

Android

Android

- # Conjunto de *software* que inclui o sistema operativo, *middleware* e aplicações
 - # Baseado em Linux (inicialmente versão 2.6)
 - # A partir da versão 4 do *Android* passou a ser usada a versão 3
 - # A partir da versão 7 do *Android* passou a ser usada a versão 4
 - # A partir da versão 11 do *Android* passou a ser usada a versão 5
 - # Suporte para as aplicações e tecnologias mais recentes na área dos dispositivos móveis (fotografia, vídeo, GPS, bússola, acelerómetro, jogos, OpenGL, SQLite... e telemóvel)



Arquitectura



System Apps

- # Conjunto de aplicações que oferecem as funcionalidades básicas esperadas por um utilizador final: Calendário, SMS, Contactos, *Browser*, ...

Java API Framework

- # APIs Java que dão acesso às funcionalidades essenciais para desenvolver aplicações para a plataforma *Android*

Native C/C++ Libraries

- # Bibliotecas C/C++ nativas que dão acesso a funcionalidades de mais baixo nível, por exemplo *OpenGL ES*, *SQLite*

- # Usadas pelas bibliotecas Java

Android Runtime

- # Máquina virtual no contexto da qual as aplicações são executadas

- # Bibliotecas com as funcionalidades básicas do Java

Hardware Abstraction Layer (HAL)

- # Camada de normalização das especificidades do hardware (*camera*, sensores, ...)

Linux Kernel

- # *Threading*, gestão de memória, Sistema de ficheiros,

...

Arquitectura – Android Runtime

Constituído por:

Conjunto de bibliotecas base similares às bibliotecas de suporte da linguagem Java

Runtime Environment (“Máquina virtual”)

Dalvick Virtual Machine (\leq API 19)

Just-In-Time (JIT) compiler

O código é compilado quando é necessário executá-lo

ART (\geq API 21 – experimental na API 19)

Ahead-Of-Time (AOT) compiler

O código é compilado em avanço

Também está disponível o JIT

Garbage Collection otimizada

Java API Framework

Sistema modular de componentes e serviços

Content Providers

- # Permite a partilha ou o acesso a certo tipo de dados do sistema e de outras aplicações (por exemplo, Contactos)

View System

- # Sistema de gestão de todas as vistas que constituem o UI (botões, listas, caixas de texto, ...)

Managers

Activity

- # Responsável pela gestão do ciclo de vida das atividades/aplicações e *back stack*

Location

- # Permite o acesso a GPS e outros sistemas de localização

Package

- # Responsável por gerir os diversos pacotes de software instalados no dispositivo

Notification

- # Responsável pela gestão da barra de notificação, permitindo que as diversas aplicações apresentem notificações ao utilizador

Resource

- # Efetua a gestão dos recursos definidos. Por exemplo:

- # escolher o melhor layout para aplicação dependendo da dimensão do ecrã, orientação do dispositivo, ...

- # adequação das mensagens à língua definida no dispositivo

Telephony

- # Implementação de todas funcionalidades relacionadas as comunicações móveis

Window

- # Gestão de todas as janelas das aplicações e eventos relacionados

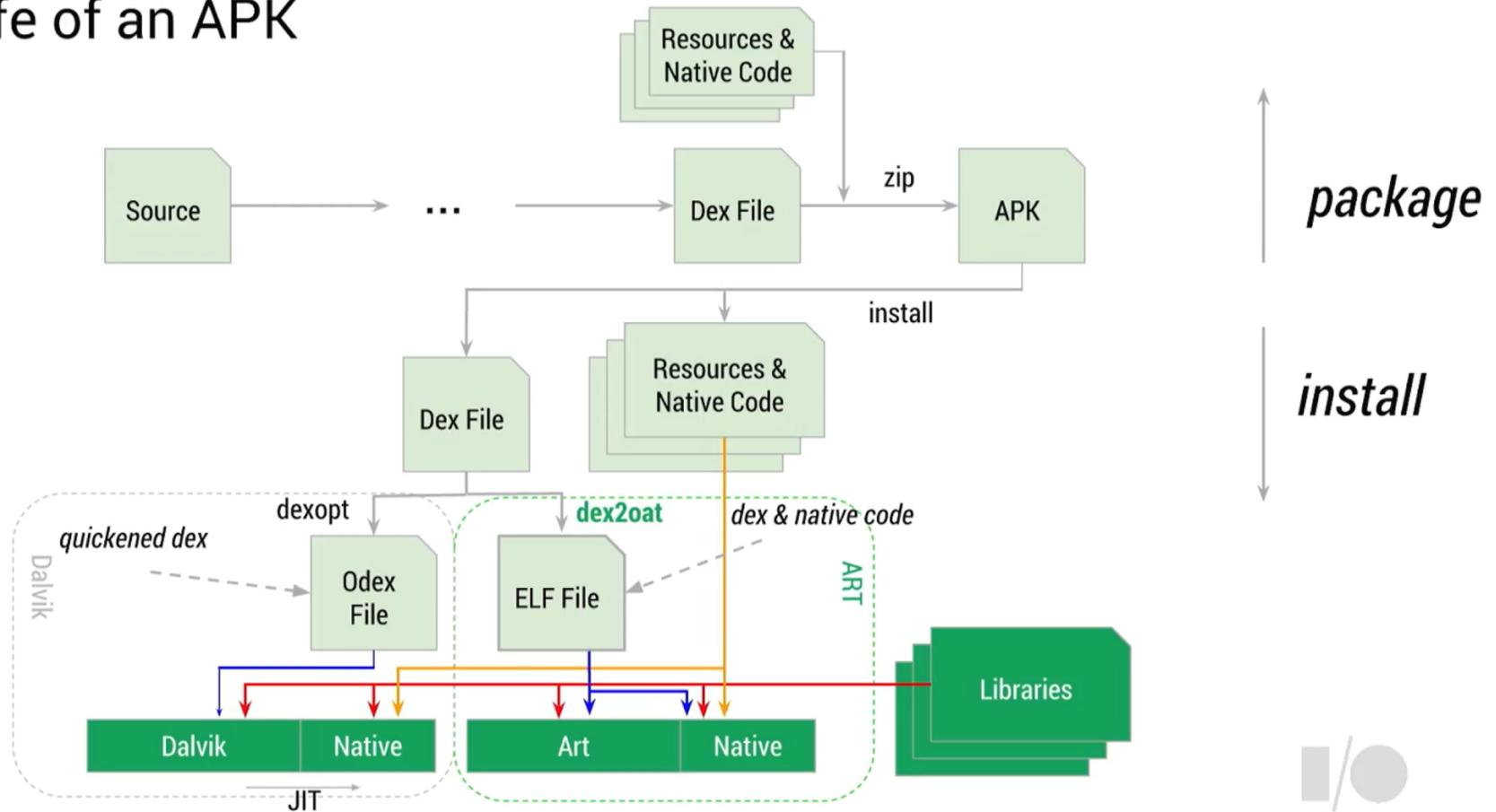
...

Aplicações Android

- # Linguagem *Java* e *Kotlin*
- # O código é compilado e é criado um *package* com extensão *.apk*
 - # No *apk* são também incluídos recursos adicionais
 - # À reunião de todos os componentes num *package apk* dá-se o nome de aplicação
 - # Mais recentemente surgiu a possibilidade de outra forma de distribuição de aplicações designada por *Bundles* (*.aab*)
 - # Este formato irá ser obrigatório a partir do final de 2021
- # Cada aplicação
 - # É executada num processo Linux independente
 - # Possui uma (instância da) máquina virtual independente das outras aplicações em execução

Ficheiros apk

The life of an APK



Componentes de uma aplicação

- # As aplicações não possuem um ponto único de entrada
 - # Possuem um conjunto de componentes que podem ser acionados sempre que necessário (por outro componente da aplicação ou outra aplicação)
- # Tipos de componentes
 - # *Activity*
 - # *Service*
 - # *Broadcast Receiver*
 - # *Content Provider*

Activity

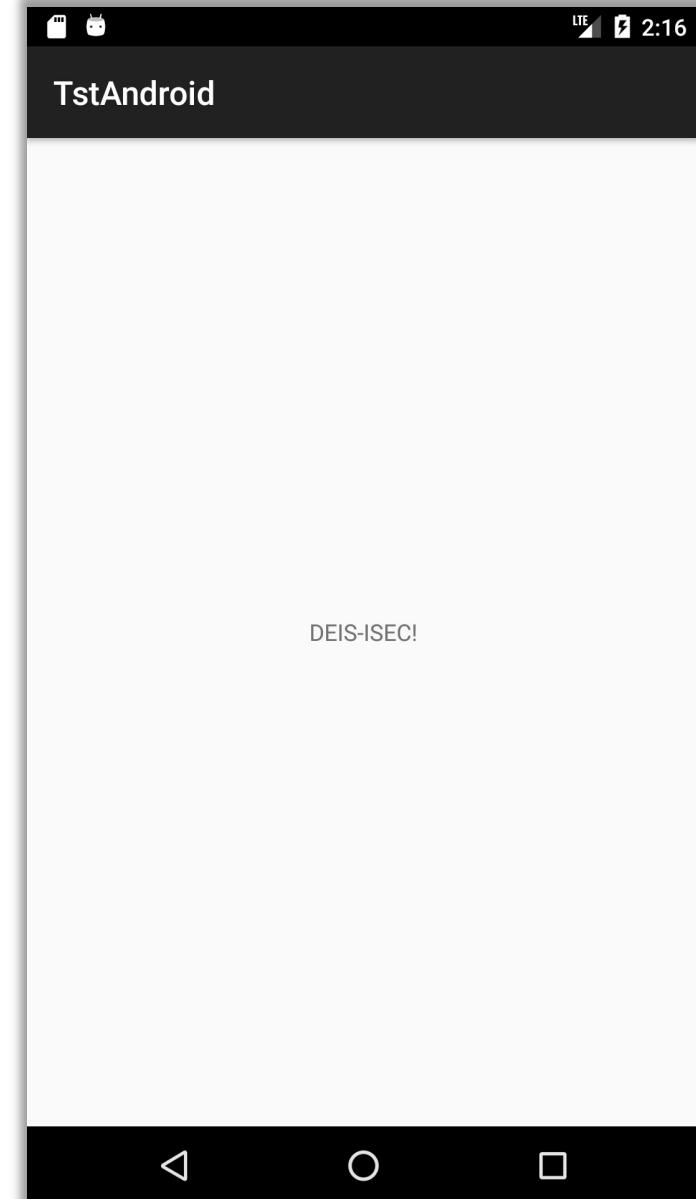
- # Este tipo de componente é caracterizado por ter uma interface visual
- # Uma aplicação pode conter várias *Activities*, classes descendentes da classe *Activity*, sendo independentes entre si
 - # Uma *Activity* pode ser classificada como sendo a primeira a ser apresentada pela aplicação
- # Normalmente ocupa todo o ecrã, mas podem existir casos em que ocupa menos espaço ou pode recorrer a janelas adicionais para pedir ou mostrar algum tipo de informação

Activity

Para criar uma atividade é necessário

- # Derivar uma classe de Activity
- # Definir os componentes visuais
- # Registar a atividade

```
class MainActivity : Activity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
}
```



Activity

- # Os conteúdos são organizados segundo uma hierarquia de objetos derivados da classe *View*
 - # Cada objeto é responsável por organizar o *layout* dos objetos dependentes (pertencentes à hierarquia)
 - # Existem objetos com diferentes objetivos
 - # Organizar outros objetos
 - # Objetos *Layout*: *AbsoluteLayout*, *FrameLayout*, *LinearLayout*, *RelativeLayout*, *ConstraintLayout*...
 - # Interagir com o utilizador (apresentação e pedido de informação)
 - # *Button*, *TextView*, *EditText*, *CheckBox*, *RadioButton*, *TimePicker*, ...
 - # A hierarquia de objetos é atribuída a uma atividade através do método `Activity.setContentView(<...>)`

Service

- # Não possui interface gráfico
 - # Usado para implementação de funcionalidades que não necessitam de pedir ou mostrar informação ao utilizador
 - # As aplicações podem recorrer a componentes *Activity* para esse fim
- # Executa em *background*
- # As classes deverão derivar da classe *Service*
- # Disponibiliza uma interface (conjunto de métodos; “API”) para que os outros componentes possam interagir com ele

```
class MyService : Service() {  
  
    override fun onBind(intent: Intent): IBinder {  
        TODO("Not yet implemented")  
    }  
}
```

Broadcast Receiver

- # Componente que permite esperar por mensagens/anúncios e reagir de forma adequada
 - # Exemplo de anúncios originados pelo sistema
 - # Nível da bateria
 - # Mudança da língua
 - # Mudança do fuso horário
 - # ...
 - # As aplicações também podem gerar anúncios (*broadcasts*)
- # Derivam da classe *BroadcastReceiver*

Broadcast Receiver

- # Uma aplicação pode conter vários *Broadcast Receivers*
- # Tal como os serviços, não possuem interface gráfico embora possam lançar atividades com esse fim
 - # Podem usar a classe `NotificationManager` para sinalizar algo (através de vibração, sons, alteração da iluminação)
 - # Podem mostrar as mensagens através de ícones adequados na barra de estado

Broadcast Receiver

Classe derivada da classe BroadcastReceiver

```
class MyReceiver : BroadcastReceiver() {  
  
    override fun onReceive(context: Context, intent: Intent) {  
        // This method is called when the BroadcastReceiver  
        // is receiving an Intent broadcast.  
        TODO("MyReceiver.onReceive() is not implemented")  
    }  
}
```

Content Provider

- # Permite a disponibilização de informação a outras aplicações
- # A informação pode ser ...
 - # gerada com base no pedido
 - # acedida a partir de uma fonte de dados adequada
 - # Não inclui mecanismos próprios de persistência de dados
 - # Os dados podem ser armazenados em ficheiros, bases de dados SQLite, consultados através de meios de comunicação, ...

Content Provider

Deriva da classe ContentProvider

Implementa um conjunto de métodos que permitem às outras aplicações a consulta e armazenamento de informação

Estes métodos são invocados pelas outras aplicações através de um objeto ContentResolver

Existem alguns *Content Provider* disponibilizados pelo sistema que permitem o acesso a informações geridas pelo mesmo

Por exemplo: Contatos, Tarefas, Imagens, Vídeos, ...

Content Provider

Classe derivada da classe ContentProvider

```
class MyContentProvider : ContentProvider() {  
  
    override fun onCreate(): Boolean {...}  
  
    override fun delete(uri: Uri, selection: String?,  
                       selectionArgs: Array<String>?): Int {...}  
  
    override fun getType(uri: Uri): String? {...}  
  
    override fun insert(uri: Uri, values: ContentValues?): Uri? {...}  
  
    override fun query(uri: Uri, projection: Array<String>?, selection: String?,  
                      selectionArgs: Array<String>?, sortOrder: String?): Cursor? {...}  
  
    override fun update(uri: Uri, values: ContentValues?, selection: String?,  
                       selectionArgs: Array<String>?): Int {...}  
}
```

Ativação de componentes: Intents

- # Como referido, a ativação de um *Content Provider* é efectuada quando um pedido é enviado através de um objeto ContentResolver
- # Os outros tipos de componentes são ativados através de mensagens assíncronas a que se dá o nome de *Intents*
 - # Um objecto Intent permite
 - # descrever o objetivo da mensagem
 - # passar valores/parâmetros entre os componentes
 - # Funções putExtra(key,value) e get<type>(key) para associar pares atributo valor com o Intent ou aceder aos valores previamente guardados (similar a um *hashmap*)

Intents

Os *Intents* podem ser

Explícitos

- # Na criação do Intent é indicada o tipo de classe de objetos que implementa o componente a ativar

Implícitos

- # Na criação do Intent indica-se a ação que se pretende executar

- # Exemplo “mostrar um mapa”, “enviar uma mensagem”, “tirar uma foto”, ...

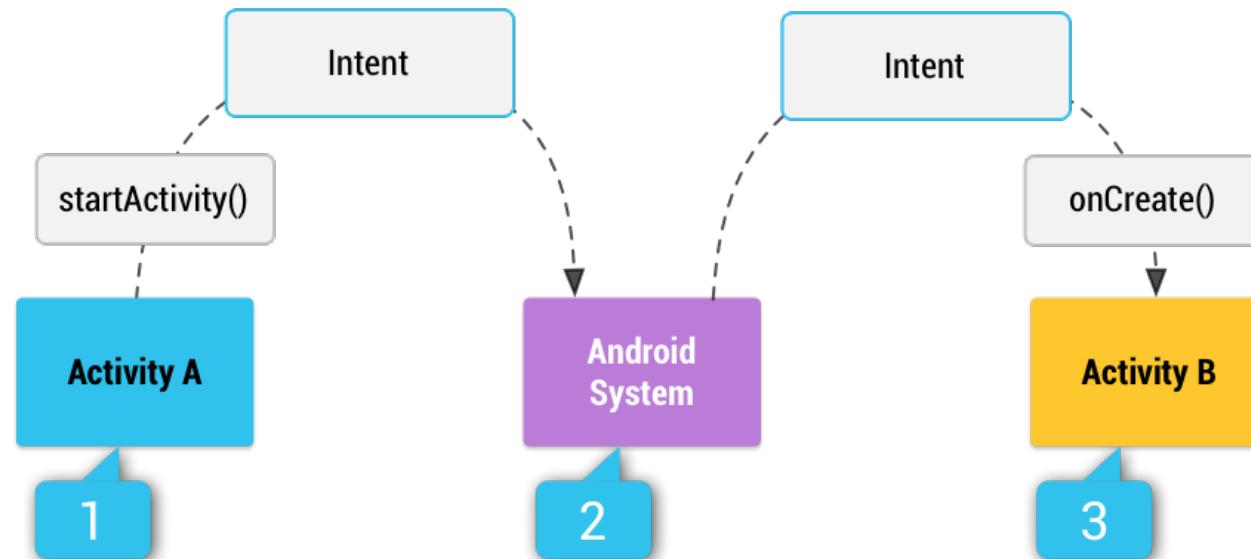
Lista de ações mais comuns

<https://developer.android.com/guide/components/intents-common.html>

Intents

É o sistema que cria e inicia a execução do componente indicado através do *Intent*

1. Um componente cria um objecto Intent com a ação
2. O sistema *Android* procura a aplicação que pode satisfazer o pedido
3. O sistema inicia o componente (cria o objeto e chama o `onCreate()`) e passa-lhe o Intent



Activação de componentes: Intents

Activity

- # Passar um objecto *Intent* através dos métodos `Context.startActivity()` ou `Activity.startActivityForResult()`
 - # O resultado pode ser acedido através do método `onActivityResult`
 - # A actividade lançada pode consultar o *intent* que lhe deu origem com o método `getIntent()`

Exemplos

Exemplo 1:

```
val intent = Intent(this,SomeActivity::class.java)
startActivity(intent)
```

Exemplo 2:

```
// Create the text message with a string
val sendIntent = Intent()
sendIntent.action = Intent.ACTION_SEND
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage)
sendIntent.type = "text/plain"
```

```
// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(packageManager) != null) {
    startActivity(sendIntent)
}
```

Intents – lista de aplicações

- # Quando existe mais do que uma opção para satisfazer o pedido é mostrada uma lista de possíveis aplicações
 - # A escolha é da responsabilidade do utilizador

- # Como forçar a apresentação da lista...

```
val sendIntent = Intent(Intent.ACTION_SEND)
val title = "Send to..."
// Create intent to show the chooser dialog
val chooser: Intent = Intent.createChooser(sendIntent, title)
// Verify the original intent will resolve to at least one activity
if (sendIntent.resolveActivity(packageManager) != null) {
    startActivity(chooser)
}
```

Activação de componentes: Intents

Activity

- # Uma actividade pode iniciar uma outra e ficar à espera de um resultado
- # Usar: `startActivityForResult(<intent>, <intID>)`
 - # O intID deve ser um *id* único para aquela actividade.
 - # Quando a segunda actividade terminar “regressa” à primeira e é chamado o método `onActivityResult`
 - # Nos parâmetros desta função são recebidos o *id*, o código de erro/sucesso e um *intent* com dados adicionais

Activação de componentes: Intents

Service

Passar um objecto Intent através do método
Context.startService(<intent>)

Broadcast receiver

Passar um objecto Intent aos métodos
Context.sendBroadCast(...),
Context.sendOrderedBroadcast(...) ou
Context.sendStickyBroadcast(...)

(Finalização de componentes)

Activity

- # `finish()`
- # Uma actividade pode finalizar outra (iniciada através do método `startActivityForResult`) usando o método `finishActivity()`

Service

- # `stopSelf()`
- # `Context.stopService()`

Broadcast receiver

- # Não é necessário terminar uma vez que só está activo enquanto está a responder a uma mensagem

Content provider

- # Não é necessário terminar uma vez que só está activo enquanto está a responder a um pedido

Ficheiro de manifesto

- # Ficheiro com nome AndroidManifest.xml
- # Possui informação (*meta-data*) essencial ao sistema, para saber
 - # “Que a aplicação existe” (segue um determinado protocolo)
 - # Os componentes que a constituem
 - # As permissões necessárias
 - # Características mínimas dos dispositivos, ...

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
    <application . . . >
        <activity android:name="com.example.project.FreneticActivity"
                  android:icon="@drawable/small_pic.png"
                  android:label="@string/freneticLabel"
                  . . . >
            </activity>
            . . .
        </application>
    </manifest>
```

Ficheiro de manifesto

Exemplos de *tags*

Definir/pedir permissões

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

Definir características mínimas

```
<uses-feature android:name="android.hardware.bluetooth" />
```

```
<uses-feature android:name="android.hardware.camera" />
```

Definição de componentes da aplicação

Activity: `<activity ... />`

Service/IntentService: `<service ... />`

Broadcast Receiver/Widget: `<receiver ... />`

Content Provider: `<provider ... />`

intent-filter

- # Normalmente são definidos no ficheiro *manifest*, no contexto do componente ao qual se aplica
 - # Também podem ser definidos em *runtime*
- # São usados para informar o sistema sobre o tipo de *intents* que são suportados pelo componente
 - # Podem existir vários *intents* para cada componente
 - # Se um componente não definir *intents* então só pode ser activado explicitamente

intent-filter

No exemplo seguinte são apresentados dois *intent-filters*

- # O primeiro *intent-filter* é comum em todas as aplicações tendo por objectivo indicar a actividade a ser activada em primeiro lugar (*entry point*)
- # O segundo *intent-filter* declara um tipo de acção que a actividade pode realizar para um tipo específico de dados

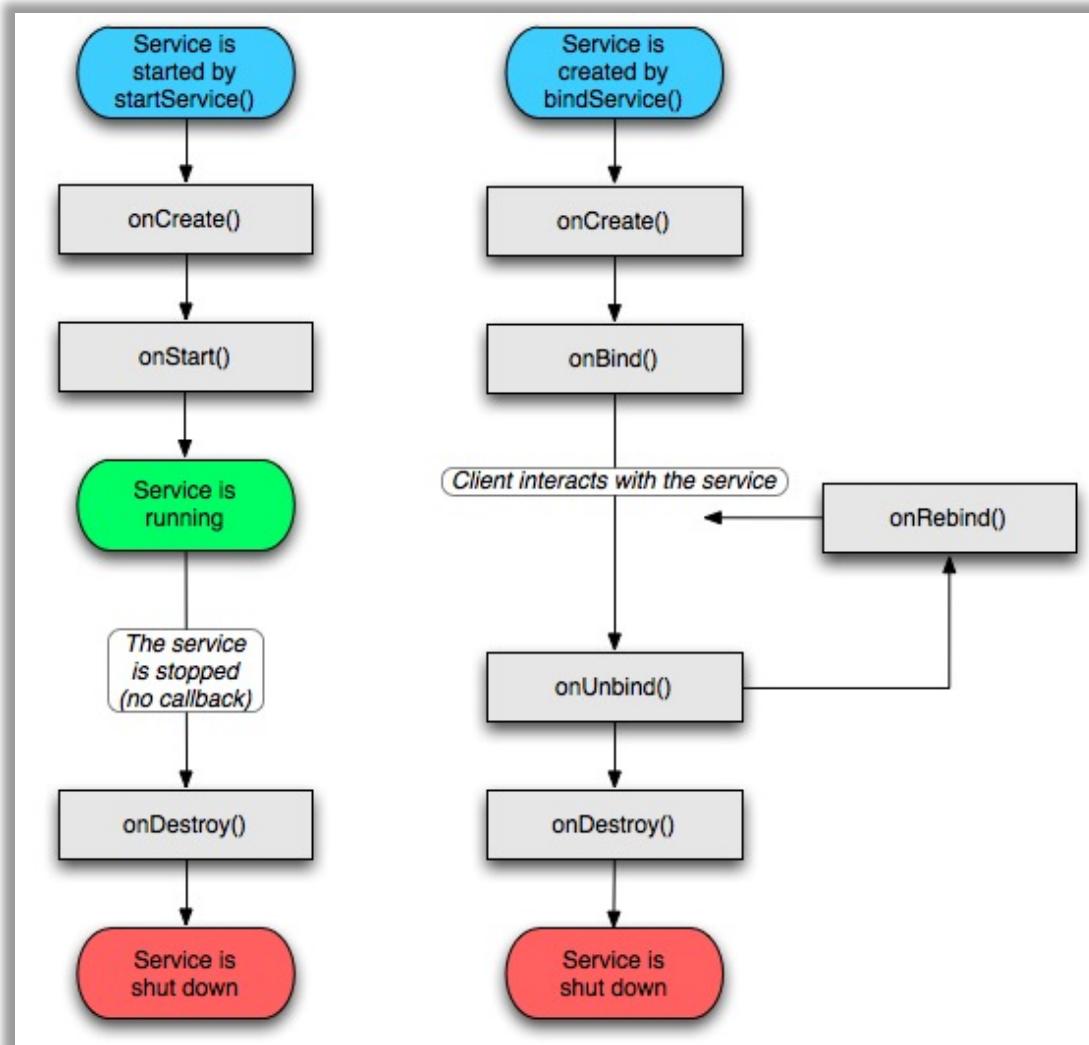
```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
    <application . . . >
        <activity android:name="com.example.project.FreneticActivity"
                  android:icon="@drawable/small_pic.png"
                  android:label="@string/freneticLabel"
                  . . . >
            <intent-filter . . . >
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter . . . >
                <action android:name="com.example.project.BOUNCE" />
                <data android:mimeType="image/jpeg" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
        . . .
    </application>
</manifest>
```

Ciclo de vida (BR e CP)

- # Os ciclos de vida dos *Broadcast Receivers* (BR) e *Content Providers* (CP) correspondem à execução dos métodos, em resposta aos pedidos efectuados
 - # Serão analisados mais tarde aquando do estudo mais aprofundado destes componentes

Ciclo de vida (Service)

(O serviço será estudado mais tarde em mais pormenor)



Ciclo de vida (Activity)

Estados possíveis:

Created

- # A actividade é criada e está disponível para as configurações iniciais

Started

- # A actividade está visível mas não permite interacção

Resumed, Active ou Running

- # A actividade está em primeiro plano no ecrã e tem o focus dos comandos do utilizador

Paused

- # Quando perde o *focus* (pode estar ainda visível)
- # A actividade pode ser finalizada pelo sistema em caso de necessidade de recursos

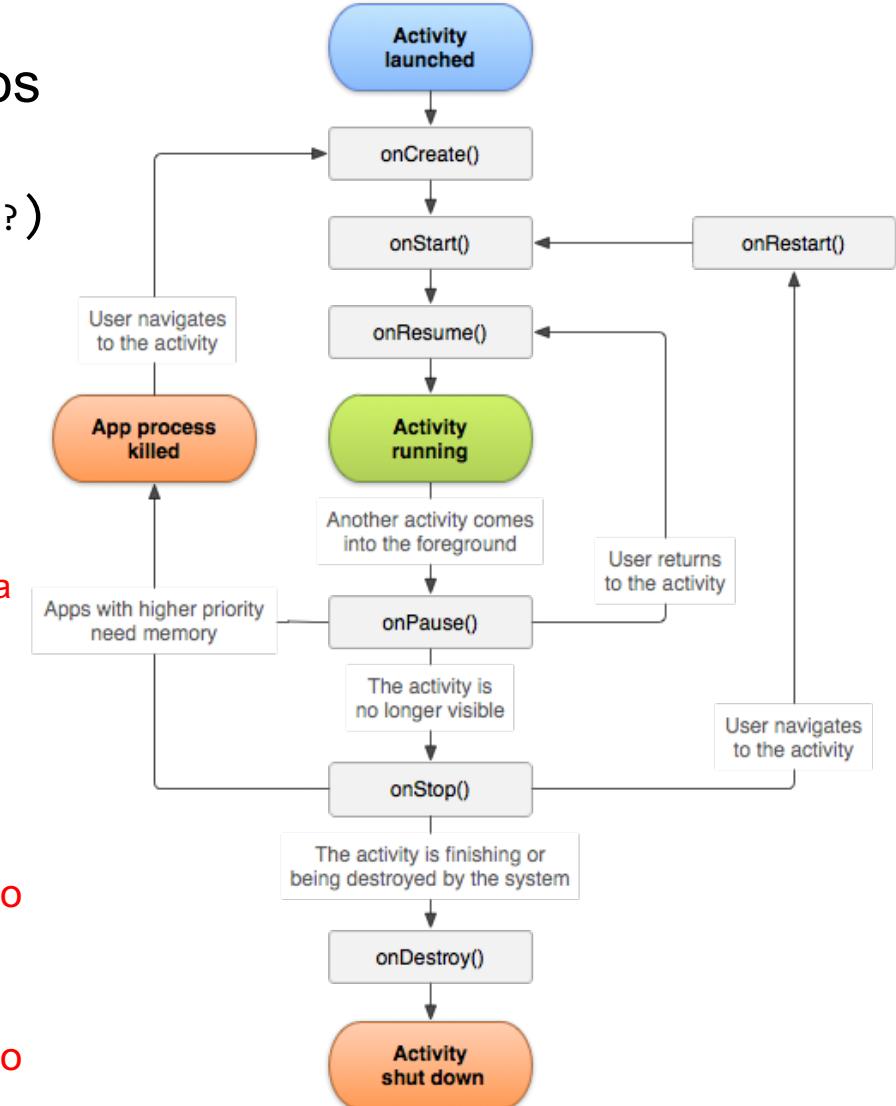
Stopped

- # Não está visível
- # Pode ainda manter informação

Ciclo de vida de uma Actividade

- # A alteração de estado é sinalizada através da chamada a diversos métodos que podem ser redefinidos

```
# fun onCreate(savedInstanceState : Bundle?)  
# fun onRestart()  
# fun onStart()  
# fun onResume()  
# fun onPause()  
    # Salvaguardar dados  
        # Este é o único método de que existe a garantia que é chamado num processo de finalização  
        # Parar processamentos que consomem CPU  
# fun onStop()  
    # Deixa de estar visível  
    # Salvaguardar dados  
    # Pode não ser chamado em caso de finalização abrupta  
# fun onDestroy()  
    # Pode não ser chamado em caso de finalização abrupta
```

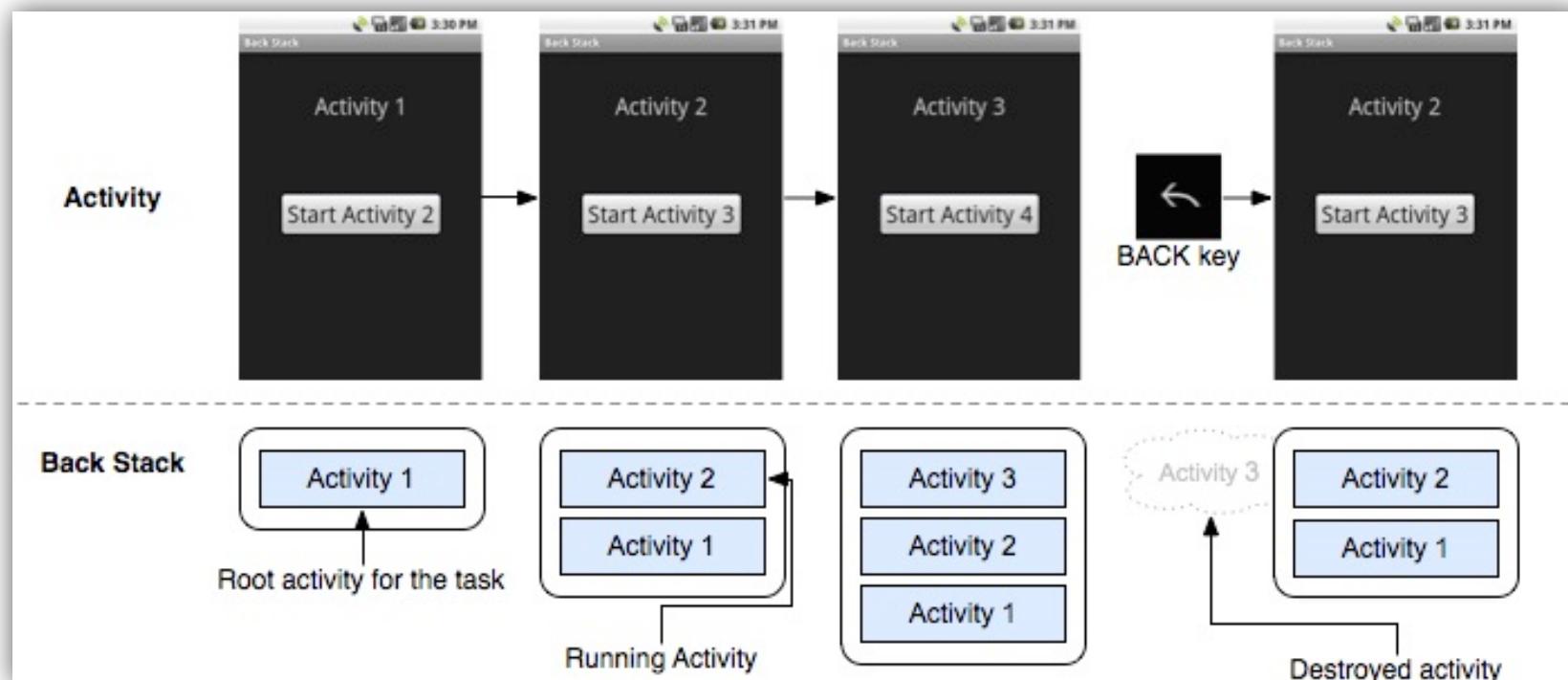


Actividades: salvaguarda de estado

- # Para salvaguardar o estado e informações necessárias deve ser redefinido o método `onSaveInstanceState()`
 - # Deve ser sempre chamado o método da classe base
 - # Possui um argumento `Bundle` onde podem ser armazenados os dados relevantes da aplicação
 - # Conjunto de métodos `put<tipo>(String key,<tipo> dados)`
 - # O objeto `bundle` é posteriormente passado no parâmetro do `onCreate` e também do método `onRestoresInstanceState`, aquando da reiniciação da atividade
 - # É normalmente usado para manter o estado entre reiniciações da aplicação por se ter rodado o dispositivo, alterando a sua orientação (*portrait vs landscape*)

Back Stack

Quando são ativadas sequencialmente várias atividades, este sistema interno controla o retorno à actividade anterior, quando a tecla ‘Back’ é usada

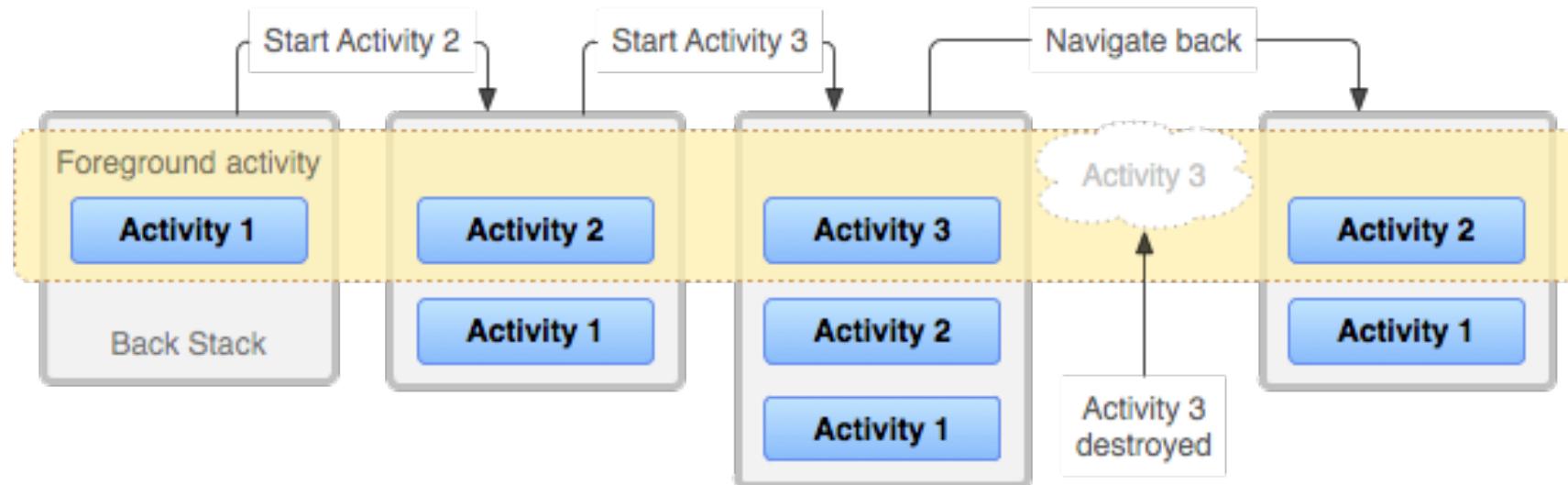


Back Stack & Single Top

- # Uma atividade pode ter configurada a flag/atributo “launchMode” com o valor “singleTop” (`FLAG_ACTIVITY_SINGLE_TOP`)
 - # Nesta situação, caso seja lançada a atividade que se encontra no topo do *Back Stack* não será criada uma nova instância
 - # O *Back Stack* não será alterado
 - # É chamado o método `onNewIntent(Intent)` para sinalizar esta situação

Back Stack

- # Podemos consultar a lista de aplicações recentes e informação sobre o *back stack* usando o adb
- # adb shell dumpsys activity activities



Objeto Application

- # Todos os componentes de uma aplicação *Android* são executados no contexto de um mesmo processo
 - # Apesar de o processo poder conter várias *threads*, os componentes são executados no âmbito da *thread* principal
- # O processo é representado numa aplicação *Android* através de um objeto Application
- # Este objeto mantém-se válido durante todo o tempo de vida da aplicação

Objeto Application

- # Uma aplicação não é obrigada a redefinir a classe Application
 - # Nesta situação é instanciado um objeto base Application
- # Para redefinir o comportamento
 - # Redefinir uma nova classe a partir da classe Application
 - # Fornecer implementações para os métodos pretendidos (onCreate, onLowMemory, onConfigurationChanged, onTrimMemory, ...)
 - # Declarar este novo tipo de objeto através do atributo android:name da estrutura <application .../> no ficheiro de manifesto
- # Para aceder a este objeto no contexto do projeto deve-se utilizar a propriedade application
 - # Em Java, usa-se a função getApplication()

Sistema de Logs

- # Podemos acompanhar a evolução da execução dos diversos processos de um dispositivo activando o módulo *Logcat*
 - # Como uma vista adicional no eclipse
 - # Através da aplicação DDMS
 - # Através de uma ligação aberta para o efeito usando o comando ADB
 - # adb logcat

Geração de mensagens de log

- # A tarefa de geração de *logs* para facilitar o processo de desenvolvimento pode ser realizado com o auxílio de métodos estáticos disponibilizados através da classe *Log*
 - # *Debug*
 - # Log.d(tag:String, msg:String)
 - # *Error*
 - # Log.e(tag:String, msg:String)
 - # *Info*
 - # Log.i(tag:String, msg:String)
 - # *Verbose*
 - # Log.v(tag:String, msg:String)
 - # *Warning*
 - # Log.w(tag:String, msg:String)
 - # “*What a Terrible Failure*”
 - # Log.wtf(tag:String, msg:String)
- # Existem versões adicionais similares mas com mais um parâmetro correspondente a um objeto *Throwable*

Interface

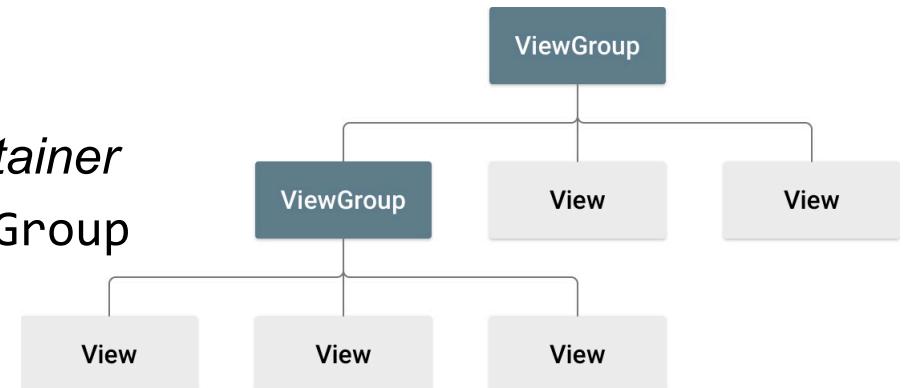
Os interfaces das aplicações são criados com base em objectos

View

- # São os objetos básicos que constituem o interface, ou seja, que permitem a interação com o utilizador através do processamento de eventos
- # Estrutura de dados que armazena informação sobre a apresentação e colocação no ecrã, bem como, o conteúdo de acordo com o objeto em causa

ViewGroup

- # Derivam de View
- # Implementam características de *container*
- # Podem conter objetos View ou ViewGroup
- # É a base para os objetos '*layout*'



Os objetos constituem uma hierarquia que irá ser atribuída a uma atividade através do método setContentView, chamado no método onCreate

Layout

Organização dos diversos elementos no ecrã

Existem vários tipos de objecto ‘*layout*’ (*ViewGroup*) que originam diferentes formas de organização dos elementos

FrameLayout

LinearLayout

RelativeLayout

TableLayout

GridLayout

AbsoluteLayout

...

ConstraintLayout (*Android Jetpack*)

Interface

Os objetos podem ser

criados programaticamente

- # Criando objetos View e derivados

- # Formando uma hierarquia de Views e atribuindo-a à atividade

construídos a partir de ficheiros xml

- # O *layout* da aplicação é definido através de xml

- # Quando associado à atividade a informação é lida do ficheiro e os objetos são criados e configurados um a um de forma automática (operação ‘inflate’)

Interface criado programaticamente

Exemplo...substituir o conteúdo do onCreate por...

```
super.onCreate(savedInstanceState)
val ll = LinearLayout(this)
val tv = TextView(this)
tv.text = "Arq. Móveis"
ll.addView(tv)
val btn = Button(this)
btn.text = "Ok"
ll.addView(btn)
btn.setOnClickListener {
    Toast.makeText(
        this,
        "Teste", Toast.LENGTH_LONG
    ).show()
}
setContentView(ll)
```

Interface definido através de XML

Cada *tag* definida em XML originará posteriormente a criação do objeto correspondente

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Arq. Móveis"
        android:id="@+id/textView"
        android:textSize="28sp"
        android:gravity="center_horizontal" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Button"
        android:id="@+id/button"
        android:layout_gravity="center_horizontal" />
</LinearLayout>
```

Layout

- # Os ficheiros XML devem ser gravados no diretório do projeto *res/layout*
- # O ambiente de desenvolvimento possui uma interface que auxilia a construção de *layouts*
- # Posteriormente os *layouts* são carregados usando identificadores que são atribuídos automaticamente aos diversos recursos

Layout

- # Assumindo que um ficheiro de *layout* é gravado com o nome `main_layout.xml` então será automaticamente criado o identificador `R.layout.main_layout`
 - # A leitura do *layout* e a criação dos objetos correspondentes, designada por operação de *inflate*, pode ser efetuada através do comando:
`setContentView(R.layout.main_layout)`

Atributos

- # Os objetos possuem um conjunto de atributos para a sua configuração
- # Um dos atributos mais importantes, para possibilitar o acesso posterior ao objeto criado, é o que permite atribuir um identificador
 - # Embora esse identificador seja um inteiro, em XML é definido da seguinte forma
 - # android:id="@+id/botao"
 - # O @ indica que é um *resource ID*
 - # O + indica que é um novo *ID*
 - # A partir do ID criado podemos ter acesso ao objecto correspondente

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text" />
```

```
val myButton: Button = findViewById(R.id.my_button)
```

Atributos

Existem dois atributos que permitem indicar a altura e largura dos mesmos e são obrigatórios para a maior parte dos componentes

- # android:layout_height
- # android:layout_width

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text" />
```

Estes atributos podem assumir os valores

- # wrap_content

- # match_parent (~~fill_parent~~)

- # Ou um valor específico para a altura e/ou largura no formato
<valor><unidade>

- # As unidades aceitas são as seguintes: *px (pixels)*, *dp (density-independent pixels)*, *sp (scaled pixels based on preferred font size)*, *in (inches)*, *mm (millimeters)*

Tipos de Layout

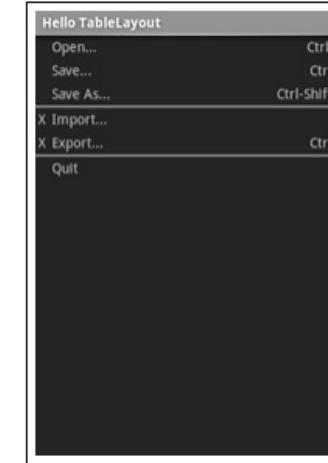
Linear Layout



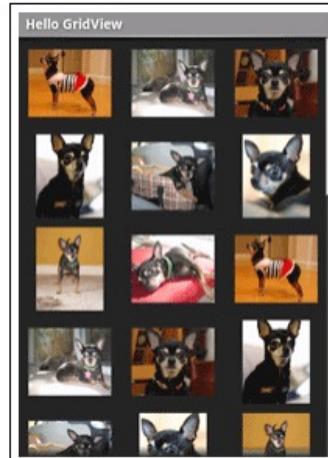
Relative Layout



Table Layout



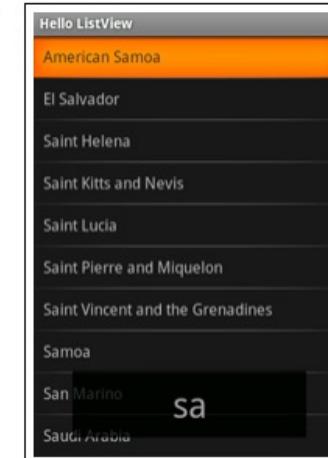
Grid View



Tab Layout



List View



LinearLayout

- # Permite organizar os objetos segundo uma sequência horizontal ou vertical
- # O alinhamento dos objetos no ecrã pode ser configurado em diversos aspetos
 - # Ocupação de espaço
 - # layout_width e layout_height
 - # Posicionamento
 - # layout_gravity
 - # center, right, left, ...
 - # Percentagem de espaço ocupado
 - # Distribuição normalizada tendo em conta os valores presentes em layout_weight

RelativeLayout

- # Permite organizar os objetos através da definição de dependências/relações entre eles
 - # O Objeto A fica à esquerda do Objeto B
 - # O Objeto C fica por baixo do Objeto D
 - # O Objeto E fica junto/fica junto da margem de baixo
 - # O Objeto F fica alinhado à direita do Objeto G

TableLayout

- # Permite a organização dos objetos em linhas
- # Similar ao comportamento das tabelas em HTML
- # Um objeto TableLayout é constituído por um ou mais TableRow
- # Quando usado individualmente, o objeto TableRow têm um comportamento análogo ao LinearLayout horizontal
- # Os objetos que constituem um TableRow não necessitam de ter as propriedades layout_width e layout_height definidas
 - # São automaticamente definidas de modo a adaptarem-se ao tamanho das linhas e colunas existentes
 - # Para cada coluna é usada a maior largura do campo existente nessa coluna, ao longo de todas as linhas

FrameLayout

- # Layout simples idealizado para apresentar apenas um item
 - # Podemos colocar vários itens desde que devidamente alinhados às diversas margens ou ao centro, usando a propriedade `layout_gravity`
- # ScrollView
 - # Deriva do FrameLayout e permite disponibilizar um espaço maior do que o ocupado no ecrã, através da utilização de `scrollbar's`
 - # No seu interior devemos colocar apenas um item, embora possa ser um ViewGroup com diversos itens

Processamento de eventos das Views

- # Para além dos eventos gerados sobre a atividade (alguns deles já experimentados no estudo do ciclo de vida de uma atividade: `onCreate`, `onPause`, ...) os componentes que formam o interface da atividade podem ser alvo de ações, as quais geram eventos
 - # Os eventos são processados no contexto de métodos específicos, exemplo:
 - # `onDraw`
 - # `onFocusChanged`
 - # `onKeyDown`
 - # `onKeyUp`
 - # `onTouchEvent`

Processamento de eventos das Views

- # Existem eventos que só são processados se a aplicação manifestar interesse nos mesmos
 - # Para registar esse “interesse” são usadas funções adequadas
 - # Formato típico: `setOn<evento>Listener`
 - # Estas funções permitem indicar os *Listeners* no contexto do qual os eventos são processados
 - # Um *Listener* corresponde a uma ou mais funções (depende do grupo de eventos a processar) declaradas através de *interfaces Java/Kotlin* adequadas

Processamento de eventos das Views

Exemplo para processar um toque num botão:

```
val btn : Button = findViewById(R.id.botao)  
btn.setOnClickListener(ObjectoOnClickListener)
```

O *ObjectoOnClickListener* pode ter como origem:

- # Implementação da interface na própria classe (*this*)
- # Implementação da interface no âmbito de uma classe criada para o efeito
- # Instanciação de um objeto através de um classe anónima
- # Implementação *inline* de um objeto de uma classe anónima – *Lambda function*

Resources

- # Podem ser definidos vários tipos de *resources*
 - # *Layouts*
 - # *Drawable*
 - # *Menus*
 - # *Values*
 - # ...
- # O mesmo *resource* pode ser definido várias vezes, de modo a se adequar ao ambiente onde vai ser visualizado
 - # *Portrait/Landscape*
 - # Língua
 - # Resolução do ecrã
 - # ...

Resources

- # As várias versões do mesmo *resource* devem possuir o mesmo nome
 - # A não sobreposição é garantida porque cada um dos ficheiros deve ser armazenado num diretório distinto dos restantes
 - # Cada diretório inclui no seu nome uma anotação que o distingue dos restantes
 - # A anotação reflete a característica diferenciadora dos restantes (língua, orientação de ecrã, ...)
- # A escolha do ficheiro com o *resource* adequado a determinado contexto é realizada pelo Resource Manager

Resources

- # Por exemplo, nos *drawables* podemos incluir *resources* com o mesmo nome, ex. `icon.ico`, em diversas subpastas correspondentes a resoluções possíveis
 - # `drawable-ldpi`
 - # Baixa densidade de pontos
 - # `drawable-mdpi`
 - # Densidade média
 - # `drawable-hdpi`
 - # Alta densidade
 - # `drawable-xhdpi`, `drawable-xxhdpi`, ...
 - # Densidade extra
 - # `drawable`
 - # A ser usado quando não são definidos *resources* mais específicos para o ambiente de execução

Resources

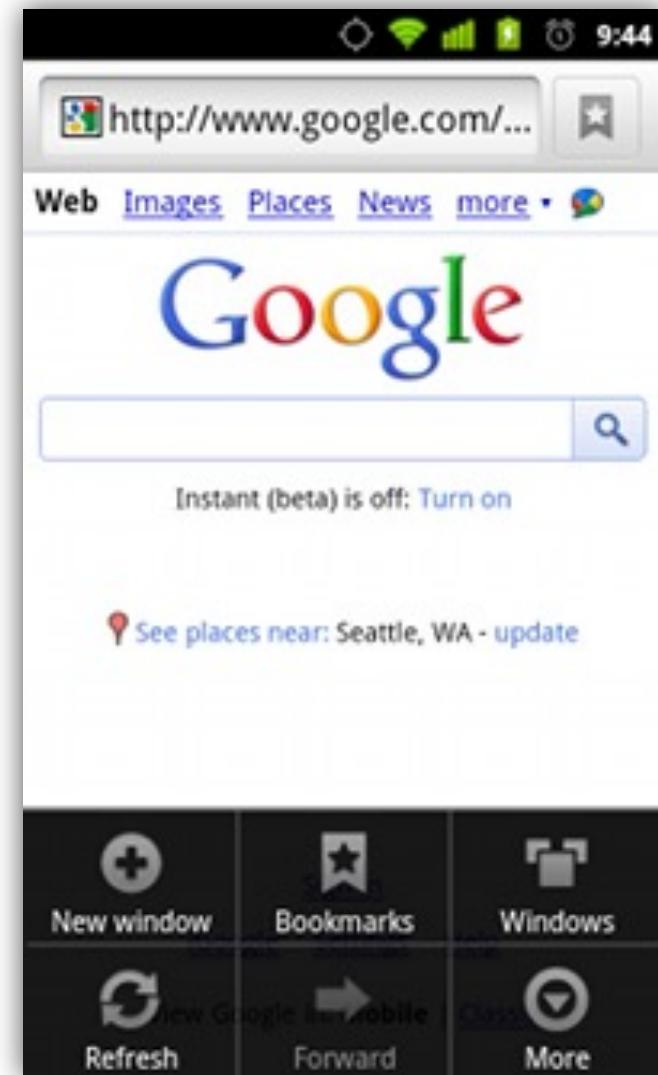
- # Da mesma forma que se podem definir *resources* diferentes de acordo com a densidade dos pixéis, podem ser definidos *resources* adequados
 - # à língua
 - # Por exemplo, para traduzir uma aplicação para a língua portuguesa, basta ter o cuidado de todas as *strings* estarem definidas no ficheiro res/values/strings e, adicionalmente, definir as mesmas *strings* (com os mesmos identificadores) na pasta res/values-pt/strings
 - # As *strings* são acedidas usando
 - # No layout: @string/<id_string>
 - # Em Java/Kotlin: Context.getString(R.string.<id_string>)
 - ... e o Resource Manager irá retornar a *string* na língua adequada
 - # à orientação do ecrã
 - # Ex: definir layouts adequados para *portrait* ou *landscape*
 - # ...

Menus

- # Os menus são úteis para disponibilizar acesso a funções usuais
- # Os menus devem ser criados no método:

```
fun onCreateOptionsMenu(menu:Menu?)
```

 - # Dinamicamente, usando métodos da classe Menu
 - # Definidos através de um ficheiro XML
- # As opções escolhidas devem ser processadas no método `onOptionsItemSelected`, o qual recebe como parâmetro uma referência para o item seleccionado



Menus

Definição das opções através de um ficheiro *xml*

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:icon="@drawable/ic_new_game"
          android:title="@string/new_game" />
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
          android:title="@string/help" />
</menu>
```

Menus

Configuração do Menu

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {  
    menuInflater.inflate(R.menu.mymenu,menu)  
    return true  
}
```

Processamento das opções selecionadas

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    when(item.itemId) {  
        R.id.new_game -> newGame()  
        R.id.help      -> showHelp()  
        else            -> return super.onOptionsItemSelected(item)  
    }  
    return true  
}
```

Menus na Action Bar

- # A partir da versão 3 passou a ser usada a barra de título
 - # inclui o *icon* da aplicação (1)
 - # permite disponibilizar algumas opções (2)
 - # permite o acesso ao menu (3)



```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
        android:icon="@drawable/ic_new_game"
        android:title="@string/new_game"
        android:showAsAction="ifRoom" />
    <item android:id="@+id/help"
        android:icon="@drawable/ic_help"
        android:title="@string/help" />
</menu>
```

Forçar a ActionBar e título

No ficheiro de manifesto colocar:

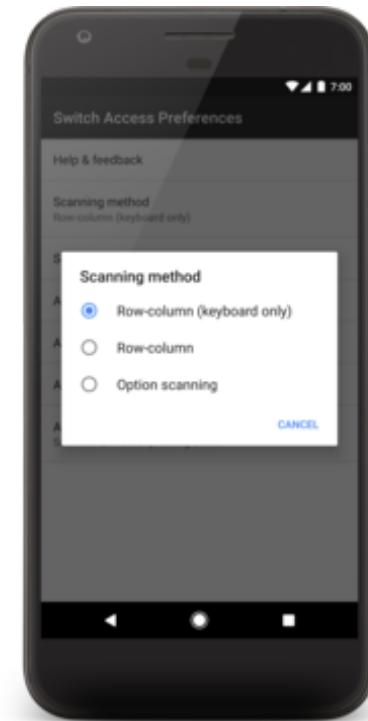
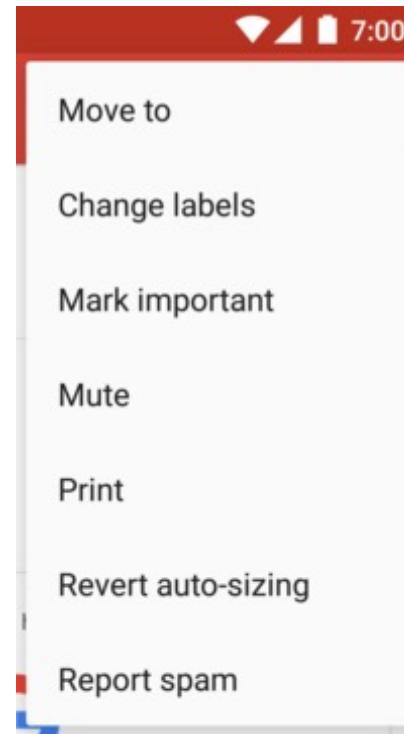
```
<application  
    android:allowBackup="true"  
    android:icon="@mipmap/ic_launcher"  
    android:label="@string/app_name"  
    android:theme=  
        "@android:style/Theme.DeviceDefault.Light.DarkActionBar" >  
  
    ...  
  
</application>
```

Nota: poderão ser usados outros valores para o tema
Ex: Theme.Holo.Light.DarkActionBar
A AppCompatActivity também dá suporte *ActionBar*

Menus de contexto "clássicos"

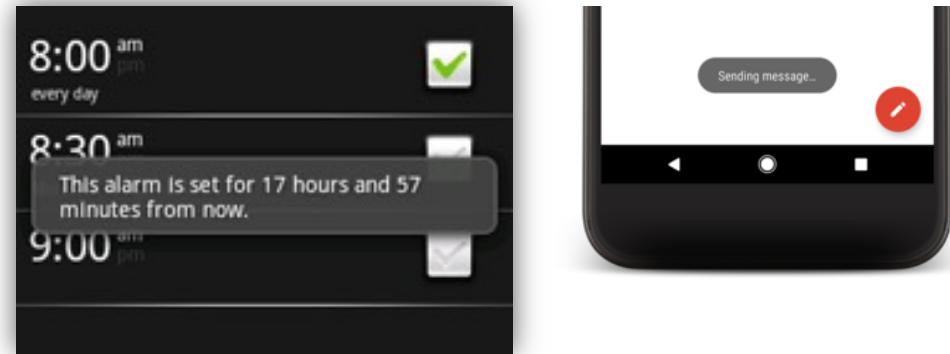
- # Menus correspondentes aos menus ativados com o botão direito do rato, nos programas para PC
- # Menus ativados com um “toque longo” adaptados ao tipo de objetivo dos diversos componentes
 - # Processar o método `onCreateContextMenu`
 - # Semelhante ao `onCreateOptionsMenu`
 - # Possui mais parâmetros de modo a dar informação sobre o componente que o pretende ativar
 - # Processar o método `onContextItemSelected`
 - # Semelhante ao `OnOptionsItemSelected`
 - # Registar as *Views* que pretendem ter menus de contexto com recurso ao método `registerForContextMenu`

Menus de contexto e PopupMenu



Geração de mensagens Toast

Existe um mecanismo, *toast*, que permite mostrar mensagens de evolução de estado ao utilizador, sem interromper a execução de outras aplicações



Classe Toast

- # Possui um método `makeText` que permite a criação deste tipo de mensagens
 - # As mensagens podem ser visualizadas durante dois períodos pré-definidos no sistema, curto e longo
 - # As mensagens são visualizadas usando o método `show()`

Geração de mensagens Toast

Podemos definir um *layout* próprio para os *Toast* seguindo os seguintes passos

- # Definir um novo recurso *Layout* (ex: `tst exemplo.xml`)
- # Usar um objecto `LayoutInflater` para criar instâncias dos objetos definidos no layout

```
val layout = LayoutInflater.inflate(R.layout.tst exemplo, null)
```

- # Criar um objeto `Toast` e associá-lo com a estrutura de objetos criada

```
val toast = Toast(applicationContext)
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0)
toast.duration = Toast.LENGTH_LONG
toast.view = layout
```

- # Visualizar o objeto `Toast` criado

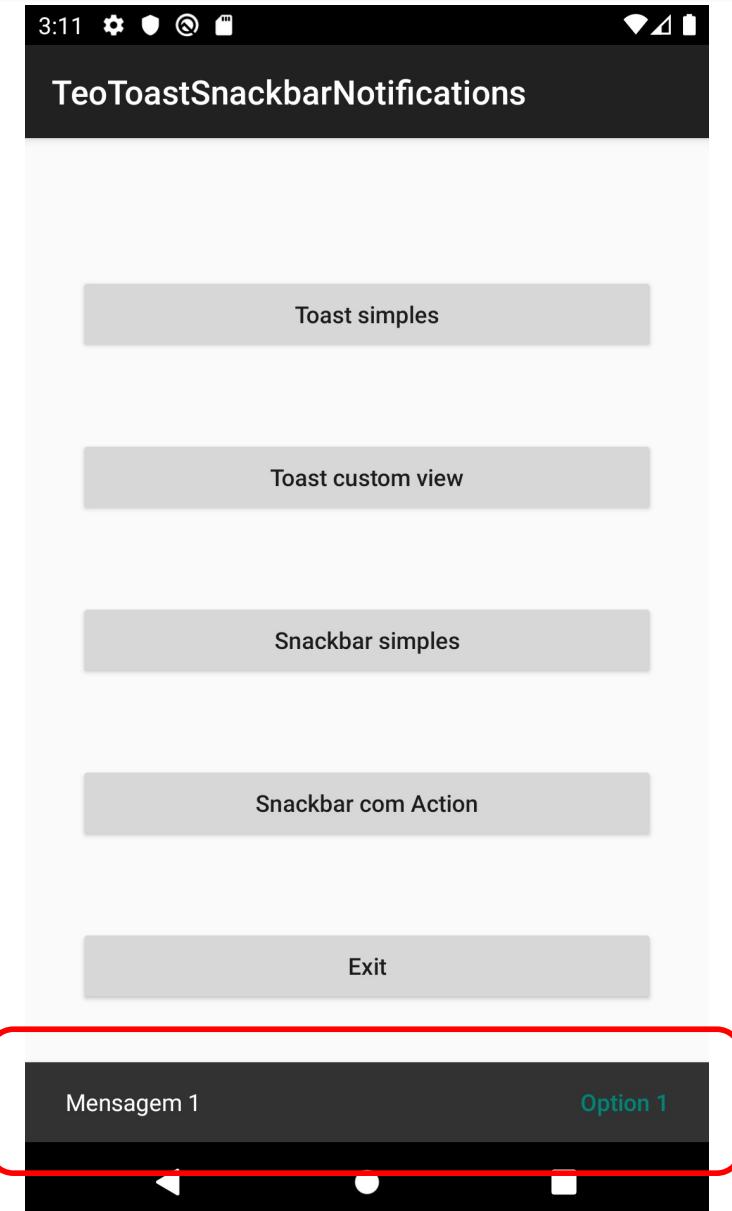
```
toast.show()
```

SnackBar

- # Permite mostrar uma mensagem de texto na parte de baixo do ecrã
- # A mensagem poderá ter associada uma ação, a qual será executada quando pressionada
- # Tal como o Toast, a Snackbar será mostrada no ecrã durante um período curto (`Snackbar.LENGTH_SHORT`) ou longo (`Snackbar.LENGTH_LONG`)
- # Permite também ficar permanente (`Snackbar.LENGTH_INDEFINITE`) a qual será escondida após um toque na ação ou através de chamada explícita à função `dismiss`
- # Componente disponibilizado através da biblioteca *material* da Google

```
implementation 'com.google.android.material:material:1.2.1'
```
- # Criada com o método `make`

```
Snackbar.make(view,R.string.msg,Snackbar.LENGTH_LONG).show()
```

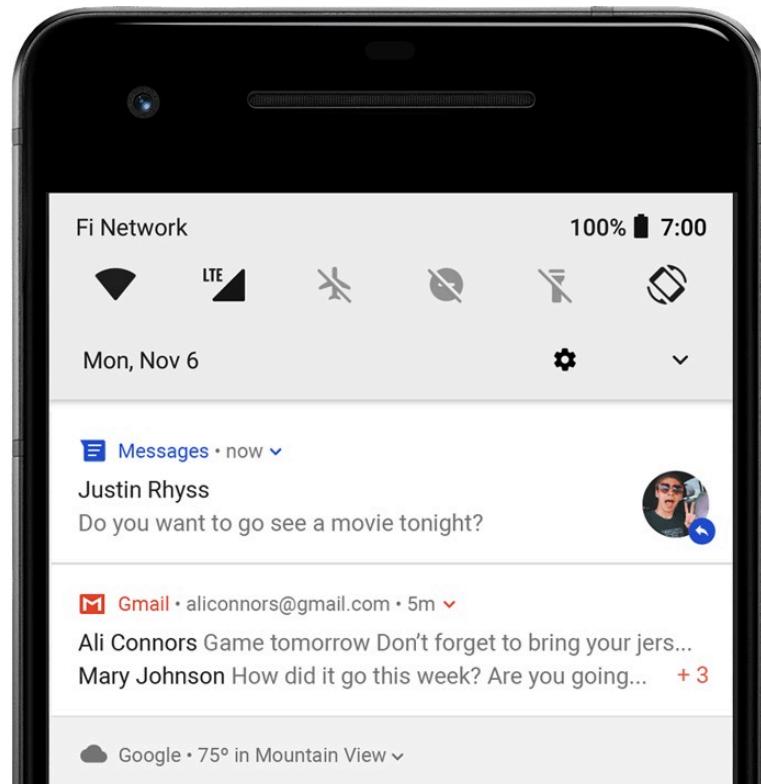


Barra de notificação de estado

- # A barra de estado surge no topo do ecrã

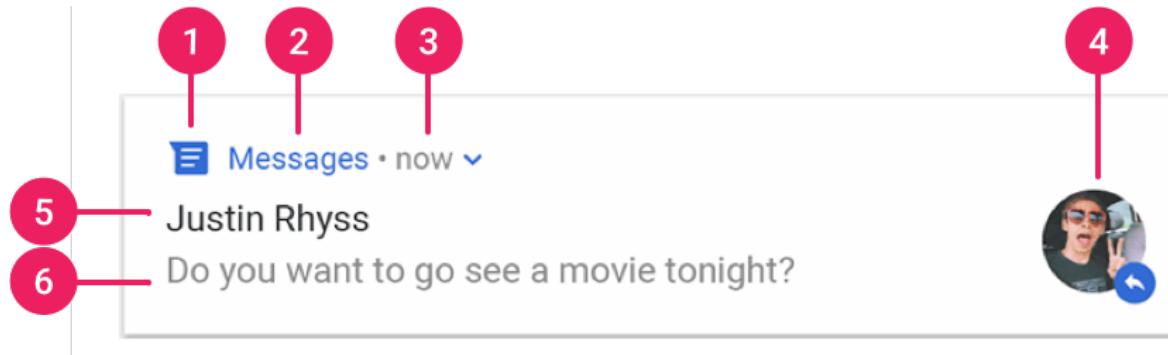


- # Deslizando a barra no sentido descendente faz aparecer a janela de notificações



Barra de notificação de estado

Elementos de uma notificação



1. Small icon
2. App name
3. Time stamp
4. Large icon
5. Title
6. Text

Canais de notificação

- # A partir da API 26 passou a ser obrigatório as notificações serem classificadas através da sua associação a “canais de notificação”
 - # Isto permite a ativação/desativação de canais de notificação por parte do utilizador (através das configurações da aplicação)
 - # Os canais são criados pela própria aplicação
 - ⌘ A criação de canais não está disponível em versões com API inferior à 26, nem está disponível via *Support Library*
 - ⌘ O código de criação de canais deverá estar protegido para não ser executado em versões mais antigas

Criação de um canal

Informação mínima para criar um canal

```
val channel_id = "amov_channel"  
val channel_name = "AMov Channel"  
val channel_importance = NotificationManager.IMPORTANCE_DEFAULT
```

Criação do canal

```
val notifMgr = getSystemService(NOTIFICATION_SERVICE) as NotificationManager  
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {  
    val channel=NotificationChannel(channel_id,channel_name,channel_importance)  
    notifMgr.createNotificationChannel(channel)  
}
```

Criação de notificações

Exemplo de informação básica para criar a notificação

```
val notif_id = 1234  
val title = "AMOV - Título"  
val text = "AMov - Texto"  
val small_icon = android.R.drawable.ic_dialog_email  
val priority = NotificationCompat.PRIORITY_DEFAULT  
... e o channelID do slide anterior
```

Criação da notificação

```
val notification = NotificationCompat.Builder(this,channel_id)  
    .setSmallIcon(small_icon)  
    .setContentTitle(title)  
    .setContentText(text)  
    .setPriority(priority)  
    .setAutoCancel(true)  
    .build()
```

Visualização da notificação

```
notifMgr.notify(notif_id,notification)
```

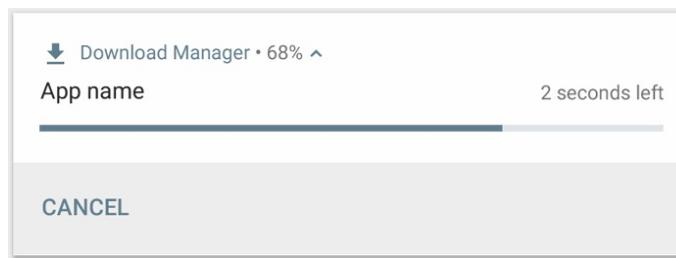
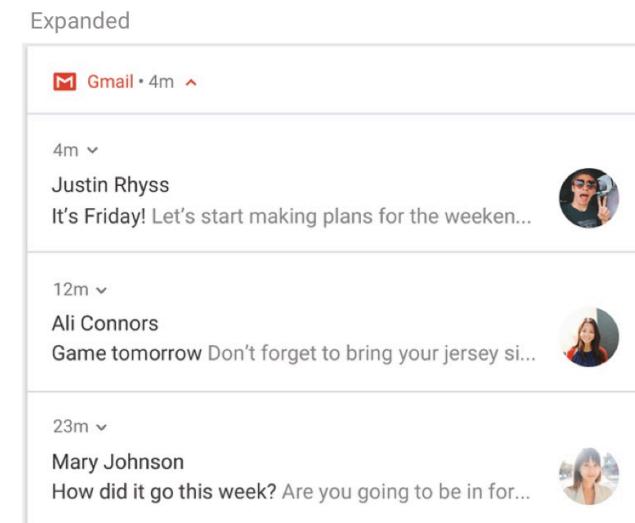
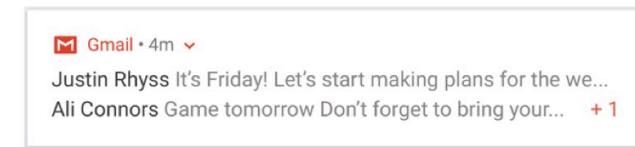
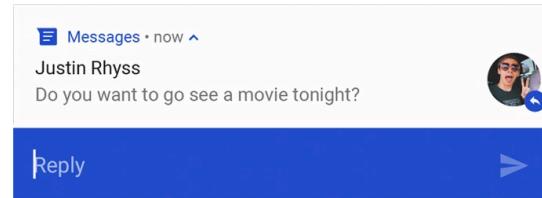
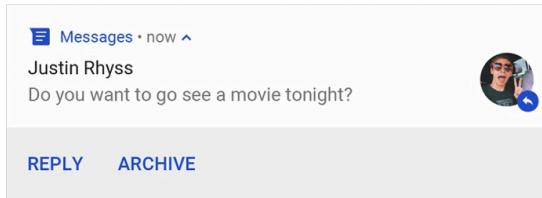
Cliques nas notificações

- # Para permitir reativar a aplicação quando é efetuado um clique na notificação deverá ser associado um PendingIntent aquando da sua criação

```
val intent = Intent(this,MainActivity::class.java).apply {  
    flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK  
}  
val pendingIntent = PendingIntent.getActivity(this,0,intent,0)  
  
val notification = NotificationCompat.Builder(this,channel_id)  
    .setSmallIcon(small_icon)  
    .setContentTitle(title)  
    .setContentText(text)  
    .setPriority(priority)  
    .setContentIntent(pendingIntent)  
    .setAutoCancel(true)  
    .build()  
  
notifMgr.notify(notif_id,notification)
```

Outros tipos de notificação

- # Existem muitas outras possibilidades de notificação que podem ser configuradas (ações personalizáveis, notificações longas/expansíveis, acumulação de notificações, barras de progresso, notificações no ecrã de bloqueio, sons, vibração, ...)



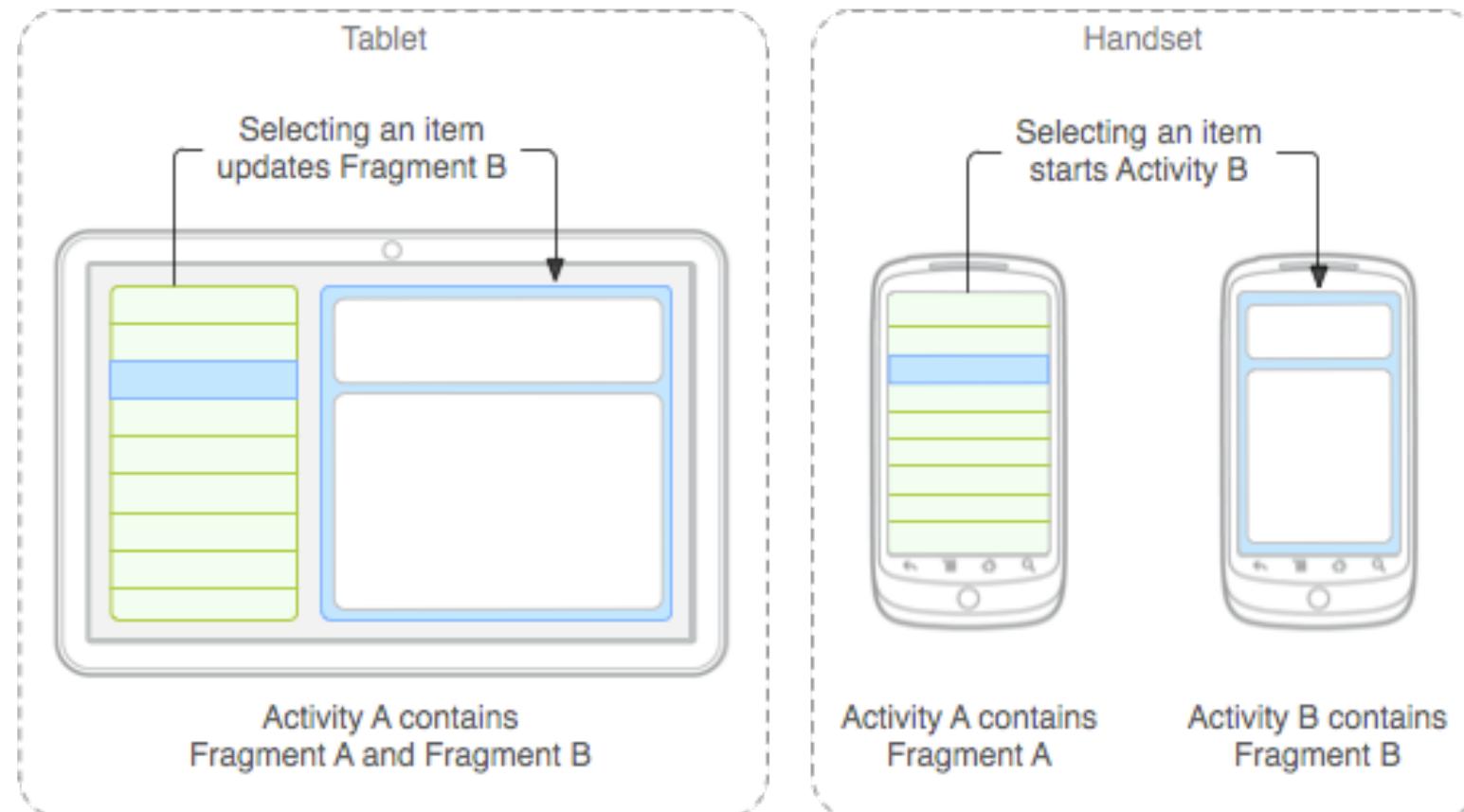
- # Consultar:

- # <https://developer.android.com/training/notify-user/build-notification>
- # <https://developer.android.com/training/notify-user/expanded>
- # <https://developer.android.com/training/notify-user/group>

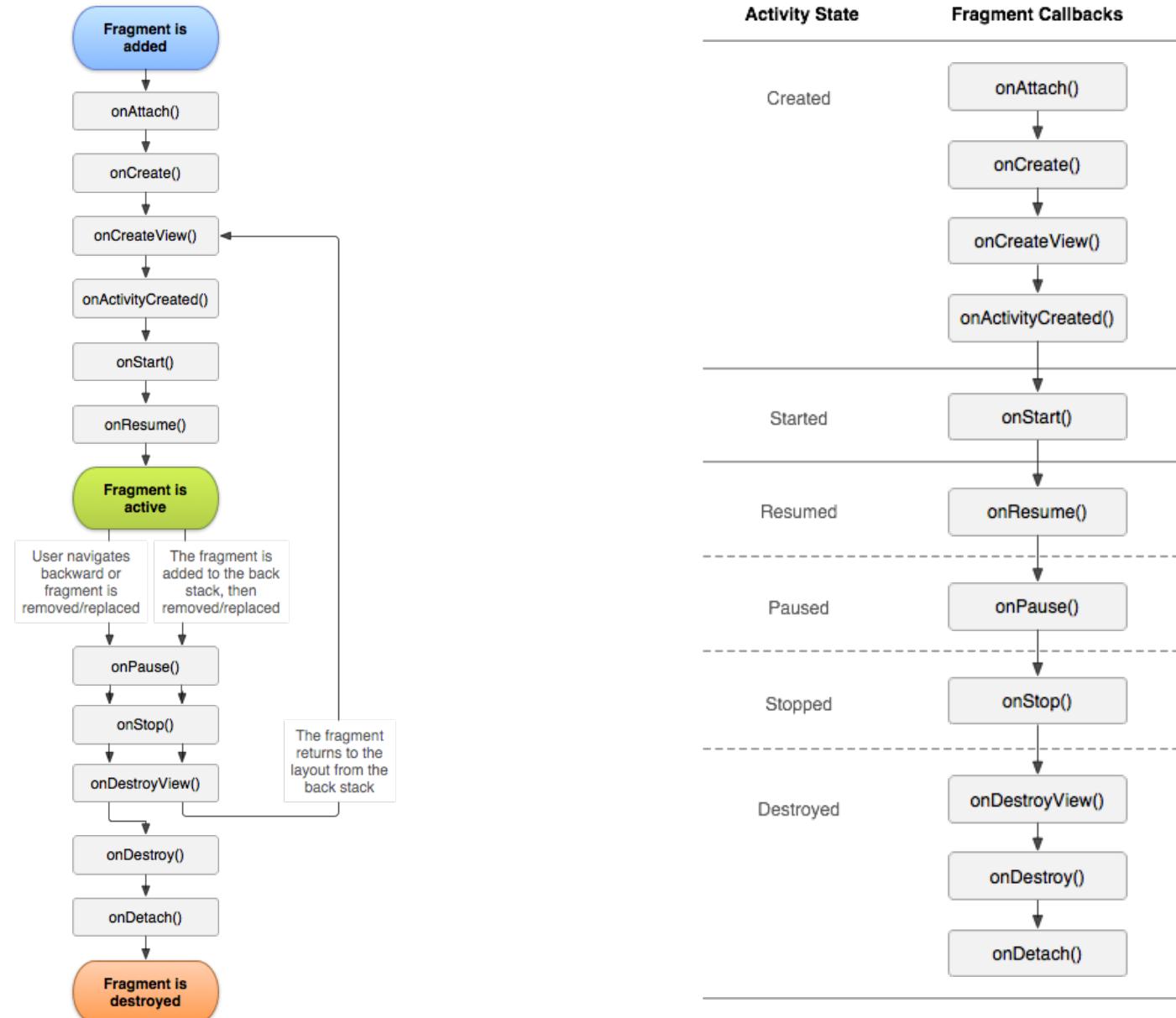
Fragment

- # Permite definir o UI por blocos
 - # Os fragmentos podem ser usados no contexto de diferentes atividades
 - # Para se adaptarem à dimensão dos dispositivos
 - # Para se adaptarem às necessidades da aplicação
- # Permitem encapsular o processamento dos componentes visuais dos fragmentos
 - # Se fossem usadas *views* repetidas nas atividades (através do uso de *include*) o processamento teria que ser repetido

Fragment



Ciclo de vida de um Fragment



Definição de um Fragment

- # A definição de um fragmento é muito similar à de uma atividade
 - # Definição do layout
 - # Através de um ficheiro xml
 - # Através de código de criação dos componentes
 - # Definição de uma nova classe como extensão da classe base Fragment

Usar fragmentos

- # Os fragmentos são incluídos no contexto de atividades
 - # Para versões anteriores à API 11 deverão ser usadas classes derivadas da FragmentActivity, usualmente a AppCompatActivity disponibilizada através da *Support Library*
- # Existem duas formas principais de usar os fragmentos
 - # Definidos através dos ficheiros XML de layout
 - # De forma programática

Exemplo com XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
    <fragment android:name="com.example.news.ArticleReaderFragment"
        android:id="@+id/viewer"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
```

Exemplo de forma programática

Inclusão de um novo fragmento

```
val fragmentManager = supportFragmentManager  
val fragmentTransaction = fragmentManager.beginTransaction()  
  
val fragment = ExampleFragment()  
fragmentTransaction.add(R.id.fragment_container, fragment)  
fragmentTransaction.commit()
```

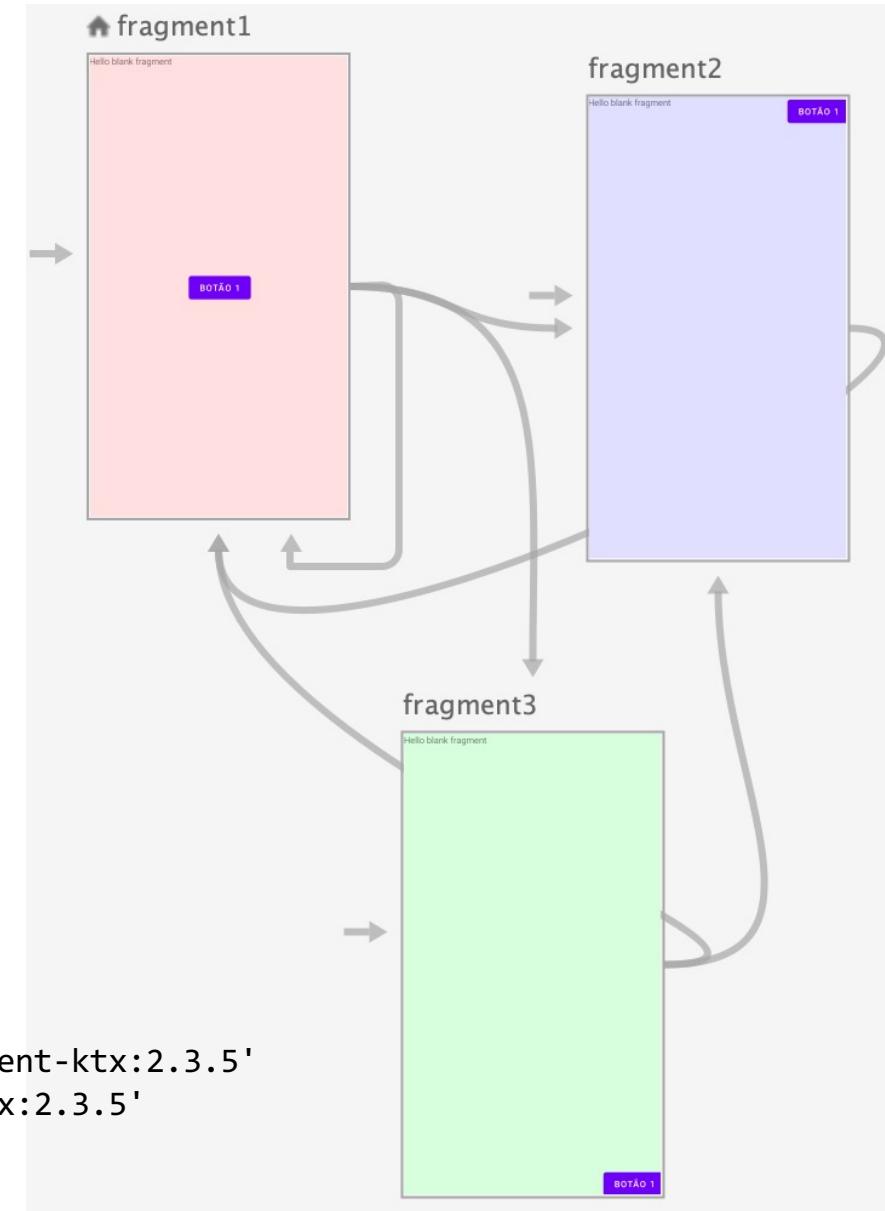
Alterar um fragmento

```
val newFragment = ExampleFragment()  
val transaction = supportFragmentManager.beginTransaction()  
transaction.replace(R.id.fragment_container, newFragment)  
transaction.addToBackStack(null)  
transaction.commit()
```

Navigation Controller

- # Com o *Android Jetpack* é disponibilizado um controlador de fragmentos (que é também um fragmento) que auxilia a tarefa de gestão do fragmento ativo entre um conjunto de fragmentos, representados através de um grafo

```
dependencies {  
    ...  
    implementation 'androidx.navigation:navigation-fragment-ktx:2.3.5'  
    implementation 'androidx.navigation:navigation-ui-ktx:2.3.5'  
    ...  
}
```



Navigation Controller

```
<!-- Exemplo de atividade que inclui um NavHostFragment-->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/tvMsg1"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Início" />

    <fragment
        android:layout_margin="16dp"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:id="@+id/fragment_base"
        android:name="androidx.navigation.fragment.NavHostFragment"
        app:defaultNavHost="true"
        app:navGraph="@navigation/base_navigation"
        />

    <TextView
        android:id="@+id/tvMsg2"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Fim" />

</LinearLayout>
```

```
<!-- Exemplo de recurso do tipo navigation: base_navigation.xml -->
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/base_navigation"
    app:startDestination="@+id/fragment1">

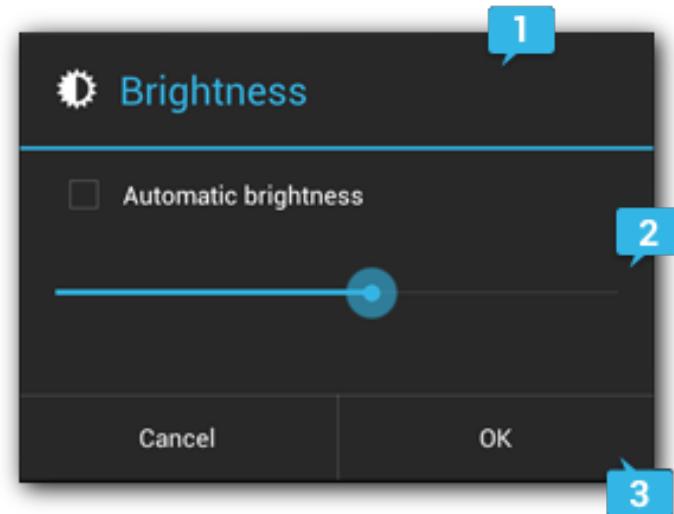
    <fragment android:id="@+id/fragment1"
        android:name="pt.isec.ans.teonavigationcontroller.Fragment1"
        android:label="fragment_1"
        tools:layout="@layout/fragment_1">
        <action android:id="@+id/action_fragment1_to_fragment2"
            app:destination="@+id/fragment2" />
        <action android:id="@+id/action_fragment1_to_fragment3"
            app:destination="@+id/fragment3" />
        <action android:id="@+id/action_fragment1_self"
            app:destination="@+id/fragment1" />
    </fragment>
    <fragment android:id="@+id/fragment2"
        android:name="pt.isec.ans.teonavigationcontroller.Fragment2"
        android:label="fragment_2"
        tools:layout="@layout/fragment_2">
        <action android:id="@+id/action_fragment2_to_fragment1"
            app:destination="@+id/fragment1" />
    </fragment>
    <fragment android:id="@+id/fragment3"
        android:name="pt.isec.ans.teonavigationcontroller.Fragment3"
        android:label="fragment_3"
        tools:layout="@layout/fragment_3">
        <action android:id="@+id/action_fragment3_to_fragment2"
            app:destination="@+id/fragment2" />
        <action android:id="@+id/action_fragment3_to_fragment1"
            app:destination="@+id/fragment1" />
    </fragment>
    <action android:id="@+id/action_global_fragment1"
        app:destination="@+id/fragment1" />
    <action android:id="@+id/action_global_fragment2"
        app:destination="@+id/fragment2" />
    <action android:id="@+id/action_global_fragment3"
        app:destination="@+id/fragment3" />
</navigation>
```

Dialogs

- # A Dialog é uma janela que permite interação com o utilizador, sobrepondo-se no ecrã e não o ocupando na sua totalidade
 - # Aparece “por cima” das restantes
- # As *dialogs* são implementações da classe Dialog
 - # Existem definidos tipos de *dialog* que facilitam a sua utilização. Ex: AlertDialog, ProgressDialog, DatePickerDialog, TimePickerDialog

AlertDialog

- # A criação de instâncias da classe AlertDialog deverão ser feitas com o auxílio da sua *nested class* Builder
- # Permite criar uma janela de diálogo constituída por 3 partes:
 1. Título e icon
 - # setTitle, setIcon, setCustomTitle
 2. Corpo
 - # mensagem
 - # setMessage
 - # lista de elementos
 - # setItems, setSingleChoiceItems, ...
 - # view/inflate ou custom view
 - # setView
 3. Botões de acção
 - # botões de ação/confirmação
 - # positive button (ok, yes, ...)
 - # setPositiveButton
 - # negative button (no, cancel, ...)
 - # setNegativeButton
 - # neutral button (remind me later, ...)
 - # setNeutralButton
- # Depois de configurada...
 - # deve ser criada usando o método create() do Builder
 - # mostrada com o método show()



AlertDialog

```
val dlg = AlertDialog.Builder(this)
    .setTitle("Título")
    .setIcon(android.R.drawable.ic_dialog_alert)
    .setMessage("Mensagem")
    .setPositiveButton("Sim",
        DialogInterface.OnClickListener { dialog, which -> . . . })
    .setNegativeButton("Não",
        DialogInterface.OnClickListener { dialog, which -> . . . })
    .setCancelable(false)
    .create()

dlg.show()
```

DialogFragment

- # Para uma melhor gestão interna e adequação ao ciclo de vida dos componentes das atividades, as janelas de diálogo deverão ser criadas no contexto de um objecto DialogFragment e geridas como fragmentos
 - # Criar um objecto derivado de DialogFragment e retornar a AlertDialog na função onCreateDialog

```
class MyDialog : DialogFragment() {  
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {  
        return AlertDialog.Builder(context!!)  
            .setTitle("Titolo")  
            . . .  
            .create()  
    }  
}
```

- # Para mostrar uma janela de diálogo criada como fragmento...

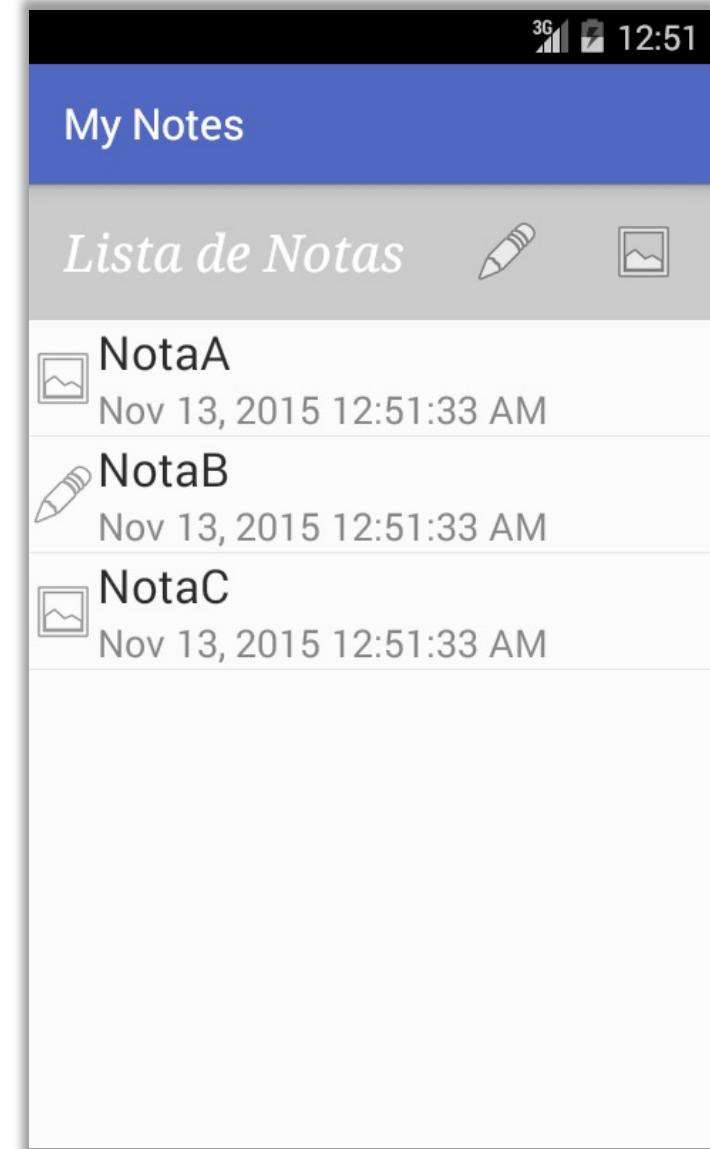
```
val dlg = MyDialog()  
dlg.show(supportFragmentManager, "my_dialog")
```

Atividades como Dialogs

- # Uma outra opção para a criação de janelas de diálogo é recorrendo a atividades
- # Para que as atividades se comportem como janelas de diálogo é necessário associar-lhes um tema adequado
 - # `@android:style/Theme.Holo.Dialog`
 - # `@style/Theme.AppCompat.Light.Dialog`
- # As atividades serão apresentadas da mesma forma que as restantes, recorrendo a um objeto Intent e chamando a função `startActivity`

ListView e ListAdapters

- # A ListView permite visualizar e selecionar listas de elementos
- # É necessário definir um ListAdapter a partir do qual são obtidos os dados a introduzir na ListView
 - # ArrayAdapter<T>
 - # Array a partir do qual são gerados dados para uma única TextView, correspondente a cada item da lista
 - # SimpleAdapter
 - # ArrayList de objectos Map que possuem os dados correspondentes a cada linha de uma lista, organizados com a ajuda de um ficheiro xml onde é definido o layout
 - # CursorAdapter
 - # SimpleCursorAdapter
 - # Custom adapter
 - # Classe derivada da BaseAdapter
 - # definir os métodos: getCount, getItem, getItemId, getView



RecyclerView

- # Permite a implementação de listas similares à ListView, mas com gestão de recursos melhorada
- # Necessita
 - # LayoutManager
 - # LinearLayoutManager
 - # GridLayoutManager
 - # StaggeredGridLayoutManager
 - # RecyclerView.Adapter
 - # Redefinir classe
 - # Configura a *recycler view* indicando o número de itens, uma função para criar um item e uma outra função para atualizar os dados de um item
 - # RecyclerView.ViewHolder
 - # Redefinir classe
 - # Representa a hierarquia de vistas correspondente a um item