Programação para Android

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Application Sandbox

- Mecanismo de segurança
- Cada aplicação roda em um processo separado, com um ID único de usuário
- Nenhuma aplicação, por padrão, tem permissão de executar qualquer operação que impacte o usuário, outras aplicações, ou o SO

APPLICATIONS

Home	Dialer	SMS/MMS	IM	Browser	Camera	Alarm	Calculator
Contacts	Voice Dial	Email	Calendar	Media Player	Photo Album	Clock	...

APPLICATION FRAMEWORK

Activity Manager	Window Manager	Content Providers	View System	Notification Manager
Package Manager	Telephony Manager	Resource Manager	Location Manager	...

LIBRARIES

Surface Manager	Media Framework	SQLite	WebKit	Libc
OpenGL|ES	Audio Manager	FreeType	SSL	...

ANDROID RUNTIME

Core Libraries
Dalvik Virtual Machine

HARDWARE ABSTRACTION LAYER

Graphics	Audio	Camera	Bluetooth	GPS	Radio (RIL)	WiFi	...

LINUX KERNEL

Display Driver	Camera Driver	Bluetooth Driver	Shared Memory Driver	Binder (IPC) Driver
USB Driver	Keypad Driver	WiFi Driver	Audio Drivers	Power Management

De que forma podemos compartilhar recursos e dados, já que aplicações estão ‘isoladas’?

Permissões

- Por conta da Sandbox, aplicações que desejam compartilhar recursos e dados devem definir de forma explícita este compartilhamento
- Usado para limitar/conceder acesso a:
  - Informações do usuário – ex.: contatos
  - APIs sensíveis a custo – ex.: SMS/MMS
  - Recursos do sistema – ex.: Câmera

Aplicações simples, como as vistas até agora, não tem permissões, ou seja… ?

Permissões: Representação

- Strings definidas no AndroidManifest.xml
- Diferentes elementos que permitem definir
  - permissões necessárias para rodar a aplicação
  - permissões que a aplicação define para outras aplicações ou componentes

Usando Permissões

- Aplicações definem as permissões necessárias para sua utilização por meio da tag:
  `<uses-permission>`

- Usuários tem de aceitar permissões para instalar a aplicação

  - isto foi alterado a partir de Android 6.0+

Exemplo

```
<uses-permission
 android:name="android.permission.CALL_PHONE" />

android.permission.CAMERA
android.permission.INTERNET
android.permission.ACCESS_FINE_LOCATION
…
```

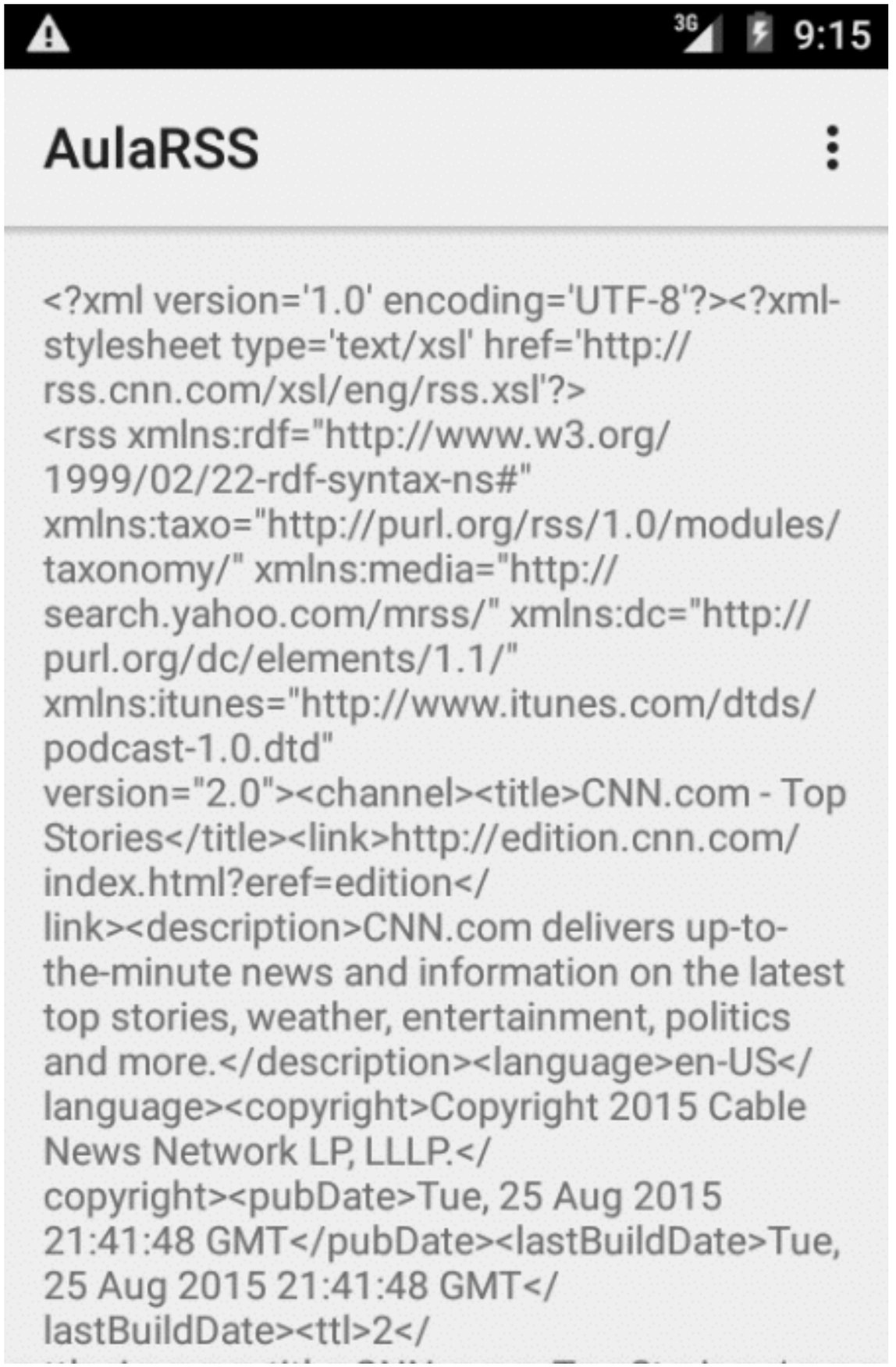
Incluindo a permissão...

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.ufpe.cin.aularss" >

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="AulaRSS"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="AulaRSS" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER"
            </intent-filter>
```

Finalmente…

Começamos a ter progresso…

Mas, ainda não é muito útil!



Quando o usuário é informado sobre permissões?

Formas de Instalação

- Via SDK Tools
- Play Store, Android 5.1-
- Play Store, Android 6.0+
- Outros meios, Android 5.1-
- Outros meios, Android 6.0+

Falhas em Permissões

- Em geral, falhas resultam no lançamento de uma **`SecurityException`**

- Isto não é garantido de ocorrer sempre, por exemplo, nos casos de **`sendBroadcast(…)`**, em que a checagem é feita após o retorno da chamada do método

- De qualquer forma, uma falha é registrada no log do sistema

Em alguns casos

- Com o tempo, uma permissão pode deixar de ser necessária, a partir de uma certa versão da API

- Analogamente, uma permissão pode passar a ser necessária a partir de uma certa versão da API

- Android resolve isto por meio dos atributos **`targetSdkVersion`** e **`android:maxSdkVersion`**

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

Atenção

- Certas permissões podem implicar na exigência de uma determinada funcionalidade

- Por exemplo, requisitar acesso a Bluetooth, faz com que o Google Play só mostre sua aplicação para dispositivos que tem esta capacidade

- Isto pode ser 'desativado', definindo como como **`false`** o atributo **`android:required`** do elemento **`<uses-feature>`**

```
<uses-feature android:name="android.hardware.camera"
              android:required="false" />
```

Checando permissões

- No momento de uma chamada ao sistema, prevenindo a aplicação de executar certas funções

- Ao iniciar uma Activity, para prevenir a aplicação de iniciar Activities de outras aplicações

- Ao enviar e receber broadcasts, para controlar quem pode receber seu broadcast ou quem pode enviar broadcasts para você

- Ao acessar e operar um content provider

- Iniciando ou se ligando a um services

Android 6.0+

- A partir de Android 6.0+, certas permissões (consideradas *dangerous*) não apenas precisam ser declaradas no Manifest
  - mas ainda precisam estar lá!

- O usuário será perguntado em tempo de **execução** se deseja conceder permissões

- Por isso não se apresentam mais estas permissões em tempo de **instalação**

Permissões afetadas

Grupo	Permissão
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	GET_ACCOUNTS, READ_CONTACTS, WRITE_CONTACTS
LOCATION	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	ADD_VOICEMAIL, CALL_PHONE, PROCESS_OUTGOING_CALLS, READ_CALL_LOG, READ_PHONE_STATE, USE_SIP, WRITE_CALL_LOG
SENSORS	BODY_SENSORS
SMS	READ_CELL_BROADCASTS, READ_SMS, RECEIVE_SMS, RECEIVE_MMS, RECEIVE_WAP_PUSH, SEND_SMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

Como saber se permissão foi concedida?

- Uso do método **`checkSelfPermission()`**
- Retorna **`PERMISSION_GRANTED`** ou **`PERMISSION_DENIED`**
- Usuário pode remover permissões nas configurações
  - isto significa que devemos sempre checar!

Dica!

```
private boolean hasPermission (String p) {
   return (
      PackageManager.PERMISSION_GRANTED
      ==
      checkSelfPermission(p)
   );
}
```

Disponibilidade…

- O método **`checkSelfPermission()`** está disponível apenas a partir da API 23

- Pode envolver a chamada em um teste da API
  - **`if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {…}`**

- Ou usar **`ContextCompat`** da biblioteca **`support-v4`**
  - **`ContextCompat.checkSelfPermission(…)`**

Como eu sei que o usuário removeu a permissão?

Sempre que possível,
então, chame
`checkSelfPermission()`

Podemos pedir permissão?

Podemos pedir permissão ao usuário?

- Uso do método **`requestPermissions()`**

  - recebe um array de Strings com as permissões solicitadas e um inteiro para identificar requisição

  - Por ex.: Podemos checar se temos determinada permissão ao criar a **`Activity`** (**`onCreate`**) e, caso não tenhamos, solicitar ao usuário

  - Após o usuário passar pelos dialogs, o método **`onRequestPermissionsResult()`** será chamado, retornando, para cada permissão solicitada, se a mesma foi concedida ou negada

Quando devo solicitar permissão ao usuário?

https://github.com/apl-devs/AppIntro

Quando **não** devo solicitar permissão ao usuário?

`requestPermissions()` é alheio a mudanças de configuração…

O que fazer se o usuário diz não?

O que fazer se o usuário diz não, e por favor, pare de pedir?

O que acontece em dispositivos antigos?

O que acontece quando configuro meu app com **`targetSdkVersion < 23`**?

https://newcircle.com/s/post/1737/2015/05/31/the-new-android-m-permissions-model

texto antigo…

O que acontece ao usuário apagar todos os dados via Settings?

`checkSelfPermission()`

- Sempre retorna **`PERMISSION_GRANTED`** se
  - app tem **`targetSdkVersion`** menor que 23
  - app roda em dispositivo mais antigo que Android 6.0

- Só retorna **`PERMISSION_DENIED`** se está usando o novo *runtime permission system*, em um dispositivo Android 6.0+, e o usuário nunca concedeu permissão ou revogou via Settings…

Dados, arquivos, assets

De que forma podemos empacotar arquivos em nossos aplicativos?

Raw Resources

- Basicamente colocar o arquivo em **`res/raw`**
- Podemos obter o arquivo por meio de chamadas a **`getResources()`**
- Usamos **`openRawResource(R.raw…)`** passando o ID do recurso, de forma a obter um objeto **`InputStream`** para ler o arquivo
- Não é possível modificar este arquivo

XML Resources

- Se o seu arquivo é um XML, melhor colocar no diretório **`res/xml`** - assume que arquivos são XML
- Acesso novamente por meio de **`getResources()`**
- Podemos então usar **`getXml(R.xml…)`** para obter um **`XmlResourceParser`** para o XML
- Também não é possível modificar este arquivo

Assets

- Empacotar dados na forma de um asset
- Criar um diretório **`assets`** no sourceset e colocar arquivos arbitrários por lá
- Acesso por meio de **`getAssets()`**
- Podemos então usar **`open(path…)`**
- Mais uma vez, não é possível modificar o arquivo

Armazenando e Manipulando Arquivos

Manipulando Arquivos

- Por meio das classes normais de I/O de Java
- Por exemplo, classe **`File`**
  - representa uma entidade do sistema de arquivos identificada pelo caminho
- A distinção em Android é **onde** lemos e escrevemos
  - não temos permissão de leitura e escrita onde bem desejarmos
  - há apenas um punhado de diretórios acessíveis

Memória Interna vs. Externa

- Internal Storage
  - dados privados, memória flash interna ao dispositivo, sempre disponível
- External Storage
  - memória pública e compartilhada, pode ser removível (cartão de memória)

Memória Interna vs. Externa

- Internal Storage
  - funciona *per-application*, arquivos escritos não podem ser lidos por outras aplicações
  - usuários com *root access* podem passar por cima
- External Storage
  - visível a todas as aplicações e ao usuário
  - qualquer entidade pode ler algo armazenado na memória externa e pode escrever/deletar…

Arquivos - Memória Interna

- **`FileOutputStream`**
  - **`openFileOutput(String name, int mode)`**
  - abre arquivo para escrita, criando o mesmo se este ainda não existir
- **`FileInputStream`**
  - **`openFileInput(String name, int mode)`**
  - abre arquivo para leitura
- **`getFilesDir()` | `getCacheDir()`**
  - retorna um objeto **`File`** apontando para a raiz dos diretórios na memória interna

Usando Memória Externa

- Originalmente (Android 1.x e 2.x), memória externa era presente apenas na forma de cartões SD
  - ainda em 2.x passou a ser partição separada na memória flash

- A partir de Android 3.0+ passou a ser simplesmente um diretório especial na mesma partição onde está contida memória interna

Onde escrever?

- Se a aplicação gerencia arquivos que não serão disponibilizados para outros apps, podemos utilizar um diretório privado na memória externa, chamando **`getExternalFilesDir()`**

- Este diretório não deve ser utilizado para mídia que pertence ao usuário, como fotos e músicas, pois é removido na desinstalação

- A partir de Android 4.4, podemos utilizar **`getExternalFilesDirs()`** para obter diretórios privados tanto da memória interna quanto da externa

Arquivos Cache

- É possível utilizar diretórios de cache caso sua aplicação precise criar arquivos temporários
- Tais arquivos podem ser deletados quando há pouco espaço de armazenamento e são removidos na desinstalação
- Acessamos o diretório cache, utilizando os métodos:
  - **`getCacheDir()`**
    - retorna o caminho absoluto do diretório
  - **`getExternalCacheDir()`**
    - retorna um objeto File representando o diretório externo

Arquivos e Memória Externa

- Se há arquivos que pertencem mais ao usuário do que ao aplicativo (fotos, músicas…) o ideal é usar **`getExternalStoragePublicDirectory(type)`**

- Tipos de diretórios
  - `DIRECTORY_MOVIES`
  - `DIRECTORY_MUSIC`
  - `DIRECTORY_PICTURES`
  - `DIRECTORY_DOCUMENTS`

Solicitando Permissão

- Para escrever ***dados públicos*** na memória externa, precisamos solicitar a permissão **`WRITE_EXTERNAL_STORAGE`**
    - Esta permissão já implica na também necessária permissão **`READ_EXTERNAL_STORAGE`**

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

Quando escrever?

- Memória externa pode estar associada a mídias removíveis, que podem desaparecer sem aviso

- Checamos estado da memória externa utilizando o método **`getExternalStorageState()`**

- `MEDIA_MOUNTED`

- `MEDIA_MOUNTED_READ_ONLY`

- `MEDIA_REMOVED`

```java
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

```java
/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

<u>Atenção:</u> Além da memória poder se tornar indisponível, não há segurança garantida sobre arquivos salvos em memória externa.

Todas as aplicações podem ler e escrever arquivos na memória externa, além do usuário poder removê-los.

Limites…

- A maioria dos devices tem um limite de 1024 arquivos abertos por processo

- Alguns devices tem limite global de 1024 arquivos abertos

Buffering & sync

- A partir de Android 3.0, o sistema de arquivos adotado foi ext4

- Realiza uma quantidade razoável de buffering

- Ao usar outra forma de persistência como preferences ou SQLite, não afeta em nada

- Ao escrever os próprios arquivos, ocasionalmente é necessário fazer chamadas a **`fsync()`**

    - De forma geral, operações I/O, como escrita de arquivos, devem ser feitas fora da main thread...

Exemplos

Persistindo Dados usando Preferences

SharedPreferences

- Maps que persistem pares de chave-valor de tipos de dados primitivos

- Automaticamente persistidos entre diferentes sessões da aplicação

- Não servem apenas para salvar preferências do usuário, mas também são úteis para tanto.

- Armazenamento de dados personalizáveis da aplicação

Acessando SharedPreferences

- **`getPreferences(int mode)`**
  - usado apenas para preferências associadas com uma dada Activity. Este é o único arquivo de preferências para a Activity, portanto não é fornecido um nome
- **`getSharedPreferences(String name, int mode)`**
  - usado quando precisamos acessar objetos **`SharedPreference`** que são globais a uma aplicação. Neste caso, passamos o nome do objeto que desejamos obter

Modificando SharedPreferences

- Chamando **`SharedPreferences.edit()`**

- Retorna uma instância do objeto **`SharedPreferences.Editor`**

    - Com este objeto, podemos manipular valores usando métodos como **`putInt(…), putString(…), remove(…), clear()`**

- As mudanças são salvas ao chamarmos o método **`SharedPreferences.Editor.commit()`** ou **`SharedPreferences.Editor.apply()`**

Lendo SharedPreferences

- Utilizando métodos para leitura disponíveis em **`SharedPreferences`**
- Por exemplo
  - **`getAll()`**
  - **`getBoolean(String key, …)`**
  - **`getString(String key, …)`**

```
public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

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

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

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

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

Testando o uso de shared preferences

PreferenceFragment

Editando preferências

- Algumas preferences vão ser resultado do uso natural da interface do seu app

- Em outros casos, podemos desejar configurar alguma preferência do usuário, em uma tela específica

- Ao invés de definir a própria UI para coletar estas configurações, podemos usar arquivos XML e **`PreferenceFragment`**

Why?!?!

Para ter forma convencional

Editando preferências

- Podemos declarar um conjunto de preferências em XML na pasta **`res/xml`**
- Esses arquivos já servem como layout de uma tela de preferências (Settings)
  - Com base no elemento **`PreferenceScreen`**
- É possível declarar valores default, gerar grupos de preferência, etc.

user_prefs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="@+id/pref_screen" >

    <EditTextPreference
        android:dialogMessage="Enter Your User Name"
        android:dialogTitle="Change User Name"
        android:key="uname"
        android:negativeButtonText="Cancel"
        android:positiveButtonText="Submit"
        android:title="User Name"
        >
    </EditTextPreference>

</PreferenceScreen>
```

Criando uma Activity para editar preferências

```
public class SettingsActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Deprecated! Devemos utilizar PreferenceFragment…

user_prefs_fragment.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/userPreferenceFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    class="course.examples.DataManagement.PreferenceFragment.ViewAndUpdatePreferencesActivity$UserPreferenceFragment"
    android:orientation="vertical" >

</fragment>
```

PreferenceFragment

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

    setContentView(R.layout.user_prefs_fragment);

}
```

```
// Fragment that displays the username preference
public static class UserPreferenceFragment extends PreferenceFragment {

    protected static final String TAG = "UserPrefsFragment";
    private OnSharedPreferenceChangeListener mListener;
    private Preference mUserNamePreference;
```

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

    // Load the preferences from an XML resource
    addPreferencesFromResource(R.xml.user_prefs);

    // Get the username Preference
    mUserNamePreference = (Preference) getPreferenceManager()
            .findPreference(USERNAME);

    // Attach a listener to update summary when username changes
    mListener = new OnSharedPreferenceChangeListener() {
        @Override
        public void onSharedPreferenceChanged(
                SharedPreferences sharedPreferences, String key) {
            mUserNamePreference.setSummary(sharedPreferences.getString(
                    USERNAME, "None Set"));
        }
    };

    // Get SharedPreferences object managed by the PreferenceManager for
    // this Fragment
    SharedPreferences prefs = getPreferenceManager()
            .getSharedPreferences();

    // Register a listener on the SharedPreferences object
    prefs.registerOnSharedPreferenceChangeListener(mListener);

    // Invoke callback manually to display the current username
    mListener.onSharedPreferenceChanged(prefs, USERNAME);

}
```

Exemplo

PreferenceFragment

Ler também

http://developer.android.com/guide/topics/ui/settings.html

SQLite

SQLite

- Banco de dados relacional em memória
- Projetado para operar com baixo footprint
- Implementa maior parte do padrão SQL92
- Suporta transações ACID
  - Atomic, Consistent, Isolated & Durable
- API não é JDBC

Schemas

- SQLite é um banco de dados relacional típico
  - Tabelas, índices, etc…
- Seu app normalmente terá um conjunto de tabelas
- Esta estrutura é normalmente referida como *schema*
- Eventualmente, o *schema* muda com o tempo…

Como o *schema* pode mudar?

Schemas

- Eventualmente, o *schema* muda com o tempo…
- Ou seja, ao atualizarmos um app, não apenas o código muda, mas a expectativa do código também…
- Isto é, o *schema* precisa ser ajustado
- A classe **`SQLiteOpenHelper`** auxilia na configuração e atualização de *schemas*

SQLiteOpenHelper

- Consolidar o código em duas situações
  - O que acontece na primeira vez que o app roda no dispositivo, após instalação?
  - O que acontece na primeira vez que uma versão atualizada do app está rodando no dispositivo, onde o *schema* foi modificado?

SQLiteOpenHelper

- Contém a lógica para criação e atualização do banco de dados, de acordo com especificação

- Estender a classe, sobrescrevendo os métodos
  - Construtor - encadeando chamada a super()...
  - **`onCreate()`** — comandos do tipo create table
  - **`onUpgrade()`** — o que fazer ao atualizar a versão do banco de dados (atualizar app)

SQLiteOpenHelper

- Construtor
- Contexto
- Nome do BD
- Versão do SCHEMA

SQLiteOpenHelper

- **onCreate()**
  - Recebe um objeto **SQLiteDatabase**, quando o banco precisa ser criado
  - Realiza configuração inicial do banco de dados
  - Pode popular dados iniciais

SQLiteOpenHelper

- **onUpgrade()**
  - Recebe objeto **SQLiteDatabase**, e valores indicando a versão antiga do SCHEMA e a nova
  - Perceba que usuários podem ‘pular’ updates…

Usando **`Helper`**

- Obtenha uma instância da subclasse de **`SQLiteOpenHelper`**
- Quando for necessário fazer consultas ou modificações, use um dos seguintes métodos
  - **`getReadableDatabase()`**
  - **`getWritableDatabase()`**

Onde manter o objeto?

- Para aplicativos triviais, basta usar um atributo de classe
- Se seu app tem múltiplos componentes usando o banco de dados, melhor usar uma instância *singleton* do objeto
  - *Por qual razão?*

Threading

- O ideal é fazer operações de I/O em threads de background

- Abrir um banco de dados é ‘barato’, mas operações de consulta, inserção, etc., não é.

- O objeto gerenciado por **`SQLiteOpenHelper`** é *thread-safe*, desde que as threads estejam usando a mesma instância

- Passar instância da classe **`Application`**

`Application`

- Instância *singleton* de **`Context`**
- Criada no processo momentos após o seu início
- Pode ser obtida em qualquer subclasse de **`Context`** com **`getApplicationContext()`**
- Vantagem é evitar *memory leaks*