

# Programação de Dispositivos Móveis

---

- Programação de Dispositivos Móveis
  - Desenvolvimento de *Interfaces* de Utilizador Orientadas por Eventos(para Dispositivos Móveis)
    - Programas Sequencias
  - Padrão Modelo-Visão-Controlador
  - Sistema Operativo *Android*
  - Aplicações *Android*
    - Processo de preparação de uma aplicação *Android*
  - Métodos
    - *onCreate()*
    - *onStart()*
    - *onResume()*
    - *onPause()*
    - *onStop()*
    - *onRestart()*
    - *onDestroy()*
  - *Intentos*
    - Instanciação de *Intentos*
    - *Intentos* Explícitos
    - *Intentos* Implícitos
    - Envio de Dados Via *Intento*
    - Começar Atividades para Obter Retorno
  - *Toasts*
  - Segurança no *Android*
    - Permissões
  - *SQLite* Databases
    - Modelo Conceptual
      - Entidade Relacionameto
    - Modelo Físico
      - Modelo por Objetos
      - Modelo Relacional
  - Componente Serviço em *Android*
  - Ciclo de Vida de um Serviço
    - *Started Service*
    - *Bound Service*
  - *Intent Service*
  - Notificações na Barra de Estado
  - Recetores de Difusão

## Desenvolvimento 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.widget.*;
...
//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.LENGTH_SHORT);
        toast.show();
    }
}
btn.setOnClickListener();
```

OU

```
import android.view.*;
import android.widget.*;
...

//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.LENGTH_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 parâmetro 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 móveis 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 estão 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 desempenho do dispositivo:
        - refrescar o ecrã;
        - renderizar páginas *web*;
        - etc.
      - inclui o ambiente de execução virtual composto por:
        - bibliotecas *Java base*;
        - máquina virtual *Dalvik*.
      - disponibiliza 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 construídas são:

- Classe **Activities**:
  - é a que permite a construção 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 componente.
- Classe **ContentProviders**:
- estes componentes são usados para fornecer conteúdos estruturados a aplicações de uma forma padrão;
- permite que aplicações 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 registre 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 *intents* 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 registre no sistema a sua vontade em receber determinados 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 pelo **javac** para ficheiros **.class**;
4. A ferramenta **dex** converte os ficheiros gerados no ponto anterior;
5. A ferramenta **apkbuilder** alimenta-se de todos os recursos 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 execução 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()`

- É chamado 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 críticas;
  - Deve **obrigatoriamente** conter uma chamada ao método `super.onDestroy()`.

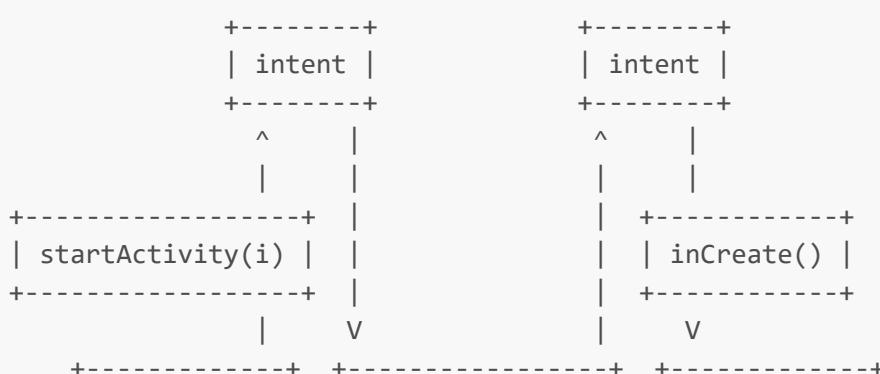
## Intentos

- 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 permite 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.





Atividade A	Sistema Android	Atividade B
+-----+	+-----+	+-----+

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 intents 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 imediatamente dá *trigger* a atividade ou serviço indicados, sem analisar filtros de intents.

```
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 intents 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(iSendMsg.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(AÇÃO QUE QUERO FAZER);
```

Para encontrar o componente certo, o SO compara o conteúdo do intento com os filtros de intents 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 intents 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`.

## Começar Atividades para Obter Retorno

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.LENGTH_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.LENGTH_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(*LENGTH\_LONG*) ou 2 segundos(*LENGTH\_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 digitalmente com uma chave privada (RSA, *Elliptic 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 depuração

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

# Processo de preparação da versão RELEASE
> gradle 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* no manifesto:

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

    <activity android:name="Activity2" />
```

```
<service android: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...

## SQLite Databases

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

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

Código tradicional:

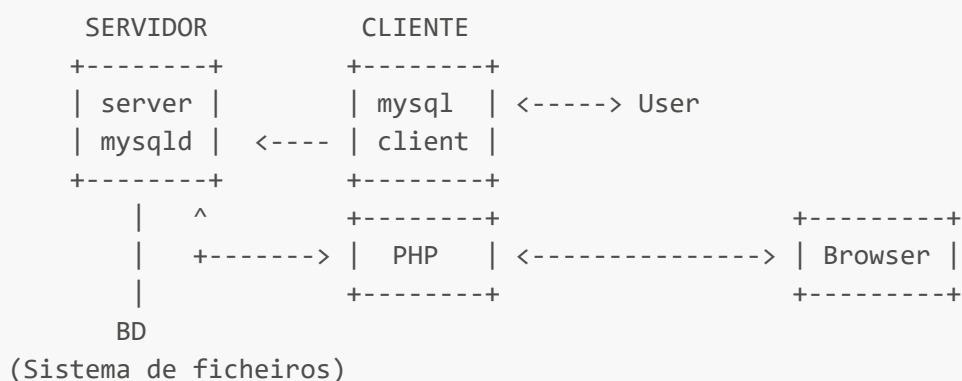
```
abrir o ficheiro
while(é possível ler linha)
    ler próxima linha
    testar se o nome Pedro está nessa linha:
        print(dessa linha)
fechar o ficheiro
```

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.

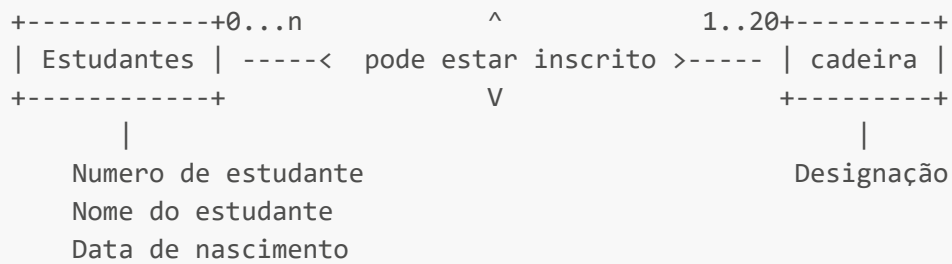


- Não tem servidor;
- Só funciona quando usamos o código.

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

## Modelo Conceptual

### Entidade Relacionameto



Cada cadeira tem um regente.

### Modelo Físico

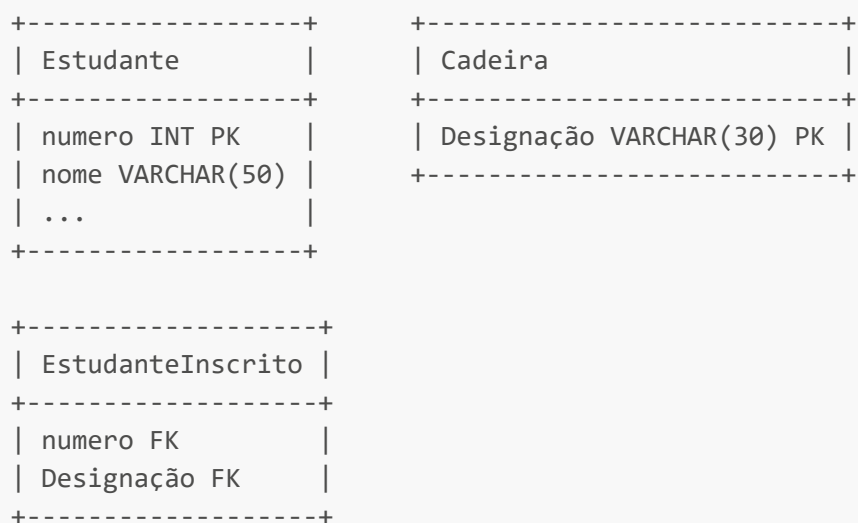
O modelo físico já tem preocupações concretas da implementação em computador.

### 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 oSQLiteDatabase;
    private AjudanteParaAbrirBaseDados oAPABD;

    @Override
    protected void onCreate(Bundle state){
        super.onCreate(state);
        setContentView(R.layout.main);
        oTextView1 = (TextView) findViewById(R.id.name);
        oTextView2 = (TextView) findViewById(R.id.avg);
        // other stuff

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

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

        ContentValues oCValues = new ContentValues();
        oCValues.put(number, new Integer(12589));
        oCValues.put(name, "Pedro");
        oCValues.put(average, new Double(10));
        oSQLiteDatabase.insert(oAPABD.TABLE_NAME1, null, oCValues);

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

        oCursor.moveToFirst();
        oTextView1.setText(oCursor.getString(0));
        oTextView2.setText(""+oCursor.getDouble(1));
    }
}

```

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

```

@Override
protected void onPause(){
    super.onPause();
    oSQLiteDatabase.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

Programa / aplicação / *software* parado num disco, que não está em execução.

Ao executar, o Programa / aplicações ou *software* transforma-se num ou mais processos e *threads*.

Thread e processo principal  
UIThread

## 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 onBind(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.LENGTH_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 serviços que são começados para fazer qualquer

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

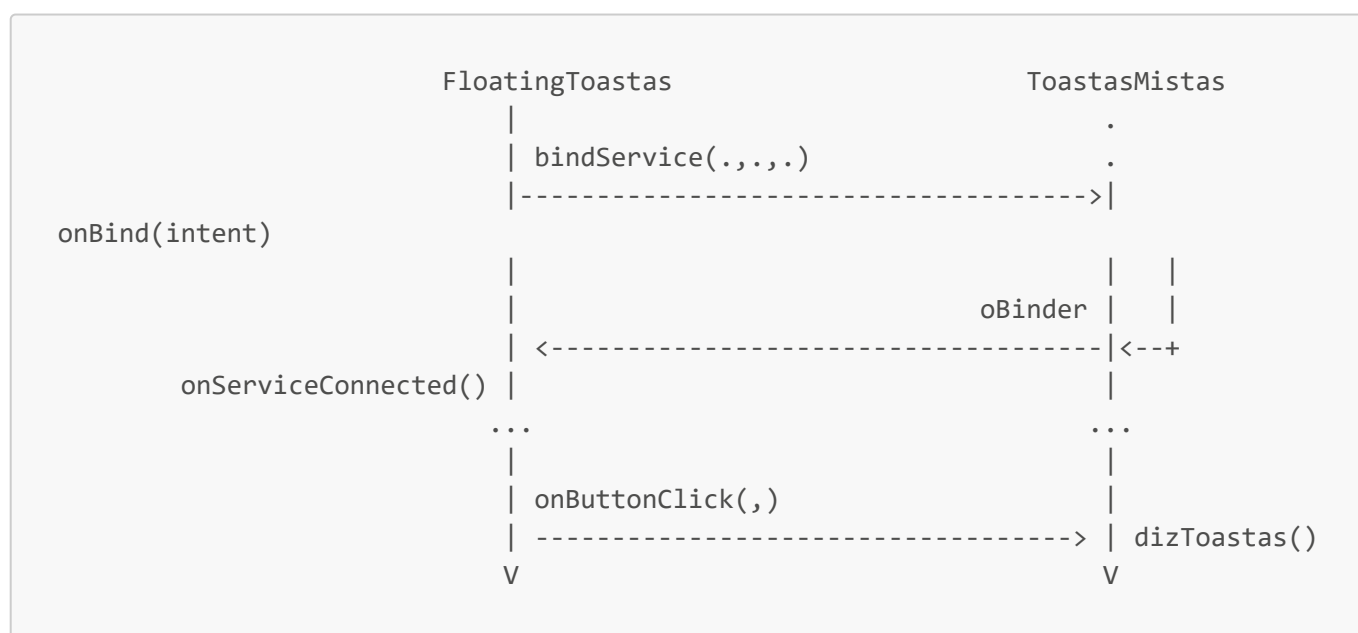
### Bound Service

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**
- **Messenger** 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.LENGTH_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 Service* 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. Registrar 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>
```