Universidade da Beira Interior
Faculdade de Engenharia
Departamento de Informática

Programação de Dispositivos Móveis
Aula 6
Licenciatura em Engenharia Informática
Licenciatura em Informática Web

Programming of Mobile Devices
Lecture 6
Degree in Computer Science and Engineering
Degree in Web Informatics

Sumário
Armazenamento e gestão de dados persistentes em aplicações móveis Android™: dados privados, partilhados e estruturados. Gestão de dados estruturados através do motor de bases de dados com suporte à linguagem *Structured Query Language* (SQL) conhecido por SQLite.

Summary
Storage and management of persistent data in Android™ mobile applications: private, shared and structured data. Management of structured data using hte databases engine with support to Structured Query Language (SQL) known as SQLite.

1 Armazenamento de Dados Persistentes
Storage of Persistent Data

1.1 Introdução
Introduction

Uma das **funcionalidades mais úteis** para a maior parte das aplicações móveis é a de **gerir e armazenar dados de forma persistente**. O Sistema Operativo (SO) Android™ disponibiliza diversas formas de o fazer, nomeadamente[1]:

1. Um **recurso/classe chamado** `SharedPreferences` **(preferências partilhadas), para se guardarem dados primitivos em pares chave-valor**;
2. **Armazenamento interno**, para se guardarem dados na **memória persistente do dispositivo**;
3. **Armazenamento externo**, para se guardarem **dados públicos na memória persistente partilhada e externa** (e.g., SDcards);
4. Bases de dados **SQLite**, para **armazenamento e acesso eficiente de dados estruturados** em bases de dados **privadas**; e
5. Formas de **acesso à rede**, para armazenamento e gestão de **dados remotos**.

É claro que **o tipo** de armazenamento ou recurso específico utilizado **irá depender dos requisitos específicos da aplicação ou funcionalidade implementada**. Por exemplo, se a ideia é guardar as **definições de utilizador ou valores de estado simples** da aplicação, as `SharedPreferences` irão compreender o recurso ideal. Se a ideia for **guardar dados não estruturados** (e.g., textos ou imagens) mas que apenas possam ser **acedidos pela própria aplicação**, o armazenamento interno concretiza uma melhor opção. Se se pretende **guardar dados com estrutura** e que mais tarde possam ser acedidos de forma eficiente, uma **base de dados relacional** será indicada.

Em baixo, discutem-se os vários recursos/classe enumerados em cima, exceto o último, eventualmente descrito num capítulo adiante. À discussão das bases de dados SQLite será dedicada uma secção, enquanto que os três primeiros serão descritos nas 3 subsecções seguintes.

1.2 Preferências Partilhadas
Shared Preferences

A implementação da classe `SharedPreferences`[2] **oferece o software necessário para guardar e recuperar dados de tipos Java primitivos**, como `booleans`, `floats`, `ints`, `longs` ou `strings`. Estes dados são **guardados em pares chave-valor**, em que **a chave é a *string* (o nome)** que define aquele valor. Apesar deste recurso ser **especialmente útil para guardar as preferências do utilizador** para uma aplicação como, por exemplo, o tamanho da letra ou definições de som, pode ser usado para **armazenar e gerir dados que precisem persistir entre sessões** de utilização de uma aplicação, desde que sejam constituídos pelos tipos mencionados antes[3].

A **instanciação de um objeto** da classe `SharedPreferences` é **feita através da invocação**

[1] Este capítulo é sobretudo baseado no `http://developer.android.com/guide/topics/data/data-storage.html` e nas referências principais desta unidade curricular.

[2] Ver `http://developer.android.com/reference/android/content/SharedPreferences.html`.

[3] Note que pode estender uma classe `PreferenceActivity` (não tratada neste curso) para melhor lidar com um menu de preferências.

do método `getSharedPreferences(string, int)` **ou** `getPreferences(int)`. O método referido em primeiro lugar deve ser usado **caso se pretenda dar um nome ao ficheiro de preferências, ou obter o ficheiro previamente guardado com esse nome**. O nome é dado no parâmetro do tipo *string*. O método aceita também um inteiro, que especifica o modo como o ficheiro deve ser guardado ou acedido (os modos de um ficheiro são discutidos nas próximas secções). **Caso só se esteja a usar o ficheiro por defeito, pode ser usado o método** `getPreferences(int)`, que aceita apenas o inteiro que define o modo de acesso ao ficheiro. Ambos os métodos são **fornecidos com o contexto da componente em utilização**. As preferências partilhadas **são ficheiros XML**, **guardados normalmente na diretoria de dados** da aplicação (dada por `/data/data/nome_pacote_aplicacao`), nomeadamente na subdiretoria `shared_prefs`:

- `.../shared_prefs/nome_dado_ao_ficheiro.xml`, caso se tenha dado um nome ao ficheiro; ou
- `.../shared_prefs/nome_pacote_aplicacao.xml`, caso se use o ficheiro por defeito.

Após se instanciar o objeto `SharedPreferences`, é possível **ler qualquer um dos valores** que armazena através de **métodos semelhantes a** `getBoolean(string,boolean)` **ou** `getInt(string,int)`, **que aceitam a chave** que define cada par, e devolvem o respetivo valor. O segundo parâmetro destes métodos é usado por conveniência, e para definir o que a função deve devolver **por defeito caso a referida chave não exista** no ficheiro.

Para **escrever dados nas preferências partilhadas, é preciso obter um segundo objeto do tipo** `Editor` **através do método** `edit()`. Este segundo objeto **disponibiliza vários métodos com prefixo** `put`, nomeadamente `putBoolean(string,boolean)` ou `putInt(string,int)`, que permitem precisamente guardar pares no objeto. Contudo, **estes pares só serão verdadeiramente armazenados no ficheiro** com caráter persistente **depois de se emitir o método** `apply()` **ou** `commit()`.

O pedaço de código seguinte exemplifica a utilização das preferências partilhadas[4]. É mostrado como se pode obter um valor booleano previamente guardado bem como este se pode ajustar após obtenção do editor respetivo. Aparentemente, o valor booleano é guardado quando se invoca o método `exit(View)` (provavelmente despoletado pelo clique num botão), imediatamente antes da atividade terminar.

```
public class SimpleNotes extends Activity {
  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    ...
    // Get the previously stored preferences
    SharedPreferences oSP = getPreferences();
    boolean bRecupera = oSP.getBoolean("recupera",
        false);
    if( bRecupera )
      ...
  }

  public void exit(View v){
    // Instantiate the Editor object
    SharedPreferences oSP = getPreferences();
    SharedPreferences.Editor oEditor = oSP.edit();
    oEditor.putBoolean("recupera", true);

    // Make the edit persistent
    oEditor.commit();
    ...
    super.finish();
  }
}
```

[4]Este código é uma adaptação de um dos exercícios práticos da aula prática 7.

1.3 Armazenamento Interno – Escrita
Internal Storage – Writting

A segunda forma de armazenamento discutida aqui refere-se a **ficheiros comuns e à memória interna do dispositivo**. Note-se que o entendimento de memória interna e externa pode depender de vários fatores. Normalmente, a **memória externa é a que não vem por defeito com o dispositivo**, ou que se **pode amover**, ou ainda que **pode ser tipicamente acedida por várias aplicações ou utilizadores**.

Para **guardar ficheiros na memória interna, nomeadamente na diretoria** `/data/data/nome_pacote_aplicacao/`, **pode usar-se o método** `openFileOutput(string, int)`, que aceita como parâmetros o **nome do ficheiro e o modo de acesso** com que o ficheiro deve ser guardado ou aberto e devolve um objeto da classe `FileOutputStream`. **Os ficheiros** criados ou editados conforme descrito nesta secção **são eliminados automaticamente pelo sistema aquando da desinstalação** da aplicação.

Tal como para os anteriores (embora não tenha sido dito explicitamente), **o dono dos ficheiros** criados com `openFileOutput(string, int)` **é dado pelo ID da aplicação** que invocou o método. Quando guardados com o modo `MODE_PRIVATE`, estes ficheiros só estão disponíveis para a respetiva aplicação móvel. Existem 4 modos de operação que interessa conhecer:

1. `MODE_PRIVATE` (valor 0), que é **o modo sugerido por defeito, que define que apenas a aplicação que criou o ficheiro ou todas aquelas que com ela partilhem o ID lhe podem aceder**. Caso o ficheiro já exista quando é tentada a abertura com `openFileOutput(string, int)`, este é primeiro eliminado e depois recriado;
2. `MODE_WORLD_READABLE` (valor 1), que define que **o ficheiro pode ser acedido para leitura por qualquer aplicação**;
3. `MODE_WORLD_WRITEABLE` (valor 2), que especifica que

qualquer aplicação pode aceder ao ficheiro para escrita; ou

4. `MODE_APPEND` (valor 32768), que determina que **a escrita de dados novos no ficheiro deve ser feita no final**, caso este já exista.

Note que, a partir da *Application Programming Interface* (API) 17, **os modos numerados com o 1 ou o 2 são desencorajados**, já que podem constituir problemas graves de segurança.

A **escrita de dados no ficheiro pela aplicação que o criou é possível em qualquer um dos modos** indicados antes, e pode ser simplesmente conseguida através de métodos como o `write(int iByte)`, o `write(byte[])` ou o `write(byte[] buffer, int offset, int count)`. O `FileOutputStream` deve ser terminado invocando o seu método `close()`. O código seguinte mostra como se pode escrever a *string* `Ola mundo!` num ficheiro chamado `ficheiro.txt` numa aplicação Android™:

```
FileOutputStream fosFile = openFileOutput("ficheiro.
    txt", Context.MODE_PRIVATE);
fosFile.write("Ola Mundo!".getBytes());
fosFile.close();
```

Repare no grau de abstração e facilidade fornecida pela API e também que os vários modos estão definidos estaticamente no `Context`, para sua conveniência.

Para além dos que já foram descritos em cima, podem-se **enunciar outros 4 métodos bastantes úteis no que toca a manipulação de ficheiros**:

- `getFilesDir()`, que devolve **o caminho absoluto** da diretoria onde os ficheiros são guardados;
- `getDir(string, int)`, que **cria a diretoria com o nome definido** no primeiro parâmetro, com as permissões de acesso definidas no segundo;
- `deleteFile(string)`, que **elimina o ficheiro** cujo nome é especificado no primeiro parâmetro; e
- `fileList()` que **devolve um vetor de *strings* com o nome de todos os ficheiros já guardados pela aplicação**.

1.4 Armazenamento Interno – *Cache*

Internal Storage – Cache

Caso o objetivo seja o de **apenas guardar dados por algum tempo**, pode **recorrer a uma *cache*** que o sistema Android™ fornece nativamente. A ideia é simplesmente **guardar os ficheiros na subdiretoria** `cache` da diretoria de dados da aplicação. O caminho para esta diretoria pode ser obtida no seio da execução através do método `getCacheDir()`. Quando os **recursos de armazenamento começarem a escassear no sistema, este começará a eliminar ficheiros contidos em subdiretorias** `cache`, pelo que **não se deve presumir que estes ficheiros estarão sempre disponíveis** e sobrevivam entre uma sessão e a seguinte. É ainda **recomendado** que os programadores **não façam depender do sistema a eliminação dos ficheiros em** `cache`, já que concretiza uma má política. Devem ser implementados métodos na aplicação que, frequentemente ou quando um determinado limite é atingido, limpem ou organizem a referida diretoria.

1.5 Armazenamento Interno – Leitura

Internal Storage – Reading

A **leitura do conteúdo de um ficheiro armazenado internamente é conseguida através da instanciação de um** `FileInputStream`, que resulta da invocação do **método** `openFileInput(string)`. O seu único parâmetro é o nome do ficheiro. O método `read (byte[] buffer, int byteOffset, int byteCount)` pode depois ser usado para devolver `byteCount` bytes para o vetor de bytes `buffer`, ou o ficheiro pode ser lido um byte de cada vez recorrendo a `read()`, que devolve um inteiro (representando um byte) e itera o cursor de leitura no ficheiro. Por defeito, a **aplicação procura o ficheiro a abrir na diretoria** `/data/data/nome_pacote_aplicacao/`.

Note que **é possível incluir um ficheiro estático no projeto** da sua aplicação. Nesse caso, deve **guardá-lo uma subdiretoria de** `res` **chamada** `raw`**.** O nome da diretoria informa o empacotador que não deve comprimir este ficheiro. O método `openRawResource(int)` é o que permite abrir ficheiros deste género **para leitura**, devolvendo um objeto da classe `FileInputStream`. O único parâmetro do método é o ID do ficheiro, escrito na forma `R.raw.nome_do_ficheiro`. Note que **não é possível escrever para ficheiros guardados em** `res/raw` (daí a qualificação de estático incluída antes).

1.6 Armazenamento Externo

External Storage

Todos os dispositivos Android™ suportam armazenamento externo partilhado que também pode ser usado para guardar dados em ficheiros de forma persistente. O meio de armazenamento pode ser **um cartão de memória externo** (tal como um cartão *Secure Digital* (SD)) ou concretizada por **uma parte do sistema de ficheiros num dispositivo não amovível (portanto interno)**. A particularidade que melhor distingue este tipo de armazenamento é o facto de **ficar disponível como uma unidade de armazenamento USB** quando o dispositivo móvel é ligado a um computador. Os ficheiros guardados no armazenamento externo **têm permissões de leitura para todos** (i.e., são `world-readable`), e **é possível que um utilizador os possa modificar via computador**.

Dado as características do armazenamento externo, **não**

deve ser presumido que este estará sempre disponível ou presente (e.g., um utilizador pode remover um cartão SD ou ligar o *smartphone* ao computador) **nem que os ficheiros que aí são guardados se mantêm inalterados de uma execução para outra** (um utilizador ou uma aplicação podem alterá-los). Consequentemente, **recomenda-se que qualquer código que manuseie armazenamento externo seja guardado por uma verificação** se este realmente existe ou está disponível. Esta verificação pode ser feita **obtendo o estado do armazenamento com o método** `getExternalStorageState()`, que devolve uma *string*, e **comparando-o com um dos estados predefinidos estaticamente** na classe `Environment`. O código seguinte mostra a implementação de um método que devolve `true` caso o armazenamento externo esteja disponível **pelo menos** (dado pelo operador `||`[5]) para leitura:

```
public boolean isExternalStorageReadable() {
  String state = Environment.getExternalStorageState();
  if (Environment.MEDIA_MOUNTED.equals(state) ||
      Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)){
    return true;
  }
  return false;
}
```

È preciso ter em atenção que, **para uma aplicação ter acesso** ao armazenamento externo, **tem que normalmente pedir essa permissão** no `AndroidManifest.xml`, nomeadamente através da inclusão de **uma das duas linhas seguintes** naquele ficheiro:

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

A exceção aplica-se ao caso em que aplicação só vai aceder a ficheiros criados por si nesse armazenamento e a versão do SO seja superior à **4.4.** Antes da versão indicada, todos os acessos teriam de ser explicitamente pedidos. Note que, **caso apenas sejam necessárias permissões de leitura, então deve ser usado o elemento com nome** `READ_EXTERNAL_STORAGE`. **O outro elemento pede permissões para escrita, e para a combinação escrita/leitura**.

O exemplo seguinte é um pouco mais elaborado que o anterior e contém mais detalhes. O objetivo do excerto de código é copiar o conteúdo do ficheiro `f1.txt`, incluído na pasta `res/raw` do projeto, para o `ficheiro.txt`, que estará alojado no armazenamento externo. A cópia só é tentada depois de ser verificado que a unidade está disponível, nomeadamente através da comparação do seu estado com a *string* `Environment.MEDIA_MOUNTED`. No caso afirmativo, é obtida a diretoria da aplicação na memória externa através de `getExternalFilesDir(null)` e aí criado o ficheiro destino. Depois disso, o objeto que representa o ficheiro é, no fundo, convertido num `OutputStream`, para onde são escritos todos os bytes lidos de `f1.txt`. Note-se que o ficheiro `f1.txt` foi aberto através do método `getResources().openRawResource(R.raw.f1)`, conforme discutido acima:

```
String state = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
  File fFile1 = new File(Environment.getExternalFilesDir(
      null), "ficheiro.txt");
  OutputStream fosFile = new FileOutputStream(fFile1);
  InputStream fisFile = getResources().openRawResource(R.
      raw.f1);
  byte[] baBuffer = new byte[fisFile.available()];
  fisFile.read(baBuffer);
  fosFile.write(baBuffer);
  fosFile.close();
  fisFile.close());
}
```

Note que o código incluído antes pode disparar exceções não tratadas no exemplo.

O método `Environment.getExternalStorageState()` **devolve todos os estados possíveis** do armazenamento externo (e.g., também pode indicar se o dispositivo está atualmente ligado a um computador). Estes estados podem ser **tratados com mais granularidade** para definir vários fluxos para o programa ou para notificar o utilizador em conformidade.

Uma determinada aplicação **pode guardar ficheiros numa diretoria do armazenamento externo que lhe é dedicada**, ou **numa que já tenha sido criada pelo sistema** (e.g., a diretoria das imagens, etc.). Caso a intenção seja guardar algo numa **diretoria de topo, pública e conhecida**, esta pode ser **procurada especificando o seu nome no primeiro parâmetro do método** `getExternalStoragePublicDirectory(string)`. Por exemplo, quando invocada com `getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES)`, o método devolve o caminho qualificado da diretoria pública que contém as imagens no armazenamento externo na forma de um ficheiro.

É preciso saber que, no caso em que os **ficheiros são guardados numa diretoria da aplicação** (i.e., construída via `getExternalFilesDir(null)`), estes **serão eliminados caso a aplicação seja desinstalada**. Por isso, não se devem guardar ou transferir dados privados importantes do utilizador para essa diretoria (e.g., uma música que o utilizador comprou via *smartphone*). Por outro lado, o facto da diretoria pertencer à aplicação **não determina necessariamente que outras aplicações (com as devidas permissões) ou o utilizador não possam manusear** os seus ficheiros. Por fim, **uma diretoria pertencente a uma aplicação não é normalmente rastreada pelo *Media scanner***. Aliás, é possível esconder o conteúdo dessas diretorias a esse programa colocando lá dentro um ficheiro chamado `.nomedia`.

2 Armazenamento de Dados Estruturados

Structured Data Storage

As secções anteriores discutiam com algum afinco a forma de armazenar e manusear dados persistentes,

[5] `||` é o operador lógico *ou exclusivo* em Java.

mas possivelmente sem estrutura, num dispositivo móvel Android™. Uma vez tendo acesso ao recurso abstrato encabeçado por **um ficheiro**, é óbvio que já existe forma de guardar dados estruturados, como por exemplo registos telefónicos ou informações de uma rede social. Contudo, por definição, **os ficheiros não fornecem, por defeito, formas eficientes e consistentes de pesquisar, aceder, modificar e eliminar dados específicos com estrutura**. Essas funcionalidades e a eficiência estariam **dependentes da inclusão de mecanismos** no código que são **típicos dos chamados sistemas de gestão de bases de dados**, mais conhecidos pela sua designação inglesa *Database Management Systems* (DBMSs).

Os **DBMSs modernos incorporam mecanismos bastante avançados** de manuseamento de dados e **interpretam**, para as bases de dados relacionais, **a linguagem estruturada de consultas**, conhecida sobretudo pelo acrónimo da sua designação inglesa *Structured Query Language* (SQL). Estes dois factos (que se conjugam a outros não referidos aqui), **permitem que os programadores** que recorrem a estes sistemas **se possam abstrair da maior parte dos detalhes de como os dados são guardados** (qual o formato), com **questões de eficiência ou regras de consistência, podendo concentrar-se apenas no conteúdo**. **Os dois sistemas operativos** para dispositivos móveis **mais populares** do mercado (iOS[6] e Android™) **recorrem ao SQLite para armazenamento e gestão de bases de dados relacionais**.

2.1 SQLite

SQLite

O **SQLite**[7] é **uma biblioteca de software que implementa um motor de bases de dados SQL**. Por construção, o SQLite **não é um DBMS**, mas apresenta **muitas das suas características** através da inclusão dos mecanismos típicos que estes sistemas usam. O SQLite é, atualmente, e também em consequência da sua integração nos sistemas operativos que dominam o mercado dos dispositivos móveis, **o motor de bases de dados mais instalado do mundo**, sendo compatível e capaz de interpretar a **norma SQL de 1992**, embora as atualizações que vai sofrendo o tragam **cada vez mais próximo de normas atuais** em alguns aspetos.

O SQLite, atualmente na versão 3.21.0 e por isso tipicamente conhecido por **SQLite3**, foi inteiramente desenvolvido na **linguagem de programação ANSI-C e é código aberto**. Com todas as suas funcionalidades ativas, **não ocupa mais do que 500 Kb**, sendo que **uma utilização normal do motor requer 250 Kb de memória**, o que se apresenta **ideal para dispositivos móveis**, onde podemos ter recursos limitados. É possível obter **toda a implementação** do SQLite **num único ficheiro** `.c` **designado por** `amalgamation.c` e a documentação técnica define-o, por causa disso, como **um motor autocontido**. Contrariamente a muitos outros DBMSs, o SQLite **não implementa o modelo Cliente/Servidor** (por isso não existe nenhum servidor) e **não requer qualquer configuração para ser usado**. Ainda assim, **tem total suporte para transações**, garantindo a a sua Atomicidade, Consistência, Independência e Durabilidade (**ACID**):

- **[Atomicidade] Uma transação** (que pode ser constituída por várias operações de leitura e escrita a uma base de dados) é uma unidade atómica de processamento, que **é realizada completamente ou**, simplesmente, **não é realizada**;
- **[Consistência]** A execução d**uma transação preserva a consistência da base de dados**, i.e., cada transação leva a base de dados de um estado consistente para outro;
- **[Isolamento]** As **atualizações feitas por várias transações concorrentes produzem o mesmo efeito que se estas fossem executadas em série**. Por vezes, tal significa que as modificações de uma transação são invisíveis para outras transações enquanto não atinge o estado `COMMITTED`;
- **[Durabilidade]** Se uma transação altera a base de dados e é `COMMITTED`, **as alterações nunca se perdem mesmo que ocorra uma falha** posterior.

Normalmente, **cada base de dados SQLite é guardada num ficheiro**, normalmente manuseada por **uma única aplicação de cada vez**. Também é **comum haver apenas uma única base de dados em determinado ficheiro**. Isto significa que o SQLite é **sobretudo usado para bases de dados de âmbito local**. Contudo, são **suportados volumes de dados de até 2 TB** e foram incluídos mecanismos que **permitem o acesso concorrente à mesma base de dados** por mais do que uma aplicação. É ainda **fornecida uma *shell***, semelhante ao cliente de muitos DBMSs, que permite **manipular o esquema, consultar e gerir a base de dados** a partir de uma interface de linha de comandos (ver em baixo). Para além das instruções mais comuns SQL, **é também possível definir *triggers* (gatilhos)** para as bases de dados. Note que o facto do SQLite ser implementado em C em poucos ficheiros (ou apenas em um, numa das variantes) permite que este **seja compilado e embutido totalmente no executável de uma aplicação**, favorecendo a sua portabilidade.

2.2 Linguagem Estruturada de Consultas

Structured Query Language

SQL é a linguagem de manipulação de bases de dados **relacionais** por excelência. Contudo, como o SQLite vem nativo às principais plataformas, estas também fornecem algum software que permite **abstrair o programador da linguagem, embora não totalmente**. In-

[6] E.g., ver `https://developer.apple.com/technologies/ios/data-management.html`.

[7] Visitar `http://www.sqlite.org/`.

dependentemente disso, o conhecimento em SQL é **importante tanto na manipulação dos dados como na depuração de uma aplicação**, já que pode ser necessário consultar ou alterar o esquema de uma base de dados para além da própria aplicação, num dos passos da depuração de um problema. Nesta secção, discute-se muito brevemente a sintaxe de algumas das instruções SQL mais importantes, tanto para o desenvolvimento como para a depuração.

A listagem seguinte mostra **a sintaxe de criação de uma tabela conforme entendida pelo SQLite**. Neste caso, a sintaxe é dada numa **forma totalmente textual**, embora os engenheiros deste motor de bases de dados prefiram uma representação visual, como de resto se mostra em baixo. **As partes** da instrução de criação da tabela **opcionais são indicadas entre parêntesis retos**. Como se pode deduzir, em SQLite é possível criar tabelas temporárias e **definir o seu esquema explicitamente a partir definições de colunas** (`ColumnDef_1,...,ColumnDef_N`) **ou herdá-lo de outras relações a partir de uma instrução de** `SELECT`.

```
CREATE [TEMP] TABLE [IF NOT EXISTS]
[Database.]TableName
[(
   ColumnDef_1, ..., ColumnDef_N, TableConstraints
)
[WITHOUT ROWID]]
[AS SELECT_STMT]
```

A sintaxe inclui uma **opção designada por** `WITHOUT ROWID`. Aquando da criação de tabelas SQLite, é **criada uma coluna especial com um ID**, usada para diversos objetivos pela própria implementação do motor. Este campo é **normalmente invisível** para o utilizador. Caso se queira **evitar explicitamente que a coluna seja criada**, é esta instrução que o vai permitir.

Caso se opte pela **definição das várias colunas, é necessário indicar obrigatoriamente o nome de cada uma**. Opcionalmente, deve ser **indicado o tipo de cada uma** (e.g., `INT`, `VARCHAR`, `FLOAT`, etc.) **e restrições adicionais** (e.g., `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `CHECK` e `FOREIGN KEY`), conforme se mostra a seguir:

```
Column_name [type] [constraint]
```

As **restrições permitem definir regras de integridade e consistência** da base de dados. Finalmente, e ainda no caso de se ter optado pela definição das colunas, é possível especificar **restrições ao nível de toda a tabela ou que se aplicam a mais do que uma coluna no final**, após todos os nomes estarem declarados. Por exemplo, caso se queira definir uma chave primária composta por `ColumnDef_1` e `ColumnDef_2`, tal pode ser conseguido colocando a restrição final `PRIMARY KEY(ColumnDef_1, ColumnDef_2)`.

A **eliminação de uma tabela** em SQLite pode ser conseguida através de uma instrução com a seguinte sintaxe (neste caso, basta indicar o nome da tabela a eliminar após declarar `DROP TABLE`):

```
DROP TABLE [IF EXISTS] [Database.]TableName
```

Recorde que a **SQL é uma linguagem declarativa**, o que basicamente significa que **as instruções definem o que deve ser feito, não como deve ser feito**. Caso as instruções sejam emitidas numa *shell*, estas devem ser normalmente sucedidas de um caracter terminador, que é muito frequentemente o ponto e virgula (`;`).

A figura **?? ilustra a sintaxe da instrução de** `SELECT` **em SQLite**. A imagem é a mesma que aparece na documentação oficial do SQLite e representa a forma preferida dos seus criadores para a expressar. De facto, a representação é útil, porque **permite construir a instrução seguindo as várias *keywords* iterativamente**. De uma forma básica e direta, pode dizer-se que a palavra `SELECT` pode ser precedido por `WITH RECURSIVE`, utilizado para *queries* em dados estruturados hierarquicamente, e que **é sempre sucedido pela discriminação das colunas que devem ser exibidas, bem como pela palavra** `FROM`**, que precede o nome da ou das tabelas para as quais a consulta é feita**. **Condições** à seleção são **introduzidas pela cláusula** `WHERE`. A **agregação** de dados pode ser conseguida por uma **cláusula de** `GROUP BY` **e restrições a esta agregação são definidas pela cláusula** `HAVING`[8]. A figura enfatiza que **a cláusula de ordenação, a ter de existir, deve ser sempre colocada após todas as outras expressões e restrições, exceto da de limitação** no número de resultados.
Note que a representação mostra os vários caminhos que podem ser seguidos para construir a instrução, evidenciando simultaneamente quais são opcionais ou não. A convergência dos vários caminhos para determinada *keyword* é sinal da sua obrigatoriedade.

As figuras **??**, **??** e **??** ilustram a sintaxe das instruções de `INSERT`, `UPDATE` e `DELETE`, respetivamente. No primeiro caso, as instruções são normalmente da forma

```
INSERT INTO Table_Name(col1, ..., coln)
VALUES(val1, ..., valn);
```

enquanto que no segundo e terceiro, as instruções são normalmente parecidas com

```
UPDATE Table_Name
SET col1 = val1, ..., coln = valn
WHERE exp;
```

```
DELETE FROM Table_Name WHERE exp;
```

A instrução de `DELETE` pode ser usada para evidenciar a necessidade de **definir chaves primárias com sentido nas relações** de uma base de dados relacional. Note-se, de um modo geral, a **eliminação de uma determinada linha numa tabela da base de dados só pode ser conseguida se existir uma coluna cujos valores identifiquem univocamente cada linha**. Caso a cláusula `WHERE` não esteja declarada nos últimos dois tipos de instruções, todas as linhas da tabela `Table_Name` serão atualizadas (no caso do `UPDATE`) ou eliminadas (no caso do `DELETE`).

[8] A cláusula `HAVING` tem o mesmo efeito de `WHERE`, mas aplica-se a dados agrupados.

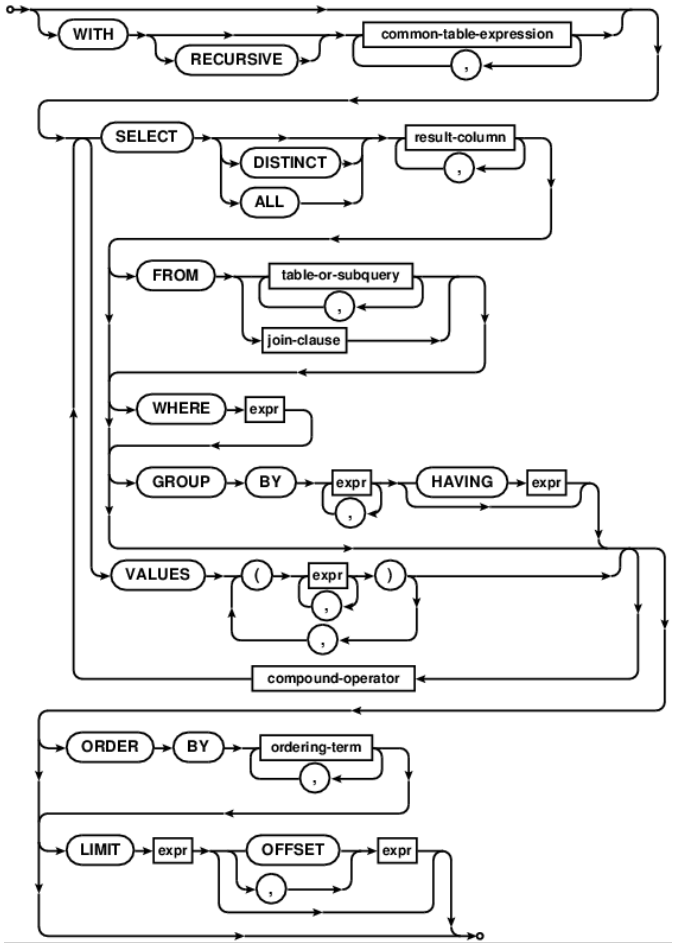

Figura 1: A instrução SQL de `SELECT` conforme entendida pelo SQLite3 (retirada de `https://www.sqlite.org/lang_select.html`).

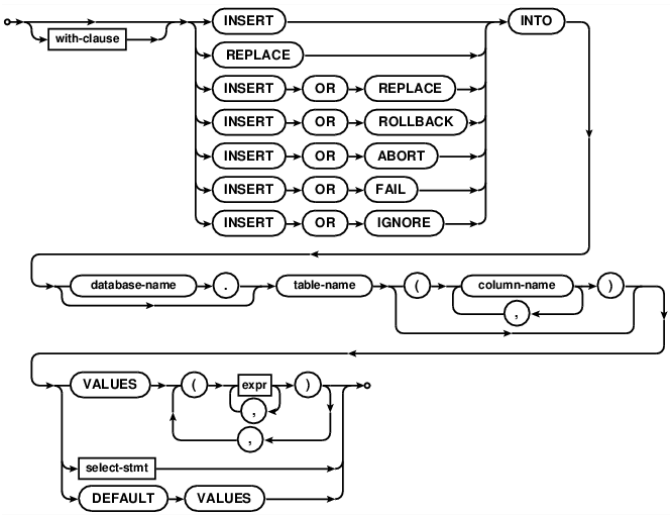

Figura 2: A instrução SQL de `INSERT` conforme entendida pelo SQLite3 (retirada de `https://www.sqlite.org/lang_insert.html`).

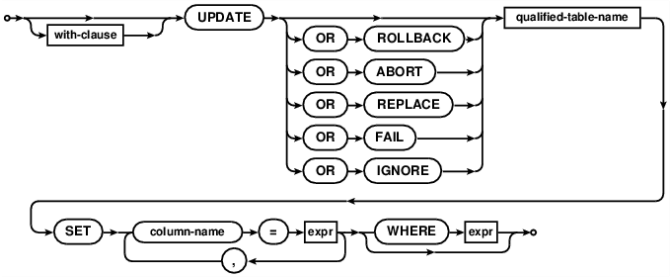

Figura 3: A instrução SQL de `UPDATE` conforme entendida pelo SQLite3 (retirada de `https://www.sqlite.org/lang_update.html`).

2.3 Bases de Dados SQLite em Android™

SQLite Databases in Android™

A utilização de bases de dados SQLite em Android™ é normalmente conseguida através de **software nos pacotes `android.database` (que contém classes genéricas para lidar com a bases de dados) e `android.database.sqlite` (que contém classes específicas para manusear bases de dados SQLite)**.

A forma **recomendada para criar ou atualizar** uma base de dados SQLite é através da **declaração e implementação de uma subclasse de `SQLiteOpenHelper`**, que requer o *import* de `android.database.sqlite`. Ao estender essa classe é **também necessário reescrever obrigatoriamente o método `onCreate()`** e, opcionalmente, o método `onUpgrade()`. O método `onCreate()` é apenas **executado da primeira vez que uma base de dados é criada**, ou seja, quando é tentada a abertura de uma base de dados e esta ainda não existe. Assim, **este método constitui o local ideal para colocar as instruções para criação da base de dados**. Em baixo, inclui-se o código Java que exemplifica a criação da base de dados `EngInf`, que apenas irá conter uma única tabela com 3 campos:

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

import android.database.sqlite.SQLiteDatabase;

public class AjudanteParaAbrirBaseDados extends
    SQLiteOpenHelper {
  private static final int DATABASE_VERSION = 2;
  private static final String DATABASE_NAME = "EngInf";
  protected static final String TABLE_NAME1 = "Student";
  protected static final String COLUMN1 = "number";
  protected static final String COLUMN2 = "name";
  protected static final String COLUMN3 = "avg";

  private static final String STUDENTS_TABLE_CREATE =
    "CREATE TABLE " + TABLE_NAME1 + " (" +
    COLUMN1 + " INT PRIMARY KEY, " +
    COLUMN2 + " VARCHAR(30), " +
    COLUMN3 + " FLOAT);";

  private static final String STUDENTS_TABLE_DROP =
    "DROP TABLE " + TABLE_NAME1 + ";";

  private static final String STUDENTS_TABLE_TEMP =
    "CREATE TEMP TABLE AlunosAux AS SELECT * FROM " +
        TABLE_NAME1 + ";";

  private static final String STUDENTS_TABLE_INSERT =
    "INSERT INTO " + TABLE_NAME1 +
    "(" + COLUMN1 + "," + COLUMN2 + ") " +
    "SELECT * FROM " + STUDENTS_TABLE_TEMP + ";";

  AjudanteParaAbrirBaseDados(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(STUDENTS_TABLE_CREATE);
```

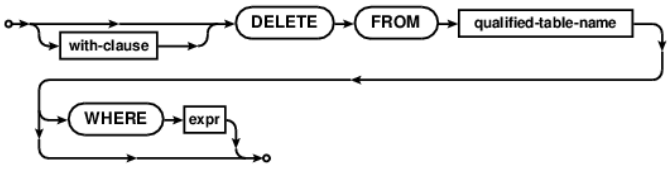

Figura 4: A instrução SQL de `DELETE` conforme entendida pelo SQLite3 (retirada de `https://www.sqlite.org/lang_delete.html`).

```
  }

  @Override
  public void onUpgrade(SQLiteDatabase db,
    int oldVersion, int newVersion){
    db.execSQL(STUDENTS_TABLE_TEMP);
    db.execSQL(STUDENTS_TABLE_DROP);
    db.execSQL(STUDENTS_TABLE_CREATE);
    db.execSQL(STUDENTS_TABLE_INSERT);
  }
}
```

Note que, no exemplo anterior, são colocados em evidência vários detalhes. Primeiro, é feita **a especificação do *package*, que deve ser igual para todas as classes do projeto**. Depois, é declarado um *import* importante, embora possam ser necessários mais. A implementação da classe começa pela **declaração de *strings* que definem o nome da tabela e das suas colunas, bem como das instruções SQL que permitem criar e atualizar a base de dados**. O **construtor padrão da superclasse é simplificado** através da definição estática de alguns valores pertencentes à base de dados em questão, pedindo apenas **o contexto da aplicação onde é criada**. A **única instrução** que o método `onCreate()` executa, neste caso, é a que cria a tabela `Student`.

O exemplo anterior é, na verdade, um pouco mais complexo do que o que é normalmente encontrado em introduções ao tema. Nele é também exemplificado como se pode fazer o ***upgrade* de uma base de dados**. Considere os dois cenários seguintes:

1. No **cenário 1, um utilizador instala e executa a aplicação pela primeira vez**;
2. No **cenário 2, um utilizador atualiza a aplicação, tendo já utilizado a versão anterior antes**. Considere que a versão anterior da aplicação fazia uso de uma base de dados mais simples, contendo apenas as colunas `number` e `name`.

No cenário 1 e durante a sua primeira execução, a aplicação chama o método `onCreate(SQLiteDatabase)` do `SQLiteOpenHelper` porque nota que a base de dados ainda não existia. Neste caso, o método criaria apenas a tabela `Student` com as 3 colunas.

No cenário 2, e durante a primeira execução após atualização, **o método `onUpgrade(SQLiteDatabase,int,int)` será invocado porque**: (i) **a aplicação nota que a base de dados já existe**; e (ii), **a versão que é passada ao construtor[9] é superior à anterior** (considere que a primeira versão da aplicação usava `DATABASE_VERSION = 1;`).

O método `onUpgrade(.,.,.,.)` executa 4 instruções SQL através do método `execSQL(String)`, que submete à base de dados a **instrução** que estiver **definida no parâmetro como uma *String***. Note que a forma de operar desta função faz do **SQL uma linguagem embutida e do Java a linguagem anfitriã**. Visto que a tabela da versão anterior da base de dados continha apenas dois campos, opta-se pelo seguinte procedimento para a sua atualização:

1. Começa-se por **fazer uma cópia temporária** da tabela `Student` para `AlunosAux`;
2. **Elimina-se a tabela** `Student`;
3. **Cria-se a nova tabela** `Student` (que já irá conter as 3 colunas); e
4. Finalmente, **copiam-se os registos anteriores para a nova tabela**, deixando um dos campos a *null*.

O procedimento aplicado é bastante genérico e deve poder ser usado, depois de adaptado, a diversos cenários de atualização. Contudo, visto que a única diferença entre a tabela anterior e a atual é o número de colunas, **o mesmo efeito poderia ter sido conseguido através de uma instrução SQL `ALTER TABLE ...`**

O excerto de código seguinte mostra como é que a classe `AjudanteParaAbrirBaseDados` pode depois ser invocada numa atividade, para além de chamar a atenção para outros detalhes. Basicamente, apenas **é necessário declarar e instanciar um objeto da classe referida, e chamar um dos métodos `getWritableDatabase()` ou `getReadableDatabase()`, para garantir que a base de dados é criada (ou atualizada) e aberta**. Note que o pacote usado nesta aplicação é o mesmo que foi usado no exemplo anterior[10]:

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

import android.database.sqlite.SQLiteDatabase;
...

public class EnfInfStudents extends Activity {
  private SQLiteDatabase oSQLiteDB;
  private AjudanteParaAbrirBaseDados oAPABD;

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
    oTView1 = (TextView) findViewById(R.id.name);
    oTView2 = (TextView) findViewById(R.id.avg);
    // ... ?other instructions?
    oAPABD = new AjudanteParaAbrirBaseDados(this);
    oSQLiteDB = oAPABD.getWritableDatabase();

    ContentValues oCValues = new ContentValues();
    oCValues.put(oAPABD.COLUMN1,new Integer(12589));
```

[9] O número da versão é último parâmetro de `super(Context,String,CursorFactory,int)`.

[10] Note que, por uma questão de legibilidade, não é feita qualquer tentativa de tratar exceções no exemplo apresentado.

```
    oCValues.put(oAPABD.COLUMN2,"Pedro");
    oCValues.put(oAPABD.COLUMN3,new Double(10));
    oSQLiteDB.insert(oAPABD.TABLE_NAME1,
      null,oCValues);

    Cursor oCursor = oSQLiteDB.query(
      oAPABD.TABLE_NAME1,
      new String[]{"name","avg"},
      null, null, null, null,"avg DESC",null);

    oCursor.moveToFirst();
    oTView1.setText(oCursor.getString(0));
    oTView2.setText(""+oCursor.getDouble(1));
  }
  @Override
  protected void onResume(){
    super.onResume();
    oSQLiteDB = oAPABD.getWritableDatabase();
  }
  @Override
  protected void onPause(){
    super.onPause();
    oSQLiteDB.close();
  }
}
```

O exemplo anterior mostra também que **o método** `getWritableDatabase()`, para além de ser o que despoleta a criação ou abertura da base de dados, também **devolve um objeto da classe** `SQLiteDatabase`**, que pode mais tarde ser usado para consultar ou aceder à base de dados**. Para se fazer uso de tal objeto, é necessário importar `android.database.sqlite;`. O método `getWritableDatabase()` **permite o acesso para leitura** (`SELECT`) **ou escrita** (`INSERT`, `DELETE` ou `UPDATE`), **enquanto que o método** `getReadableDatabase()` **apenas permite operações de consulta**.

Repare que o código incluído antes define uma atividade que contém pelo menos duas etiquetas de texto no seu *layout*. Estas duas etiquetas são instanciadas ainda antes da base de dados ser aberta e depois são preenchidas com o nome e média do melhor aluno na base de dados. Pelo meio, é ainda inserido um novo registo na tabela (o aluno com o número 12589, chamado *Pedro* e com a média 10).

O exemplo anterior também ilustra **a utilidade dos métodos** `onPause()` **e** `onResume()`, que ainda não tinham sido implementados até à data nesta unidade curricular. Neste caso, **revelam-se ideais para fechar ou reabrir a base de dados**, respetivamente. Quando a aplicação se preparar para sair de foco ou terminar, **convém fechar o acesso à base de dados**. Caso se volte à aplicação, convém reabrir a ligação, até porque os dados podem ter sido alterados entretanto.

As **bases de dados criadas no contexto de uma aplicação ficam tipicamente guardadas na diretoria de dados** atribuída a essa aplicação, dentro de uma **subdiretoria chamada** `databases`. Isto significa que ao criar uma base de dados chamada `EngInf` no contexto de uma aplicação cujo *package* é `pt.ubi.di.pmd.exstorage2`, por exemplo, a base de dados ficará guardada em `/data/data/pt.di.ubi.pmd.exstorage2/databases/EngInf.db`. **Por defeito, esta base de dados só estará acessível, pelo nome, para qualquer classe da respetiva aplicação**, mas não para qualquer outra.

2.4 `SQLiteDatabase`

`SQLiteDatabase`

Um objeto da classe `SQLiteDatabase` **representa determinada base de dados e disponibiliza** um conjunto de **métodos de conveniência** para a sua consulta e manipulação. Embora seja necessário algum conhecimento **SQL** para usar esses métodos, alguns deles **não requerem que se usem as *keywords* que lhe são características**. **Alguns dos métodos mais interessantes** neste contexto são:

```
long insert(String table, String nullColumnHack,
    ContentValues values)
```

que aceita o **nome da tabela**, um vetor de *Strings* com o **nome das colunas** e os **vários valores a inserir** num objeto da classe `ContentValues`. O método **devolve o ID da linha inserida** ou -1 em caso de erro.

```
int delete(String table, String whereClause, String
    [] whereArgs)
```

que aceita o **nome da tabela** como primeiro parâmetro, **a cláusula** `WHERE` **em forma de *String*** no segundo parâmetro, e um conjunto de valores, a substituir pelo caracter `?`, se este for usado na cláusula `WHERE`. O método **devolve o número de linhas afetadas**.

```
int update(String table, ContentValues values,
    String whereClause, String[] whereArgs)
```

cuja descrição é semelhante à anterior. Note que a *String* referente à cláusula `WHERE` deve conter apenas a definição da condição, e não a própria palavra `WHERE`.

```
execSQL(String sql)
```

que **executa instruções SQL que não devolvem resultados** (ideal para instruções de `CREATE`, `ALTER` ou `DROP TABLE`).

```
Cursor rawQuery(String sql, String[] selectionArgs,
    CancellationSignal cancellationSignal)
```

que permite **executar uma instrução de** `SELECT` **totalmente definida como uma *String*** passada no primeiro argumento. Caso se queira usar pré-processamento e **garantir o saneamento na entrada de dados**, pode-se **definir a *query* com pontos de interrogação (`?`)**, que são depois substituídos pelos valores dados no *array* de *Strings* em segundo lugar. Para o exemplo apresentado antes, o trecho de código seguinte iria obter a(s) linha(s) que contivessem o nome `Pedro`:

```
rawQuery("SELECT * FROM Student WHERE name=?;",
    String[] {"Pedro"}, null);
```

Esta forma de definir os valores **permite**, por exemplo, **evitar ataques de injeção de código SQL**. Caso o parâmetro pelo qual se se quer consultar a base de dados seja oriunda da interface com o utilizador, e introduzida por ele(a), podia-se correr o risco de se estar a enviar para o motor SQLite uma *query* contaminada e com código malicioso. **Ao colocar um `?` e ao definir as *Strings***

à parte, qualquer *input* vindo do utilizador é garantidamente interpretado como uma *String*, e não como código.

Finalmente, o método

```
Cursor query(String table, String[] columns, String
    selection, String[] selectionArgs, String
    groupBy, String having, String orderBy, String
    limit)
```

permite também **submeter uma instrução de** `SELECT` à base de dados, mas **especificando cada parte da sintaxe em *Strings* diferentes**. No exemplo apresentado antes, são pedidos todos os registos de alunos ordenados pela média das classificações (`avg`), da maior para a menor. A cláusula de ordenação é definida no penúltimo parâmetro (`avg DESC`). Realça-se a importância de **definir os vários nomes dados às tabelas e colunas estaticamente** na implementação da classe `SQLiteOpenHelper`, já que isso **aumenta a escalabilidade do código e evita erros**, facilitando também a construção das *queries*. Para **consultas mais elaboradas**, pode fazer-se uso da **classe** `SQLiteQueryBuilder`.

2.5 Cursores
Cursors

Cada método de consulta a bases de dados **devolve um objeto da classe** `Cursor` (`android.database.Cursor`)[11]. Esta classe vem **resolver aquele que é conhecido como o *problema da impedância***, que se **refere ao facto dos dados devolvidos por uma linguagem como a SQL terem tamanhos que não podem ser estimados à partida**, contrariamente ao que acontece em linguagens procedimentais ou orientadas por objetos. E.g., não se sabe, à partida, quantas linhas a maior parte das *queries* SQL devolvem.

Num determinado momento, e se devidamente inicializado, **um objeto** `cursor` **aponta para uma linha do conjunto de dados devolvido**, **disponibilizando um conjunto de métodos de conveniência para navegar** nesse conjunto de dados. Por exemplo, contém os métodos `getCount()` e `getPosition()`, para saber o número de linhas do conjunto de dados e a linha para que aponta atualmente; ou os métodos `moveToFirst()`, `moveToLast()` ou `moveToNext()` para **navegar para o início, para o fim ou para a próxima linha** do conjunto de dados a que se refere, respetivamente.

O objeto `cursor` facilita também **os métodos que permitem obter os valores de tipos primitivos** (neste caso Java) **para cada uma das colunas** da linha para onde aponta. Estes métodos **assumem normalmente a forma** `get[type](int)`. Por exemplo, `getInt(int)` ou `getString(int)` devolvem o inteiro ou a *String* na coluna cujo índice é dado como parâmetro ao método, respetivamente. **Caso não se saiba o índice da coluna** a que se quer aceder, mas **o seu nome seja conhecido, pode recorrer-se ao método** `getColumnIndex(String)`, cujo primeiro parâmetro define precisamente esse nome, devolvendo -1 em caso de erro, ou o índice (começando em 0) da coluna pretendida.

[11] Ver `http://developer.android.com/reference/android/database/Cursor.html`.

2.6 Depuração de Bases de Dados
Debugging Databases

A **depuração de aplicações móveis** que criem ou manipulem ficheiros ou bases de dados no sistema **passa** necessariamente **por verificar se esses ficheiros existem ou contêm os dados necessários e no formato certo**. No caso do Android™, **a ferramenta *Android Monitor***, ou mais especificamente a funcionalidade de **gestor de ficheiros que incorpora**, pode ser **usada para verificar a existência de ficheiros nas diretorias do sistema**. Ver o conteúdo de ficheiros normais também será relativamente simples. Contudo, a consulta ou manipulação do conteúdo de uma base de dados, bem como do seu esquema irá requerer uma ferramenta dedicada para esse efeito.

Existem **aplicações para abrir e navegar em bases de dados SQLite** com interface gráfica e **em modo linha de comandos**, sendo que a última pode revelar-se **interessante por permitir, por exemplo, que se aceda a uma base de dados através da *shell*** fornecida pelo `adb` na plataforma Android™. De resto, **o SDK Android™ já inclui a ferramenta** de linha de comandos `sqlite3` (na diretoria `tools`) e o **Mac OSX também a disponibiliza de fábrica**. A emissão do comando `sqlite3 nome-ficheiro-bd` conduz o utilizador uma *shell* para manipulação da base de dados.

Enquanto que, **para o iOS, a depuração de uma base de dados requer que o ficheiro respetivo seja copiado do dispositivo móvel real para o computador**[12], **no sistema Android™ pode-se abrir a base de dados diretamente no dispositivo** (virtual ou real), **desde que se possuam permissões de acesso** à mesma. O conjunto de passos e *outputs* que concretizam o procedimento a tomar neste caso pode ser estruturado da seguinte forma:

1. Começa-se por **obter informação acerca dos dispositivos móveis** (virtuais ou reais) reconhecidos pelo `adb` com o comando
   `$ adb devices`,
   O comando anterior deve produzir um *output* parecido com o que se inclui em baixo

   ```
   emu1-5554
   emu2-5555
   ...
   ```

2. Sabendo o nome do dispositivo alvo, **consegue-se acesso através de uma *Bourne shell*** com o co-

[12] Caso esteja a utilizador um simulador, o ficheiro da base de dados já está disponível na árvore do sistema de ficheiros desse simulador no disco local.

mando
`$ adb -s emu1-5554 shell`.
Note que, caso só exista um dispositivo alvo, o excerto `-s emu1-5554` é desnecessário, já que esta parte apenas define qual o dispositivo para o qual se vai estabelecer a ponte (*bridge*).

3. Finalmente, **executa-se a ferramenta** `sqlite3` **para acesso e manipulação à base de dados**. O **nome** da ferramenta deve ser **seguido do nome e caminho completo da base de dados** no sistema, caso se esteja a executar da raiz do sistema de ficheiros. O comando será semelhante ao seguinte
`$ sqlite3 /data/data/pt.di.ubi.pmd.exstorage2/databases/exdatabase.db`.
Alternativamente, pode navegar até à diretoria que contém a base de dados e simplesmente emitir o comando `$ sqlite3 nome-basedados.db`. Ao entrar na *shell* assim despoletada, deve ser mostrado um `output` parecido com o seguinte

```
SQLite version 3.6.20
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

e a *prompt* será semelhante a
`$ sqlite>`

Apesar do seu aspeto modesto, **a ferramenta disponibiliza todas as funcionalidades que permitem a total consulta e manuseamento da informação na base de dados e do seu esquema**. Para além de ser possível a **introdução direta de** todas as **instruções SQL** que interpreta (seguidas de um ponto e vírgula (`;`)), ainda **providencia um conjunto de comandos específicos** e úteis. Estes comandos são normalmente **precedidos por um ponto (`.`) e podem ser listados escrevendo** `.help`. Alguns dos mais interessantes são brevemente descritos na tabela seguinte:

`.backup`	Permite f**azer uma cópia de segurança** de uma base de dados para determinado ficheiro (**se** o ficheiro contiver apenas uma base de dados, **corresponde a fazer uma cópia desse ficheiro**);
`.dump`	**Produz um *script* SQL** com todas as instruções que definem determinada base de dados, bem como o seu conteúdo;
`.load`	Que **carrega o conteúdo de um ficheiro** especificado como parâmetro, e contendo dados devidamente separados por um caracter bem definido, **para uma tabela à escolha**;
`.restore`	**Recupera uma base de dados** a partir de uma cópia construida, e.g.„ a partir de `.backup`;
`.schema`	**Mostra o conjunto de comandos** `CREATE TABLE` e seus homónimos **que**, no fundo, **definem o esquema** da base de dados;
`.tables`	**Lista o nome de todas as tabelas** da base de dados;
`.exit`	Para **sair** do SQLite.

Note que alguns dos comandos descritos antes aceitam parâmetros de entrada que não foram especificados na tabela, embora alguns tenham sido referidos na descrição.

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.