Universidade da Beira Interior
Faculdade de Engenharia
Departamento de Informática

Programação de Dispositivos Móveis
Aula 8
Licenciatura em Engenharia Informática
Licenciatura em Informática Web

Programming of Mobile Devices
Lecture 8
Degree in Computer Science and Engineering
Degree in Web Informatics

Sumário
Criação de recetores de eventos que atuam ao nível do sistema ou de uma aplicação móvel. Acesso e manipulação de conteúdos partilhados. Definição de um provedor de conteúdos no âmbito de determinada aplicação móvel. Recetores de difusão e provedores de conteúdos Android™.

Summary
Creation of event receivers acting at the system or mobile application level. Acess to, and manipulation of, shared contents. Definition of a content provider in the scope of a given mobile application. Broadcast receivers and content providers in Android™.

1 Recetores de Difusão
Broadcast Receivers

1.1 Introdução
Introduction

Um *recetor de difusão* (adaptação da designação inglesa *Broadcast Receiver*) é **uma componente das aplicações móveis Android™ que permite que estas se registem para receber eventos provenientes do sistema ou de outra aplicação**. Estes **eventos** são também **representados por intentos**. Contudo:

- Os **intentos usados no contexto de inicialização de Serviços ou atividades são entregues** pelo sistema **a um único componente**, pelo que não podem ser usados para avisar, por exemplo, diversas aplicações acerca de determinado evento. Os **intentos enviados em difusão chegam a todas as aplicações com recetores registados para receber esses eventos**;
- Os **intentos** que são **lançados em difusão são diferentes** dos que são usados para iniciar atividades ou Serviços, pelo que **nunca atingem filtros de componentes destes dois componentes**;
- Por outro lado, os **intentos usados por atividades ou Serviços para despoletar outras componentes do mesmo género nunca chegam aos filtros dos recetores de difusão**.

Este componente é bastante importante no sistema Android™, visto **permitir avisar um conjunto de aplicações**, registados para o efeito, **acerca da ocorrência de determinado evento** no sistema ou aplicação **de uma forma eficiente**. É, por exemplo, este componente que permite que determinada aplicação reaja à receção de uma chamada de voz ou ao facto da bateria estar com pouca carga. Nestes casos, o sistema difunde eventos direcionados a determinado filtro de intentos, e todas as aplicações com um recetor registado ficam avisadas de que esse evento está a acontecer, ou já aconteceu.

A emissão de um intento em difusão **pode ser feita por qualquer aplicação do sistema**. Ainda assim, há que ter em conta que **a receção de alguns intentos em difusão pode requerer o pedido de algumas permissões no manifesto**. Caso este recurso não existisse, teria de ser simulado através da multiplicação e envio de intentos para todas as aplicações que o desejassem receber.

Algumas das ações definidas como padrão na classe `Intent` para difusão são mostradas na tabela seguinte. Intentos representando estas ações são **normalmente enviados pelo sistema Android™** (i.e., por várias das suas aplicações e gestores): Dos exemplos apresenta-

`ACTION_TIMEZONE_CHANGED`	O fuso horário foi mudado;
`ACTION_BOOT_COMPLETED`	Terminou o **arranque** do sistema;
`ACTION_PACKAGE_ADDED`	Foi **adicionada uma nova aplicação**;
`ACTION_PACKAGE_REMOVED`	Foi **removida uma aplicação**;
`ACTION_BATTERY_CHANGED`	O estado da **bateria** do dispositivo mudou;
`ACTION_POWER_CONNECTED`	O dispositivo foi **ligado à corrente elétrica**;
`ACTION_POWER_DISCONNECTED`	O dispositivo foi **desligado da corrente elétrica**;
`ACTION_SHUTDOWN`	O dispositivo está **prestes a ser desligado**.

dos em cima, e que concretizam alguns dos mais típicos intentos difundidos pelo sistema Android™ pode, por exemplo, dizer-se que a ação `ACTION_PACKAGE_ADDED` é enviada em difusão pela aplicação nativa *Package Manager* para enfatizar o facto de que são as aplicações que geram estes intentos, e que qualquer aplicação o pode fazer, desde que tenha uma razão para isso (e.g., avisar outras que determinados dados já estão disponíveis). Outros eventos mais específicos, mas relativos ao sistema, são originados pelos respetivos gestores caso a funcionalidade a que se referem esteja ativa. Por exemplo, caso o dispositivo móvel suporte telefonia, o respetivo gestor (`TelefonyManager`) também estará disponível, e difundirá intentos da sua responsabilidade, nomeadamente o `DATA_SMS_RECEIVED_ACTION`[1], caso seja recebida uma nova SMS.

Existem **duas formas de registar recetores**:

- **Estaticamente**, a partir da inclusão de uma *tag* `<receiver>` **no manifesto da aplicação**;
- **Dinamicamente**, i.e., **programaticamente**, no **código da implementação de um dos outros componentes de determinada aplicação**. Esta opção faz **uso do método** `registerReceiver()`, disponível no contexto da aplicação.

1.2 Registo no Manifesto e Implementação
Registering in the Manifest and Implementation

Quando definido **estaticamente**, o recetor de difusão é **registado aquando do arranque do sistema Android™ ou logo que um pacote é instalado**. O excerto de XML seguinte mostra a declaração estática de um recetor chamado `BootCompletedReceiver`, acompanhado pela definição de um filtro de intentos que *filtra* (ou, neste caso, *captura*) o evento `BOOT_COMPLETED`. Isto significa que este recetor é despoletado logo que o sistema termina o arranque:

```
<receiver
  android:name="ReceiverForAlarms" android:priority="100">
  <intent-filter>
    <action android:name="android.intent.action.
        BOOT_COMPLETED" />
  </intent-filter>
  <intent-filter>
    <action android:name="pt.di.ubi.pmd.exservice.
        ServiceAlarms"/>
    <category android:name="android.intent.category.DEFAULT
        " />
  </intent-filter>
</receiver>
```

Note que **há recetores para certo tipo de eventos que não podem ser declarados estaticamente** e que alguns requerem o pedido de permissões para funcionar.

[1]A definição de intentos de difusão disponíveis para alguns dos gestores é feita na implementação de algumas classes que a eles dizem respeito. Por exemplo, a ação `DATA_SMS_RECEIVED_ACTION` está definida em `https://developer.android.com/reference/android/provider/Telephony.Sms.Intents.html`.

Por exemplo, o recetor definido antes precisa de uma permissão para aceder aos eventos que solicita, nomeadamente de:

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

O facto de alguns recetores só poderem ser definidos dinamicamente **deve-se sobretudo a questões de performance do sistema**. Os recetores, **se em grande número ou incidência, podem degradar a performance**, já que **são invocados cada vez que o evento** que escutam **é despoletado** no sistema. Por exemplo, enquanto que o evento de `BOOT_COMPLETED` só é despoletado depois do arranque, os eventos do tipo `ACTION_TIME_TICK` são despoletados a **todos os minutos**. Neste caso, o ideal é que uma aplicação só esteja à escuta por esse evento, se dele necessitar, enquanto está em execução. Por este motivo, o evento referido em último não pode ser capturado por recetores definidos estaticamente.

O excerto de código seguinte mostra, para já, a forma de **implementar um recetor de difusão**. De um modo suscito, pode dizer-se que **tal é conseguido através da extensão da classe** `BroadcastReceiver`**, que requer importar** `android.content.BroadcastReceiver`**, e da reescrita do método** `onReceive(.,.)`:

```
package pt.di.ubi.pmd.exservice;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class ReceiverForAlarms extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // Este recetor faz parte do mesmo
    // pacote que o ServiceAlarms.
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);
  }
}
```

Quando um evento/intento representando uma das ações definidas nos filtros é despoletado, **o sistema Android™ entrega esse intento a todos os recetores registados** (e autorizados) **através da invocação do método referido**. Note que **o ciclo de vida de um recetor difusão é unicamente determinado por esse método**. Conforme ilustrado, o componente Recetor de Difusão **afigura-se ideal para despoletar Serviços ou Atividades aquando da ocorrência de determinado evento**. No código incluído antes, mostra-se inclusive como é que o Serviço de alarmes, dado como exemplo na aula anterior, pode ser iniciado aquando do arranque do sistema, já que o recetor está definido para esse evento, e a sua implementação do método `onReceive(.,.)` define um intento explicito para o Serviço.

1.3 Registar Recetores de Difusão Programaticamente

Registering Broadcast Receivers Dynamically

Na secção anterior foi enfatizado o facto de **alguns recetores de difusão não poderem ser declarados estaticamente por motivos de performance** do sistema (i.e., usá-los dessa forma pode levar a uma degradação da experiência do utilizador ou da capacidade de resposta do dispositivo de uma maneira geral). Por isso, a plataforma disponibiliza **formas para registar ou eliminar um registo de um recetor de difusão programaticamente**, mais especificamente através dos métodos:

- `registerReceiver(.,.)`, que **aceita o objeto recetor de difusão** a registar e **o objeto filtro de intentos** a que este deve ficar à escuta; e
- `unregisterReceiver(.)`, que **aceita apenas o objeto recetor de difusão cujo registo deve ser eliminado**. Note que este objeto (recetor de difusão) é normalmente implementado num ficheiro à parte, conforme também já ilustrado na secção anterior.

De modo a facilitar o entendimento desta forma de registar recetores de difusão, considere o seguinte excerto de código, acompanhado pelo seguinte cenário: considere que queria que qualquer serviço no sistema Android™ fosse capaz de disparar o serviço `ServiceAlarms` (discutido na aula anterior) através de um intento em difusão para o filtro `pt.di.ubi.pmd.exservice.ServiceAlarms`. Esta funcionalidade específica já era conseguida pela inclusão do filtro no manifesto exibido na secção anterior, é claro, mas assuma, adicionalmente, que queria que o recetor apenas estivesse disponível enquanto o utilizador estivesse a utilizar a atividade `FloatingAlarms`. Neste caso, o recetor difusão deveria ser registado no método `onResume()`, sendo o seu registo anulado (preferencialmente) no método `onPause()`:

```
package pt.di.ubi.pmd.exservice;

import android.app.Activity;
import android.content.IntentFilter;
...

public class FloatingAlarms extends Activity {
  private final ReceiverForAlarms oMyReceiver =
        new ReceiverForAlarms();

  @Override
  protected void onResume(){
    super.onResume();
    IntentFilter oIF = new IntentFilter("pt.di.ubi.
        pmd.exservice.ServiceAlarms");
    registerReceiver(oMyReceiver, oIF);
  }

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
  }

  @Override
  protected void onPause(){
    super.onPause();
    unregisterReceiver(oMyReceiver);
  }

  public void onButtonClick(View v){
    Intent oIntent = new Intent(this, ServiceAlarms.
        class);
    startService(oIntent);
  }
}
```

Note que, no excerto de código anterior, e para além dos métodos de interesse para esta secção, é também utilizada uma classe que ainda não havia sido referida antes. **A classe `IntentFilter`, disponível em `android.content`, permite a definição dinâmica** (i.e., programaticamente) **de filtros de intentos**, sendo obrigatória a sua utilização aquando do registo de um recetor de difusão, já que constitui o segundo argumento de `registerReceiver(.,.)`. Para terminar esta parte da discussão, resta dizer que, em baixo, mencionar-se-á como é que outra atividade ou Serviço pode enviar um intento para o recetor com este filtro.

1.4 Enviar Intentos em Difusão

Broadcasting Intents

A *Application Programming Interface* (**API**) **21 recomenda apenas duas formas de enviar intentos em difusão**, embora anteriormente fossem disponibilizadas mais. Essas duas formas/métodos são:

- Intentos em difusão **não ordenados, enviados através do método `sendBroadcast(.)`, que aceita o intento** a enviar como parâmetro de entrada. Este tipo de intentos **são enviados para o sistema Android™, que os entrega a todos os recetores** que estejam registados para os receber, sem nenhuma ordem em particular. Isto significa também que **este método (`sendBroadcast()`) é assíncrono, i.e., retorna imediatamente à componente** que o chamou depois de ser entregue ao sistema. Como tal, **não permite que sejam recebidos resultados da execução dos recetores ou abortar determinado intento** enviado em difusão;
- Intentos em difusão **ordenados, enviados através do método `sendOrderedBroadcast(.,.)`, que aceita o intento** a enviar **e uma *String*** como parâmetros de entrada. **Estes tipos de intentos são enviados para o sistema Android™, que os entrega a todos os recetores que estejam registados para os receber**, mas **por uma ordem em particular**, que pode ser **definida no manifesto através do atributo `android:priority="integer"` na *tag* `receiver`**. Este método **também é assíncrono** relativamente à componente que a invocou **mas permite**[2], por exemplo, **que um recetor ajuste dados**

[2]Basicamente, este facto significa que a componente que invocou o

que são enviados com o intento para outros recetores, antes que estes recebam o intento. **Também permite que determinado recetor aborte o intento antes de este chegar a outros recetores com prioridade inferior**.

Os dois métodos referidos antes estão **incluídos na classe `Context`, por conveniência**. Esta classe **engloba muitos dos métodos fulcrais para o desenvolvimento de aplicações móveis**, sendo **uma das maiores**, cuja **importância** é assim de fácil justificação[3]. A classe contém **mais métodos com assinaturas semelhante às anteriores**, nomeadamente um `sendBroadcast(.,.)` que aceita também uma *String* para as permissões necessárias do recetor, e um `sendBroadcastAsUser(.,.,.)`, que permite enviar um intento em difusão em nome de determinado utilizador/aplicação, ou um `sendOrderedBroadcast(.,.,.,.,.,.,.)` com 7 argumentos, para **um maior controlo sobre determinada difusão**. O método referido em último permite receber retorno através de um truque. Na verdade, um dos argumentos indica um recetor do mesmo intento que é enviado, e o sistema operativo trata-o como sendo o último a recebê-lo (portanto, pode receber o retorno que outro lhe coloque).

O pequeno trecho de código seguinte mostra como uma Atividade, numa outra aplicação diferente da anteriormente referida, pode enviar um intento em difusão para o recetor que foi registados antes:

```
package pt.di.ubi.pmd.exservice;

import android.app.Activity;
...

public class ExActivity extends Activity {

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
  }

  public void onButtonClick(View v){
    Intent oIntent = new Intent("pt.di.ubi.pmd.
        exservice.ServiceAlarms");
    sendOrderedBroadcast(oIntent, null);
  }
}
```

Por curiosidade, atente nos dois detalhes seguintes: (i) o registo do recetor no ficheiro `AndroidManifest.xml` (incluído já nesta aula) contém um atributo `android:priority`, que só foi discutido nesta secção; (ii) **o intento** criado no método `onButtonClick()` **é implícito, mas aponta diretamente para o filtro** que foi definido naquele manifesto, sendo enviado usando o método `sendOrderedBroadcast()`. Assim sendo, e dada a prioridade definida no manifesto, e a menos que outra aplicação defina uma prioridade maior, **este intento será entregue a esse recetor, antes de ser entregue a outros** que tenham o mesmo filtro. Por outro lado, **como o segundo parâmetro de `sendOrderedBroadcast(.,.)` está a `null`, os recetores registados para este intento não precisam ter pedido qualquer permissão** para o poder receber. O parâmetro da permissão pode ser usado para definir também quais os recetores que podem receber determinado intento.

Considere ainda que a implementação do recetor de difusão continha uma linha adicional com a **invocação do método `abortBroadcast()`**, conforme se mostra a seguir:

```
package pt.di.ubi.pmd.exservice;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class ReceiverForAlarms extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // Este recetor faz parte do mesmo
    // pacote que o ServiceAlarms.
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);

    // Note a seguinte linha de codigo:
    abortBroadcast();
  }
}
```

Neste caso, e **só porque o intento foi enviado com o método `sendOrderedBroadcast()`, este recetor podia cancelá-lo**, impedindo que outros recetores (com menos prioridade) o recebessem. Note que quando é usado `sendBroadcast()`, todos os recetores registados para determinado intento o recebem, independentemente da ordem. Isto significa que **intentos enviados com `sendBroadcast()` não podem ser abortados**.

Caso fosse necessário propagar resultados entre recetores, e assumindo que estes eram invocados pelo sistema depois deste receber um intento enviado com `sendOrderedBroadcast()`, poder-se-iam usar os métodos `getResultData()` e `setResultData(.)`, entre outros, para obter e adicionar dados que eram transmitidos com o evento, como se mostra a seguir:

```
package pt.di.ubi.pmd.exservice;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class ReceiverForAlarms extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // Este recetor faz parte do mesmo
    // pacote que o ServiceAlarms.
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);

    // Note as seguintes linhas de codigo
```

método continua a executar normal e imediatamente após o ter emitido, já que o sistema retorna imediatamente, embora os vários recetores fiquem a executar em paralelo para além da aplicação ou componente.

[3] Ver `http://developer.android.com/reference/android/content/Context.html`.

```
    String sData = getResultData();
    sData = sData + " Another receiver!";
    setResultData(sData);
  }
}
```

O excerto de código anterior mostra como um determinado recetor (chamado `ReceiverForAlarms`) iria processar uma *String* vinda de outro recetor anterior, antes de a enviar para o próximo. Neste caso, iria usar o método de conveniência `getResultData()`, que devolve uma *String*, adicionando-lhe uma nova frase `Another receiver!` antes de voltar a ajustar aquele parâmetro. Caso houvessem 3 recetores a receber o referido intento, o último a recebê-lo iria obter uma `String` semelhante a: `Another receiver! Another receiver! Another receiver!`.

O registo de recetores de difusão é uma funcionalidade **algo delicada do sistema**, principalmente porque pode constituir **um ponto de entrada (leia-se vetor de ataque) para determinada aplicação**. Se o recetor estiver **acessível para todas as aplicações, pode ser despoletado, por exemplo, por uma aplicação maliciosa**[4], que pode tentar subverter o fluxo normal da aplicação através da manipulação do intento ou com outras estratégias. Estes **problemas resolvem-se combinando vários recursos e definições**, nomeadamente através do ajuste de parâmetros no manifesto (e.g., ajustando atributo `android:exported`). Contudo, **caso os eventos em difusão sejam apenas dirigidos a objetos do mesmo processo, pode recorrer ao gestor** `LocalBroadcastManager`, que lhe ajuda a **registar recetores e a enviar intentos em difusão de âmbito local**. Neste caso, **intentos externos ao processo não chegam ao recetores assim declarados, e os intentos gerados não chegam a outros recetores do sistema**. Para comunicações locais, esta solução é **muito mais eficiente** que a que foi antes discutida.

2 Provedores de Conteúdos

Content Providers

2.1 Introdução

Introduction

Os **Provedores de Conteúdos constituem o quarto tipo de componentes Android™** discutidos neste curso. É um componente deste tipo que **permite gerir o acesso a um repositório de dados de uma determinada aplicação**. A utilização de um Provedor de Conteúdos é **sobretudo direcionada a aplicações diferentes daquela que o criou**, já que essa tem acesso direto ao que é disponibilizado. O componente é fulcral para o objetivo em questão, já que **permite controlar, de uma forma consistente e central**, o acesso a recursos que, de outra forma, deveriam ser privados. **Oferece uma interface padrão para aceder aos dados que lida automaticamente com comunicação entre-processos e segurança** no acesso aos dados.

É comum que **uma aplicação que implemente um Provedor de Conteúdos também forneça uma *User Interface* (UI) para esses conteúdos**, já que faz sentido que seja essa aplicação a maior interessada em lidar com todos os detalhes dos dados que fornece. No exemplo incluído neste capítulo mostra-se como se pode construir um Provedor de Conteúdos para uma base de dados que guarda os filmes favoritos de um utilizador. Este exemplo foi explorado numa aula prática anterior, e a respetiva aplicação continha formas de ver, aceder, inserir, atualizar ou eliminar dados da base de dados, embora sem aceder ao Fornecedor de Conteúdos.

A funcionalidade deste componente é concretizada, na verdade e por desenho, **por dois módulos de software diferentes** (os **Fornecedores de Conteúdos** em si, e os **clientes desses Fornecedores**), disponíveis em **classes diferentes**. A discussão seguinte aborda, por isso, os seguintes tópicos:

1. **Como criar e declarar um Provedor de Conteúdos** em determinada aplicação Android™;
2. **Como aceder a um Provedor de Conteúdos disponível numa outra aplicação**.

2.2 Criar um Provedor de Conteúdos

Creating a Content Provider

A criação de um provedor de conteúdos é tido como **um processo algo elaborado**, principalmente porque deve **comportar uma fase de análise da real necessidade de o implementar, a modelação e estruturação dos dados e a implementação/declaração** propriamente dita. Após se decidir que deve ser efetivamente criado um Provedor de Conteúdos, **os passos estritamente necessários à sua criação são** os seguintes:

1. **Modelar a forma como os dados são armazenados e implementar a lógica necessária para os criar**. Os fornecedores de conteúdos podem disponibilizar **dados na forma de ficheiros** ou **dados estruturados**, normalmente armazenados em bases de dados relacionais SQLite;
2. **Definir uma String que determina, de forma unívoca no universo Android™, o Provedor de Conteúdos** a implementar. Esta `String` é tipicamente chamada de **autoridade (*authority*)**, e a documentação oficial sugere que seja usado **um esquema parecido ao que é usado para definir domínios e recursos na Internet**, **mas de forma reversa**. E.g., enquanto que para o site do Departamento de Informática se usa `di.ubi.pt`, no

[4] Mais sobre este assunto em `http://developer.android.com/reference/android/content/BroadcastReceiver.html#Security`.

caso das aplicações Android™ e dos Provedores de Conteúdos usa-se `pt.ubi.di.nome_applicacao` e `pt.ubi.di.nome_aplicacao.nome_provedor`. Este esquema é, na maioria das vezes, suficiente para garantir que a autoridade é única à escala global.

3. **Implementar a classe** `ContentProvider` (disponível no pacote `android.content`), **reescrevendo alguns dos seus métodos** (`onCreate()`, `query()`, `insert()`, ...), como de resto é costume para outras componentes de aplicações Android™, como as Atividades ou Serviços;

4. **Declarar o novo componente no manifesto Android™**, bem como **definir as permissões necessárias** que outra aplicação que lhe quer aceder deve ter.

De modo a cristalizar melhor os passos enunciados em cima, discute-se a seguir **um exemplo** de uma implementação de um Provedor de Conteúdos. Note que este exemplo elabora (e introduz algumas alterações) ao que já foi feito numa das aulas práticas anteriores. A ideia da aplicação e do respetivo Provedor é lidar e **disponibilizar uma lista de filmes favoritos de determinado utilizador**. O exemplo e discussão serão focados na disponibilização de conteúdos da forma **dados estruturados guardados numa base de dados SQLite, que é o mais comum em aplicações Android™**. Esta explicação pode ser generalizada, em algumas partes, para a disponibilização de ficheiros. Contudo, quando se lida com ficheiros, o retorno dos Fornecedores de Conteúdos é constituído por *handlers* para esses recursos, e não por `Cursores`.

Conforme sugerido no passo 1, a primeira parte da implementação de um Provedor de Conteúdos deve ser focada na **modelação dos dados e na lógica necessária para os criar/atualizar**. O primeiro trecho de código mostra a implementação de uma classe que estende a `SQLiteOpenHelper`, conforme já discutido numa aula anterior. Conforme irá notar, são declaradas **algumas variáveis de conveniência relativas à base de dados**, nomeadamente a *String* contendo a instrução **SQL embutida** para criação da única tabela da base de dados. Note que este exemplo é básico, visto que apenas irá referir uma única tabela, para facilitar a explicação. Para exemplos mais elaborados, sugere-se a consulta da documentação em `http://developer.android.com/guide/topics/providers/content-provider-creating.html`.

Relativamente ao código que já foi disponibilizado antes, o seguinte excerto de código difere apenas no nome da chave primária, que aqui se chama `_ID` e no facto do nome da base de dados ser enviado como um parâmetro no construtor da classe, em vez de estar definido estaticamente como uma *String* no início da classe:

```
package pt.ubi.di.pmd.exstorage2;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;

public class AjudanteParaAbrirBD extends
    SQLiteOpenHelper {
  private static final int DB_VERSION = 1;
  protected static final String
          TABLE_NAME = "Movie";
  protected static final String COL1 = "_ID";
  protected static final String COL2 = "name";
  protected static final String COL3 = "year";

  private static final String CREATE_MOVIE =
    "CREATE TABLE " + TABLE_NAME +
    " ("+COL1+" INTEGER PRIMARY KEY,  "
        +COL2+" VARCHAR(50), "
        +COL3+ " INT);";

  public AjudanteParaAbrirBD(Context context, String
       DB_NAME) {
    super(context, DB_NAME, null, DB_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(CREATE_MOVIE);
  }
  @Override
  public void onUpgrade(SQLiteDatabase db, int
       oldVersion, int newVersion) {
    db.execSQL("DROP TABLE " + TABLE_NAME + ";");
    db.execSQL(CREATE_MOVIE);
  }
}
```

É **útil definir a chave primária das tabelas da base de dados afetas a um Fornecedor de Conteúdos com o nome** `_ID`, já que **alguns objetos permitem carregar o conteúdo de um** `Cursor` **para uma** `View` (e.g., `ListView`) **automaticamente**, assumindo que um campo com o nome `_ID` é fornecido.

Tendo a criação e atualização da base de dados assegurada, passa-se para a implementação do Provedor de Conteúdos propriamente dito, ilustrado de seguida:

```
package pt.ubi.di.pmd.exstorage2;

import android.content.ContentProvider;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.net.Uri;

public class ProvedorFavFilmes extends
    ContentProvider {
  // Sera necessario um handle para a
  // base de dados a abrir:
  private AjudanteParaAbrirBD oAPABD;

  // O nome da base de dados:
  private String DBNAME = "FavoriteMovies";

  // O objeto que ira permitir
  // aceder a base de dados:
  private SQLiteDatabase oSQLiteDB;

  private String sAuthority =
    "pt.ubi.di.exstorage2.provedor";

  @Override
  public boolean onCreate() {
    // Criar um objeto do tipo AjudanteParaAbrirBD.
    // Este metodo devolve muito rapido porque a
```

```
    // base de dados apenas e aberta quando
    // oSQLiteDB.getWritableDatabase() for invocado:
    oAPABD = new AjudanteParaAbrirBD(
            getContext(),
            DBNAME
        );
        return true;
    }

  // Implementa o metodo query do Provedor:
  @Override
  public Cursor query(
      Uri uri,
      String[] projection,
      String selection,
      String[] selectionArgs,
      String sortOrder){
    // Abrir a base de dados para leitura:
    oSQLiteDB = oAPABD.getReadableDatabase();
    Cursor oCursor = oSQLiteDB.query(
      "Movie", projection, selection, selectionArgs,
      null, null, sortOrder);
    return oCursor;
  }

  // Implementa o metodo insert do Provedor:
  @Override
  public Uri insert(Uri uri, ContentValues oCValues)
      {
    // Inserir aqui, eventualmente, algum codigo
    // relacionado com a gestao de erros, etc.

    // Abrir a base de dados:
    oSQLiteDB = oAPABD.getWritableDatabase();
    int iId = (int) oSQLiteDB.insert(oAPABD.
        TABLE_NAME, null,oCValues);
    oSQLiteDB.close();
    return new Uri(sAuthority+"/"+iId);
  }
}
```

Conforme se mostra em cima, **é necessário importar e estender a classe** `android.content.ContentProvider`. Neste caso, é também necessário **importar as classes** `SQLiteDatabase` (já discutida anteriormente) **e a classe** `Uri`**, disponível no pacote** `android.net`. Relembre que o acrónimo URI **expande para *Uniform Resourse Identifier*** e que, portanto, **é este objeto que permite identificar, de forma única, um determinado recurso num determinado domínio**.

O código incluído anteriormente reescreve apenas dois métodos da classe `ContentProvider` (embora pudessem ser implementados mais):

- **O método** `onCreate()`**, que deve ser usado para inicializar o Provedor de Conteúdos**, nomeadamente para inicializar **um *handler* apara a abertura da base de dados**, se for o caso. Este método é **executado na *thread* principal** logo que a aplicação que o contém é iniciada e, por isso, **não deve conter operações demoradas**. Por exemplo, **pode conter a instanciação do objeto que ajuda a criar ou abrir a base de dados, mas não deve conter a chamada para a abertura dessa base de dados** (e.g., não deve conter `getWritableDatabase()`). As operações mais demoradas, como a abertura da base de dados para escrita, deve ser feita apenas quando for necessária, nomeadamente dentro de métodos como `insert()`, `delete()` ou `update()`. Este **método devolve** `true` **caso o Provedor tenha sido criado com sucesso, e** `false` **no caso contrário**;
- O **método** `insert(.,.)` que, neste caso, apenas se limita à abertura da base de dados, e à **inserção dos valores provenientes do cliente deste Provedor** (contidos em `oCValues`). Note que o método `insert(.,.)` **aceita um** `Uri` **como primeiro parâmetro e também devolve um objeto desta classe**. O `Uri` recebido deve **apontar especificamente para o recurso que se quer utilizar no âmbito do** `INSERT`**, enquanto que o devolvido deve apontar para a linha inserida**. Neste caso, o `Uri` de entrada não é utilizado no âmbito do método, porque apenas existe uma tabela onde inserir dados, mas este podia ser, por exemplo, **parecido com** `content://pt.ubi.di.exstorage2.provedor/Movie`. Já **o de saída seria semelhante a** `content://pt.ubi.di.exstorage2.provedor/Movie/2`**, em que o** `2` **apontava para a linha específica criada pelo** `INSERT` dentro da tabela `Movie`.

O último passo consiste em declarar o componente no ficheiro `AndroidManifest.xml`. A seguir inclui-se um exemplo do que poderia ser o conteúdo deste ficheiro para a aplicação discutida em cima:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
 xmlns:android="http://schemas.android.com/apk/res/android"
 package="pt.ubi.di.pmd.exstorage2"
 android:versionCode="1"
 android:versionName="1.0">

<uses-sdk android:minSdkVersion="8"
  android:targetSdkVersion = "21" />

<permission android:name="pt.ubi.di.pmd.perm-provider"
  android:label="Permission for Movies Provider"
  android:description="@string/act_permission_desc"
  android:protectionLevel="dangerous" />

<application android:label="@string/app_name"
             android:icon="@drawable/ic_launcher">
  <activity android:name="FavoriteMovies"
            android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.
          LAUNCHER" />
    </intent-filter>
  </activity>

  <provider android:name="ProvedorFavFilmes"
   android:authorities="pt.ubi.di.pmd.exstorage2.provedor"
   android:permission="pt.ubi.di.pmd.perm-provider" />
</application>
</manifest>
```

Repare nos detalhes seguintes:

- O segundo *tag* incluído no elemento `manifest` corresponde à **definição de uma permissão nova**, chamada `pt.ubi.di.pmd.perm-provider`;

- Dentro do elemento `application` é **definido um elemento** `activity` **e um elemento** `provider`, cujo nome é `ProvedorFavFilmes`;

- Dentro da *tag* `provider` é **definida a *authority* deste Provedor de conteúdos**, representada pelo URI `pt.ubi.di.pmd.exstorage2.provedor`. **Não se podem declarar Provedores de uma forma dinâmica**, pelo que têm de ser obrigatoriamente declarados no `AndroidManifest.xml`; e

- **A permissão** referida no primeiro item desta lista **é aplicada ao provedor através do atributo** `android:permission`. Ao usar este atributo está imediatamente a definir a **permissão para aceder ao Provedor de Conteúdos tanto para leitura, como para escrita**. Caso quisesse **diferenciar** o tipo de permissões necessárias para os dois casos, podia primeiro declará-las com nomes diferentes e depois **usar os atributos** `android:readPermission` **e** `android:writePermission` para obter esse tipo de granularidade.

Relembre-se que, ao serem declaradas as permissões necessárias para aceder a determinado componente de uma aplicação Android™, o utilizador tem de a aceitar e fornecer explicitamente à aplicação que a requisita aquando da sua instalação. A aplicação anterior não está a requisitar permissões, mas sim a declará-las. Isto significa que serão as aplicações que quiserem utilizar o Provedor que as terão de pedir.

2.3 Aceder a um Provedor de Conteúdos
Accessing a Content Provider

Uma aplicação pode **aceder aos dados de determinado Provedor de Conteúdos recorrendo a um objeto da classe** `ContentResolver`. Note que **este objeto assumirá o papel de cliente no modelo de comunicação** entre a aplicação que disponibiliza o conteúdo e aquela que o recebe. **Para cada método disponível no cliente, será despoletado um método equivalente no servidor** (Provedor), **sendo os resultados entregues ao primeiro**. O `ContentResolver` oferece assim, na aplicação cliente, **os métodos básicos de obtenção** (i.e., `query()`) **e edição** (i.e., `insert()`, `update()` e `delete()`) **do conteúdo**. É inclusive usado o **acrónimo CRUD para referir o conjunto de funcionalidades disponibilizadas**, abreviando **C**reate, **R**etrieve, **U**pdate e **D**elete. Repare que **os dois objetos executam em diferentes processos** (pertencentes a cada uma das aplicações), mas **lidam automaticamente com as comunicações** entre os mesmos.

O acesso a um Provedor de Conteúdos requer tipicamente o **preenchimento das seguintes condições**:

1. **Pedido de permissões** para o Provedor específico no `AndroidManifest.xml`;
2. **Conhecimento prévio** (ou forma de obter essa informação em tempo de execução) **do URI** do Provedor de Conteúdos;
3. **Instanciação de um objeto da classe** `ContentResolver`**, seguida da utilização dos métodos** que esta disponibiliza para acesso aos conteúdos. Opcionalmente, pode fazer-se **diretamente uso dos métodos estáticos para acesso a esses conteúdos**.

A **obtenção da informação acerca das permissões** necessárias **ou do URI** do Provedor de Conteúdos **é conseguida normalmente através da consulta da documentação do próprio Provedor**. Por exemplo **o sistema Android™ já fornece uma panóplia de Provedores de Conteúdos para uso em aplicações**, nomeadamente para acesso aos registos das chamadas (`content://call_log/calls`), contactos (`content://contacts/people`), *bookmarks* (`content://browser/bookmarks`), etc. Os URIs respetivos são mencionados na documentação oficial, sendo que, por vezes, **também estão disponíveis via variáveis estáticas e públicas nas classes que implementam os Provedores**. Por exemplo, a classe `android.provider.UserDictionary` contém uma *String* chamada `CONTENT_URI` com essa informação específica.

Nesta secção faz-se referência a uma nova aplicação Android™ que irá fazer uso do Provedor de Conteúdos implementado antes. Considere que a aplicação tinha como objetivo mostrar não só os filmes favoritos de um utilizador, mas também as músicas. Caso a aplicação `FavoriteMovies` estivesse instalada, esta nova, e melhorada, aplicação com nome `MoviesAndMusique` **iria tentar importar os filmes que o utilizador já tivesse colecionado, através do Provedor**. Para já, **é necessário pedir as permissões necessárias no manifesto** Android™. O conteúdo do ficheiro seria, assim, algo semelhante a:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
 xmlns:android="http://schemas.android.com/apk/res/android"
 package="pt.ubi.di.pmd.otherapp"
 android:versionCode="1"
 android:versionName="1.0">

<uses-sdk android:minSdkVersion="8"
  android:targetSdkVersion = "21" />

<uses-permission android:name="pt.ubi.di.pmd.perm-provider"
      />

<application android:label="@string/app_name"
             android:icon="@drawable/ic_launcher">
  <activity android:name="MoviesAndMusique"
            android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.
          LAUNCHER" />
    </intent-filter>
  </activity>
</application>
</manifest>
```

Note que o manifesto incluído antes pede a permissão `pt.ubi.di.pmd.perm-provider`, sem a qual não poderia aceder ao Provedor. De seguida mostra-se a implementação da atividade `MoviesAndMusique`, que irá **tentar obter os valores da tabela `Movie` e exibi-los imediatamente numa `ListView`**, tudo na função `onCreate()`:

```
package pt.ubi.di.pmd.moviesmusique;

import android.app.Activity;
import android.os.Bundle;

import android.net.Uri;
import android.database.Cursor;
import android.content.ContentValues;

import android.view.View;
import android.widget.ListView;
import android.support.v4.widget.SimpleCursorAdapter;

public class MoviesAndMusique extends Activity
{
  private Uri oUriProvider =
    new Uri("pt.ubi.di.pmd.exstorage2.provedor");
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Cursor oCursor = getContentResolver().query(
    oUriProvider,
    new String[]{"name","year"},
    null,  // Criterio de selecao
    null,  // Parametros do criterio de selecao
    null); // Ordem

    if ( (null == mCursor ) || (mCursor.getCount() < 1)) {
       // Caso nao sejam devolvidos resultados
       TextView  oTV = (TextView) findViewById(R.id.TV);
       oTV.setText("No data available!");
    } else {

      ListView oLV = (ListView) findViewById(R.id.lv);

      SimpleCursorAdapter oAdptr = new SimpleCursorAdapter(
        this,           // Contexto da aplicacao
        R.layout.line, // Um XML a definir uma linha
        oCursor,        // O cursor com o resultado
        new String[]{"name","year"},   // Nomes das colunas
        new int[]{R.id.ED1, R.id.ED2}, // IDs das views
        0);             // Flags opcionais

      // Ajusta o adaptador a respetiva widget
      oLV.setAdapter(oAdptr);
    }
  }

  public void onINSERTclick( View v ){
    ContentValues oCValues = new ContentValues();
    EditText oED1 = (EditText) findViewById(R.id.name);
    EditText oED2 = (EditText) findViewById(R.id.year);
    oCValues.put("name", oED1.getText().toString() );
    oCValues.put("year", new Integer( oED2.getText().
        toString() ));
    Uri iId = getContentResolver().insert(oUriProvider,
        oCValues);
  }
}
```

O excerto de código anterior contém bastantes detalhes de interesse:

1. Em primeiro lugar, contém **dois *imports* para as classe dos pacotes `android.widget` e `android.support.v4.widget`** nunca referidas anteriormente. **A `ListView` é um contentor que permite mostrar itens numa lista com orientação vertical com *scroll***. Esta `ListView` **pode ser automática e dinamicamente preenchida a partir de um cursor, usando um adaptador**, e.g., da classe `SimpleCursorAdapter`;

2. Em segundo lugar, é **instanciado um objeto da classe `Uri`** no início da implementação da Atividade `MoviesAndMusique`, **que aponta para o Provedor de Conteúdos** definido antes. Este objeto é **depois usado nos métodos `query()` e `insert()`**;

3. Em terceiro lugar, **é feita uma tentativa de obter os dados do Provedor de Conteúdos através do método `query()`, que funciona de modo muito semelhante ao que é usado para aceder a uma base de dados SQLite, embora o primeiro parâmetro seja agora um URI**, e não o nome de uma tabela. Na verdade, em muitos casos, **este URI deve apontar, de forma unívoca para a tabela a que se quer aceder** (neste caso, como só existe uma tabela, basta apontar para o Provedor). Note ainda que **o método `query()` devolve um objeto da classe `Cursor`**;

4. Em quarto lugar, é feita uma verificação se o cursor está vazio ou é nulo, ajustando uma mensagem no *layout* caso isso aconteça;

5. Caso contrário, **é feita uma tentativa de colocar o conteúdo do cursor devolvido pela *query* numa `ListView`**, conforme referido em cima;

6. Por último, é também exibida a implementação do método `onINSERTclick()`, que permite **fazer uma inserção no Provedor de Conteúdos através do método `insert(Uri,ContentValues)`.**

Note que ficam ainda alguns detalhes por especificar, nomeadamente na implementação do Provedor. Na verdade, **não foram implementados os métodos para atualização ou eliminação de registos**, tal como **não foi programada lógica para lidar com URIs mais específicos** (e.g., um URI que especificasse o nome da tabela e o número da linha a devolver numa *query*). Adicionalmente, não foi mostrado o conteúdo de todos os XML usados mas, para clarificar a forma de atuação do adaptador entre a `ListView` e o `Cursor`, mostra-se a seguir o conteúdo do ficheiro `line.xml`, que define como cada linha da lista deve ser desenhada:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
    com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <EditText
      android:layout_width="0dp"
      android:layout_height="fill_parent"
      android:gravity="left"
      android:id="@+id/ED1"
      android:layout_weight="2"
    />
    <EditText
      android:layout_width="0dp"
      android:layout_height="fill_parent"
      android:gravity="center"
```

```
        android:id="@+id/ED2"
        android:layout_weight="1"
    />
</LinearLayout>
```

2.4 URIs dos Conteúdos

Content URIs

Os URIs usados no contexto de aplicações Android™ e, mais concretamente, no contexto de Provedores de Conteúdos, **são recursos muito poderosos**, apesar de não terem alvo de uma análise muito elaborada nas secções anteriores. Normalmente, e neste contexto, **um URI pode ser usado para determinar desde o Provedor de Conteúdo até à linha da tabela que se quer obter**. Um URI pode ser **decomposto nas seguintes partes**: `<standard_prefix>://<authority>/<data_path>/<id>`. Cada uma das partes pode então ser descrita da seguinte forma:

- A parte `<standard_prefix>` **determina o protocolo ou tipo de recurso** (e.g., para os Provedores de Conteúdo é sempre `content`, para recursos na *World Wide Web* (WWW) é normalmente `http`);
- **A `<authority>` determina unicamente o recurso** a que se quer aceder;
- **A `<data_path>` especifica a parte do recurso** a que se quer aceder, e.g., nomes de tabelas ou páginas específicas;
- **A parte do `<id>` é normalmente utilizada para transportar parâmetros para a parte do recurso especificado**. Em Provedores de Conteúdos é comum incluir um número nesta parte do URI que identifica a linha que se quer obter, atualizar ou eliminar para determinada tabela mencionada no `<data_path>`.

A titulo de exemplo, pode indicar-se o URI `content://contacts/people/13`, que aponta para o contacto número 13 guardado na tabela `people` do Fornecedor de Conteúdos `contacts`. O URI `content://call_log/calls/1` (que aponta para outro Fornecedor do sistema Android™) aponta para a primeira chamada no registo de chamadas, guardado numa tabela que provavelmente se chama `calls`.

O **tratamento do URI num Provedor de Conteúdos significa decompô-lo em várias partes, e lidar com essas partes** (nomeadamente com o nome do recurso e com os parâmetros) **caso a caso**. A plataforma fornece algum software que facilita este tratamento, nomeadamente através da **classe** `UriMatcher`[5]

Nota: o conteúdo exposto na aula e aqui contido não é (nem deve ser considerado) suficiente para total entendimento do conteúdo programático desta unidade curricular e deve ser complementado com algum empenho e investigação pessoal.

[5] Ver `http://developer.android.com/guide/topics/providers/content-provider-creating.html`.