Programação de Dispositivos Móveis

- Recetores de Difusão

Deselvolvimento de *Interfaces* de Utilizador Orientadas por Eventos(para Dispositivos Móveis)

As Interfaces de utilizador orientadas por eventos são utilizadas na maior parte das aplicações móveis modernas.

> (...)um **Evento** corresponde a uma ação ou ocorrência que um programa foi capaz de detetar e tratar.

O sistema operativo(SO) é que está encarregue de capturar o evento e de comunicar os eventos aos programas/aplicações.

> (...)é o programa que esperar pelo utilizador(...)

A programação por eventos é (normalmente) feita por **modelo de delegação de eventos** baseado nas entidades:

- **Controlos**, fonte do evento;
- **Consumidores** (*Listeners*), correspondem às rotinas de tratamento desses dados;
- ***Interfaces*** são a forma normalizada de comunicação entre as duas entidades.

```
        import android.view.*;
        import android.widgets.*;
        ...
        //a proxima linha liga o botao na interface ao btn (a este Codigo):
        btn = (Button) findViewById(R.id.btn);

        OnClickListener p = new OnClickListener(){
            @override
            public void onClick(View v){
                Context context = getApplicationContext();
                TOast toast = Toast.makeText(context, "Clicou no botao!",
Toast.LENGH_SHORT);
                toast.show();
            }
        }
        btn.setOnClickListener();
```

OU

```
    import android.view.*;
    import android.widgets.*;

    ...

    //a proxima linha liga o botao na interface ao btn (a este Codigo):
    btn = (Button) findViewById(R.id.btn);
    // este formato tem a vantagem de nao alterar todo o metodo e so alterar para
o objeto que queremos
```

```
    btn.setOnClickListener(
        new OnClickListener(){
        @override
        public void onClick(View v){
            Context context = getApplicationContext();
            TOast toast = Toast.makeText(context, "Clicou no botao!",
Toast.LENGH_SHORT);
            toast.show();
            }
        }
    );

    btn2 = (Button) findViewById(R.id.btn2);
    btn2.setOnClickListener(

    );
```

Objetos interativos:

- Conhecidos com *widgets* ou controlos;
- Representação gráfica, nomeadamente botões *on/off*;
- São reutilizáveis e disponibilizados numa biblioteca/*framework*:
    - para *Android*:
        - `android.widget`;
        - `android.view`.
- Dois tipos genéricos:
    - Objetos interativos de **I/O**;
    - Contentores.
- A comunicação entre objetos interativos pode ser feita de 3 maneiras:
    - manipulação direta de propriedades de outros objetos;
    - avisando o elemento imediatamente acima na hierarquia(acerca das alterações a fazer);
    - gerando outros (pseudo-)eventos, já que não são exatamente gerados por humanos.
- O *handler* recebe o próprio objeto interativo como paramêtro de entrada;
- O objeto interativo onde o evento é gerado é designado por fonte.

Programas Sequencias

PROS:

- Funcionam sempre em linha de comandos;
- Aguentam todas as funções que quiserem (à custa de variáveis e de uma estratificação em árvore e divisão de modos).

CONS:

- Pouco intuitivos;
- Funções disponíveis só após mudança de modo;
- Quem controla a interação é o programa.

Padrão Modelo-Visão-Controlador

O padrão de arquitetura para aplicações móvies é o padrão **Modelo-Visão-Controlador**. Implementar esta arquitetura significa normalmente dividir o projeto de uma aplicação em vários ficheiros e pastas, cada um afeto a uma destas partes:

1. Modelo - Esquema de dados, e acesso e manipulação desses dados (*SQL* + *PHP*);
2. Visão - Aspeto visual de aplicação (*HTML* + *CSS*);
3. Controlador - Código que liga o aspeto visual aos dados (*Javascript*, *PHP*).

Tem o objetivo de favorecer a sua escalabilidade e manutenção, bem como a portabilidade e a reutilização de código. Este padrão permite uma maior grau de abstração sendo muito popular para o desenvolvimento de aplicações *web* e móveis.

Sistema Operativo *Android*

A plataforma *Android* é composta por:

- uma pilha de *software*, com várias camadas:
    - é composta por 4 camadas:
        - Camada do *Kernel*(núcleo):
            - camada mais baixa;
            - fornece os serviços base;
            - possui uma arquitetura de permissões(para restringir acesso a recursos);
            - possui mecanismos padrão de gestão de memória e de processos;
            - possui suporte a comunicações em rede;
            - camada em que estam presentes as *drivers*;
            - o *kernel* do *android* difere dos *kernels* para *desktop*.
        - Bibliotecas nativas:
            - segunda camada a contar de baixo;
            - implementadas em `C` ou `C++`;
            - estão encarregues de atividades críticas relacionadas com o sempenho do dispositivo:
                - refrescar o ecrã;
                - renderizar páginas *web*;
                - etc.
            - inclui o ambiente de execução virtual composto por:
                - bibliotecas *Java base*;
                - máquina virtual *Dalvik*.
            - disponibliza um conjunto de classes `Java`.
        - *Framework* aplicacional:
            - contém *software ou recursos que as aplicações *Android* podem necessitar e reutilizar;
            - alguns destes componentes são:
                - gestor de pacotes;
                - gestor de janelas;
                - gestor de recursos;
                - gestor de atividades;
                - provedores de conteúdos;
                - gestor de localização;

            - gestor de notificações.
        - Camada de aplicação:
            - camada de topo;
            - inclui as aplicações que:
                - veem com o sistema;
                - são instaladas pelo utilizador.
            - pode incluir aplicações de desenvolvimento;
            - o utilizador interage com esta camada diretamente.
- um *kit* de desenvolvimento de *software*(SDK);
- extensa documentação.

Aplicações *Android*

Os 4 blocos fundamentais sobre os quais as aplicações *Android* são construidas são:

- Classe `Activities`:
    - é a que permite a contrução de *interfaces* gráficas;
    - uma atividade deve corresponder a uma única ação ou interação que o utilizador pode fazer;
    - são as atividades que dão origem às janelas;
    - uma aplicação vai suportar mais do que uma funcionalidade, provavelmente terá de conter mais do que uma atividade.
- Classe `Services`:
    - é uma componente aplicacional que pode executar operações de longa duração em segundo plano e não fornece uma *interface* de utilizador;
    - um serviço pode ser despoletado(*triggered*) por outra compinente.
- Classe `ContentProviders`:
- estes componentes são usados para fornecer conteúdos estruturados a aplicações de uma forma padrão;
- permite que aplicaçoes comuniquem entre si ou que usem dados que são partilhados por todo o sistema;
- a *interface* para os `ContentProviders` chama-se `ContentResolver`.
- Classe `BroadcastReceiver`:
    - executa e processa eventos;
    - tem o objetivo de permitir que determinada aplicação se registe no sistema como capaz de lidar com determinado evento e, quando esse evento acontece, esta seja chamada pelo sistema operativo para o processar;
    - outras aplicações criam estes eventos através da definição de intentos e difundem-nos para o sistema utilizando um método como o `sendBroadcast()`;
    - são componentes importantes em termos de *performance* e funcionalidade;
    - são estes que permitem que uma determinada aplicação registe no sistema a sua vontade em receber determindados eventos.

Processo de preparação de uma aplicação *Android*

Para uma aplicação *Android* poder ser utilizada num dispositivo *Android*, tem que ser gerado um ficheiro `.apk`. Os passos que o sistema toma para tal são:

1. A ferramenta de empacotamento de recursos *Android*(`aapt`) agrupa os ficheiros com recursos, nomeadamente o `AndroidManifest.xml` e os ficheiros de *layout* e compila-os. Neste passo é também gerado o ficheiro `R.java` que contém referências para os vários recursos, para que possam ser usadas no âmbito da implementação da aplicação e para comodidade do programador;
2. A ferramenta `aidl` converte *interfaces* definidas na linguagem **AIDL** em *interfaces* `Java`;
3. Todo o código `Java`entretanto gerado (`R.java` + *interface*) e de implementação da aplicação é compilado palo `javac` para ficheiros `.class`;
4. A ferramenta `dex` converte os ficheiros gerados no ponto anterior;
5. A ferramenta `apkbuilder`alimenta-se de todos os resursos que foram compilados, bem como os que não são compiláveis para produzir um arquivo `.apk`;
6. O arquivo `.apk`é posteriormente assinado digitalmente;
7. Finalmente, e caso a aplicação esteja a ser assinada para produção, o arquivo `.apk` deve ser ainda alinhado com a ferramenta `zipalign`. Este último passo permite reduzir a utilização de memória aquando da execuçaõ da aplicação num dispositivo *Android*.

Métodos

`onCreate()`

- O método é chamado quando uma atividade é criada pela primeira vez;
- É neste método que a configuração estática deve ser feita, nomeadamente:
  - criação ou ajuste da *interface* de utilizador;
  - ligação de dados e recursos com objetos da interface;
  - a colocação de lógica aplicacional para lidar com eventos em objetos interativos;
  - recuperação do estado anterior.
- O método é chamado com um parâmetro da classe `Bundle`;
- Pode conter o estado da atividade no momento em que esta foi pausada ou parada;
- Este método é sempre seguido de `onStart()`;
- Deve **obrigatoriamente** conter uma chamada a `super.onCreate()` ou será lançada uma exceção, e a atividade poderá não funcionar.

`onStart()`

- É chamado quando a atividade está próxima de ficar visível;
- Torna-se ideal para colocar código que reajuste o estado da aplicação com dados provenientes dos sensores ou guardados no sistema;
- É sempre seguido de `onResume()`;
- Deve **obrigatoriamente** conter uma chamada ao método `super.onStart()`.

`onResume()`

- É *triggered* quando a atividade está a transitar de um estado invisível ou tapada para o primeiro plano;
- É neste método que se devem colocar instruções que inicializem e corram animações ou que toquem sons;
- Depois de executar, a atividade fica no estado de execução e o utilizador pode interagir com ela;
- É sempre seguido de `onPause()`.
- Deve **obrigatoriamente** conter uma chamada ao método `super.onResume()`.

`onPause()`

- É *triggered* quando a atividade está a perder o foco;
- Não se deve fazer nada demasiado moroso no âmbito deste método, dado que o sistema não evolui para a nova atividade enquanto esta não terminar(`return`);
- É normalmente usada para guarda dados persistentes que a atividade esteja a editar;
- Concretiza o lugar ideal para:
    - gerir a paragem de animações e outras operações que requeiram recursos de computação;
    - fechar a transição para a nova atividade;
    - fechar recursos externos que são de acesso exclusivo.
- O armazenamento do estado da aplicação pode normalmente ser feito recorrendo ao método `onSaveInstanceState(Bundle)`;
- É normalmente, mas não necessariamente, seguido de `onStop()`;
- Caso a atividade fique ainda visível, mas em segundo plano, então encontra-se num estado pausado e pode evoluir para `onResume()`;
- Só se a atividade ficar completamente invisível é que o método seguinte é chamado;
- É possível que o sistema mate processos relativos a atividades que foram pausadas, em caso de necessidade expressiva;
- Deve **obrigatoriamente** conter uma chamada ao método `super.onPause()`.

`onStop()`

- È chamdado quando a atividade já não está visível para o utilizador e pode ser utilizado para fazer *caching* de alguns dados para o caso da atividade ser retomada mais à frente;
- As atividades paradas têm uma maior probabilidade de serem terminadas por falta de memória que as pausadas;
- Caso o utilizador volte a navegar para as mesmas, é chamado o método `onRestart()`, seguido de `onStart()` e de `onResume()`;
- Caso a tarefa em que se encontra venha a ser terminada, o fluxo evolui para `onDestroy()`;
- Não se deve aguardar por este método para guardar o estado da atividade, dado que este pode nunca vir a ser *triggered*;
- Deve **obrigatoriamente** conter uma chamada ao método `super.onStop()`.

`onRestart()`

- Apenas é chamado quando uma atividade foi previamente colocada em segundo plano e depois um utilizador volta a navegar para a mesma;
- Deve conter código que permite recuperar dados que hajam sido guardados em `onStop()`;
- Se estão a ser usados ponteiros dinâmicos para recursos do sistema, é aqui que se devem refrescar esse conteúdos;
- Deve **obrigatoriamente** conter uma chamada ao método `super.onRestart()`.

`onDestroy()`

- É invocado quando a atividade está para terminar normalmente, quer programaticamente, quer por ação do utilizador;
- Algumas ações básicas que se devem incluir aqui incluem a libertação de recursos computacionais, nomeadamente *threads* que tenham sido criadas no contexto da atividade;

- Este método pode não ser chamada caso a atividade seja terminada de modo abrupto;
- Não deve conter a implementação de funcionalidades criticas;
- - Deve **obrigatoriamente** conter uma chamada ao método `super.onDestroy()`.

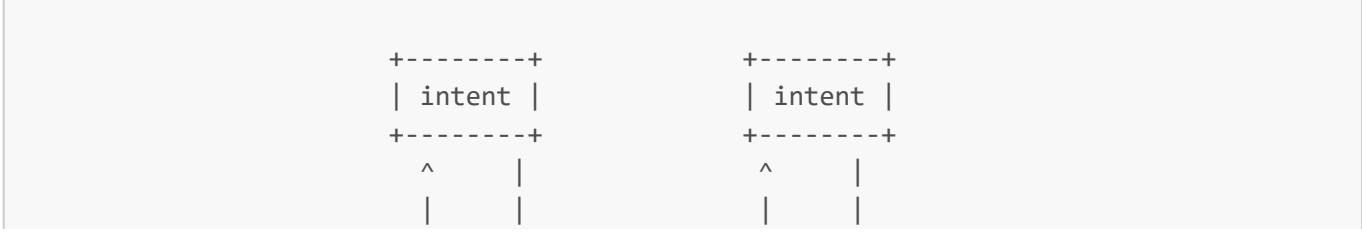
Intentos

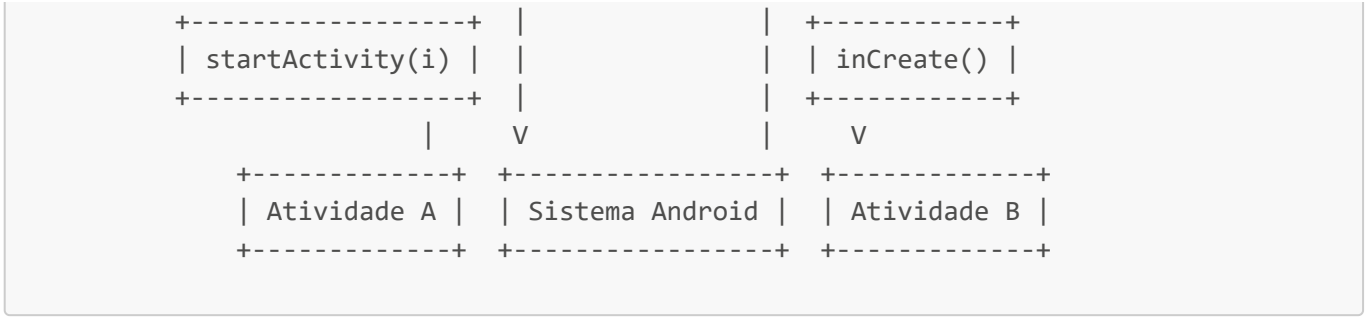
- Intentos são objetos (objetos mensagem);
- É possível dizer: Queria que alguém me abrisse esta imagem (em vez de ser o próprio programa a ter que abrir a imagem);
- Os intentos são enviados (todos) para o sistema operativo;
- São os intentos que permitem ir de uma Atividade para outra;
- Podem ser usados para:
  - dar *trigger* a uma atividade ou um serviço;
  - emitir um evento em difusão.
- Não são usados no contexto da componente `ContentProvider`:
  - É possível:
    - dar *trigger* a uma nova atividade através do método `startActivity(Intent)`, que aceita um intento a definir a ação que deve ser executada;
    - 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)`.
- Os `Services` podem ser executados recorrendo ao método `startService(Intent)`, que também aceita o intento a definir o serviço e eventualmente alguns dados que este deve processar.
- É possível obter uma ligação duradoura a um serviço através de `bindService(Intent, ServiceConnection, int)`, que permirte que um serviço esteja associado à execução de determinada atividade ou outro serviço, sendo terminado quando estes terminarem também;
- É possível emitir *broadcasts* para o sistema através da instanciação de um intento e da sua passagem como parâmetro nos métodos `sendBroadcast()`, `sendOrderedBroadcast()`, ou `sendStickyBroadcast()`.
- Existem dois tipos básicos de intentos:
  1. Intentos explícitos;
  2. Intentos implícitos.

Instanciação de Intentos

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 encontrar o seu destino e ser permitido, de o enviar para execução.

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.





Um intento pode conter:

- o nome do componente destino(opcional);
- a ação a efetuar(especificada através de uma *string* pré-definida);
- os dados(compostos por um *Uniform Resource Locator* e pelo tipo *Multi-Purpose Internet Mail Extensions* do conteúdo para onde aponta);
- a categoria(uma *string* que indica o tipo de componente);
- os extras(que são pares chave-valor usados para transferir dados adicionais);
- *flags*(funcionam como meta-dados para o objeto `intent`).
- Os intentos são os objetos que permitem evoluir de uma componente para outra:

```
//Activity1
    Intent i = new Intent(...);
    startService(i);
    bindService(..., i, ...);

//Activity1
    Intent i = ...
    sendBroadcast(i);
    sendOrderedBroadcast(i);
```

```
Aplicação A           SO              Aplicação B
intento i
i.putExtra("nome", ...);
  putExtra(String, int)
  putExtra(String, double)
  putExtra(String, String)
...
startActivity(i)
    i --------------->[ i ]----------------->i
```

Intentos Explícitos

- Especificam univocamente a componente que deve ser *triggered* pelo nome qualificado no SO;

- São normalmente usados quando se:
  - quer dar *trigger* a outra componente da própria aplicação;
  - sabe exatamente o nome da classe ou atividade destino.
- Quando são criados, o SO imetidatamente dá *trigger* a atividade ou serviço indicados, sem analisar filtros de intentos.

```
import android.content.Intent;
...
Intent i = new Intent(this, Activity2.class);
//                     ^           ^
//                     |           |
//                     |      Nome da atividade 2
//                  contexto (package)
startActivity(i);
```

Trigger de intento que pertence a mesma aplicação que este código;

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

Trigger de intento que **não** pertence a mesma aplicação que este código;

- Quando um intento é definido desta forma, o SO não esboça qualquer tentativa de encontrar as aplicações que o possam tratar.

Intentos Implícitos

- São intentos para os quais não é especificado o nome ou pacote do componente a executar;
- 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;
- São particularmente úteis para quando se quer fazer uso de uma funcionalidade que outra aplicação do sistema possa oferecer, sem especificar exatamente qual;

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

```
Intent i = new Intent(Intent.ACTION_SEND);
i.putExtra(Intent.EXTRA_TEXT, "Estou na aula de PDM!");
startActivity(i);

Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse(imagens/imagem.jpg));
i.setType("img/image");
startActivity(i);
```

```
Intent i = new Intent(NOME e PACKAGE da componente a abrir);
OU
Intent i = new Intent(AÇÂO QUE QUERO FAZER);
```

Para encontrar o componente certo, o SO compara o conteúdo do intento com os filtros de intentos 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 *triggered*. 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 pedaço de código **XML** seguinte mostra o aspeto de elementos `intent-filter` no ficheiro `AndroidManifest.xml`.

```
<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>
```

Caso um programador queria 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 na plataforma**.

Envio de Dados Via Intento

Existem **várias formas de enviar dados** através de um intento:

- Através da **indicação de um URI**;
- 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. O intento pede ao OS que lhe abra qualquer aplicação que permita `VER`, de alguma forma, os dados que lhe está a passar.

```
```

```
    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 ua chave que pode ser usada para devolver o valor colocado no segundo parâmetro** do método no destino.

```
    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, definido na `Activity2` termina o exemplo anterior.

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

De forma mais simplistica:

Na atividade 1

```
    Intent i1 = new Intent("GOTO ACTIVITY2");
    startActivityForResult(i1, request_code);

    ...

    onActivityResult(request_code, result_code, Intent){
        //O tratamento de quando o resultado voltar.
    }
```

Na atividade 2

```
    ...
    Intent iResponse = new Intent();
    iResponse.putExtra("string1", "Hello. How are you?");
```

```
        setResult(RESULT_OK, iResponse);
        super.finish();
```

Toasts

```
        import android.widget.Toast;
        ...
        Toast oToast = new Toast(this);
        oToast.setDuration(Toast.LENGH_LONG);
        oToast.setText("This is a toast message.");
        oToast.show();
        ...
```

Em termos de funcionalidade é equivalente a:

```
        import android.widget.Toast;
        ...
        Toast.makeText(this, "This is a toast message.", Toast.LENGH_LONG).show();
```

As *Toasts* são objetos interativos(*widgets*). As *Toasts* nunca recebem foco e por isso o utilizador nunca pode realmente interagir com elas. As *Toasts* têm duração de 3.5 segundos(`LENGH_LONG`) ou 2 segundos(`LENGH_SHORT`).

Segurança no Android

Permissões

1. Cada vez que instalamos uma aplicação em *Android*, é criado um novo utilizador do sistema. Breves exceções:

   1.1. Todas as aplicações vão assinadas digitalemente com uma chave privada (RSA, *Eliptic Curves*)

   ```
        shared_id = True //no AndroidManifest
   ```

   1.2. Algumas aplicações assinadas com a chave de sistema têm acesso a tudo.

2. Cada vez que correm uma aplicação, esta corre numa máquina virtual Java diferente/**distinta**/única de todas as outras;

3. Cada máquina virtual corre com o ID de utilizador da sua aplicação;

4. Quando se instala uma aplicação, todos os ficheiros dessa aplicação ficam com as seguintes permissões no Linux:

```

```

```
          dono    grupo   outros
1122      rw-      ---     ---
```

SDK Android
Software Development Kit Android

Quando queremos **desenvolver** software, temos de ter o kit de desenvolvimento. Diferente de correr a aplicação. Quando vou desenvolver uma aplicação Java, eu preciso do JDK.

```
> gradle AssembleDebug
# Isto prepara o apk para ser instalado em versão de deputação

> adb install -r app-debug.apk
# Esta aplicação já estava assinada digitalmente

# Processo de preparação da versão RELEASE
> grable build

app-release-unsigned.apk
# 1. assinar
# 2. alinhar

# É preciso ter chaves de developer

jarsigner (v1)
apksigner (v2 e v3)
```

Estamos a fazer uma aplicação. Estamos com vontade de dar acesso a umas das atividades da aplicação

1. Temos de **definir** uma permissão com a *tag permission* mp manifesto:

```
<manifest>
    <permission
    (...)
    />
<application android:permission="Toda">
    <activity android:name="Activity1"
    android:permission="special" />

    <activity android:name="Activity2" />
    <service andrpid:name="Service1" />
    <receivers android:name = "Receiver1" />
    <providers android:name="Provider1" />
</manifest>
```

Estamos a fazer uma aplicação, mas queremos usar os recursos de outra. Há que pedir...

Armazenamento de Dados Persistentes

Umas das funcionalidades mais úteis para a maior parte das aplicações móveis é a de **gerir e armazenar dados de forma persistente**. O SO *Android* disponibiliza diversas formas de o fazer, nomeadamente:

1. Um **recurso/classe** chamado `SharedPreferences`, **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**;
4. Bases de dados ***SQLite***, para **armazenamento e acesso eficiente de dados estruturados** em bases de dados **privadas**;
5. Formas de **acesso à rede**, para **armazenament e gestão de dados remotos**.

Preferências Partilhadas

A implementação da classe `SharedPreferences` 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* que define aquele valor.

A instanciação de um objeto da classe `SharedPreferences` é feita através da invocação do método:

- `getSharedPreferences(string, int)`:
  - caso se pretenda dar um nome ao ficheiro de preferências;
  - caso se pretenda obter o ficheiro previamente guardado com esse nome.
- `getPreferences(int)`:
  - caso só se esteja a usar o ficheiro por defeito;
  - aceita apenas o `int` que define o modo de acesso ao ficheiro.

Ambos os métodos são fornecidos com o contexto da componente em utilização.

As **Preferências Partilhas** são ficheiros `XML`, guardados normalmente na diretoria de dados da aplicação, nomeadamente na diretoria:

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

O pedaço de código a seguinte exemplifica a utilização das preferências partilhadas.

```
    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();
        }
    }
```

Armazenamento Interno

Escrita

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;
  - o modo de acesso com que o ficheiro deve ser guardado ou aberto.
- devolve um objeto da classe `FileOutputStream`.

Os ficheiros criados ou editados conforme descrito nesta seção são eliminados automaticamente pelo sistema aquando da desinstalação da aplicação.

Existem 4 modos de operação que interessa conhecer:

- `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;
- `MODE_wORLD_READABLE`(valor 1), que define que o ficheiro pode ser acedido para leitura por qualquer aplicação;
- `MODE_WORLD_WRITABLE`(valor 2), que especifica que qualquer aplicação pode aceder ao ficheiro para escrita;
- `MODE_APPEND`(valor 32768), que determina que a escrita de dados novos no ficheiro deve ser feita no final, caso este já exista.

A escrita de dados no ficheiro pela aplicação que o criou é possível em qualquer um dos modos indicados antes. 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();
```

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;
- `fileList()` que devolve um vetor de *strings* com o nome de todos os ficheiros já guardados pela aplicação.

Cache

A memória *cache* serve para guardar dados por algum tempo. A maneira como funciona é que os dados são guardados em ficheiros que por si são guardados na subdiretoria `cache` da diretoria de dados da aplicação. O caminho para esta diretoria pode ser obtido no seio da execução através do método `getCacheDir()`. Quando os recursos de armazenamento começam a escassear no sistema, este começará a **eliminar ficheiros** contidos em subdiretorias `cache`. É por isso que não se deve presumir que estes ficheiros estarão sempre disponíveis e sobrevivam entre uma sessão e a seguinte. Devem ser implementados métodos na aplicação que limpem/organizem a diretoria `cache`.

Leitura

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 `int`(representando um `byte`) e itera o cursos de leitura no ficheiro. Por defeito, a aplicação procura o ficheiro a abrir na diretoria `/data/data/nome_pacote_aplicacao/`.

É possível incluir um ficheiro estático no projeto da aplicação. Neste caso, deve-se guardá-lo numa subdiretoria de `res` chamada `raw`. Este nome 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`. **Não é posível escrever para ficheiros guardados em** `res/raw`.

Armazenamento Externo

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(ex.: cartão *Secure Digital*(SD));
- concretizado por uma parte do sistema de ficheiros num dispositivo não amovível(interno).

O que melhor distingue este tipo fe armazenamento é o facto de ficar disponível como uma unidade de armazenamento USB quando o dispositivo é ligado a um computador. Os ficheiros guardados no armazenamento externo **têm permissões de leitura para todos**(`world-readable`), e é possível que um

utilizador os possa modificar via outro dispositivo computacional compatível. **Não deve ser presumido que este estará sempre disponível ou presente**, num que **os ficheiros que aí são guardados se mantêm inalterados de uma execução para outra**. Recomenda-se que qualquer código que manuseie armazenamento externo seja guardado por uma verificação se este realmente existe ou está disponível. O código seguinte mostra a implementação de um método que devolve `true` caso o armazenamento externo esteja disponível **pelo menos 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;
    }
```

Para uma aplicação ter acessoa ao armazenamento externo, tem que normalmente pedir essa permissão no `AndroidManifest.xml`, 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"/>
```

O objetivo do excerto de código seguinte é copiar o conteúdo do ficheiro `fi.txt`, na pasta `res/raw` para o `ficheiro.txt`, alojado no armazenamento externo. A cópia só é tentada depois de ser verificado que a unidade está disponível, nomeandamente 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 `fi.txt`.

```
    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();
        fosFile.close();
    }
```

O método `Environment.getExternalStorageState()` devolve todos os estados ossíveis do armazenamento externo. 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. 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 `getExternalPublicDirectory(Environment.DIRECTORY_PICTURES)`, o médoto devolve o caminho qualificado da diretoria pública que contém as imagens no armazenamento externo na forma de um ficheiro. No caso em que os ficheiros são guardados numa diretoria da aplicação, estes serão eliminados caso a aplicação seja desinstalada.

SQLite Databases

SLQ significa *Structured Query Language*(Linguagem Estruturada de Consultas).

Não se diz como se faz, apenas se diz o que se quer.

Código tradicional:

```
abrir o ficheiro
while(é possivel ler linha)
    ler próxima linha
    testar se o nome Pedro está nessa linha:
        print(dessa linha)
fechar o ficheiro
```

Código SQL:

```
SELECT * FROM User WHERE nome="Pedro"
```

As bases de dados SQL servem para guardar, gerir e aceder a dados de uma forma eficiente.

O SQLite é um **motor** de bases de dados.

```
 SERVIDOR          CLIENTE
+--------+        +--------+
| server |        | mysql  | <-----> User
| mysqld |  <---- | client |
+--------+        +--------+
    |   ^         +--------+                    +---------+
    |   +-------> |  PHP   | <----------------> | Browser |
    |             +--------+                    +---------+
   BD
(Sistema de ficheiros)
```

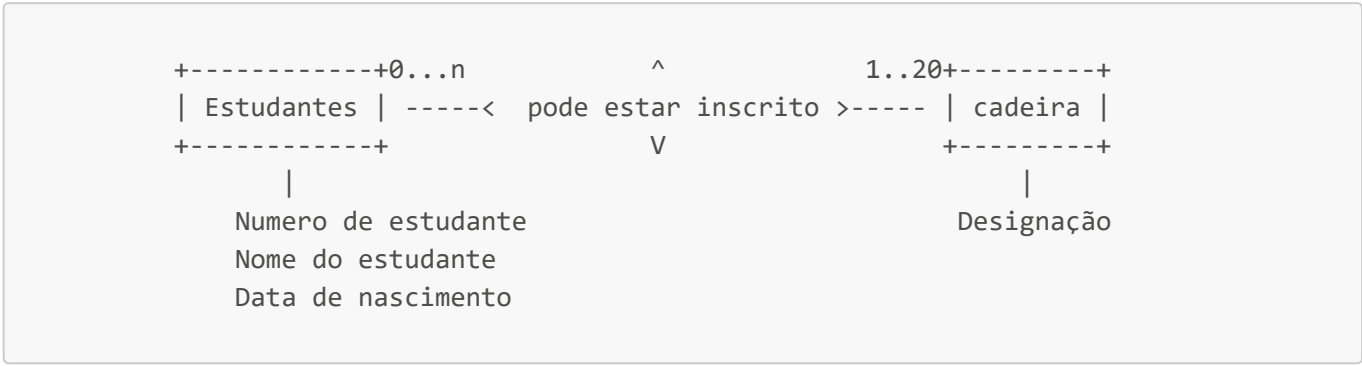
- Não tem servidor;

- Só funciona quando usamos o código.

O sistema de gestão de bases de dados / motor de bases de dados mais instalado do mundo é o SQLite.

Modelo Conceptual

Entididade Relacionameto



Cada cadeira tem um regente.

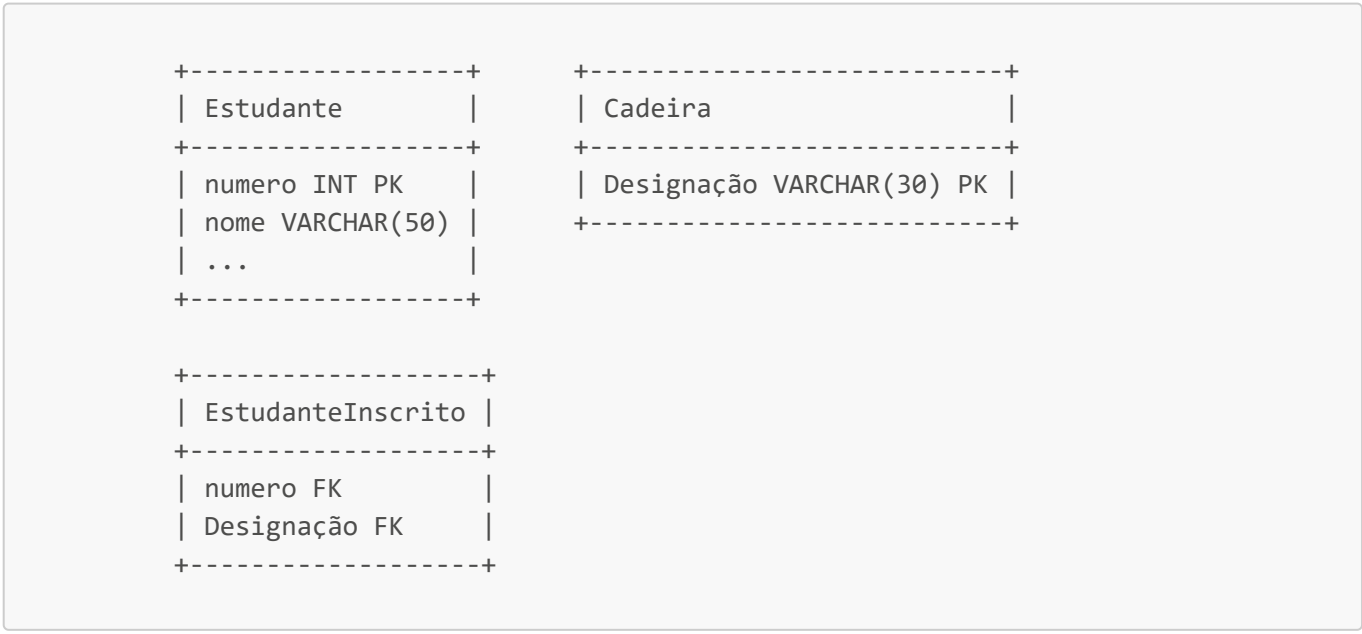
Modelo Físico

O modelo físico já tem preocupações concretas da implementação em computador.

Modelo por Objetos

```
Public Class Estudante
int numero;
String nome;
Date birthday;
```

Modelo Relacional



```
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 stuff

        oAPABD = new AjudanteParaAbrirBaseDados(this);
        oSQLiteDB = oAPABD.getWritableDatabase();

        oSQLiteDB.insert("User", VALUES);
        oSQLiteDB.delete("User", "name=? AND number=?", new String[2] =
{"Pedro","12589"});
        oSQLiteDB.update();
        oSQLiteDB.query();

        ContentValues oCValues = new ContentValues();
        oCValues.put(number, new Integer(12589));
        oCValues.put(name, "Pedro");
        oCValues.put(average, new Double(10));
        oSQLiteBD.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.geDouble(1));
    }
```

Os **Cursores** resolvem o problema de **Impedância**.

```
    @Override
        protected void onPause(){
            super.onPause();
            OSQLiteDB.close;
        }
    @Override
    public void onCreate(SQLiteDataBase db){
```

```
                db.execSQL(STUDENTS_TABLE_CREATE);
            }
            @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);
            }
        }
```

onCreate é o método do SQLiteOpenHelper que é executado quando o programa corre pela primeira vez.

O método **onUpgrade** de um objeto SQLiteOpenHelper é invocado quando a versão da base de dados muda aquando de uma nova atualização

Componente Serviço em *Android*

As aplicações *Android* correm numa máquina virtual `java`. Assim, determinada instância de execução corresponde apenas a uma processo, onde primariamente corre a *main thread*(*UI thread*). Um modelo em que é apenas usada uma única *thread* não pode responder a todos os cenários de utilização de uma aplicação móvel. O uso de ua só *thread* significa que qualquer conjunto de operações é executado de uma forma sequencial, pelo que a presença de uma operação lenta irá incorrer uma situação em que a aplicação deixa de responder. O *Android* define o tempo máximo de reação das suas aplicações para os 5 segundos. Depois disso o sistema assume controlo, mostrando a mensagem *Application not Responding*(ANR) ao utilizador.

A solução passar por colocar diferentes tarefas a executar de forma assíncrona, o que se concretiza em colocá-las a correr em diferentes processos ou diferentes *threads*. A plataforma *Android* suporta o processamento em segundo plano através de **4 formas distintas**:

1. A classe `Threads` está disponível em `java.lang.Thread` e pode ser usada para **processamento assíncrono**. É precisso ter em conta que as *thread* criadas desta forma não podem atualizar a *interface* de utilizador diretamente;
2. A classe `Handler`(`android.os.handler`) permite definir um manípulo na *thread* principal, para o qual se podem enviar mensagens ou código para ser executado. Neste caso, declara-se a tarefa que se quer executar de forma assíncrona dentro de uma nova *thread*, e todas as operações de atualização da *interface* de utilizador são definidas dentro de uma classe `Runnable`, enviadas para serem realizadas pela *thread* principal através do método `post(Runnable)`;
3. A classe `AsyncTask` (`android.os.AsyncTask`) pode ser usada para criar *threads* que facilmente comunicam com a *UI thread*, já que define 4 métodos que podem ser reescritos, sendo que alguns correm na *thread* em segundo plano, enquanto que outros correm na *thread* onde o objeto é criado, que normalmente corresponde à *thread* principal.
4. Finalmente, a classe `Service`(`android.app.Service`).

Definição de Serviço

Um `Service`não é mais do que uma forma de anunciar o desejo de uma aplicação *Android* executar uma operação, ou então definir uma forma de fornecer funcionalidades a outras aplicações, que se podem vincular

ao serviço para obter essas funcionalidades.

- Um serviço **não é um processo separado**;
- Um serviço **não é, nem cria, uma thread*.

A não ser que seja estritamente definido em contrário, um serviço corre no mesmo processo da aplicação. Caso se queira fazer trabalho demorado dentro de um serviço, o programador deve declarar explicitamente uma nova *thread*.

Um `Service` pode ser distinguido de uma `Activity` pelo facto de que o `Service`não tem uma *interface* de utilizador.

Existem dois tipos de serviços: 1.`Started Service`, que são colocados em execução por outro componente através do método `startService()`. Uma vez iniciados, os serviços sem vínculo podem correr indefinidamente em segundo plano, mesmo que o componente que os colocou em execução seja destruído. Os serviços deste tipo fazem apenas uma operação e não devolvem resultados para a componente que os invocou. Depois de cumprirem o seu destino, o serviço deve auto terminar-se; 2. `Bounded Service`, que são colocados em execução ou criado o vínculo através do método `bindService()`. Este tipo de serviço providencia formas definir uma *interface* que permite que outros componentes da mesma aplicação, ou de aplicações diferentes, interajam com o serviço numa arquitetura cliente-servidor, enviando-lje pedidos e obtendo resultados. É possível que várias componentes se liguem simultaneamente a um serviço e este só sobrevive enquanto tiver componentes vinculadas.

Um mesmo Serviço pode funcionar em modo `started` ou `bounded`, dependendo de como é invocado e dos métodos da sua superclasse que são implementados.

Ciclo de Vida de um Serviço

```
        public class ExampleService extends Service{
            @Override
            public void onCreate(){...}

            @Override
            public int onStartCommand(Intent intent, int flags, int startId){...}

            @Override
            public IBinder onBind(Intent intent){...}

            @Override
            public boolean onUnbind(Intent intent){...}

            @Override
            public void onRebind(Intent intent){...}

            @Override
            void onDestroy(){...}
        }
```

O ciclo de vida de um Serviço sem Vínculo tem 3 métodos.

O ciclo de vida de um Serviço com Vínculo tem 4 métodos.

Os Serviços, apesar de correrem sem *interface* gráfica, correm na mesma *thread* (e portanto processo) da *interface* gráfica.

```
        while(true):
            ;

            JVM
             |
             | aplicação a correr
             | Activity, Service
             | new Thread(código)
             +-------+
             |       | código
             |       |
             V       V
          UIThread  thread secundária
                    toast
```

Para os serviços é preciso fazer `@Override` no método `onStartCommand(Intent intent, int flags, int startId)`.

Eu sei que um serviço é começado / serviço sem vínculo quando o método que é implementado é o `onStartCommand`. Para ser um serviço com vínculo, tem

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

        import android.app.Service;
        import java.lang.Thread;
        import java.lang.Runnable;
        ...

        public class ServiceAlarms extends Service {
            @Override
            public int onStartCommand (Intent intent, int flags, int startId) {
                Runnable oTask = new Runnable(){
                    @Override

                    //adicionar gancho

                    public void run(){
                        try{
                                for(int i = 1; i < 11; i++){
                                    gancho.handle(Toast.makeText(this, "Warning #"
+ i, Toast.LENGH_SHORT).show());
                                    Thread.sleep(3000);

                                }
                                stopSelf();
```

```
                        } catch(InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }

                new Thread(oTask).start();

                //Se o processo for morto, nao voltar a tentar o onStartCommand
            }
        }
```

Para implementar um serviço deve ser extendida a classe `Service`.

Started Service

São serviçoes que são começados para fazer qualquer

Para ser um serviço sem vínculo, tem de implementar o `onStartCommand`.

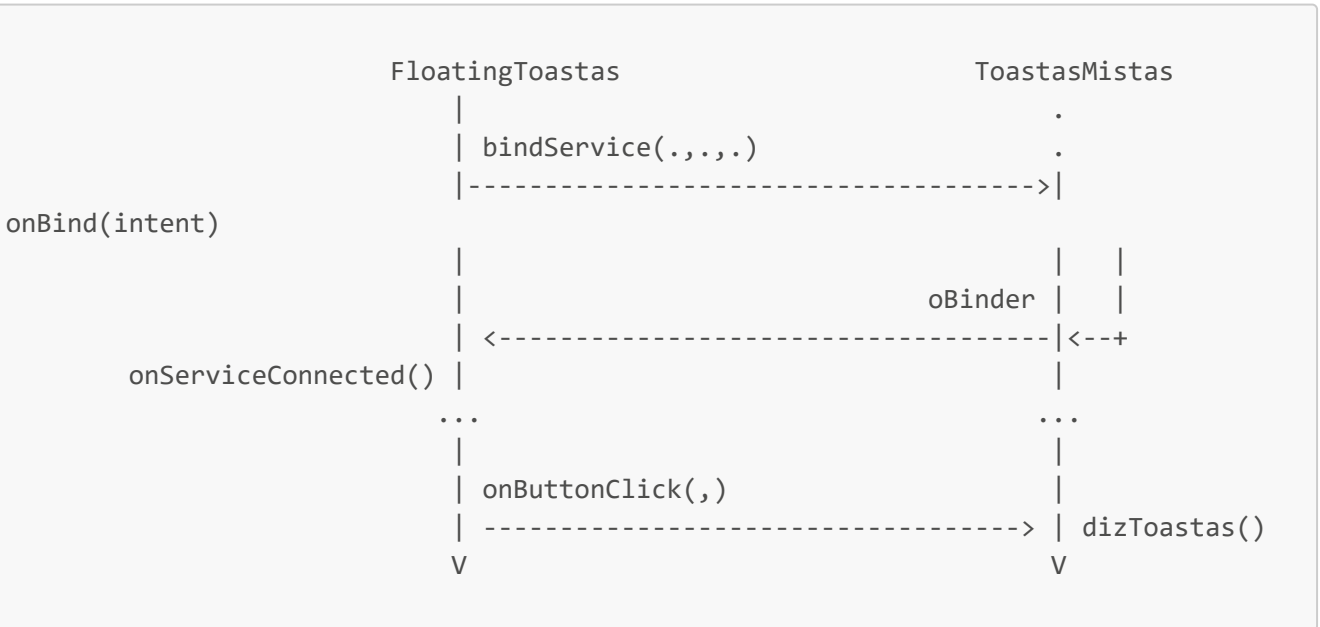
Bound Service

São serviços a que outros blocos das aplicações se podem ligar e desligar.

Uma *Activity* pode-se ligar a um *Bound Service*. Outra *Activity* pode-se ligar ao mesmo *Bound Service*. n *Activities* podem-se ligar ao mesmo *Bound Service*.

As 3 formas básica de definir um vínculo para um Serviço são:

- `Binder`
- `Messanger` e `Handler` (Gancho)
- `Android Interface Definition Language` (AIDL)



Um serviço com vínculo termina quando todos os componentes a ele ligados terminarem.

Os serviços que necessitam do método `stopSelf()` são os Serviços sem vínculo.

Intent Service

```
        public class ServiceAlarms extends IntentService {
            public ServiceAlarms(){
                super("ServiceAlarms");
            }

            @Override
            protected void onHandleIntent(Intent intent) {
                for(int i = 1; i < 11; i++){
                    gancho.handle(Toast.makeText(this, "Warning #" + i,
Toast.LENGH_SHORT).show());
                    Thread.sleep(3000);

                }
            }
        }
```

O código anterior está mal implementado e vai resultar no *Toast* a correr numa *Thread* paralela a *Thread* da *UI*.

A classe `Intent Servce` já chama automaticamente ...

Notificações na Barra de Estado

```
        package pt.di.ubi.pmd.exservice;
        import android.app.Service;
        import android.app.NotificationManager;
        ...

        public class ServiceAlarms extends Service {
            //A proxima variavel define um numero que
            //identifica a notificação e que pode ser
            // usado para mais tarde a atualizar.
            private int iID = 100;

            @Override
            public int onStartCommand(Intent intent, int flags, int startId){
                ...
                NotificationCompat.Builder oBuilder =
                new NotificationCompat.Builder(this)
                .setSmallIcon(R.drawable.icone_notificacao)
                .setContentTitle("Floating Alarms")
                .setContentText("O Servico dos alarmes terminou. Carregue aqui
para o reiniciar.");

                NotificationManager oNM =
(NotificationManager).getSystemService(Context.NOTIFICATION_SERVICE);
```

```
                //O oNM passa a ser uma representação do NotificationManager
dentro da vossa aplicação.
                oNM.notify(iID, oBuilder.build());
            }
        }
```

Recetores de Difusão

Aplicação *Android*

- Activities (têm *interface gráfica*)
- Services (não têm *interface gráfica*)
- *BroadcastReceivers*
- *ContentProviders*

Quero fazer uma aplicação que abra automaticamente quando o sistema acaba de carregar:

1. Implementar essa aplicação, nomeadamente uma atividade de interface;
2. Implementar, dentro do meu projeto, um `BroadcastReceiver`:
    1. Registar o `BroadcastReceiver` no `AndroidManifest.xml`;
    2. Implementar o `BroadcastReceiver` em código, nomeadamente o método `onReceive()`:

```
public class ReceiverForAlarms extends Activity{

    @Override
    onReceive(){
        Intent i = new Intent(this, Main.class);
        startActivity(i);
    }
}
```

```
public class Main extends Activity{
    ORecetor oR = new ORecetor;

    onCreate(){
        super.onCreate();
        ...
        registerReceiver(oR);
    }

    onDestroy(){
        unregisterReceiver(oR);
    }
}
```

Declarar atividade dentro do `AndroidManifest.xml` :

```
        <application>
            <activity>
            ...
            </activity>

            <service>
            ...
            </service>

            <receiver>
            ...
            </receiver>
        </application>
```