Persistência de dados

SharedPreferences, Internal Storage e External Storage

Armazenando informações ...

Há diferentes formas de se fazer persistência de dados em Android, cada qual mais adequada a um propósito.

Diferenciam-se basicamente em termos de visibilidade (se são privadas ou compartilhadas com outros app) e quanto ao espaço/volume que demandam.

- Vale ressaltar que em dispositivos mobile o espaço de armazenamento é bastante restrito em relação aos computadores e não convém manter grandes quantidades de dados localmente.
- Por outro lado, os custo de pacotes de dados junto as companhias de telefonia podem torna proibitivo a transferência de dados para sistemas de armazenamento remoto através da nuvem.

Formas de persistência

SharedPreferences

Permite salvar e recuperar dados de tipos primitivos na forma de pares chave-valor. As informações são persistidas na sessão do usuário, permanecendo mesmo após o termino da execução do app.

Internal Storage

Armazenamento em arquivos (*streams*) feito diretamente no dispositivo e privativos a aplicação. Quando o usuário desinstala o app esses arquivos também são removidos.

External Storage

Similar ao anterior, porém a informação é colocada no “armazenamento externo” do dispositivo (que pode ser um cartão de memória removível ou interno ao aparelho).

SharedPreferences

Usualmente associados a parâmetros ou preferências do app.

Obtendo um objeto:

- getSharedPreferences(String nome, int modo)
- getPreferences(int modo)

Gravando:

- Obter um objeto SharedPreferences.Editor com o método edit()
- Gravar com putTipo(chave, valor)
- Persistir com o método commit()

Escrevendo:

- Ler com getTipo(chave)

```
Context context = getActivity();
SharedPreferences sharedPref =
    context.getSharedPreferences(getString(R.string.pref_file),
                                 Context.MODE_PRIVATE);
SharedPreferences sharedPref =
     getActivity().getPreferences(Context.MODE_PRIVATE);

SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.high_score_key), newHighScore);
editor.commit();

int defaultValue =
       getResources().getInteger(R.integer.high_score_default);
int highScore =
       sharedPref.getInt(getString(R.string.high_score_key),
                         defaultValue);
```

Internal Storage

Grava dados num arquivo privado no armazenamento interno.

A classe dos dados a serem persistidos deve ser Serializable.

Processo:

- Chamar getFileStreamPath(“arq”), que retornará um FileOutputStream;
- Usando ObjectOutputStream, escrever no arquivo com writeObject(); ou
- Usando ObjectInputStream, escrever no arquivo com readObject();
- Fechar os fluxos com close().

```
//Para Gravar
String FILENAME = "arquivo_de_objetos_serializados";
File file = getFileStreamPath(FILENAME);

FileOutputStream fos = new FileOutputStream(file);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(objeto);
oos.close();
fos.close();

//Para Ler
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
ObTeste retorno = (ObTeste) ois.readObject();
fis.close();
ois.close();
```

External Storage

Mesmo esquema funcional utilizado no esquema Internal Storage.

O app requer permissões de escrita.

É necessário verificar o estado do sistema de armazenamento externo do dispositivo antes.

Use getExternalFilesDir() para abrir um File que representa o diretório de armazenamento externo do aplicativo.

- Requer um parâmetro type que especifica o subdiretório desejado. Este método irá criar o diretório apropriado, se necessário.

Caso queira salvar arquivos que não são específicos da sua aplicação e que não devam ser excluídos quando o app for desinstalado, use Environment.getExternalStoragePublicDirectory()

- Informe como parâmetro o tipo de diretório público desejado.
- Ex: Environment.DIRECTORY_MUSIC, Environment.DIRECTORY_RINGTONES, ou null para receber a referencia do diretório raíz.

Verificando estado do armazenamento externo

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

If ( Environment.MEDIA_MOUNTED.equals(state) ) {                    // Podemos ler e escrever
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) { // Só podemos ler a mídia
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {                                                                  // não é possível ler nem escrever
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

Acessando o armazenamento externo coletivo:

```
// Acessando a pasta pública de Downloads
File dir = Environment.getExternalStoragePublicDirectory(
                                        Environment.DIRECTORY_DOWNLOADS);
File arquivo = new File(dir, "teste.obj");

FileOutputStream fos = new FileOutputStream( arquivo );
ObjectOutputStream oos = new ObjectOutputStream( fos );
oos.writeObject( objeto );
```

Persistência de dados

SQLite Database

SQLite Database

O SQLite é um banco de dados relacional, nativo no Android e que disponibiliza os recursos de um SGBD para aplicações que precisam persistir dados estruturados em suas aplicações.

A forma de utilizá-lo é difere-se um pouco das práticas tradicionais, por conta do gerenciamento feito pelo próprio Android. Recomenda-se a definição de um Esquema e Contrato.

A classe SQLiteOpenHelper fornece uma API para gerenciamento do BD, oferecendo referencias ao banco e permitindo ao sistema realizar operações de criação e atualização apenas quando necessárias e não durante a inicialização da aplicação.

Para usá-la crie uma subclasse que reescreva os métodos onCreate() e onUpgrade().

Criando o Esquema

Consiste na criação de uma classe que encapsule as definições (descrição) do banco de dados.

- Em outras palavras, definir uma classe que especifique o nome das tabelas, assim como os nomes e tipos de cada um de seus campos.

Pode ser criada como uma *inner class* na classe que transaciona com o banco.

- Subclasse de SQLiteOpenHelper

```
// Exemplo de Inner class esquema para um BD
public static class FilmesDesc
                    implements BaseColumns {
    public static final String
    TABELA_NOME    = "Filmes",
    COL_TITULO             = "titulo",
        TIPO_TITULO        = "text";
    COL_SUBTITULO          = "subtitulo",
        TIPO_SUBTITULO  = "text",
    COL_GENERO             = "genero",
         TIPO_GENERO     = "text",
    COL_AVALIACAO          = "avaliacao",
         TIPO_AVALIACAO = "float";
}
```

Criando o Esquema

```
public static final String SQL_CRIAR_TABELA  =
      "CREATE TABLE " + FilmesDesc.TABELA_NOME + " (" +
            FilmesDesc._ID + " INTEGER PRIMARY KEY," +
            FilmesDesc.COL_TITULO    + " " + FilmesDesc.TIPO_TITULO    + "," +
            FilmesDesc.COL_SUBTITULO + " " + FilmesDesc.TIPO_SUBTITULO + "," +
            FilmesDesc.COL_GENERO    + " " + FilmesDesc.TIPO_GENERO    + "," +
            FilmesDesc.COL_AVALIACAO + " " + FilmesDesc.TIPO_AVALIACAO + " )";

public static final String SQL_APAGAR_TABELA =
     "DROP TABLE IF EXISTS " + FilmesDesc.TABELA_NOME;
```

A subclasse de SQLiteOpenHelper

```
public class FilmesBDHelper
        extends SQLiteOpenHelper {

    public static final
        int DATABASE_VERSION = 1;
    public static final
        String DATABASE_NAME = "Filmes.db";

    // método construtor
    public FilmesBDHelper(Context context) {
        super(context, DATABASE_NAME,
                null, DATABASE_VERSION);
    }

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

    @Override
    public void onUpgrade(SQLiteDatabase db,
                int oldVersion, int newVersion){
        db.execSQL(SQL_APAGAR_TABELA);
        onCreate(db);
    }
}
```

Implementado o “Contrato”: inserção

```
public boolean inserir(Filme filme) {
        SQLiteDatabase db = this.getWritableDatabase(); // conecta ao BD para gravação
        ContentValues tupla = new ContentValues();       // objeto para a composição da tupla
        tupla.put( FilmesDesc._ID, filme.getId() );                          // info: chave
        tupla.put( FilmesDesc.COL_TITULO, filme.getTitulo() );               // info: título
        tupla.put( FilmesDesc.COL_SUBTITULO, filme.getSubtitulo() );  // info: subtítulo
        tupla.put( FilmesDesc.COL_GENERO, filme.getGenero() );               // info: gênero
        tupla.put( FilmesDesc.COL_AVALIACAO, filme.getAvaliacao() ); // info: nota de avaliaç
        db.insert( FilmesDesc.TABELA_NOME, null, tupla);  // insere os dados na tabela
        return true;
}
```

Implementado o “Contrato”: consulta por chave

```
public Filme getFilme(int id) {
        Filme filme = null;                         // instância de retorno
        SQLiteDatabase db = this.getReadableDatabase(); // modo leitura
        String[] colunas = {                        // colunas de deverão ser retornadas:
                FilmesDesc._ID,                     //  - chave
                FilmesDesc.COL_TITULO,              //  - título
                FilmesDesc.COL_SUBTITULO,           //  - subtítulo
                FilmesDesc.COL_GENERO,              //  - gênero
                FilmesDesc.COL_AVALIACAO   }; //  - avaliação
        String criterio = FilmesDesc._ID + " = ?";              // critério de consulta (parâmetros)
        String[] valorArgumentos = { Integer.toString(id) } // valor dos parâmetros
```

Implementado o “Contrato”: consulta por chave

```
  cursor.moveToFirst();              // posiciona o cursor da projeção na primeira tupla
  if (!cursor.isAfterLast()) {       // se há algum registro é porque a consulta retornou informação
     filme = new Filme();            // então instanciamos o objeto e populamos seus atributos ...
     filme.setId( cursor.getInt( cursor.getColumnIndex(FilmesDesc._ID)) );
     filme.setTitulo( cursor.getString( cursor.getColumnIndex(FilmesDesc.COL_TITULO)) );
     filme.setSubtitulo( cursor.getString( cursor.getColumnIndex(FilmesDesc.COL_SUBTITULO)) );
     filme.setGenero( cursor.getString( cursor.getColumnIndex(FilmesDesc.COL_GENERO) ) );
     filme.setAvaliacao( cursor.getFloat( cursor.getColumnIndex(FilmesDesc.COL_AVALIACAO) ) );
  }
  if (!cursor.isClosed())  cursor.close();    // antes de sair, encerre as operações
  return filme;
}
```

Implementado o “Contrato”: consultar todos

```
public ArrayList<Filme> getTodosFilmes() {
        ArrayList<Filme> lstFilmes = new ArrayList<Filme>();
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor =
          db.rawQuery("select * from " + FilmesDesc.TABELA_NOME, null);
        cursor.moveToFirst();
        while( !cursor.isAfterLast() ) {
            // crie um novo objeto, popule seus atributos e adicione no vetor
        }
        if (!cursor.isClosed()) cursor.close();
        return lstFilmes;
}
```

Implementado o “Contrato”: consultar todos

```
// detalhando o conteúdo do laço ...
while( !cursor.isAfterLast() ){
   Filme filme = new Filme();  // instancie um novo objeto e popule seus atributos
   filme.setId( cursor.getInt( cursor.getColumnIndex(FilmesDesc._ID)) );
   filme.setTitulo( cursor.getString( cursor.getColumnIndex(FilmesDesc.COL_TITULO)) );
   filme.setSubtitulo( cursor.getString( cursor.getColumnIndex(FilmesDesc.COL_SUBTITULO)) );
   filme.setGenero( cursor.getString( cursor.getColumnIndex(FilmesDesc.COL_GENERO)) );
   filme.setAvaliacao( cursor.getFloat( cursor.getColumnIndex(FilmesDesc.COL_AVALIACAO)) );
   lstFilmes.add( filme );          // adicione o novo objeto ao vetor, e
   cursor.moveToNext();             // mova o cursor para a próxima tupla
}
```