Universidade da Beira Interior
Faculdade de Engenharia
Departamento de Informática

Programação de Dispositivos Móveis
Aula 5
Licenciatura em Engenharia Informática
Licenciatura em Informática Web

Programming of Mobile Devices
Lecture 5
Degree in Computer Science and Engineering
Degree in Web Informatics

Sumário
Discussão de um dos conceitos fundamentais da filosofia de implementação de aplicações Android™: intentos. Análise de intentos implícitos e explícitos, bem como da forma como estes podem ser usados para transportar informação e ligar componentes de uma aplicação ou mais. Introdução ao tema da segurança na plataforma Android™, dando especial ênfase à arquitetura de permissões e controlo de acesso sobre a qual elabora.

Summary
Discussion of one of the fundamental concepts of the implementation philosophy for Android™ applications: the intent objects. Analysis of implicit and explicit intents, as well as of the means that can be used to transfer data between components of one or more applications. Introduction to the security subject, paying special attention to the permissions and access control architecture in which the Android™ platform elaborates on.

1 Intentos
Intents

1.1 Introdução
Introduction

Anteriormente, foi dito que as **aplicações Android™ são constituídas por vários componentes**, nomeadamente ***atividades***, ***fornecedores de conteúdos***, ***recetores de difusão*** e ***serviços***. Foi também dito que **é a ordem pela qual esses componentes são organizados que dita o fluxo e as funcionalidades** oferecidas pelas mesmas. Contudo, para além da forma como as atividades são tratadas pela pilha de retrocesso de tarefas, nada foi dito acerca da forma como se pode **despoletar uma dessas componentes, transitar de uma delas para outra, ou como se pode transportar dados entre as mesmas**. É precisamente neste ponto da discussão que o conceito de **intento** ganha relevância.

> Na gíria específica do universo Android™, um intento (da designação inglesa ***intent***) **é um objeto que encapsula, de uma forma abstrata, a intenção de determinada componente fazer uma ação**. **Dependendo da especificidade** do intento, esta ação **pode ser executada por uma componente bem definida** da mesma aplicação ou de outra, capturada pelo Sistema Operativo (SO) e **entregue a uma de várias componentes** que podem fazer essa ação, **ou ser descartada**. Um intento é descartado **caso a componente alvo não exista ou caso não haja componentes capazes de lidar com a mesma**.
> Na documentação oficial[a], é descrito como **um objeto mensagem** que pode ser usado para pedir uma ação a outra componente.
>
> [a] Ver, e.g., `http://developer.android.com/guide/components/intents-filters.html`.

Note-se que, ao discutir as atividades anteriormente, foi enfatizado o facto de **as componentes serem consideradas como unidades algo isoladas de uma aplicação Android™**, no sentido do seu ciclo de vida ser também fortemente determinado e gerido pelo próprio SO. **Os intentos**, e a forma como, em último caso, determinam a forma de evoluir de uma aplicação, constituem **mais um elemento desta filosofia de programação e execução**. Em vez do fluxo de execução estar completamente determinado programaticamente (o que é também possível através de intentos), **a evolução** de um componente para outro faz-se através da **formalização daquilo que ainda se quer fazer a seguir**, em vez de o especificar imediatamente, permitindo **potenciar a modularidade do código**. Há quem defenda que esta filosofia de implementação em geral, e os intentos em particular, contribuem significativamente para o sucesso da plataforma, porque **permitem o desenvolvimento de aplicações mais ricas**, que usufruem de **funcionalidades fornecidas por outras aplicações** através de um **mecanismo muito simples**.

Os intentos podem ser usados para **despoletar uma atividade** ou um **serviço**, ou para **emitir um evento em difusão** (i.e., a um *BroadcastReceiver*). Os intentos **não são usados** no contexto da componente *fornecedores de conteúdos*:

- É possível **despoletar uma nova atividade através do método** `startActivity(Intent)`, que aceita

um intento a definir a ação que deve ser executada e, opcionalmente, o nome da componente que a deve executar. É possível **passar dados para a nova atividade incluindo-os no objeto instanciado**, e também **obter dados no final da execução da atividade**, **através do método** `startActivityForResult(Intent)` (ver adiante).

- Os **serviços** (componentes que executam ações em segundo plano) **podem ser executados** (ou executar ações para determinada aplicação) **recorrendo ao método** `startService(Intent)`, que também **aceita o intento** a definir o serviço e eventualmente **alguns dados que este deve processar**. É ainda **possível obter uma ligação** (da classe `ServiceConnection`) **duradoura a um serviço** através de `bindService(Intent,ServiceConnection,int)`, que **permite que um serviço esteja associado à execução** de determinada atividade ou outro serviço, sendo **terminado quando estes terminarem também** (é útil quando determinado serviço só deve funcionar enquanto a aplicação ou atividade estiver a ser executada). Estes intentos serão discutidos noutro capítulo.

- É possível **emitir *broadcasts* para o sistema** (e para que outras aplicações os recebam) através da **instanciação de um intento** e da sua passagem como parâmetro nos **métodos** `sendBroadcast()`, `sendOrderedBroadcast()`, **ou** `sendStickyBroadcast()`. Estes métodos serão também discutidos posteriormente, quando se abordarem os *BroadcastReceivers* com mais detalhe.

Existem **dois tipos básicos de intentos**: (i) **intentos explícitos** e (ii) **implícitos**. As secções seguintes descrevem estes dois tipos com mais detalhe, apresentando alguns exemplos, após ser listada e brevemente descrita a informação que, de uma maneira geral, estes objetos podem conter. Adiante discute-se também a forma de enviar e receber dados através dos mesmos.

1.2 Instanciação de Intentos
Instantiating intents

A figura incluída a seguir demonstra como é que, de um ponto de vista abstrato, o mecanismo associado aos intentos deve funcionar. Basicamente, **quando um componente de uma aplicação quer começar outro** componente, **instancia um intento e envia-o para o sistema**, que fica **responsável por identificar, verificar as permissões e**, em caso de o encontrar o seu destino e ser permitido, **de o enviar para execução**.

Portanto, para que o mecanismo funcione, o objeto da classe `Intent` **tem de necessariamente transportar alguma informação** que permita ao sistema efetuar essa tarefa, nomeadamente a **ação a efetuar, e o nome ou categoria do componente que deve receber o intento**.

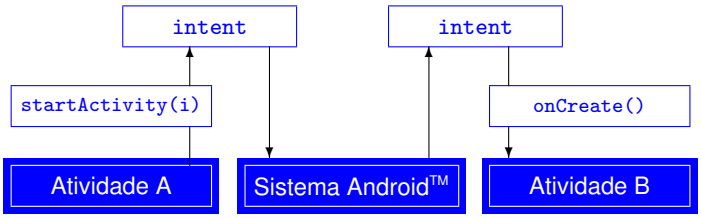

Figura 1: O mecanismo inerente ao uso de `intents` para despoletar atividades em Android™.

Adicionalmente, pode ainda **conter dados** que o componente destino usa para efetuar a ação pretendida. Assim, um intento **pode conter**:

1. O **nome do componente destino** – esta informação é opcional, mas é a que no fundo **distingue intentos explícitos** (caso contenha esta informação) **de implícitos** (caso contrário). O facto é que a plataforma permite que uma componente **determine apenas a ação** a ser executada no intento, deixando ao **critério do SO a escolha da componente que a vai fazer**. Esta **escolha** pode adicionalmente **ser baseada na categoria especificada ou nos dados incluídos**. Esta informação é corretamente definida recorrendo a objetos da classe `ComponentName` (ver exemplo em baixo), que podem ser incluídos no construtor do intento ou ajustados através de métodos como `setComponent()`, `setClass()` ou `setClassName()`;

2. A **ação a efetuar** – que normalmente se **especifica através de uma *string* pré-definida** (e.g., `Intent.ACTION_SEND`, que determina a **ação de enviar/partilhar algum conteúdo**[1]) e disponível na classe `Intent`. **Quase todos os construtores aceitam esta *string***, exceto o vazio, o de cópia e aquele que pode ser usado para lançar uma componente especifica e bem definida (i.e., `Intent (Context, Class)`). **Caso o nome da componente destino não seja especificado, esta informação é obrigatória**. A ação também pode ser especificada através do método `setAction()`. Para além das ações fornecidas pelo sistema na classe `Intent`, **é possível configurar ações que determinada componente de uma aplicação aceita no** `AndroidManifest.xml`. Nesse caso, e para se fazer uso dessa ação, a *string* que a refere deve **conter também o nome do pacote dessa aplicação**.

3. Os **Dados** – **compostos por um** *Uniform Resource Locator* (**URI**) **e pelo tipo MIME** (*Multi-Purpose Internet Mail Extensions*) **do conteúdo para onde aponta**. O tipo de dados é **normalmente determinado pela ação do intento** (e.g., se a ação for do tipo `ACTION_EDIT`, o URI deve apontar para o documento a editar). Contudo, **é recomendado que o tipo seja sempre explicitamente ajustado**, para

[1] Para uma lista extensa de *strings* e ações disponíveis no Android™, ver `http://developer.android.com/reference/android/content/Intent.html`.

que **o intento seja melhor filtrado. Tomar essa opção evita que, por exemplo, um visualizador de imagens seja colocado como opção ao utilizador quando este tenta abrir um ficheiro de música**. O URI e o tipo de dados **podem ser ajustados usando os métodos** `setType(String)` **e** `setData(URI)`, respetivamente. Caso seja necessário ajustar ambos, então deve ser utilizado o `setDataAndType(.,.)`.

4. A **Categoria** – que é **uma *string* que indica o tipo de componente** que pode lidar com determinado evento implícito. Na realidade, **é possível definir mais do que uma categoria para cada intento**, mas a maior parte dos intentos não usa este recurso. A classe `Intent` contém uma série de categorias *hard-coded* que podem ser prontamente usadas neste contexto. Um exemplo dessas categorias é a `CATEGORY_BROWSABLE`, que basicamente determina que o intento pode fluir para qualquer aplicação capaz de exibir o conteúdo de *links*.

5. Os **Extras** – que são **pares chave-valor** usados para **transferir dados adicionais** necessários para determinada ação. Tal como algumas ações usam URIs com uma configuração específica, também outras podem fazer uso de dados adicionais. Por exemplo, quando se usa a ação `ACTION_SEND`, podem-se usar os extras com chaves `EXTRA_EMAIL` ou `EXTRA_SUBJECT` para definir o endereço de e-mail destino ou assunto da mensagem, respetivamente, já que algumas aplicações de *e-mail* fazem uso dele para preencher automaticamente o endereço do destinatário. Para colocar estes valores em intentos, pode-se **recorrer aos vários métodos** `putExtra()`**, que aceitam sempre o valor da chave no primeiro parâmetro e o valor a transportar no segundo**. Também se podem **definir todos os extras num objeto da classe** `Bundle`, passando-o depois ao intento.

6. *Flags* – **funcionam como meta-dados para o objeto** `intent` e podem, e.g., ser **usados para instruir o sistema em como deve lançar ou manipular as atividades ou serviços lançados por esse objeto**. Por exemplo, podem ser usadas para definir se uma atividade que é despoletada deve ou não aparecer na lista de atividades recentes ou não. O ajuste desta informação é normalmente conseguida através de `setFlags()`.

1.3 Intentos Explícitos
Explicit Intents

Os **intentos explícitos especificam univocamente a componente que deve ser despoletada** pelo seu **nome qualificado** no SO (i.e., declarando o **pacote** e o **nome da classe**). Este tipo de intentos é normalmente usado **quando se quer despoletar outra componente da própria aplicação** ou quando se sabe exatamente o nome da classe ou atividade destino. Quando um intento destes é criado, **o SO imediatamente despoleta a atividade ou serviço indicados**, **sem analisar filtros de intentos**.

Esta secção contém dois trechos de código Java que **exemplificam a criação de dois intentos explícitos**. No primeiro, demonstra-se a forma como **tipicamente se despoleta uma atividade** específica, **da qual se sabe o nome** (i.e., a classe Java) e que **potencialmente pertence à mesma aplicação** da componente que a está a chamar. Note-se que, neste caso, é usado o construtor `Intent (Context, Class)`, sendo que **o contexto é usado pelo sistema para determinar parcialmente para onde é que o intento deve ser enviado**, e que `Class` é o nome da classe (i.e., do `ficheiro.class`) que implementa o componente destino. Note ainda que **o uso da classe** `Intent` **requer que se faça a importação de** `android.content.Intent`.

```
import android.content.Intent;
...
Intent intent1 = new Intent(this,Activity2.class);
startActivity(intent1);
```

O segundo exemplo mostra também um **intento explícito, mas com destino a uma aplicação diferente da que o instanciou**. Neste caso, o intento é inicializado recorrendo ao construtor vazio, sendo depois **o pacote e o nome da componente ajustados** através de `new Component(String pkg, String cls)`. Neste caso, o intento deverá culminar na abertura da calculadora que vem por defeito no SO Android™.

```
import android.content.Intent;
...
Intent iCalc = new Intent();
iCalc.setComponent(
  new ComponentName("com.android.calculator2","com.
      android.calculator2.Calculator"));
startActivity(iCalc);
```

Note que **quando um intento é definido desta forma**, o SO **não esboça qualquer tentativa de encontrar as aplicações** que o possam tratar, falhando apenas se o componente destino não existir. Isto também significa que os filtros de intentos definidos no `AndroidManifest.xml` (ver em baixo) não são consultados para intentos deste tipo.

1.4 Intentos Implícitos
Implicit Intents

Os **intentos implícitos** são aqueles para os quais **não é especificado o nome ou pacote do componente a executar**. Em vez disso, **é declarada uma ação geral a ser desenvolvida** pela componente recetora e eventualmente **uma ou mais categorias** a que esta deve pertencer, bem como **dados adicionais**. Estes intentos são **particularmente úteis** para quando se quer fazer **uso de uma funcionalidade que outra aplicação do sistema possa oferecer**, sem especificar exatamente qual. Por exemplo, uma aplicação pode querer mostrar uma

imagem ao utilizador, apesar de não possuir essa funcionalidade. Nesse caso, pode emitir um intento com ação `ACTION_VIEW` e aguardar que SO lhe localize uma aplicação capaz de lidar com essa ação específica.

Note que, quando um componente não é indicado pelo seu nome canónico aquando da configuração do intento, **é necessário ao menos especificar uma ação para esse intento**. É através da ação, e opcionalmente através da categoria e dados, que **o SO encontra uma potencial componente para lidar com o intento**. O exemplo seguinte ilustra a instanciação e emissão de um intento deste tipo.

```
import android.content.Intent;
...
Intent iSendMsg = new Intent(Intent.ACTION_SEND);
iSendMsg.putExtra(Intent.EXTRA_TEXT,"Testing an
    implitic intent!");
iSendMsg.setType("text/plain");
// The following line will assess if an
// activity will resolve this particular intent
if( iSendMsg.resolveActivity(getPackageManager()) !=
     null )
  startActivity(iSendMsg);
```

Para **encontrar o componente certo**, o SO **compara o conteúdo do intento com os *filtros de intentos*** (da designação inglesa *content filters*) declarados no `AndroidManifest.xml` para os elementos `activity`. Se **apenas uma correspondência** entre os dois for encontrada no conjunto de todas as aplicações, então **a componente respetiva é despoletada**. Caso haja **mais do que uma correspondência**, o sistema mostra **uma caixa de diálogo ao utilizador**, a partir da qual pode **escolher interativamente qual deve tratar a ação**. O utilizador pode inclusive definir uma aplicação por defeito para aquela ação. Caso **não exista nenhuma** aplicação capaz de acolher o intento, a aplicação **pode ser terminada ou continuar**, caso a possibilidade tenha sido levada em conta durante a implementação.

O pedaço de código XML seguinte mostra o aspeto de elementos `intent-filter` no ficheiro `AndroidManifest.xml`. Cada elemento desses inclui um elemento `action` e **pode** incluir vários elementos `category`. Os tipos de dados (ver acima) também podem ser definidos através do elemento `data`. Note-se que é possível definir um `intent-filter` para cada componente de uma aplicação Android™ (para serviços e recetores de difusão), conforme sugere a hierarquia do ficheiro XML representado.

```
<activity android:name="ShareActivity">
 <intent-filter>
  <action android:name="android.intent.action.SEND"/>
  <category android:name="android.intent.category.DEFAULT"/
      >
  <data android:mimeType="text/plain"/>
 </intent-filter>
</activity>
```

É claro que, caso um programador queira que a sua **aplicação seja capaz de receber intentos implícitos** de outras, **terá de definir os filtros no manifesto**. Estes filtros **não precisam ser os que já estão definidos na plataforma (e listados na classe** `Intent`), embora **a definição de novos possa não ser muito proveitosa**, já que outros programadores podem não os conhecer.

Este tipo de intentos **encabeçam**, na realidade, um recurso **bastante poderoso, maximizando a funcionalidade e modularidade do sistema e das aplicações**, sendo muito simples encontrar exemplos da sua utilização. Por exemplo, ao usar o gestor de ficheiros e ao escolher a opção de partilhar um documento em particular, o utilizador é confrontado com um conjunto de aplicações que podem ser usadas para o efeito (e.g., a aplicação *Gmail*, *Dropbox* ou *Facebook*). O que no fundo aconteceu é que foi emitido um intento para partilha (`Intent.ACTION_SEND`), e as várias aplicações com filtros para este intento e registadas no sistema foram mostrados ao utilizador pelo sistema, para que este possa escolher o que quer utilizar. A forma como os filtros de intentos são definidos permite também que qualquer programador de aplicações Android™ possa facilmente **colocar a sua aplicação na lista que trata de determinada ação**.

1.5 Envio de Dados Via Intento

Sending Data Via Intent

Existem **várias formas de enviar dados** através de um intento, nomeadamente através da **indicação de um URI** ou através de uma **lista de pares de valores** designada por `Extras`. O exemplo seguinte mostra como se pode declarar um intento definindo a ação (`ACTION_VIEW`) e um URI, ambos passados diretamente ao construtor. Como se pode constatar, os dados da localização são passados dentro do URI, bem como a etiqueta a mostrar nas coordenadas (i.e., `Covilha`). O intento pede ao SO que lhe abra qualquer aplicação que permita `VER`, de alguma forma, os dados que lhe está a passar. E.g., se a aplicação Google Maps estiver instalada, estará registada como sendo capaz de processar estes dados, sendo o URI enviado para e processado num dos seus componentes.

```
import android.content.Intent;
...
Intent intent = new Intent(android.content.Intent.
    ACTION_VIEW, Uri.parse(
  "geo:0,0?q=40.2857325,-7.5012379 (Covilha)"));
startActivity(intent);
```

O exemplo seguinte ilustra o envio de dados via pares de valores. Depois de se instanciar o intento, basta fazer uso do método `putExtra(string, .)` para definir um novo par. **A primeira *string* constitui uma chave** que pode ser **usada para devolver o valor colocado no segundo parâmetro** do método no destino. Note que **existem vários métodos** `putExtra(string,.)` **para os vários tipos primitivos disponíveis no Java** (entre outros e.g., `Strings`), nomeadamente `int`, `double`, `byte`, etc.

```
import android.content.Intent;
...
Intent iActivity = new Intent(this,Activity2.class);
```

```
iActivity.putExtra("string1","This string is going
    to Activity2.");
startActivity(iActivity);
```

Para reaver os valores enviados como extras, **obtém-se primeiro o intento** no componente destino através de `getIntent()`, e **depois o valor do par** através de um método `getTypeExtra("ID")` adequado. O trecho de código seguinte termina o exemplo começado antes. Note que este trecho de código estará definido na `Activity2`, despoletada em cima.

```
import android.content.Intent;
...
Intent iCameFromActivity1 = getIntent();
String s = iCameFromActivity1.getStringExtra("
    string1");
```

1.6 Obtenção de Resultados Via Intento
Obtaining the Result Via Intent

Tal como é possível enviar dados para a componente destino através de intentos, também **é possível receber resultados de uma atividade no retorno**. Em baixo incluem-se dois trechos de código Java que implementam este processo em particular. O primeiro pedaço de código mostra que é criado **um intento explícito e passado ao método `startActivityForResult(Intent,int)`, que despoleta a segunda atividade**. Mais abaixo, também se evidencia a **rescrita de um método chamado `onActivityResult(int, int, Intent)`, que serve de função retorno** (*callback function*), e que é **chamada automaticamente quando um intento regressa com uma resposta**. O `REQ_CODE` é usado para **identificar várias respostas**, caso a aplicação tenha despoletado vários intentos.

```
import android.content.Intent;
...
private static final int REQ_CODE = 10;
{
  ...
  Intent iNewAct = new Intent(this,Activity2.class);
  startActivityForResult(iNewAct,REQ_CODE);
  ...
}
...
@Override
protected void onActivityResult(int reqCode, int
    rCode, Intent iData){
  if( ( reqCode == REQ_CODE ) & rCode == RESULT_OK )
    String sHello = iData.getExtra("string1");
  ...
}
```

Repare-se que **o intento que regressa à atividade inicial não é o mesmo que partiu para a segunda componente**, conforme se evidencia em baixo. O seguinte trecho de código mostra uma **rescrita do método `finish()`, que pode ser chamado no código de uma componente para a terminar**. Neste método é instanciado um intento, alimentado ao método `setResult(int, Intent)` juntamente com **um inteiro que determina o sucesso** (-1=`RESULT_OK`) **ou insucesso da tarefa** (0=`RESULT_CANCELED`).

```
import android.content.Intent;
...
@Override
public void finish(){
  Intent iResponse = new Intent();
  iResponse.putExtra("string1","Hello. How are you?"
      );
  setResult(RESULT_OK,iResponse);
  super.finish();
}
...
```

2 Permissões
Permissions

2.1 Introdução
Introduction

A **arquitetura de segurança do Android™ elabora** simultaneamente **em mecanismos que o núcleo do SO Linux disponibiliza** (nomeadamente o sistema de controlo de acesso a ficheiros) **e em mecanismos adicionais que implementa**, baseados sobretudo em filtros[2]. Um dos **alicerces base** da arquitetura consiste na **assunção de que uma aplicação, por defeito, não tem permissão para executar operações que possam ter impacto adverso noutras aplicações, no SO ou para o utilizador**. Esta restrição aplica-se, portanto, a **dados privados** que possam estar no dispositivo (e.g., contactos ou fotos), **à leitura e escrita em ficheiros** pertencentes a outras aplicações, e **aceder a recursos considerados protegidos ou sensíveis**, como redes de comunicação, câmara, etc. É esta assunção que formaliza o conceito de *sandbox*.

Como **cada aplicação** Android™ executa numa *sandbox*, estas **têm que explicitamente pedir permissões para usar recursos** não fornecidos, de forma nativa, por essa *sandbox*. Estas permissões **são declaradas estatica e explicitamente no manifesto** da aplicação, e **o sistema pede o consentimento ao utilizador** para o usufruto dos recursos **aquando da instalação** ou **durante a execução**. Na verdade, em versões anteriores à 6.0, as permissões teriam de ser **todas aceites** pelo utilizador aquando da instalação, ou a aplicação nem sequer instalava. A partir da versão 6.0, é possível dar permissões durante a execução da aplicação, sendo exibida pelo SO uma caixa de diálogo para esse efeito. Assim, para aplicações desenhadas para a versão 6.0 ou superior, é necessário adicionar código na aplicação que verifique se uma permissão já foi pedida, e que despolete o processo de aceitação em caso negativo.

A documentação oficial define que, no caso do Android™, **o conceito de *sandbox* não está intimamente relacionado com a máquina virtual Java**, e que esta não

[2]Esta secção é parcialmente inspirada em `http://developer.android.com/guide/topics/security/permissions.html`.

deve ser entendida como a tecnologia que garante a segurança por isolamento. Na verdade, **qualquer tipo de aplicação, tenha ela sido implementada em Java, nativa ou híbrida, é sandboxed** através do mecanismo referido em baixo, que emana do núcleo do SO.

2.2 Controlo de Acesso e IDs do Utilizador
Access Control and User IDs

Num SO Linux, cada utilizador tem um IDentificador (*user* ID) que é **usado** para, por exemplo, **controlar o acesso desse utilizador, ou dos processos que ele corre, a ficheiros ou recursos do sistema**. Por exemplo, o utilizador de um SO Linux com identificador `1000` não terá acesso ao ficheiro seguinte

```
rw- -- -- root root file.xxx,
```

já que as permissões indicam claramente que apenas o utilizador `root`, com user ID `0`, lhe pode aceder.

Durante o **processo de instalação de uma aplicação** Android™, o **SO atribui-lhe um IDentificador de utilizador que é único nesse dispositivo** (i.e., noutro dispositivo, o *user* ID até pode ser diferente deste, mas único nesse contexto). **A identidade não muda** durante o período em que a aplicação está instalada no dispositivo. Dado que **o núcleo Linux garante o controlo de acesso ao nível dos processos**, só este facto **assegura** que **determinada aplicação não possa aceder aos recursos de outra diretamente, ou corram no mesmo processo**. Contudo, a partir da **inclusão de um atributo** `sharedUserId` no *tag* do pacote no `AndroidManifest.xml`, **é possível forçar que duas aplicações diferentes corram com o mesmo ID**. Nesse caso, e também por questões de segurança, **os dois pacotes são tratados como sendo a mesma aplicação**, partilhando o mesmo ID e permissões em termos de acesso a recursos do sistema. Esta possibilidade **depende**, contudo, **do facto das duas aplicações estarem assinadas com a mesma chave privada**. Caso contrário, um programador malicioso poderia tentar desenvolver uma aplicação cujo manifesto a acopla-se a uma outra para fins nefastos.

Após instalada, e a menos que expressamente indicado em contrário, **todos os dados guardados** por determinada aplicação ficarão, portanto, **associados ao ID que lhe foi atribuído**. Os **métodos** que normalmente são **usados para criar ficheiros em aplicações Android™** são o `getSharedPreferences(String, int)`[3], `openFileOutput(String, int)` ou o `openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory)`. Todos esses métodos **aceitam uma *flag*** (um `int`) que determinada as **permissões com que os respetivos ficheiros são criados**. Para permitir que outras aplicações acedam aos dados para leitura ou escrita, podem-se **usar os modos** `MODE_WORLD_READABLE` **e** `MODE_WORLD_WRITEABLE`, respetivamente, embora sejam atualmente fortemente desencorajados. Isto provoca a criação de ficheiros com o *User* ID da aplicação, mas com as seguintes permissões:

```
usr grp oth Usr ID GRP ID file.xxx
rw- --- rw- App_ID App_ID file.xxx
```

A **forma ideal de partilhar recursos** de uma aplicação é através de **Provedores de Conteúdos, Serviços ou Recetores de Difusão**.

[3] E.g., ver `http://developer.android.com/reference/android/content/Context.html`.

2.3 Assinatura Digital da Aplicação
Application Digital Signature

Conforme já mencionado durante a discussão do **processo de preparação** de uma aplicação Android™, **os arquivos** `.apk` **têm de ser assinados digitalmente para serem aceites pelo SO**. **A chave pública correspondente à chave privada que assina a aplicação deve estar num certificado X.509**. É comum inclusive dizer-se que o arquivo deve ser assinado com um certificado (o que pode ser entendido como um abuso de linguagem). O certificado **pode ser auto-assinado** (i.e., **não precisa sequer ser assinado por uma autoridade de certificação**) e **identificar univocamente o autor da aplicação**. **Ao contrário de outros gigantes de software, não há qualquer intervenção da Google** na produção destes certificados, pelo que **devem ser construidos localmente** usando ferramentas fornecidas, e.g., pelo Java *Software Development Kit* (SDK). O **principal objetivo** destes certificados é precisamente o de **distinguir os autores das aplicações** possibilitando, por exemplo, que **o sistema forneça ou negue o acesso de uma aplicação aos recursos ou componentes de outra**, ou permita que seja **dado a mesma ID a duas aplicações diferentes**. Caso duas aplicações estejam assinadas com a mesma chave, **o sistema irá permitir, sem perguntar ao utilizador**, que uma das aplicações **aceda aos recursos ou componentes da outra**, desde que a primeira declare o pedido de permissão no manifesto, e a segunda defina essa permissão (também no manifesto) com o nível de proteção (`android:ProtectionLevel`) *SignatureLevel*.

A ferramenta `keytool` **pode ser usada para criar o certificado X.509** e um par de chaves RSA através da combinação de opções seguinte:

```
$ keytool -genkey -v -keystore
chaveiro.keystore -alias nome_chaves -keyalg
RSA -keysize 2048 -validity 9150
```

Note que o certificado gerado fica guardado no ficheiro `chaveiro.keystore`, que o tipo de chaves é RSA, que o seu tamanho é de 2048 bits e que a sua validade é de 9150 dias, que pode ser decomposto na multiplicação 366 dias $\times$ 25 anos. Este valor foi aqui ajustado para enfatizar que, atualmente, a Google apenas aceita

certificados com validade superior a 25 anos. O *alias* é o nome utilizado em baixo para assinar a aplicação.

A preparação da versão *release* da aplicação Android™ prossegue depois com a compilação da mesma usando o comando:

`$ gradle build` ou `$ ./gradlew build`

e com a assinatura do pacote resultante (`app/build/outputs/app-release-unsigned.apk`) usando a **ferramenta `jarsigner` (v1) ou `apksigner` (v2 e v3), fornecida com o SDK**:

```
$ jarsigner -verbose -sigalg SHA1withRSA
-digestalg SHA1 -keystore chaveiro.keystore
app-release-unsigned.apk nome_chaves
```

Note que **o algoritmo de assinatura digital utilizado no comando anterior é o `SHA1withRSA`**, mas que as versões mais recentes dos esquemas de assinatura de aplicações Android já suportam outros algoritmos mais atuais, baseados nomeadamente em SHA256 e SHA512 e em Curvas Elípticas. Note também que é possível configurar o `gradle` (inclusive no IDE) para fazer a assinatura automaticamente. É recomendado que se verifique que o arquivo ficou de facto assinado digitalmente com um comando semelhante ao seguinte:

```
$ jarsigner -verify -verbose -certs
app-release-unsigned.apk
```

O primeiro esquema de assinatura digital usado em Android™ (conhecido como v1), fazia uso da ferramenta `jarsigner`, disponivel no próprio JDK, assinar os pacotes `apk`. Mais recentemente, o esquema de assinaturas foi revisto e melhorado duas vezes (v2 e v3), tendo também sido desenvolvida uma ferramenta específica para assinatura dos `apk`, conhecida por `apksigner`. O funcionamento desta nova ferramenta é, contudo, semelhante ao da antiga.

O arquivo criado com o comando `$ gradle build` **não está alinhado** (aos bytes), conforme requerido no processo de preparação de uma aplicação Android™. Um último passo consiste, portanto, na emissão de um comando parecido com o seguinte, que faz uso da ferramenta `zipalign`:

```
$ zipalign -v 4 application.apk
application-aligned.apk
```

Note que, **para além deste alinhamento, o arquivo não deve sofrer quaisquer outra modificação após ter sido assinado digitalmente**, visto que tal irá invalidar a assinatura. O alinhamento garantirá apenas que os dados não compilados (recursos) começam todos com um alinhamento específico em relação ao início do ficheiro, o que tipicamente provoca uma redução na quantidade de RAM consumida pela aplicação.

2.4 Pedir Permissões no Manifesto

Ask for Permissions in the Manifest

Por defeito, **não são dadas quaisquer permissões a aplicações** Android™. O sistema **veda acesso a todos os recursos que estão para além daqueles incluídos no projeto ou dos que são criados pela própria aplicação aquando da sua execução**. Isto significa que sem o **pedido explícito de permissões no manifesto**, conforme descrito a seguir, uma determinada aplicação móvel só terá acesso aos ficheiros incluídos na pasta `res` e aos que entretanto criar, que são normalmente guardados num componente de armazenamento interno [4] ou externo. O pedido de permissões é **feito através da colocação de uma ou mais *tags* `<uses-permission>`, cujo atributo `android:name` específica o recurso a que se quer ter acesso**. O elemento `<uses-permission>` está **contido obrigatoriamente no elemento `<manifest>`** (e não em `<application>` ou `<activity>`, pelo que **se aplica a toda a aplicação**. A seguir inclui-se um exemplo de um pedido de permissão para ler o registo de chamadas do sistema[5]:

```
<manifest xmlns:android="http://schemas.android.com/
    apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.
        permission.READ_CALL_LOG" />
    ...
</manifest>
```

Durante a instalação da aplicação no Android™, **o *instalador de pacotes* dá algumas permissões à aplicação**, mediante várias condições e cenários. Por exemplo, **algumas permissões são dadas automaticamente por via da verificação das assinaturas digitais**, enquanto que **outras são dadas** por serem consideradas inofensivas para o sistema. **Antes da versão 6.0**, o SO pedia ao **utilizador**, de forma interativa, que explicitamente concedesse todas as permissões durante a instalação. Nessas versões, **uma aplicação não pode ser instalada se falhar pelo menos uma das permissões**. A partir da versão 6.0, as **permissões têm de ser pedidas e dadas durante a execução das aplicações**, uma escolha feita para mostrar mais claramente (ao utilizador) em que pontos do funcionamento da aplicação essas permissões fazem sentido.

A partir da versão 6.0 do sistema operativo é absolutamente necessário incluir, no código da aplicação, instruções que verifiquem se uma permissão já foi dada e, em caso negativo, instruções que as peçam ao utilizador. De uma forma simplificada (i.e., omitindo alguns detalhes), para se pedir uma permissão com sucesso, deve-se:

1. **Definir** a permissão no manifesto, como descrito em cima (sempre necessário);
2. **Verificar** se a permissão já foi dada, recorrendo ao método `checkSelfPermission(String)` (ver trecho de código seguinte); e
3. **Pedir** a permissão com o método `requestPermissions(String[], int)` caso ainda

[4] No caso de ser guardado num dispositivo de armazenamento interno, os ficheiros ou bases de dados são tipicamente guardadas em `/data/data/nome_do_pacote`.

[5] Mais permissões típicas no SO Android™ em `http://developer.android.com/reference/android/Manifest.permission.html`.

não tenha sido concedida antes, que funciona de forma assíncrona (retornando imediatamente).

O trecho de código seguinte está alinhado com o exemplo dado nesta secção, dado fazer a verificação se a permissão `READ_CALL_LOG` pedida no manifesto (em cima) já foi dada, pedindo-a caso contrário. Note-se que se podem pedir várias permissões simultaneamente, dado o método `requestPermissions(String[], int)` aceitar um *array* de *Strings* (uma para cada permissão). O método referido em último retorna imediatamente, e **há um método de *callback* que o sistema operativo invoca automaticamente** (`onRequestPermissionsResult(int, Strings[], int)`) quando o utilizador aceita ou rejeita uma permissão, não discutido aqui.

```
if( checkSelfPermission(Manifest.permission.
    READ_CALL_LOG) != PackageManager.
    PERMISSION_GRANTED )
    requestPermissions(new String[]{Manifest.
        permission.READ_CALL_LOG},1);
else {
...
}
```

Quando uma aplicação tenta **aceder a um recurso para o qual não tem permissão** (e.g., porque não a declarou, verificou e pediu), **é normalmente enviada** para o componente em questão **uma** `SecurityException`. Embora essa exceção possa nem sempre ser disparada[5], **problemas de permissões são quase sempre reportados no *log*** do sistema. Se a componente respetiva não estiver preparada para lidar com a exceção, a aplicação pode terminar abruptamente.

2.5 Definir Permissões no Manifesto
Define Permissions in the Manifest

A definição de uma nova permissão para acesso a uma determinada aplicação, ou a um dos seus componentes, **também é feita no** `AndroidManifest.xml`. Neste caso, usam-se **um ou mais elementos** `<permission />`**, que devem estar forçosamente dentro do elemento** `<manifest>`. Estes elementos devem ter pelo menos **2 atributos** definidos: (i) `android:name` e (ii) `android:protectionLevel`. Também é recomendado definirem-se sempre os atributos `android:label` e `android:description`. Em baixo exemplifica-se como se pode declarar uma nova permissão chamada `com.me.app.myapp.permission.actividade`, que em baixo é aplicada para guardar o acesso a uma atividade específica:

```
<manifest xmlns:android="http://schemas.android.com/apk/res
    /android"
  package="com.me.app.myapp" >
  <permission android:name="com.me.app.myapp.permission.
      actividade"
    android:label="@string/activity_permission"
    android:description="@string/act_permission_desc"
    android:permissionGroup="android.permission-group.
        CAMERA"
    android:protectionLevel="dangerous" />
    ...
</manifest>
```

[5]Alguns métodos só reportam o falhanço em aceder a um determinado recurso ao devolver o resultado (`return`), e não à cabeça.

Note que o trecho XML anterior **apenas define a permissão**. Esta **ainda não foi aplicada** a nenhum componente particular.

O atributo `protectionLevel` **é necessário e caracteriza o risco que está associado a uma permissão**. É possível encontrar uma descrição dos vários níveis de proteção em `http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel`:

- Por exemplo, **se o nível for** `normal`**, significa que esta permissão** não deve ter impacto para o utilizador ou para o sistema, e que portanto **pode ser dada automaticamente aquando da instalação** de qualquer aplicação que a peça (i.e., sem autorização explícita do utilizador);
- **Se for** `dangerous` (como no exemplo anterior), **então terá que ver com dados pessoais do utilizador ou recursos mais sensíveis do sistema**, pelo que será necessário perguntar-lhe explicitamente;
- Para além das duas anunciadas antes, existem ainda **os níveis** `signature` **e** `signatureOrSystem`, que definem que a **permissão deve apenas ser dada a outras aplicações assinadas com o mesmo certificado, ou a aplicações de sistema** (ou assinadas com o mesmo certificado das aplicações de sistema), respetivamente.

O **nome da permissão** é **usado para a identificar de forma única em todo o sistema** (daí conter o **nome qualificado** do pacote). Este nome **é também usado no atributo** `android:permission` **dos elementos** `<activity>`**,** `<service>`**,** `<receiver>` **e** `<provider>`**, que especifica a que componentes é que a permissão declarada se aplica realmente**.

É **conveniente providenciar sempre um** `label` **e uma** `description`. É o conteúdo das respetivas *strings* que **é mostrado ao utilizador quando o sistema quer pedir que seja dada ou negada permissão**. Sugere-se que sejam ambas muito claras e que a descrição seja composta por apenas duas ou três frases. A primeira frase descreve a permissão, enquanto que as restantes podem informar o utilizador do que pode acontecer se essa permissão for abusada por, e.g., *malware*. O `label` deve indicar, em poucas palavras, o que é que se está a proteger. Note que, no exemplo anterior, é enfatizado o facto de se estarem **a usar *strings* definidas no ficheiro** `strings.xml` **na pasta** `res`. Para uma maior escalabilidade e portabilidade, os vários aspetos do desenvolvimento de uma aplicação móvel, neste caso Android™, devem ter esta abordagem em consideração. A seguir mostra-se um exemplo para estes dois atributos, que vem no seguimento do anterior:

```
<string name="activity_permission">Usa a camara do
    dispositivo!</string>
```

```
<string name="act_permission_desc">Permite que a
    aplicacao aceda a camara do dispositivo para
    tirar fotos. Pode ser usada por software
    malicioso para obter dados ou imagens da sua
    vida pessoal.</string>
```

O `permission-group` é usado para agrupar várias permissões pedidas por uma aplicação na caixa de diálogo que é mostrada ao utilizador. Por isso, será indicado usar um dos valores já existentes na documentação oficial.

Num dispositivo Android™, **é possível ver as permissões que cada aplicação está a usar via *Settings* $\longrightarrow$ *Applications***. Pode-se **recorrer ao comando** `pm list permissions -s` (dentro da *shell* fornecida por `adb shell`) para **obter uma ideia de quais são as permissões atualmente definidas** (por várias aplicações) no sistema:

```
$ adb shell pm list permissions -s
```

O output será semelhante a:

```
All Permissions:

Network communication: view Wi-Fi state,
create Bluetooth connections, full Internet
access, view network state

Your location: access extra location provider
commands, fine (GPS) location, mock location
sources for testing, coarse (network-based)
location

Services that cost you money: send SMS
messages, directly call phone numbers
...
```

2.6 Aplicar Permissões a Componentes

Aplicar Permissions to Components

É possível **aplicar as permissões com bastante granularidade** no sistema operativo Android™. Depois de definida (e nomeada), uma permissão é **aplicada a determinados componentes ou a toda a aplicação através do atributo `android:permission="name_of_the_permission"`**. A **permissão mais específica sobrepõe sempre a menos específica**. E.g., uma permissão aplicada a uma *activity* terá precedência relativamente a uma permissão aplicada à *application*. De uma forma breve, a forma de atuação das permissões para os vários componentes de uma aplicação Android™ pode ser definida da seguinte forma:

- As permissões **aplicadas a uma atividade restringem que componentes é que a podem despoletar**. As permissões são verificadas quando é invocado o método `startActivity()` ou `startActivityForResult()`. Caso as permissões não sejam concedidas, é disparada uma `SecurityException`;
- As permissões **aplicadas a um serviço** (`<service>` *tag*) **restringem também quem pode começar ou associar-se ao mesmo**. São verificadas aquando da invocação de `startService()`, `stopService()` ou `bindService()`. Eventualmente, podem-se verificar permissões durante a execução de um serviço através do método `checkCallingPermission(string)`, em que a *string* indica o nome da permissão que se quer verificar;
- As permissões **aplicadas a Recetores Difusão são definidas nas *tags* `<receiver>` e restringem quais as aplicações que podem enviar eventos para esses recetores**. Neste caso, **a permissão só é validada após o método `sendBroadcast()` devolver** (i.e., `return`), já que **é o sistema que faz a tentativa de entrega ao recetor**, e não o próprio emissor. Por isso, **o emissor tem de esperar pelo retorno do sistema**. Neste caso, **nunca será levantada uma exceção**. É possível que um recetor forneça também a permissão numa *string* aquando da invocação do método `sendBroadcast()`, para controlar programaticamente os emissores que podem enviar mensagens para o componente. De igual forma, o componente emissor pode especificar a permissão aquando da invocação de `sendBroadcast()`;
- Finalmente, as permissões **aplicadas a fornecedores de conteúdos são definidas na *tag* `<provider>` e são usadas para restringir o acesso a determinado conteúdo**. Contudo, **para estes componentes é possível definir dois atributos** para as permissões que determinam se um aplicação especifica pode **ler** (`android:readPermission`) **ou escrever** (`android:writePermission`) no fornecedor de conteúdos [6]. As **permissões são verificadas na primeira invocação** do provedor de conteúdos. O método `query()` requer permissões de leitura, enquanto que os métodos `insert()`, `update` e `delete()` requerem a permissão de escrita.

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.

[6] Nota: é possível definir controlos de acesso ainda mais granulares para este tipo de componentes, nomeadamente relacionados com permissões a *Uniform Resource Locators* (URIs), através dos quais estes componentes são normalmente acedidos. Estes controlos não são aqui discutidos.