Tópicos Avançados de Programação

Aula 4

Professor Marcelo Rodrigues do Nascimento

Conversa Inicial

Em nossa aula 3, aprendemos a enviar e receber mensagens entre Activities e a criar um objeto POJO que represente uma estrutura personalizada, entendo ainda como serializá-los.

Com esses passos, aprendemos como desenvolver aplicativos com um nível de complexidade um pouco maior. Desde o instanciamento de novas classes e sua utilização, como o EditText, estamos a cada aula aprofundando nosso conhecimento no desenvolvimento de aplicativos.

Nesta aula, utilizaremos todos os conceitos já vistos para desenvolver um aplicativo com a funcionalidade de armazenamento de dados, desde configurações de seu aplicativo até um cadastro simplificado de amigos.

Confira no material *on-line* a videoaula com os comentários iniciais do professor Marcelo.

Contextualizando

Softwares em geral possuem como característica comum a capacidade de gravação de registros. Sejam configurações de utilização, pontuação máxima obtida por um jogador ou até mesmo registros em um banco de dados, esta funcionalidade é utilizada intensamente em todos os ramos do desenvolvimento.

Daremos agora início à persistência de dados, gravando em disco as preferências de interface de nosso usuário. Também utilizaremos o conhecimento acumulado nas últimas aulas para criar um aplicativo que nos permitirá efetuar um cadastro simplificado de amigos utilizando o banco de dados nativo do Android, SQLite.

Aprenderemos as características principais deste banco, o desenvolvimento de adaptadores de dados e também sua utilização em *layouts* de listagem de dados, que nos servirão para mostrar os registros gravados atualmente.

Confira no vídeo disponível no material *on-line* a contextualização feita pelo professor Marcelo.

Tema 1 – Persistindo valores com SharedPreferences

A plataforma Android fornece várias maneiras para armazenamento de dados em suas aplicações. Uma delas é conhecida como SharedPreferences e permite que tipos primitivos sejam salvos no formato “chave,valor”. Uma vez salvos, estes dados podem ser acessados de qualquer lugar de dentro de seu aplicativo, tanto para leitura quanto para escrita. No entanto, estes dados não poderão ser acessados a partir de outros aplicativos.

Então por que utilizaríamos o SharedPreferences?

Seu uso é comum em situações onde você deve guardar alguma informação primitiva (ou seja, tipos simples) entre sessões de utilização de seu aplicativo. Por exemplo, você poderia guardar o nome do usuário, suas configurações desejadas ou até mesmo a maior pontuação atingida em um jogo.

Suas preferências podem ser salvas em um ou mais arquivos, dependendo de sua necessidade. Para preferências em nível de Activity, utiliza-se a chamada **getPreferences()**. Para preferências em nível de aplicação (ou seja, que podem ser acessadas de qualquer parte da aplicação utiliza-se a chamada **getSharedPreferences()**.

O acesso aos dados armazenados é feito através da instância de um objeto SharedPreferences, e a partir daí utilizamos métodos get para recuperação do valor.

Exemplo para acesso de preferencias em nível de Activity:

```
String nome;

SharedPreferences prefs = getPreferences(MODE_PRIVATE);

prefs.getString(“nome”,nome);
```

Neste exemplo, instanciamos o objeto prefs a partir de SharedPreferences em nível de Activity através do método **getPreferences()** e então preenchemos nossa variável **nome** com os valores encontrados neste objeto, cuja chave de localização seja igual a “nome”.

Caso a informação estivesse armazenada a nível de aplicação, apenas a forma de acesso mudaria. Neste caso, nosso objeto prefs seria instanciado a partir da chamada getSharedPreferences, passando-se como argumentos o nome do arquivo a ser acessado e também o modo de acesso.

Exemplo:

```
String nome;

SharedPreferences prefs =

getApplicationContext().getSharedPreferences(“minhas preferencias”, MODE_PRIVATE);

prefs.getString(“nome”,nome);
```

Ao utilizarmos o método getSharedPreferences() de dentro do contexto de nossa aplicação, informamos que desejamos os valores armazenados no arquivo “minhaspreferencias”. Mais uma vez, lembre-se de que estas preferências são acessíveis a partir de qualquer parte do seu aplicativo, mas são inacessíveis a outros aplicativos.

A gravação de dados em SharedPreferences também é bastante simplificada. Devemos instanciar um objeto de edição a partir de SharedPreferences.Editor, chamando o método de SharedPreferences edit(), deixando assim o arquivo de preferências pronto para edição e finalmente lançando seus valores com o comando **putString()**, **putBoolean()**, **putFloat()**, **putInt()** ou **putLong()** ou **putStrigSet()**. A gravação (persistência) dos dados é feita a partir dos comandos apply() e commit().

O método **apply()** salva seus dados para a memória imediatamente em arquivo e em uma thread separada, garantindo assim que não haja nenhum bloqueio na thread principal, ou seja, seu aplicativo não irá congelar. Essa técnica foi introduzida a partir da API 9 (Android 2.3) e é a preferida.

O método **commit()** salva o dado diretamente em arquivo, mas o processo é executado dentro da thread principal do aplicativo, forçando a parada de todos os processos de seu aplicativo até que o arquivo esteja salvo. Este método retornará **true** caso o arquivo seja gravado com sucesso e **false** em caso de falha. Utilize o método **commit()** somente se necessitar de confirmação imediata da gravação dos registros em arquivo ou caso esteja desenvolvendo para dispositivos abaixo da API 9.

Exemplo de gravação:

SharedPreferences prefs = getApplicationContext().getSharedPreferences(“minhasprefere ncias”, MODE_PRIVATE);

Editor editor = prefs.edit();

editor.putString(“nome”,”Marcelo”);

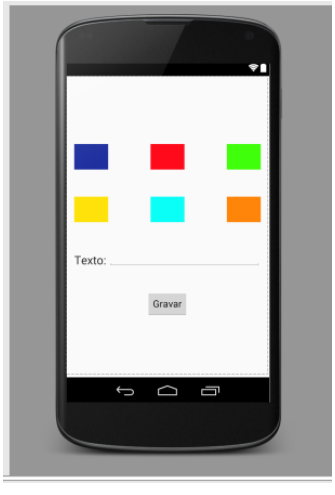
editor.apply();

Vamos agora colocar este conhecimento em prática. Criaremos um aplicativo que armazenará e recuperará as preferências de cor de fundo e texto em uma TextView. Abra o Android Studio, inicie um novo projeto chamado SharedPrefs. Em company domain coloque: aula4.grupouninter.com.br

Nosso dispositivo alvo continua sendo Phone and Tablet, utilizando a API 19. Crie também uma Activity Vazia e dê a ela o nome de MainActivity, com seu layout activity.main.

Uma vez que o projeto tenha sido gerado corretamente, modifique o *layout* de sua MainActivity para que reflita o *layout* da Figura 1.

Figura 1 – Layout Main Activity.



Precisaremos de 6 componentes Button, com sua propriedade Text vazia e seu background representando uma das 6 cores em hexadecimal a seguir:

#FF15279A

#ff0011

#37ff00

#ffe100

#00fff7

#ff8000

Também precisaremos de uma TextView com o valor “Texto”, um componente PlainText (para armazenamento do valor digitado pelo usuário) e um botão para Gravação das preferências do usuário. Lembre-se de, ao criar seus componentes, atribuí-los ids de forma que estes sejam facilmente identificáveis. Para este exemplo, utilizaremos os seguintes ID’s:

Botões de cores:

btnAzul, btnVermelho, btnVerde, btnAmarelo, btnCeleste e btnLaranja.

TextView: txtTexto

Plain Text: edtNome

Botão de gravação: btnGravar

Tente gerar este *layout* sozinho, para se familiarizar com o conceito de RelativeLayout.

De qualquer maneira, acesse o material *on-line* e confira o XML para sua referência.

Em nossa interface, ao clicarmos em cada um dos botões coloridos na tela, o fundo deve ser alterado.

Ao clicarmos no botão “Gravar”, a cor atual de nosso fundo deverá ser gravada em nossas SharedPreferences, bem como o valor digitado em nossa **EditText** (edtNome). Logo, precisamos criar objetos que representem cada um dos componentes que sofrem qualquer tipo de alteração ou interação com o usuário.

Vamos começar pelos botões de cores:

```
Button btnAzul;
Button btnVermelho;
Button btnVerde;
Button btnAmarelo;
Button btnCeleste;
Button btnLaranja;
```

Após sua declaração, devemos inicializá-los vinculando-os ao seu representante no *layout*:

```
btnAzul = (Button) findViewById(R.id.btnAzul);
btnVermelho = (Button) findViewById(R.id.btnVermelho);
btnVerde = (Button) findViewById(R.id.btnVerde);
btnAmarelo = (Button) findViewById(R.id.btnAmarelo);
btnCeleste = (Button) findViewById(R.id.btnCeleste);
btnLaranja = (Button) findViewById(R.id.btnLaranja);
```

Agora, devemos redefinir o Listener de cada botão, para que possamos utilizar o evento desejado por nós:

```
btnAzul.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});

btnVermelho.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});

btnVerde.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});

btnAmarelo.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});

btnCeleste.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});

btnLaranja.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});
```

Para que possamos ter acesso às suas propriedades, no entanto, é necessário que instanciemos e vinculemos um objeto que represente este *layout*.

Antes de mais nada, determine ao RelativeLayout um ID chamado: layoutPrincpal, então crie nosso objeto *layout*:

```
...
Button btnLaranja;
RelativeLayout layout;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
...
```

E finalmente o instancie:

```
...
btnCeleste = (Button) findViewById(R.id.btnCeleste);
btnLaranja = (Button) findViewById(R.id.btnLaranja);
layout = (RelativeLayout) findViewById(R.id.layoutPrincipal);

btnAzul.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
...
```

Pronto! Podemos agora modificar o comportamento de nossos botões para que, ao serem pressionados, modifiquem o *background* de nosso layoutPrincipal, mudando assim a cor de fundo da tela. Vamos, antes de mais nada, criar uma variável que armazenará o valor da cor escolhida:

```
String cor = "#FFFFFF";
```

Para finalmente aplicá-la ao nosso *layout*, faz-se como a seguir:

```
btnAzul.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        cor = "#FF15279A";

layout.setBackgroundColor(Color.parseColor("#FF15279A"));
    }
});

btnVermelho.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View view) {
        cor = "#ff0011";

layout.setBackgroundColor(Color.parseColor("#ff0011"));
    }
});

btnVerde.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        cor = "#37ff00";

layout.setBackgroundColor(Color.parseColor("#37ff00"));
    }
});

btnAmarelo.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        cor = "#ffe100";

layout.setBackgroundColor(Color.parseColor("#ffe100"));
    }
});

btnCeleste.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        cor = "#00fff7";
        layout.setBackgroundColor(Color.parseColor("#00fff7"));
    }
});

btnLaranja.setOnClickListener(new View.OnClickListener() {
    @Override
```

```
    public void onClick(View view) {
       cor = "#ff8000";

layout.setBackgroundColor(Color.parseColor("#ff8000"));
    }
});
```

Se executarmos nosso aplicativo, veremos que agora nossos botões coloridos mudam a cor de fundo. Nosso campo de texto também permite a digitação de valores, mas o botão de gravação não executa nenhuma ação. Vamos declarar então o botão de “Gravar”:

```
Button btnGravar;
```

Instanciá-lo, vinculando-o ao componente correspondente no *layout:*

```
btnGravar = (Button) findViewById(R.id.btnGravar);
```

Finalmente, deve-se redefinir seu Listener OnClickListener:

```
btnGravar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});
```

Para que possamos manipular o componente **EditText** (edtNome), devemos também declará-lo:

```
EditText edtNome;
```

e, finalmente instancia-lo

```
edtNome = (EditText) findViewById(R.id.edtNome);
```

Agora, podemos instanciar o objeto SharedPreferences e permitir sua edição, gravando assim nossas preferências:

```
btnGravar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        SharedPreferences prefs =
getPreferences(MODE_PRIVATE);
        SharedPreferences.Editor editor =  prefs.edit();
        editor.putString("nome", edtNome.getText().toString());
        editor.putString("cor", cor);
        editor.apply();
    }
});
```

Agora que estamos gravando corretamente as preferências do usuário, vamos criar um método que as carregue sempre que o aplicativo for iniciado:

```
public void carregarPrefs(){
    SharedPreferences prefs =
getPreferences(MODE_PRIVATE);
    edtNome = (EditText) findViewById(R.id.edtNome);
    edtNome.setText(prefs.getString("nome",""));
    cor = prefs.getString("cor", "#FFFFFF");
    layout = (RelativeLayout)
findViewById(R.id.layoutPrincipal);
    layout.setBackgroundColor(Color.parseColor(cor));
}
```

Não esqueça de chamar o método carregarPrefs() logo após a criação ao final do método onCreate de nossa Activity:

```
...
            editor.apply();
        }
    });

    carregarPrefs();
}
...
```

Acessando o material *on-line*, você confere o código completo de nossa MainActivity para sua referência.

Para mais informações sobre os valores com SharedPreferences, confira no material *on-line* a videoaula do professor Marcelo.

Tema 2 – Salvando dados utilizando o SQLite

Conforme vimos em nossa primeira aula, a plataforma Android oferece suporte nativo para o SQLite. O SQLite é uma pequena biblioteca, implementada em C que oferece acesso a uma base de dados racional SQL e suportando até 2 TB de dados. Seu armazenamento é feito em um arquivo de texto dentro do dispositivo, e não existe a necessidade de se estabelecer qualquer tipo de conexão JDBC ou ODBC por exemplo.

Duas classes são utilizadas para a criação do banco de dados:

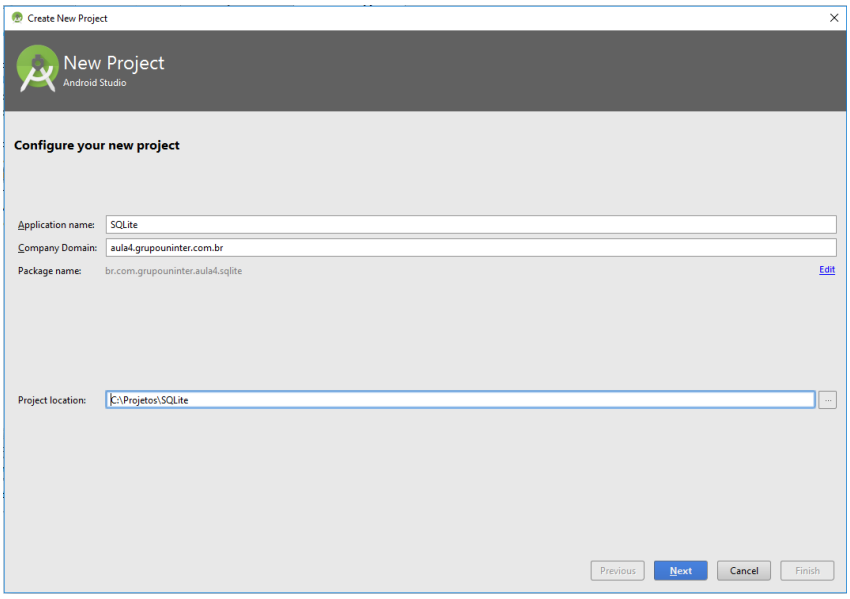
SQLiteDataBase – contém os métodos de manipulação de dados;

SQLiteOpenHelper – responsável pela criação e versionamento do banco.

Para ilustrar a utilização do SQLite para persistência de dados, vamos partir do suposto que desejamos criar uma aplicação que armazene dados de nossos amigos, que permita busca entre estes dados, sua recuperação e atualização.

Crie um novo projeto chamado SQLite, e em company Domain digite: aula4.grupouninter.com.br:

Figura 2 – Projeto SQLite



Deixe como dispositivos alvo Phone and Tablet e o SDK mínimo como API 19. Crie seu projeto com uma Activity vazia, deixando como nome da Activity **MainActivity** e como nome do *layout* **activity_main**.

Antes de darmos início ao desenvolvimento, devemos definir o escopo do projeto: para este exemplo armazenaremos o nome, telefone e *e-mail* de nossos amigos em um banco de dados SQLite.

Antes que possamos começar a gravar os dados no banco de dados, devemos criar a classe que nos auxiliará na criação e manutenção de nosso banco de dados e tabelas. Crie uma nova classe chamada DBHelper, clicando com o botão direito no pacote br.com.grupouninter.aula4.sqlite e em **New** -> **Java Class.**

```
package br.com.grupouninter.aula4.sqlite;

/**
 * Created by Marcelo on 07/08/2016.
 */
public class DBHelper {
}
```

Agora, devemos herdar o comportamento da classe SQLiteOpenHelper, garantindo assim que o comportamento padrão do SQLite seja repassado à nossa classe DBHelper. Ao estendermos a classe SQLiteOpenHelper somos solicitados a declarar os métodos padrão **onCreate()** e **onUpgrade().**

O método **onCreate** é chamado pela primeira vez quando a criação da tabela é necessária. Após a criação das tabelas, este método não mais será executado.

O método **onUpgrade** será chamado quando a versão do banco de dados é atualizada. Imagine que na primeira vez que o aplicativo foi executado a versão do banco de dados era 1, e durante uma atualização do mesmo a estrutura do banco de dados sofreu alteração. Ao mudarmos a versão do banco de dados para 2, o método **onUpgrade** é chamado e nos permite tomar as ações necessárias.

```
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {

}

@Override
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i,
int i1) {

}
```

Também somos solicitados a declarar o construtor de nossa classe. O construtor padrão do SQLiteOpenHelper solicita por argumentos: Context, nome do banco de dados, objeto responsável pela seleção de um cursor apropriado e versão do banco de dados.

Então, antes de mais nada, vamos criar as variáveis estáticas responsáveis por armazenar o nome de nosso banco de dados e sua versão:

```
private static final String NOME_BANCO_DADOS =
"CadastroAmigos";
private static final int VERSAO_BANCO_DADOS = 1;
```

Agora, podemos criar o construtor para nossa classe DBHelper, passando à classe matriz (ou mãe) as varáveis que acabamos de criar:

```
public DBHelper(Context context) {
   super(context, NOME_BANCO_DADOS, null,
VERSAO_BANCO_DADOS);
}
```

Perceba que onde deveríamos passar o objeto responsável pela seleção de um cursor apropriado estamos passando um valor nulo. Discutiremos isso mais à frente. Nossa estrutura de armazenamento será:

Nome da tabela: amigos

Campo: id
Tipo : INTEGER (auto incremento)

Campo: nome
Tipo: TEXT

Campo: telefone
Tipo: TEXT

Campo: *e-mail*
Tipo: TEXT

```
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {
    // comando sql responsável pela criação da estrutura da tabela
amigos
    String tabela_amigos =  "CREATE TABLE amigos ("
            + "id INTEGER PRIMARY KEY autoincrement,"
            + "nome TEXT,"
            + "telefone TEXT,"
            + "email TEXT"
            + ")";
    // executando o sql de tabela_amigos
    sqLiteDatabase.execSQL(tabela_amigos);
}
```

No método onUpgrade, definiríamos o que deve ser executado caso a versão de nossa base de dados sofra alteração. Por exemplo, podemos acrescentar novos campos à tabela, efetuar um *backup* e assim por diante. Para efeito deste exemplo, manteremos este método sem alterações (esta é a primeira versão de nossa tabela).

Agora que temos a estrutura de nossa tabela de Amigos definida, devemos criar um POJO que representará esta estrutura em nosso sistema.

Crie uma nova classe chamada “Amigo” e defina nela as propriedades de acordo com a estrutura de nossa tabela:

```
package br.com.grupouninter.aula4.sqlite;

/**
 * Created by Marcelo on 07/08/2016.
 */
public class Amigo {

    private int id;
    private String nome;
    private String telefone;
    private String email;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
```

```
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Iremos agora definir em nossa classe DBHelper o método que garante a inserção de um registro. Como nossa tabela solicita os dados de um amigo (id, nome, telefone e *e-mail*), passaremos nosso POJO como argumento ao método de inserção e utilizaremos seus getters para acessar seus valores:

```
public void inserir(Amigo amigo){
    // abrimos a conexão com o arquivo que armazena o banco de dados
    SQLiteDatabase db = this.getWritableDatabase();
    // criamos um statement que recebera a SQL e seus valores a serem passados ao banco
    SQLiteStatement stmt = db.compileStatement("INSERT INTO amigos (nome, telefone, email) " +
            "VALUES (?,?,?)");
    // passamos o valor de nome de amigo ao primeiro argumento de nossa statement (?)
    stmt.bindString(1, amigo.getNome());
    // passamos o valor de telefone ao segundo argumento de nossa statement (?)
```

```
    stmt.bindString(2, amigo.getTelefone());
    // passamos o valor de email ao terceiro argumento de
nossa statement (?)
    stmt.bindString(3, amigo.getEmail());
    // executamos o statement
    stmt.execute();
    // fechamos o objeto Statement
    stmt.close();
    // fechamos o arquivo de banco de dados
    db.close();
}
```

Agora, devemos implementar o método que atualizará os dados de nosso amigo. Novamente passaremos como argumento um POJO com os dados de nosso amigo:

```
public void atualizar(Amigo amigo){
    // abrimos a conexão com o arquivo que armazena o banco
de dados
    SQLiteDatabase db = this.getWritableDatabase();
    // criamos um statement que recebera a SQL e seus valores
a serem passados ao banco
    SQLiteStatement stmt = db.compileStatement("UPDATE
amigos SET nome=?, telefone=?, email=? "+
            "WHERE id = ?");
    // passamos o valor de nome de amigo ao primeiro
argumento de nossa statement (?)
    stmt.bindString(1, amigo.getNome());
    // passamos o valor de telefone ao segundo argumento de
nossa statement (?)
    stmt.bindString(2, amigo.getTelefone());
    // passamos o valor de email ao terceiro argumento de
nossa statement (?)
    stmt.bindString(3, amigo.getEmail());
    // passamos o valor de id ao quarto argumento de nossa
statement (?)
    stmt.bindLong(4, amigo.getId());
    // executamos o statement
    stmt.execute();
    // fechamos o objeto Statement
    stmt.close();
    // fechamos o arquivo de banco de dados
    db.close();
}
```

Agora, devemos criar o método que excluirá um determinado amigo ao receber um ID por argumento:

```
public void excluir(int id){
    // abrimos a conexão com o arquivo que armazena o banco de dados
    SQLiteDatabase db = this.getWritableDatabase();
    // criamos um statement que recebera a SQL e seus valores a serem passados ao banco
    SQLiteStatement stmt = db.compileStatement("DELETE FROM amigos WHERE id = ?");
    // passamos o valor de id de amigo ao primeiro argumento de nossa statement (?)
    stmt.bindLong(1, id);
    // executamos o statement
    stmt.execute();
    // fechamos o objeto Statement
    stmt.close();
    // fechamos o arquivo de banco de dados
    db.close();
}
```

Nosso próximo passo será retornar os dados de um determinado amigo ao receber seu ID por argumento. Neste caso, receberemos o ID do amigo e retornaremos o objeto POJO devidamente preenchido:

```
  public Amigo retornarAmigo(int id){
    // abrimos a conexão com o arquivo que armazena o banco de dados
    SQLiteDatabase db = this.getWritableDatabase();
    // comando sql que retornará o id, nome, telefone e email de um amigo com base no id passado
    String query = "SELECT id, nome, telefone, email FROM amigos WHERE id = ?";
    // definimos um cursor - objeto que receberá os registros que atendam os requisitos da query criada
    Cursor cursor = db.rawQuery(query, new String[] {String.valueOf(id)});
    // movemos o cursor para o primeiro registro encontrado
    cursor.moveToFirst();
    // instanciamos o objeto amigo
    Amigo amigo = new Amigo();
    // definimos suas propriedades com base nas colunas existentes no cursor
    amigo.setId(cursor.getInt(0));
    amigo.setNome(cursor.getString(1));
    amigo.setTelefone(cursor.getString(2));
```

```
    amigo.setEmail(cursor.getString(3));
    // fechamos o arquivo do banco de dados
    db.close();
    // retornamos o pojo devidamente preenchido
    return amigo;
}
```

Para sua referência, acesse o material *on-line* e veja o código completo de nosso arquivo DBHelper.java.

Agora que finalizamos a implementação do DBHelper, vamos criar a interface que permitirá ao usuário o cadastramento de um amigo. Como você deve imaginar, criaremos uma nova Activity que conterá campos para nome, telefone e e-mail, além de um botão para gravar e outro para cancelar.

Crie então uma Activity em branco chamada AmigoActivity, com seu *layout name* chamado activity_amigo e clique em **Finish.**

No arquivo de *layout* desta activity, arraste o componente Small Text para mostrar o ID do amigo, o componente Plain Text para armazenar o nome, telefone e e-mail do amigo e dois componentes Buttons, para as ações de Gravar e Cancelar. Defina as seguintes propriedades:

Componente Small Text:
nome: txtID
text: #

Componente Plain Text:
nome: edtNome
hint: Nome

Componente Plain Text:
nome: edtTelefone
hint: Telefone

Componente Plain Text:
nome: edtEmail
hint: E-mail

Componente Button:
nome: btnCancelar
text: Cancelar

Componente Button:
Nome: btnGravar
Text: Gravar

Tente criar este *layout* seguindo as recomendações acima. Isso aumentará sua intimidade com o desenvolvimento de *layouts* relativos. Para sua referência, acesse o material *on-line* e veja o arquivo XML que representa o *layout* do arquivo activity_amigo.xml.

Com nosso *layout* definido, retorne à classe AmigoActivity.java e instancie os objetos necessários para o funcionamento deste *layout*:

```
TextView txtID;
EditText edtNome;
EditText edtTelefone;
EditText edtEmail;
Button btnCancelar;
Button btnGravar;
```

Na criação de nosso componente, instancie os componentes Button referentes às ações de Cancelar e Gravar:

```
btnCancelar = (Button) findViewById(R.id.btnCancelar);
btnGravar = (Button) findViewById(R.id.btnGravar);
```

Agora, vamos redefinir o Listener OnClickListener dos dois botões, para que possamos definir seu comportamento:

```
btnCancelar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});

btnGravar.setOnClickListener(new View.OnClickListener() {
    @Override
```

```
    public void onClick(View view) {

    }
});
```

Em nosso botão de Cancelar, tudo que desejamos é encerrar esta activity. Para tanto, basta que executemos o método **finish()**. Para o botão Gravar, desejamos que o conteúdo de nossos campos nome, telefone e *e-mail* seja armazenado em um POJO da classe Amigo, e então enviado à classe DBHelper para gravação.

Antes de mais nada, devemos instanciar os objetos que armazenam o conteúdo digitado por nosso usuário, para que possamos fazer uso de seus valores:

```
edtNome = (EditText) findViewById(R.id.edtNome);
edtTelefone = (EditText) findViewById(R.id.edtTelefone);
edtEmail = (EditText) findViewById(R.id.edtEmail);
```

Após isso, passamos ao objeto amigo os valores coletados nos componentes de edição de texto:

```
Amigo amigo = new Amigo();
amigo.setNome(edtNome.getText().toString());
amigo.setTelefone(edtTelefone.getText().toString());
amigo.setEmail(edtEmail.getText().toString());
```

Finalmente, enviamos o conteúdo de amigo para nossa classe DBHelper, para que seja persistido:

```
DBHelper db = new DBHelper(view.getContext());
db.inserir(amigo);
```

Para sua referência, acesse o material *on-line* e confira o código completo (até o momento) da classe AmigoActivity.java.

Já temos então uma Activity que nos permite a inclusão de um novo amigo, só nos resta chamá-la a partir de nossa Main Activity. Para isso, no *layout* activity_main.xml, remova o componente TextVew com o texto “Hello World!” e adicione um componente Button com as seguintes propriedades:

id: btnAmigo
text: Inserir Amigo

No arquivo MainActivity.java, declare o objeto que faz referência a este componente:

```
Button btnAmigo;
```

Depois, instancie-o:

```
btnAmigo = (Button) findViewById(R.id.btnAmigo);
```

Finalmente, vamos redefinir seu Listener, chamando a ActivityAmigo:

```
btnAmigo.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(view.getContext(),
AmigoActivity.class);
        startActivity(intent);
    }
});
```

Execute o aplicativo. O que aconteceu? Seu cadastro foi gravado corretamente?

Para mais informações sobre o salvamento dos dados utilizando o SQLite, confira no material *on-line* a videoaula do professor Marcelo.

Tema 3 - Adapters

Agora que nosso aplicativo está incluindo registros, necessitamos comunicar ao usuário a respeito da presença deles, criando um uma interface que liste os registros inseridos em nosso banco de dados. No entanto, antes de apresentar ao usuário estes registros, devemos criar uma classe que se responsabilizará pelo acesso a estes registros, e também pela forma como estes devem ser renderizados na interface com o usuário.

Estamos falando de adaptadores. Um adaptador é um objeto que cria uma ponte entre uma AdapterView e os dados para aquela View. O adaptador provê acesso aos dados e também é responsável por criar uma View para cada registro presente no conjunto de dados.

Para que possamos entender um pouco mais sobre como um adaptador funciona, considere o seguinte: em nossa classe DBHelper, criaremos um método que retornará todos os registros encontrados no banco de dados. Como esses registros serão retornados deste método? Seu retorno se dará através de uma ArrayList (lista de objetos) do tipo Amigo.

Ou seja, executaremos uma query que retorne todos os registros de nossa tabela, iteraremos por esses registros e, a cada iteração, criaremos um novo objeto do tipo Amigo, preenchendo-o e então adicionando-o à ArrayList de retorno.

Mas como será feita a manipulação destes dados? Qual registro será representado por qual componente em uma interface?

É neste momento que o adaptador entra em ação e renderiza corretamente cada registro, enviando-o ao componente correto e também gerando uma nova view para cada nova linha (registro) encontrado.

Nosso adaptador será desenvolvido a partir da classe Base Adapter. Vamos criar então uma nova classe em nosso projeto, chamada de AmigoAdapter, estendendo-a a partir de BaseAdapter.

Ao estendermos nossa classe a partir de BaseAdapter, somos solicitados a implementar os métodos getCount(), getItem(), getItemId e getView().

```
package br.com.grupouninter.aula4.sqlite;

import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;

/**
 * Created by Marcelo on 07/08/2016.
 */
public class AmigoAdapter extends BaseAdapter {
    @Override
    public int getCount() {
        return 0;
    }

    @Override
    public Object getItem(int i) {
        return null;
    }

    @Override
    public long getItemId(int i) {
        return 0;
    }

    @Override
    public View getView(int i, View view, ViewGroup
viewGroup) {
        return null;
    }
}
```

O primeiro passo na criação de um adaptador é definir qual objeto armazenará os dados que serão enviados a ele. Vamos então definir uma List de Amigos:

```
private List<Amigo> amigos;
```

Devemos também armazenar o Contexto no qual este adaptador está sendo renderizado (ou seja, o componente responsável por sua renderização):

```
Context context = null;
```

E o objeto responsável por "inflar" o *layout* (ou seja, renderizá-lo):

```
private LayoutInflater inflater;
```

Podemos agora criar o construtor para nossa classe, que inicializará os objetos declarados:

```
public AmigoAdapter(Context context, List<Amigo> amigos){
    this.amigos = amigos;
    this.inflater = LayoutInflater.from(context);
    this.context = context;
}
```

Para implementarmos o método getCount(), devemos retornar a quantidade de registros encontrados em nossa List de amigos. Para isto, basta utilizarmos o método size():

```
@Override
public int getCount() {
    return this.amigos.size();
}
```

O método getItem é responsável por retornar o objeto do tipo Amigo que se encontra em uma determinada posição de nossa List.

```
@Override
public Object getItem(int i) {
   return this.amigos.get(i);
}
```

O método getItemID é responsável por retornar o id do objeto do tipo Amigo em determinada posição:

```
@Override
public long getItemId(int i) {
   return i
}
```

Finalmente chegamos ao método responsável por renderizar os valores encontrados em um *layout* definido. Antes de mais nada, devemos criar este *layout* a ser preenchido.

Em sua janela de projeto, expanda a pasta **res** e clique com o botão direito em ***layout*****,** selecionando a opção **New** -> **Layout resource file.**

Chame este novo arquivo de amigo_layout e pressione OK.

Neste novo *layout,* necessitaremos de componentes que serão utilizados para gerar um registro em uma listagem de dados. Cada registro deverá mostrar o código do amigo, seu nome, telefone e *e-mail*.

Adicione um componente Small Text com as seguintes propriedades:

name: txtIDAmigo
text : #

Adicione um componente Large Text com as seguintes propriedades:

name:txtNomeAmigo
text: Nome

Adicione um componente Medium Text com as seguintes propriedades:

name: txtTelefoneAmigo
text: Telefone

Adicione um componente Medium Text com as seguintes propriedades:

name: txtEmailAmigo
text: E-mail

Para sua referência, acesse o material *on-line* e veja o XML completo do arquivo amigo_layout.txt.

Agora que temos o *layout* que representa um registro em uma listagem de registros, podemos implementar o método **getView()**, que se encarregará de renderizar registro a registro os valores encontrados em nossa List, atribuindo os valores de cada propriedade a seu componente correto dentro de um *layout*.

```
@Override
public View getView(int i, View view, ViewGroup viewGroup) {
    // Atribuímos ao objeto Amigo o registro localizado em nosso
método getItem(pos)
    final Amigo amigo  = (Amigo) getItem(i);

    // caso nosso objeto View não possua um layout atribuído a
ele, instanciamos o mesmo, passando como
    // argumento o layout que utilizaremos para representar um
registro
    if (view == null) {
        view = inflater.inflate(R.layout.amigo_layout, null);
    }
    // instanciamos os objetos que corresponderão aos
componentes em nosso layout
    TextView idAmigo = (TextView)
view.findViewById(R.id.txtIDAmigo);
    TextView nomeAmigo = (TextView)
view.findViewById(R.id.txtNomeAmigo);
    TextView telefoneAmigo = (TextView)
view.findViewById(R.id.txtTelefoneAmigo);
    TextView emailAmigo = (TextView)
view.findViewById(R.id.txtEmailAmigo);

    // iniciamos estes objetos

    idAmigo.setText(amigo.getId());
    nomeAmigo.setText(amigo.getNome());
    telefoneAmigo.setText(amigo.getTelefone());

    return view;

}
```

Em nossa classe DBHelper, necessitamos agora desenvolver um método que liste todos os registros encontrados no banco de dados e os agrupe em uma lista de objetos do tipo Amigo. Abra seu arquivo DBHelper.java e implemente o método listar():

```
    public ArrayList<Amigo> listar(){
        // criamos uma List que armazenará objetos do tipo Amigo
        ArrayList amigos = new ArrayList<Amigo>();
        // abrimos a conexão com o arquivo que armazena o banco
    de dados
        SQLiteDatabase db = this.getWritableDatabase();
        // Buscamos todos os registros disponiveis no banco de
    dados ordenados por nome
```

```
    String query = "Select id, nome, telefone, email FROM
amigos ORDER BY nome";
    Cursor cursor = db.rawQuery(query, null);
    // iteramos por todos os registros localizados com base em
nossa query
    while (cursor.moveToNext()) {
        // instanciamos um novo objeto amigo
        Amigo amigo = new Amigo();
        // preenchemos este objeto com os valores armazenados
no cursor
        amigo.setId(cursor.getInt(0));
        amigo.setNome(cursor.getString(1));
        amigo.setTelefone(cursor.getString(2));
        amigo.setEmail(cursor.getString(3));
        // adicionamos o objeto preenchido à nossa List de
amigos
        amigos.add(amigo);
    }
    // retornamos o objeto Amigos ao método que o chamou
    return amigos;
}
```

Para sua referência, acesse o material *on-line* e veja o código completo de nosso arquivo DBHelper.java.

Até o momento, criamos nosso arquivo DBHelper, que nos auxilia na manipulação do banco de dados (incluir registros, excluir registros, alterar um registro e listar todos os registros disponíveis), além de implementarmos nosso Adapter (adaptador) que servirá de ponte entre nosso banco de dados e a interface que desejamos apresentar ao usuário.

O que nos resta agora é modificar o *layout* de nossa MainActivity, adicionando um componente de ListView para que nossos registros possam ser visualizados pelo usuário. Em seu arquivo activity_main.xml, arraste de sua janela de componentes uma ListView e modifique sua propriedade name para listAmigos.

Para sua referência, acesse o material on-line e veja o arquivo xml completo do arquivo activity_main.xml.

Agora devemos criar o objeto que representa esta ListView:

```
ListView listAmigos;
```

Também necessitaremos do objeto AmigoAdapter:

```
AmigoAdapter amigoAdapter;
```

Finalmente, necessitaremos de nosso objeto DBHelper, que nos auxiliará na manipulação do banco de dados:

```
DBHelper dbHelper;
```

Vamos agora inicializar a ListView e também nosso DBHelper:

```
listAmigos = (ListView) findViewById(R.id.listAmigos);
dbHelper = new DBHelper(this);
```

Finalmente, vamos criar um método que instancie corretamente nossa ListView, preenchendo-a com os valores oriundos de nosso adaptador:

```
public void listaAmigos(){

    ArrayList<Amigo> amigos = dbHelper.listar();
    amigoAdapter = new AmigoAdapter(this,amigos);
    listAmigos.setAdapter(amigoAdapter);

}
```

Lembre-se de adicionar a chamada a este método na última linha de seu método onCreate em sua MainActivity:

```
listaAmigos();
```

Ao executarmos o projeto agora, os registros que foram previamente inseridos por você estão visíveis na ListView. Experimente adicionar um novo registro.

O que acontece quando você fechar a Activity de cadastro? Sua ListView não foi atualizada com o novo registro.

Isso ocorre porque não houve nenhuma notificação de que o conteúdo do adaptador sofreu alteração.

Para resolver esta situação, vamos aproveitar o ciclo de vida de sua Activity. Sempre que uma Activity volta a primeiro plano, o último método executado por ela é o onResume, lembra-se?

Redefina então o onResume() para que este execute também a chamada ao método listaAmigos(). Desta maneira, sempre que a MainActivity for iniciada, o conteúdo da ListView será atualizado.

Como este método ocorre todas as vezes que uma Activity inicia, não existe a necessidade de chamar o método listaAmigos() ao final do **onCreate(),** portanto remova-o.

Execute o aplicativo. Mesmo após removermos o método **listaAmigos()** do **onCreate()**, sua ListView continua sendo corretamente populada. Experimente agora inserir um novo registro. O que acontece quando você clica no botão gravar?

Para sua referência, acesse o material *on-line* e veja o código completo de nosso arquivo MainActivity.java.

Para mais informações sobre os adapters, confira no material *on-line* a videoaula do professor Marcelo.

Na prática

- Altere o aplicativo de SharedPreferences para que este tenha a opção de voltar a cor de fundo e o campo de texto aos seus valores padrão, ou seja: branco e texto vazio.
- Altere o aplicativo de SharedPreferences para que este grave suas preferências de forma acessível a todas as Activities do aplicativo. Crie então uma nova Activity que receba o valor armazenado na propriedade “nome” e permita sua alteração e gravação. Esta gravação (caso ocorra) deverá refletir seu resultado imediatamente na Activity Principal. Dica: como podemos fazer para retornar valores entre Activities?
- Altere o aplicativo SQLite para que seja possível alterar os dados de um amigo.
- Altere o aplicativo SQLite para que seja possível excluir os dados de um amigo.
- Altere o aplicativo SQLite para que seja possível efetuar uma filtragem na ListView que contém os amigos.

Síntese

Nesta aula, foram apresentados os conceitos de persistência de dados em dispositivos Android. Aprendemos a gravação de parâmetros simples, utilizando o objeto SharedPreferences, sua recuperação e também sua utilização na redefinição de uma Interface.

Aprendemos também a respeito do banco de dados nativo do Android – SQLite, analisando suas principais características, seus objetos de acesso e sua configuração. Criamos os objetos de gestão do banco de dados. Conhecemos o objeto Adaptador, que é utilizado como forma de ligação entre a estrutura de dados e o componente de visualização.

Utilizamos também o conhecimento previamente adquirido para desenvolver uma interface que permitiu ao nosso usuário inserir novos registros e visualizar seu conteúdo.

Confira no material *on-line* o vídeo de síntese do professor Marcelo.

Referências

ADAPTER. Disponível em: <https://developer.android.com/reference/android/widget/Adapter.html>. Acesso em: 25 ago. 2016.

DEITEL, Paul; DEITEL: Harvey; WALD, Alexander. **Android 6 para Programadores:** Uma Abordagem Baseada em aplicativos. Porto Alegre: Bookman, 2016.