Padrão de Desenho: *Singleton*

Desenvolvimento Móvel em Android

Persistência de Dados: #SQLite

Sónia Luz, sonia.luz@ipleiria.pt
David Safadinho, david.safadinho@ipleiria.pt
Departamento de Engenharia Informática
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Leiria
1º Semestre - 2021/2022

Persistência de Dados

- Considerando que os nossos contactos passam a ser persistentes
  - Devem ser partilhados por todas as atividades da aplicação;
  - Mas neste caso quantas instâncias de **GestorContactos** criamos?
  - Onde?
  - Quantas instâncias queremos que sejam criadas?

Padrão de desenho de software: *Singleton*

- Em Java existe um tipo de classe designado:
  - ***Singleton***
    - É um padrão de desenho de software;
    - Garante a existência de apenas uma instância de uma classe;
    - Mantém um ponto global de acesso ao objeto instanciado;

Padrão de desenho de software: *Singleton*

- Uma classe ***Singleton***
  - Permite definir uma única instância;
  - Partilhada por todos os objetos que lhe acedem;
- Deve conter um atributo estático e privado do tipo do ***Singleton*** que representa a instância;
- Deve conter um construtor privado
  - Para apenas ser invocado dentro da classe;

Padrão de desenho de software: *Singleton*

- O acesso exterior à instância é efetuado através do método **getInstance()**;
  - Deve ser **static**
    - Garantindo que é um método da classe;
  - Deve ser **synchronized**
    - Garantindo que não é possível ser acedido por 2 *threads* em simultâneo;
  - Deve garantir que o construtor só é invocado uma única vez
    - Se ainda não existir instância do objeto;

Criação da classe *Singleton*

- Aplicando este conceito ao nosso projeto
  - Pretendemos substituir a nossa classe **GestorContactos**
  - Começamos por criar a classe **SingletonContactos**

```
public class SingletonContactos {
    private static SingletonContactos INSTANCE = null;

    public static synchronized SingletonContactos getInstance()
    {
        if( INSTANCE == null ){
          INSTANCE = new SingletonContactos();
        }
        return INSTANCE;
    }
    private SingletonContactos() {
    }
}
```

Criação da classe *Singleton*

- Na classe **SingletonContactos** adicionamos o atributo para guardar os contactos atualizados
  - Que instanciamos no construtor;

```
public class SingletonContactos {
    private static SingletonContactos INSTANCE = null;
    private LinkedList<Contacto> contactos;

    private SingletonContactos() {
        contactos = new LinkedList<>();
        //só enquanto os dados não estiverem devidamente guardados
        adicionarDadosIniciais();
    }
```

Criação da classe *Singleton*

- Adicionamos os métodos que devem aceder/atualizar os contactos da nossa aplicação:

```
public class SingletonContactos {
    public LinkedList<Contacto> getContactos() {
        return new LinkedList<>(contactos);
    }
    //métodos para leitura e escrita em ficheiros binários : armazenamento interno
    public void lerContactos(Context context){
        ...
    }
    public void gravarContactos(Context context){
       ...
    }
    //método para adicionar um contacto à lista de contactos
    public void adicionarContacto(Contacto contacto) {
        contacto.setFoto(R.drawable.foto);
        contactos.add(contacto);
    }
}
```

Instanciação da classe *Singleton*

- Para colocarmos a aplicação a funcionar através da instância do ***Singleton***
  - Devemos substituir todos os acessos à classe **GestorContactos**;
  - Através da chamada do método **SingletonContactos.getInstance().getContactos();**
  - Remover o atributo do tipo **GestorContactos** de todas as atividades que o contenham;
  - Remover a classe **GestorContactos** da aplicação;

Instanciação da classe *Singleton*

- Exemplo na atividade **ListViewContactos**

```
public class ListViewContactosActivity extends AppCompatActivity {
...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
...
        if(savedInstanceState == null) {
            //ler contactos do armazemento interno através de ficheiros binários locais
            //this.gestorContactos = new GestorContactos(this);
            //this.gestorContactos.lerContactos();
            SingletonContactos.getInstance().lerContactos(this);
              this.contactos = SingletonContactos.getInstance().getContactos();
        }else{
            this.contactos = (LinkedList<Contacto>)
                    savedInstanceState.getSerializable(ESTADO_CONTACTOS);
        }

        if(this.contactos == null) {
            this.contactos = SingletonContactos.getInstance().getContactos();
            //this.gestorContactos.adicionarDadosIniciais();
        }
...
}
```

Persistência de Dados

- Opção de armazenamento de dados no Android:
  - Bases de Dados SQLite
    - Armazenar dados estruturados numa base de dados privada;

Base de Dados SQLite

- O SQLite é um Base de Dados (BD) relacional
  - Open-source que suporta comandos SQL;
- Cada aplicação Android pode criar várias bases de dados que irão ficar armazenadas no sistema;
- Para fazer o acesso a uma base de dados SQLite dentro da plataforma Android
  - Vamos utilizar uma API de acesso que já vem incluída no pacote SDK;

Base de Dados SQLite

- As classes seguintes vão ser utilizadas para a criação da base de dados:
  - **SQLiteDatabase**: Classe que contém os métodos de manipulação da BD;
    - Classe usada para implementar os comandos SQL;
  - **SQLiteOpenHelper**: Classe responsável pela criação da BD e também responsável pelas versões da mesma;

Base de Dados SQLite

- Ao colocar uma classe a estender/herdar da classe **SQLiteOpenHelper**
  - O Android Studio obriga o programador a implementar dois métodos importantes para o correto funcionamento e criação da BD:
    - **onCreate()**
    - **onUpgrade()**

Base de Dados SQLite

- O método **onCreate()**
  - É chamado quando a aplicação cria a BD pela primeira vez;
    - Neste método devem estar todas as diretrizes de criação e os dados iniciais da BD;

Base de Dados SQLite

- O método **onUpgrade()**
  - É responsável por atualizar a BD com alguma informação estrutural que tenha sido alterada;
    - Este método é chamado sempre que uma atualização for necessária;
      - Para não haver qualquer tipo de inconsistência de dados entre a BD existente no dispositivo Android e a nova BD que a aplicação irá utilizar;

Base de Dados SQLite

- Para criar a BD precisamos de uma nova classe **ModeloBDHelper**

```
public class ModeloBDHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "contactosDB";
    private static final int DB_VERSION = 1;

    private final SQLiteDatabase database;
    //alterar para apenas receber o contexto, e o factory fica a null
    public ModeloBDHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
        this.database = this.getWritableDatabase();
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
```

Base de Dados SQLite

- Para implementar o método **onCreate**()

```
public class ModeloBDHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "contactosDB";
    private static final String TABLE_NAME = "Contacto";
    public static final String ID_CONTACTO = "id";
    public static final String NOME_CONTACTO = "nome";
    public static final String FOTO_CONTACTO = "foto";
...
    @Override
    public void onCreate(SQLiteDatabase db) {
        String createContactoTable = "CREATE TABLE " + TABLE_NAME +
                "( " + ID_CONTACTO + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                NOME_CONTACTO + " TEXT NOT NULL, " +
                FOTO_CONTACTO + " INTEGER" +
        ");";
        db.execSQL(createContactoTable);
    }
```

Base de Dados SQLite

- Para implementar o método **onUpgrade**()

```
public class ModeloBDHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "contactosDB";
    private static final String TABLE_NAME = "Contacto";
  ...
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        this.onCreate(db);
    }
```

Base de Dados SQLite

- Após isso é necessário adicionar os métodos que definem o CRUD sobre a BD
  - **Create;**
  - **Read;**
  - **Update;**
  - **Delete;**

Base de Dados SQLite

- Implementação do método de criação **adicionarContacto**()
  - O comando **insert** tem como primeiro parâmetro o nome da tabela;
  - O segundo parâmetro indica o que fazer quando o **ContentValues** está vazio
    - Se estiver a null não vai inserir nenhuma linha na BD

```
public void adicionarContacto(Contacto contacto){
    ContentValues values = new ContentValues();

    values.put(ID_CONTACTO, contacto.getId());
    values.put(NOME_CONTACTO,contacto.getNome());
    values.put(FOTO_CONTACTO, contacto.getFoto());

    this.database.insert(TABLE_NAME, null, values);
}
```

Base de Dados SQLite

- Implementação do método de leitura **getAllContactos**()
  - Neste caso vamos precisar de uma variável do tipo **Cursor**;
  - Porque todas a consultas devolvem um objeto desse tipo

```
public LinkedList<Contacto> getAllContactos(){
    LinkedList<Contacto> contactos = new LinkedList<>();
    Cursor cursor = this.database.rawQuery("SELECT * FROM " + TABLE_NAME, null);

    if(cursor.moveToFirst()){
        do{
            contactos.add(new Contacto(cursor.getLong(0),
                    cursor.getString(1),
                    cursor.getInt(2)));
        }while(cursor.moveToNext());
    }
    return contactos;
}
```

Base de Dados SQLite

- Implementação do método de update **guardarContacto**()
  - Aqui vamos devolver se existe ou não o contacto com o respetivo id
    - Para indicar sucesso ou não na atualização dos dados;

```
public boolean guardarContacto(Contacto contacto){
    ContentValues values = new ContentValues();

    values.put(ID_CONTACTO, contacto.getId());
    values.put(NOME_CONTACTO,contacto.getNome());
    values.put(FOTO_CONTACTO, contacto.getFoto());

    return this.database.update(TABLE_NAME, values,
            "id = ?", new String[]{"" + contacto.getId()}) > 0;
}
```

Base de Dados SQLite

- Implementação do método de remoção **removerContacto**()
  - Vai apagar o contacto se encontrar o respetivo id;

```
public void removerContacto(long idContacto){
    this.database.delete(TABLE_NAME, "id = ?",
            new String[]{"" + idContacto});
}
```

Base de Dados SQLite

- Ainda podem acrescentar outros métodos para manipulação dos dados:
  - Pesquisar por um campo;
  - Remover todos;
  - ...;

Base de Dados SQLite

- Assim, onde deve ser criada a instância da BD?
  - Na classe **SingletonContactos** para ficar disponível para todas as atividades;
- Onde se deve aceder para guardar os dados/alterações
  - No método **onPause**() da atividade
    - Mas obrigada a apagar todos os registos e voltar a adicionar todos os que estão na lista;
      - Não é o mais correto porque se pode tornar um processo mais moroso e "pesado";
  - Deve ser **sempre** que se quer efetuar uma operação sobre a BD

Base de Dados SQLite

- Na classe **SingletonContactos** será necessário:
  - Um método para iniciar a BD passando-lhe o ***Context***
    - Porque não é possível fazê-lo através do construtor do ***Singleton***;
  - Um método para ler os dados da BD;
  - E para atualizar na BD?
    - Criar um método para gravar tudo;
    - Ou cada método para apenas executar cada operação diretamente
      - Esta é a opção mais correta;

Base de Dados SQLite

- Na classe **SingletonContactos**

```
public class SingletonContactos {
    private static ModeloBDHelper modeloDB = null;
    //metodo para iniciar a BD passando-lhe o contexto
    //se ainda não tiver sido instanciada
    public static void iniciarBD(Context context) {
        if(modeloDB == null)
            modeloDB = new ModeloBDHelper(context);
    }
    public void lerBD(){
        this.contactos = modeloDB.getAllContactos();
    }
    public void gravarBD(){
        modeloDB.removerAllContactos(); //se já existir, remover todos os contactos
        for (Contacto contacto: contactos) {
            modeloDB.adicionarContacto(contacto);
        }
    }
```

Base de Dados SQLite

- Na classe **SingletonContactos**
  - Alguns dos métodos a implementar / atualizar

```
public void adicionarContacto(Contacto contacto) {
    contacto.setFoto(R.drawable.foto);
    contactos.add(contacto);
    //adicionar também à BD
    modeloDB.adicionarContacto(contacto);
}

public void removerContacto(Contacto contacto){
    if(contactos.contains(contacto)) {
        contactos.remove(contacto);
       //remover também da BD
        modeloDB.removerContacto(contacto.getId());
    }
}
```

Base de Dados SQLite

- Na atividade **ListViewContactos** adicionar o uso da BD para ler e guardar dados

```
public class ListViewContactosActivity extends AppCompatActivity {
...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
...
        SingletonContactos.iniciarBD(this);
        if(savedInstanceState == null) {
            SingletonContactos.getInstance().lerBD();
            this.contactos = SingletonContactos.getInstance().getContactos();
        }else{
            this.contactos = (LinkedList<Contacto>)
                    savedInstanceState.getSerializable(ESTADO_GESTOR_CONTACTOS);
        }
        if(this.contactos == null) {
            this.contactos = SingletonContactos.getInstance().getContactos();
        }
...
    }
```

Desafio:

- Criar uma base de Dados SQLite
  - Para gerir os dados dos contactos
    - Ler, gravar, adicionar, remover;

Fontes e Mais Informação

- Classe *Singleton* em Java
  - https://www.javaworld.com/article/2073352/core-java/simply-singleton.html
- Opções de Armazenamento
  - https://developer.android.com/training/data-storage
- Armazenamento em Base de Dados SQLite
  - https://developer.android.com/training/data-storage/sqlite
- Armazenamento em Base de Dados Room
  - https://developer.android.com/training/data-storage/room

Próximo Tema:

Persistência de Dados: Características de uma API REST e Requisitos de Acesso com Aplicações Android

- Opções de Armazenamento
  - https://developer.android.com/training/data-storage
- Armazenamento recorrendo a Ligação de Rede
  - https://developer.android.com/guide/topics/connectivity
- Introdução ao JSON
  - http://www.json.org/json-pt.html
- Classes JSON disponíveis no Android Studio
  - https://developer.android.com/reference/org/json/package-summary.html
- *Transmitting Network Data Using Volley*
  - https://developer.android.com/training/volley/index.html