
    return file;
}
```

SQLite

- Base de datos embebida en el dispositivo (en un archivo de la memoria interna)
- Maneja consultas (queries) y sentencias DML y DDL del standard SQL
- Podemos tener acceso de lectura o de escritura
- Esta base de datos es privada para cada aplicación

Creación de la BD

- Se extiende la clase `SQLiteOpenHelper`
- Se llama al método `onCreate()` siempre que sea la primera vez que se utiliza la base de datos... pero como sabemos eso?
- Se ejecuta el onUpgrade cuando se requiere hacer un cambio en la estructura de la base de datos... pero como Android sabe que hay que hacer upgrade?

Creación de la BD

```
public class DiccionarioHelper extends SQLiteOpenHelper{
        private static final int DATABASE_VERSION = 1;

            private static final String SQL_USUARIOS=
                  "CREATE TABLE IF NOT EXISTS T_USUARIO " +
          "(id integer primary key AUTOINCREMENT, username TEXT, password TEXT );";


      public DiccionarioHelper(Context context, String name,
                    CursorFactory factory, int version) {
                super(context, name, factory, version);
                                }

                                }

                                @Override
                public void onCreate(SQLiteDatabase db) {

                                db.execSQL(SQL_USUARIOS);

                            Log.i("SQLiteEjemplo" , " Se creo la BD");
                                }

                                @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
                            // TODO Auto-generated method stub

                                }

                                }
```

Operaciones en la BD

Inserción de datos

```
private void registrarUsuario(){
DiccionarioHelper dbHelper = new DiccionarioHelper(this, "SQLITEEJEMPLO", null, 1);
SQLiteDatabase db = dbHelper.getWritableDatabase();

ContentValues valores = new ContentValues();
valores.put("username", "pablito");
valores.put("password", "123");
db.insert("T_USUARIO",  null,  valores);
db.close();
Log.i("SQLiteEjemplo" , " Nuevo");
}
```

Modificación de datos

```
private void modificarUsuario(){
DiccionarioHelper dbHelper = new DiccionarioHelper(this, "SQLITEEJEMPLO", null, 1);
SQLiteDatabase db = dbHelper.getWritableDatabase();

ContentValues valores = new ContentValues();
valores.put("username", "pablito_mod");
valores.put("password", "123");
db.update("T_USUARIO", valores, "id=1", null);
db.close();
Log.i("SQLiteEjemplo" , " Modificacion");
}
```

Operaciones en BD

Método para eliminación

```
                    private void eliminaciónUsuarios(){
        DiccionarioHelper dbHelper = new DiccionarioHelper(this, "SQLITEEJEMPLO", null, 1);
                        SQLiteDatabase db = dbHelper.getWritableDatabase();

                            db.delete("T_USUARIO", null, null);

                                    db.close();
                                       }
```

Consultas

- Se utiliza el método `query()` para hacer consultas, que acepta como parámetros:
  - `distinct`: misma función que la palabra DISTINCT en SQL
  - `table`: nombre de la tabla
  - `columns`: columnas a retornar
  - `selection`: filtro del query (cláusula WHERE)
  - `selectionArgs`: valores de los filtros (de ser necesario)
  - `groupBy`: filtro para declarar como agrupar filas (igual que GROUP BY en SQL)
  - `having`: filtro que declara que grupos se mostrarán (igual que HAVING en SQL)
  - `orderBy`: ordenamiento
  - `limit`: número máximo de registros

Cursores

- Interface que nos permite tener acceso de lectura/escritura a un set de datos devuelto por la base de datos.
- No son thread-safe
- Métodos importantes:
  - `close()`: Para cerrar el cursor. Siempre hacerlo!
  - `get<type>(columnIndex)`: Nos retorna el valor según el índice de la columna
  - `getCount()`: Número total de registros
  - `moveToNext()`: Mueve el puntero al siguiente registro. Devuelve falso si es el último. Útil para bucles.