



Programação de Dispositivos Móveis

Aula 4

Licenciatura em Engenharia Informática

Licenciatura em Informática Web

Sumário

Descrição dos blocos fundamentais que constituem as aplicações Android™ e análise do seu processo de preparação e compilação. Foco na componente Activity, descrevendo com detalhe o ciclo de vida e os mecanismos de gestão das atividades no sistema operativo.

Programming of Mobile Devices

Lecture 4

Degree in Computer Science and Engineering

Degree in Web Informatics

Summary

Description of the fundamental components that may be used to compose an Android™ application and analysis of its building process. Thorough discussion of the Activity component, namely of their life cycle and of the mechanisms available in the operating system for managing them.

1 Aplicações Android™

Android™ Applications

1.1 Introdução

Introduction

As **aplicações Android™** são constituídas por vários **componentes**, que **colaboram** entre si, e que a **plataforma despoleta e corre** quando necessário. Cada um desses componentes **tem o seu próprio objetivo** e, portanto, a sua própria *Application Programming Interface* (API). Dado que, em Android™, as aplicações nativas correm numa máquina virtual Java, cada um destes componentes são também **implementados em Java**. Os **4 blocos fundamentais** sobre os quais as aplicações Android™ são construídas são:

1. Atividades (Activities);
2. Serviços (Services);
3. Fornecedor de Conteúdos (ContentProviders); e
4. Recetores Difusão (Broadcastreceiver).

Note que, no fundo, uma aplicação Android™ é constituída por um conjunto destes blocos, **organizados de forma a disponibilizarem determinada funcionalidade**. Cada um dos componentes referidos anteriormente são discutidas com um pouco mais de detalhe nas secções seguintes e também adiante neste curso.

1.2 Atividade

Activity

A classe Atividade **é a que permite a construção de interfaces de utilizador gráficas** (Graphical User Interface (GUI)) sendo, por isso, uma **classe fundamental** para a maior parte das aplicações Android™. É ela que fornece a base para que os utilizadores **manipulem, adicionem ou obtenham informação de uma aplicação**. Os **outros 3 componentes não fornecem GUI** e, ou correm em segundo plano para potencialmente auxiliar as atividades principais, ou servem para disponibilizar, de uma forma uniforme, dados privados (e geridos pela aplicação) a outras aplicações.

Por **definição¹**, **uma atividade deve corresponder a uma única ação ou interação que o utilizador pode fazer**, como por exemplo, escrever uma mensagem, ver uma página ou ler um e-mail. **São elas que dão origem às janelas** onde os *widgets* se estruturam numa unidade consistente. As atividades são normalmente apresentadas ao utilizador como janelas que **ocupam o ecrã do dispositivo por inteiro** (ou quase). Isto significa que, se uma aplicação vai **suportar mais do que uma funcionalidade, provavelmente terá de conter mais do que uma atividade**. O Android™ lida com estas atividades de uma forma muito precisa que é necessário conhecer. Por isso, as atividades serão discutidas com mais detalhe adiante nesta aula.

Note que as atividades concretizam, no fundo, a forma como foram pensadas as aplicações **para dispositivos móveis**, inicialmente com **ecrãs pequenos**. Nesse caso, faria ainda mais sentido que cada atividade (cada ecrã) suportasse apenas uma interação simples com o utilizador. **Atualmente**, e com a proliferação de dispositivos com ecrãs maiores, **é possível que algumas atividades**

¹E.g., ver <http://developer.android.com/reference/android/app/Activity.html>.

sejam desenhadas de modo a suportar várias funcionalidades simultaneamente.

O excerto de código seguinte contém o código de uma aplicação HelloWorld muito simples, mas que serve de exemplo. Repare em pelo menos 3 detalhes: (i) a classe Activity foi importada para este ficheiro de código (`import android.app.Activity`); (ii) a classe principal estende esta classe (`extends Activity`), e (iii) é feita a reescrita (`@Override`) de um método chamado `onCreate()`. Ainda se poderia dizer que o fluxo de execução evolui da seguinte forma: primeiro é restaurado o estado da atividade (`super.onCreate()`), e em segundo é construída a interface (`setContentView()`).

```
package pt.ubi.di.pi.helloworld;

import android.app.Activity;
import android.os.Bundle;

public class HelloWorld extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

1.3 Serviço

Service

Um **serviço** é um componente aplicacional que pode executar operações de longa duração em segundo plano e não fornece uma interface de utilizador. Um serviço pode ser **despoletado por outra componente** com o intuito de que este continue a fornecer determinada funcionalidade **mesmo que o utilizador mude** para outra aplicação ou atividade. Um **exemplo prático** que mostra a utilidade de Serviços em Android™ é por exemplo o de **ouvir música**. A maior parte das aplicações existentes para o efeito permitem que o utilizador escolha a música via interface gráfica, para depois poder utilizar outras aplicações enquanto a escolha toca em segundo plano. Este componente permite que uma aplicação se associe a um serviço em execução e interaja com o mesmo ou troque informação com outras aplicações².

1.4 Fornecedor de Conteúdos

Content Provider

Como o próprio nome indica, estes **componentes são usados para fornecer conteúdos estruturados a aplicações** Android™ de uma forma padrão. Em última análise, é este componente que **permite que aplicações comuniquem entre si ou que usem dados que são partilhados por todo o sistema**, como por exemplo os contactos de telefone de um utilizador. **A interface** para os fornecedores de conteúdos **chama-se ContentResolver**

²Mais info em <http://developer.android.com/guide/components/services.html> e adiante.

e quanto um pedido lhe é endereçado, este **verifica se a aplicação tem os privilégios necessários para lhe aceder** (que devem ser especificados no manifesto). No caso afirmativo, o ContentResolver **envia o pedido para o fornecedor de conteúdos respetivo, previamente registado no sistema** para o efeito³.

Note que, se uma aplicação não tiver a necessidade de partilhar os dados que produz ou que precisa com outras aplicações, então não precisa de fornecedores de conteúdos. Estes fornecedores de conteúdos implementam normalmente a lógica necessárias para fornecer ou atualizar os dados de uma determinada aplicação uma forma uniforme e segura. É um conjunto de métodos que permitem que outras aplicações manipulem esses dados. Note também que **o Android™ já vem de fábrica com alguns fornecedores de conteúdos padrão**⁴, tipicamente relacionados com informação de contactos, calendários e ficheiros multimédia. Os fornecedores de conteúdo serão alvo de discussão adiante.

1.5 Recetor de Difusão

BroadcastReceiver

O componente **recetor de difusão escuta e processa eventos** na plataforma Android™. Na **arquitetura de software** para troca de mensagens conhecido como **publish-subscribe**, as instâncias destes componentes **tomam o papel de subscritores**. O seu **objetivo é o 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 intenções (`intents`) e **difundem-nos para o sistema utilizando um método como o `sendBroadcast()`**, para o caso de um ou mais recetores estarem registados para tratar a mensagem. Tanto os recetores de difusão como os *intents* serão discutidos com mais detalhe adiante.

Os recetores difusão são **componentes importantes em termos de performance e funcionalidade** de um sistema Android™. São eles que permitem que uma determinada aplicação registre no sistema a sua vontade em receber determinados eventos. **Caso não existisse** um componente com esta forma de funcionamento, cada vez que um fosse despoletado um evento que pudesse ter impacto em todas as aplicações, **o sistema seria obrigado a enviar esse evento para todas elas**.

Para obter uma ideia prática e concreta da utilidade destes componentes, considere, por exemplo, que determinada aplicação muda a cor de fundo de uma etiqueta de texto de acordo com a carga na bateria. Nesse caso, pode instanciar um recetor de difusão para eventos relacionados com a carga, sendo que o sistema envia esses eventos, caso ocorram, apenas para aquele recetor, que

³Mais em <http://developer.android.com/reference/android/content/ContentProvider.html> e adiante.

⁴Ver <http://developer.android.com/reference/android/provider/package-summary.html>.

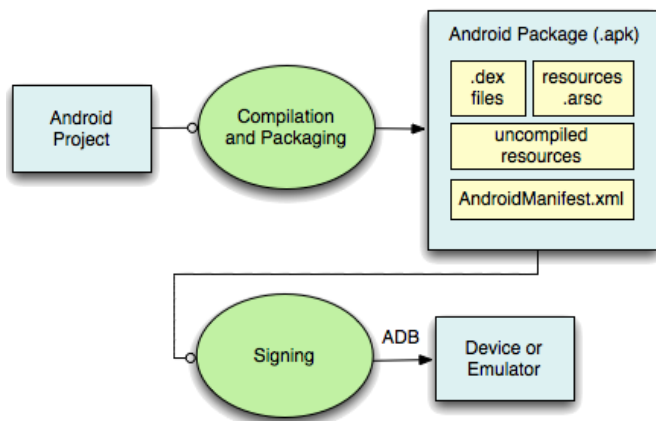


Figura 1: Processo de compilação simplificado de uma aplicação Android™.

pode então atuar em conformidade. É um componente deste tipo que permite que o gestor de notificações escute e mostre mensagens SMS na barra de notificações quando estas chegam.

1.6 Processo de Preparação de uma Aplicação Android™

Building an Android™ Application

Após implementar uma aplicação, é necessário prepará-la para que possa ser instalada num dispositivo com Android™ e correr na máquina virtual java (Dalvik ou ART). A figura ?? ilustra o processo de compilação da aplicação de uma forma simplificada. Ambas as figuras foram adaptadas da documentação oficial disponível no *Uniform Resource Locator* (URL) <http://developer.android.com/tools/building/index.html>. Durante o processo de preparação da aplicação, alguns ficheiros do projeto são compilados sendo depois empacotados juntamente com outros recursos num arquivo com extensão .apk⁵. O arquivo, conforme esquematizado, contém: os ficheiros .dex (que são os ficheiros .class compilados para o *byte code* Dalvik), uma versão em binário do *AndroidManifest.xml*, um ficheiro contendo metadados acerca dos recursos em *resources.arsc*, e também todos os recursos que a aplicação usa em formato original, nomeadamente ícones e ficheiros multimédia (e.g., ficheiros de som).

Note que o processo de preparação do arquivo .apk final inclui assinar digitalmente o pacote. Este mecanismo usa **criptografia de chave pública** (normalmente recorrendo ao sistema criptográfico Rivest, Shamir e Adleman (RSA)), para o qual é necessário um par de chaves pública e privada e um certificado de chave pública. Tanto o ambiente de desenvolvimento *Android Studio* como o *Gradle* assinam automaticamente os pacotes com uma chave de depuração aquando da compilação de aplicações (estas chaves estão normalmente guardadas em \$HOME/.android/debug.keystore). Contudo, a

⁵Extensão que deriva da designação inglesa *Application Package*.

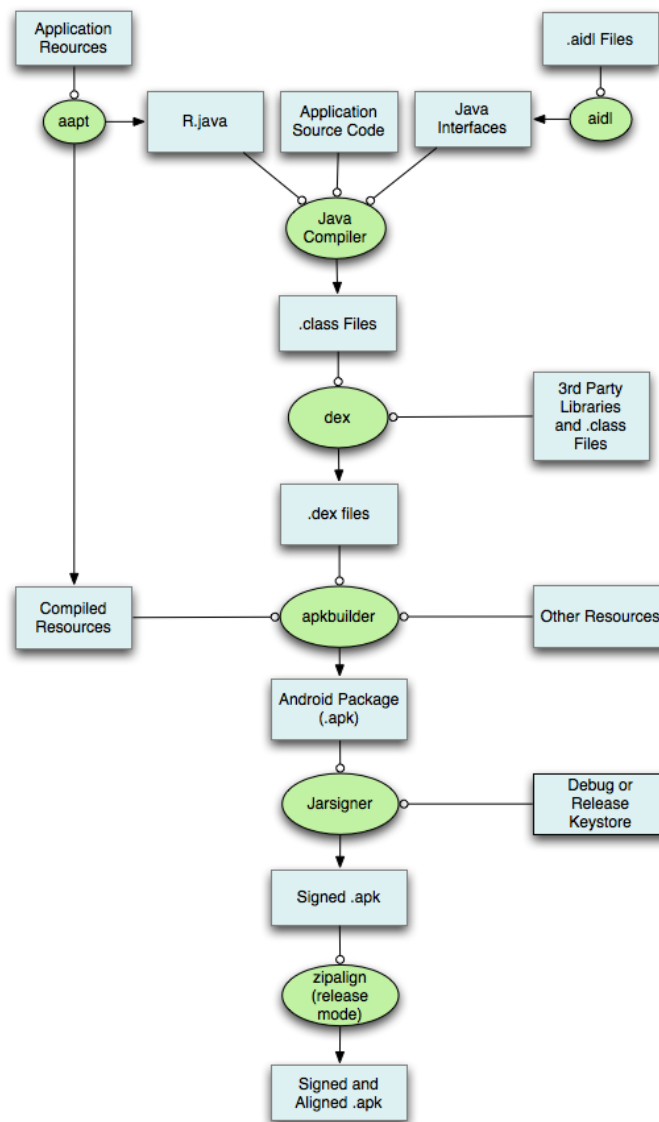


Figura 2: Processo de preparação de uma aplicação Android™.

publicação da aplicação na **versão release** na loja **Google Play** requer que o programador tenha um certificado que o identifica (ou que identifica as suas aplicações univocamente no universo Android™). Estas chaves e certificado podem ser geradas localmente (e o certificado pode ser auto-assinado), sendo que este apenas terá de provar que possui a chave privada para que este seja considerado. Este certificado não precisa, por isso, de ser assinado por nenhuma autoridade certificadora⁶.

A publicação na *Google Play* requer uma subscrição de 25\$ (aproximadamente 20 euros)⁷, a configuração de alguns detalhes (por exemplo, o preço da aplicação) e o preenchimento de alguns requisitos relativos, e.g., ao tamanho da aplicação, capturas de ecrã para apresentar a aplicação, etc. Em princípio, e caso a aplicação

⁶Mais informação acerca deste assunto em <http://developer.android.com/tools/publishing/app-signing.html#cert>.

⁷A subscrição *Google Play* é única. No caso da Apple é anual (certa de 100\$ anuais para programadores em nome individual).

não contenha *malware*, será publicada ao fim de pouco tempo⁸.

O processo de preparação da aplicação Android™ encontra-se ilustrado **com bastante detalhe** na figura ???. Como se pode constatar, o processo pressupõe a utilização de **várias ferramentas** e a **iteração por diversos passos**:

1. A ferramenta de empacotamento de recursos Android™ (da designação inglesa *Android Asset Packaging Tool* (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 referencias para os vários recursos**, para que possam ser usadas no âmbito da implementação da aplicação e para comodidade do programador;
2. A ferramenta *aidl*⁹ converte interfaces definidas na linguagem AIDL em interfaces Java;
3. **Todo o código Java** entretanto gerado (R.java + interfaces) 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* (***apk packager* em versões mais recentes**) **alimenta-se então de todos os recursos que foram compilados, bem como os que não são compiláveis (como imagens) 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™.

1.7 R.java e strings.xml

R.java **and** strings.xml

Note que, durante a descrição anterior, foi referida a **geração automática de um ficheiro de recursos chamado R.java**. Este ficheiro é colocado dentro da diretoria *generated/source/r* e **não deve ser modificado pelo programador em situação alguma**. Contém a definição de várias **classes estáticas**, nomeadamente a **classe id** e **string** que permite que os **programadores possam aceder a recursos contidos na diretoria res com mais facilidade**, através de instruções semelhantes a: *R.string.app_name*

⁸Para mais informação sobre este assunto, pode consultar o <http://developer.android.com/tools/publishing/preparing.html>.

⁹Da designação inglesa *Android Interface Definition Language* (AIDL)

A título de exemplo, inclui-se a seguir o conteúdo de um ficheiro R.java:

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.
 * This class was automatically generated by the
 * aapt tool from the resource data it found. It
 * should not be modified by hand.
 */
package pt.ubi.di.pmd.acalculator;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int SUM=0x7f050002;
        public static final int number1=0x7f050000;
        public static final int number2=0x7f050001;
        public static final int result=0x7f050003;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
    }
}
```

Este ficheiro é **gerado pela ferramenta aapt**, que **vasculha dentro das diretorias** com recursos (nomeadamente na */res*) e nos respetivos **ficheiros XML**. Se encontrar **imagens**, atribui-lhes um **ID** (um inteiro de 32 bits) na **classe drawable**; se encontrar **strings definidas no ficheiro strings.xml**, atribui-lhes um **ID** na **classe string**; e se encontrar **ficheiros XML de layout**, atribui um **ID** a cada entrada que contiver um atributo semelhante a *android:id="@+id/nome-do-atributo"* na **classe id**. Note-se que, no último caso apontado, **só são criados IDs para os recursos cujo atributo android:id comece com o @+**, sendo que é o + que determina que é necessário criar o ID aquando da compilação.

A seguir incluem-se exemplos do conteúdo do ficheiro *strings.xml* e de um excerto do ficheiro de *layout* *main.xml*, respetivamente.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ACalculator</string>
</resources>
```

```
<EditText
    android:id="@+id/number1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:inputType="number"
/>
```

É de notar que, graças ao ficheiro R.java, os vários recursos são acedidos através de chamadas a esta classe e às suas subclasses dentro do código Java. Contudo, **entre ficheiros XML, os vários recursos são citados recorrendo a entradas semelhantes a @string/nome-dado-às-string**.

1.8 AndroidManifest.xml

AndroidManifest.xml

Uma **estrutura/ficheiro fundamental em todas as aplicações Android™** é o ficheiro conhecido por **AndroidManifest.xml**. Este ficheiro, cujo **nome tem de ser exatamente** o que foi enunciado antes, tem de estar presente na **pasta principal do projeto**, sendo depois compilado durante o processo de preparação da aplicação. Este ficheiro **apresenta informação** essencial acerca da aplicação **ao sistema Android™** que precisa ser conhecida antes que este a possa executar (e.g., a aplicação *Home* lê este ficheiro para saber o nome da aplicação e apresentá-la no ecrã respetivo). Entre outras funcionalidades, este Manifesto serve os **seguintes objetivos**:

- Indica o nome dado ao pacote (*package*) Java para esta aplicação, e que serve como **identificador único** para a aplicação no sistema;
- Enumera as várias atividades, serviços, recetores de difusão, e fornecedores de conteúdos que compõem a aplicação;
- Indica o nome de **todas as classes que implementam as componentes** indicadas antes, bem como **as suas capacidades**, em termos de intents e mensagens que são capazes de processar;
- Declara as **permissões que a aplicação precisa para aceder a partes protegidas** do sistema **ou para interagir com outras aplicações**;
- Também **declara as permissões que outras aplicações precisam ter** para aceder a funcionalidades daquela a que o manifesto se aplica;
- Pode identificar a **API mínima que a aplicação requer**;
- Lista **eventuais bibliotecas que precisam ser ligadas** para a execução da aplicação.

A título de exemplo, em baixo mostra-se o conteúdo de um **AndroidManifest.xml** (note que o nome da aplicação estará na **strings.xml**, e que este recurso é referenciado por `@string/app_name`):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/
    android"
  package="pt.ubi.di.pmd.acalculator"
  android:versionCode="1"
  android:versionName="1.0">
  <uses-sdk android:minSdkVersion="8"
    android:targetSdkVersion="18" />
  <application
    android:label="@string/app_name"
    android:icon="@drawable/ic_launcher">
    <activity
      android:name="ACalculator"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.
          action.MAIN" />
```

```
        <category android:name="android.intent.
          category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Repare que, no manifesto exemplificado antes, a **atividade chamada ACalculator é configurada como sendo o ponto de entrada da aplicação** (ou, mais precisamente, da tarefa) através da definição dos atributos `<action android:name="android.intent.action.MAIN"/>` e `<category android:name="android.intent.category.LAUNCHER"/>`.

2 A Classe Activity

The Activity Class

As atividades (Activities) são os objetos que, por excelência, fornecem os meios para os utilizadores interagirem com as aplicações. Por definição, as atividades **devem ser modulares**, no sentido de **cada uma delas suportar apenas uma ação** que o utilizador pode fazer. Se esta definição for tida como garantida, as aplicações Android™ (ou pelo menos aquelas que oferecem uma interface gráfica) **corresponderão a sequências ordenadas de atividades**, assim agrupadas para atingir as funcionalidades pretendidas. No jargão específico do ecossistema Android™, estas sequências são designadas por tarefas:

Uma tarefa é um conjunto ordenado de atividades que um utilizador percorre para obter determinada funcionalidade no contexto de uma aplicação Android™.

Note que **uma aplicação pode ter várias tarefas**, dependendo das funcionalidades que disponibiliza, e ainda que **não é requisito que todas as atividades de determinada tarefa pertençam necessariamente à mesma aplicação**. Por exemplo, quando está a utilizar uma aplicação para mudança de *wallpaper*, esta pode, a determinada altura, evoluir para uma atividade da aplicação *gallery*. Este facto é dado como **uma das grandes potencialidades do Android™**.

Devido ao lugar de destaque que as atividades e, por conseguinte, as tarefas, ocupam, **o sistema contém um conjunto de mecanismos que ajudam o utilizador a navegar pelas várias atividades**. Um desses mecanismos é designado por **pilha de retrocesso de tarefas** (da designação inglesa *Task Backstack*¹⁰). O sistema **gere automaticamente o ciclo de vida** das atividades (ver em baixo), e garante que estas são **devidamente inicializadas, suspensas ou retomadas, bem como destruídas em caso de falta de memória**.

¹⁰Ver <http://developer.android.com/guide/components/tasks-and-back-stack.html>.

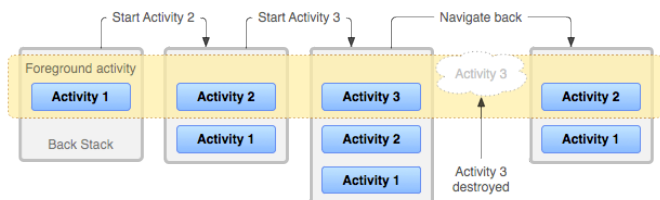


Figura 3: Representação da pilha de retrocesso de atividades mantida pelo Android™, bem como da sua evolução ao longo da execução de um tarefa (obtida de referência em nota de rodapé).

2.1 Pilha de Retrocesso de Tarefas

Task Backstack

Considere que determinado utilizador está a começar uma tarefa. Por exemplo, pode estar a começá-la partindo do ecrã *Home*, a partir do qual despoleta uma aplicação, que lhe mostra a primeira atividade (Activity_1). Esta atividade é a **primeira da tarefa e é colocada imediatamente no topo da pilha**. Se o utilizador **avança para outra atividade** através da interação com algum *widget*, a **atividade anterior é recalcada** (passada para segundo plano), e a nova atividade (Activity_2) passa para o topo da pilha. Repare-se que, como a Activity_1 **deixa de estar visível, o sistema suspende-a**, podendo vir a **retomá-la** mais tarde. Caso a tarefa avance para uma terceira atividade (Activity_3), as outras são recalcadas ainda mais na pilha, e assim sucessivamente. Eventualmente, **se uma ordem de navegar para trás for emitida, se atividade atual for destruída programaticamente ou se o sistema decidir, por algum motivo, terminar a atividade atual, a que estava imediatamente antes na pilha é retomada**, neste caso a Activity_2. Normalmente, a navegação para trás é conseguida através do botão *Back*. A figura ?? demonstra o funcionamento da pilha de retrocesso de tarefas para o exemplo que foi discutido antes.

Note que é o mecanismo *Backstack* que **permite uma navegação tão intuitiva nos dispositivos com Android™**, e que permitem que as tarefas sejam definidas como referido em cima. É também importante referir que **o ciclo de vida das atividades não está sob o controlo das aplicações que as usam, mas do próprio utilizador e do sistema** (e.g., que pode ter de terminar uma atividade suspensa por falta de recursos, que mais tarde terá de ser recreada se a tarefa voltar até esse ponto). Torna-se crucial que o programador conheça melhor o ciclo de vida de uma atividade, que se discute a seguir.

2.2 Ciclo de Vida de uma Atividade

Activity Lifecycle

Uma atividade pode estar em **um de 3 estados funda-**

mentais:

1. A atividade está **ativa ou em execução**, se estiver **totalmente visível em primeiro plano**;
2. Diz-se que está **pausada se perdeu o foco**, mas ainda está **parcialmente visível** (e.g., está parcialmente escondida por uma atividade de dimensões mais pequenas ou semi-transparente). Uma **atividade pausada mantém o estado e continua ligada ao gestor de janelas**, mas pode ser destruída se o sistema estiver com problemas de memória;
3. Diz-se que está **parada se estiver completamente tapada por outra atividade**. Neste caso também **mantém o estado, mas será mais depressa destruída** caso o sistema precise de memória.

Note que o sistema também **contém mecanismos que permitem que o estado das atividades que são destruídas**, por falta de recursos, **seja retomado** para aquele que tinham antes dessa decisão.

O diagrama incluído na Figura ?? esquematiza **os vários estados de uma atividade** e também as várias transições que podem percorrer, assim como **as ações que podem despoletar essas mudanças**.

O sistema Android™ **tenta manter os processos relativos a aplicações na memória o máximo tempo possível**, para **favorecer a agilidade** do sistema. Contudo, chegará ao ponto em que terá de **matar processos** antigos quando a memória estiver cheia ou próxima de se esgotar. A **decisão estará intimamente ligada com o estado de interação com o utilizador**. Os **4 estados seguintes** estão ordenados por precedência na decisão de destruir ou não determinada atividade:

1. Por vezes existem processos que já **não estão a lidar com quaisquer atividades ou outros componentes**. Estes processos são os primeiros a ser destruídos;
2. As **atividades em segundo plano não visíveis** podem ser eliminadas caso seja preciso, visto não estarem a ser usadas naquele momento. Se necessário, podem ser **retomadas novamente pelo sistema** adiante;
3. As atividades que possam estar **visíveis mas que não sejam o foco atual da interação** podem ter de ser eliminadas, e recuperadas adiante se necessário;
4. **Em último caso, o sistema elimina as atividades no topo da pilha de retrocesso de tarefas**, que são as que são consideradas como as mais importantes. Esta situação **pode acontecer quando o sistema está a fazer paging**, sendo a ação necessária para manter a fluidez da interface.

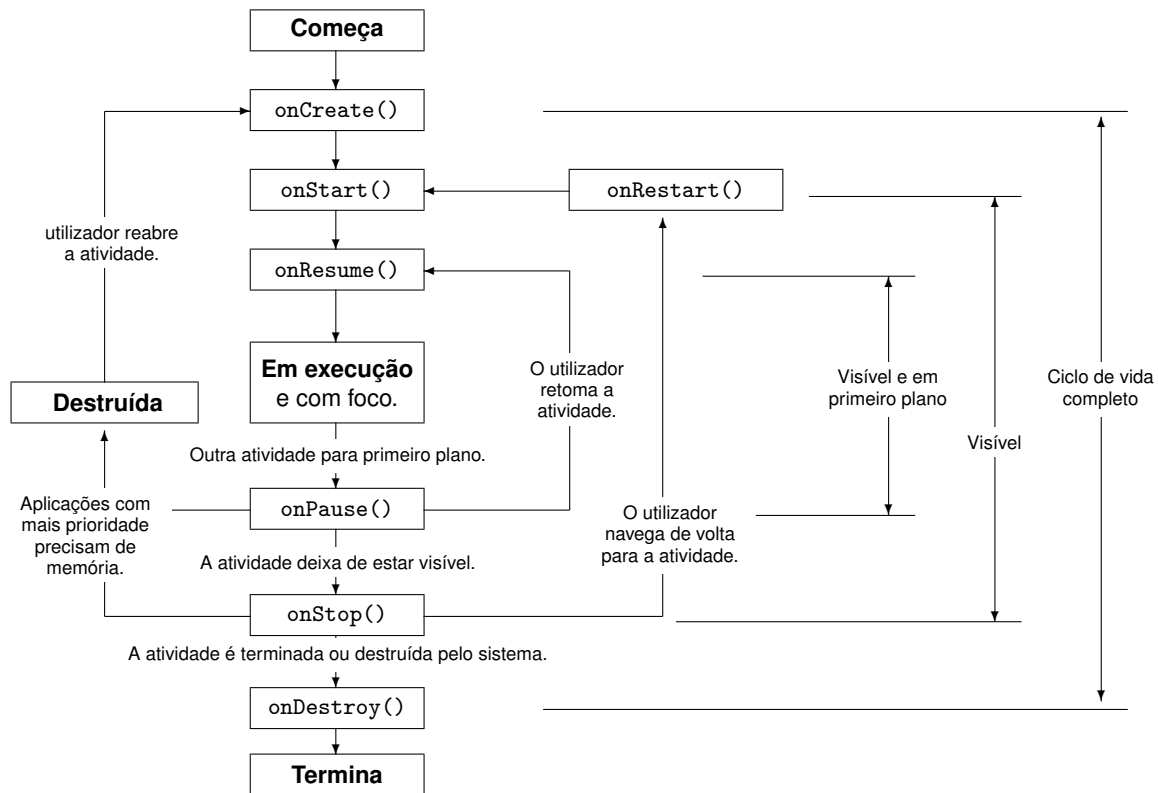


Figura 4: Ciclo de vida de uma atividade.

De modo a perceber melhor o diagrama, considere que iria utilizar uma aplicação que tinha um temporizador de 1 minuto, depois do qual terminava. Quando abria a aplicação, e.g., no seu ícone, o sistema despoletava automaticamente o método `onCreate(bundle)` da atividade principal (MAIN). Logo de seguida, era chamado o método `onStart()` e depois o `onResume()`. Neste ponto de execução, deveria existir já uma interface gráfica no ecrã relativa a esta aplicação. Considere que esperava então os restantes 55 segundos que faltavam para o minuto. Automaticamente, o sistema chamava então o método `onPause()`, seguido de `onStop()` e, finalmente, de `onDestroy()`. Assim que o método `onStop()` era chamado, a aplicação deixava de estar visível. Cada um dos métodos do ciclo de vida das Atividades é estudado com mais detalhe nas secções seguintes.

2.3 Método `onCreate()`

onCreate() Method

O método `onCreate(Bundle)` é **chamado quando uma atividade é criada pela primeira vez** (note que já não é mais chamado durante o ciclo de vida, mesmo que haja mudanças da atividade entre o primeiro e segundo plano). É neste método que **a configuração estática deve ser feita**, nomeadamente a **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 e recuperação do estado anterior**. Note que o método é chamado com um parâmetro da classe `Bundle`, forne-

cido pelo sistema Android™, e que pode conter o estado da atividade no momento em que esta foi pausada ou parada (sugere-se guardar o estado da atividade aquando da chamada ao método `onPause()`).

Este método é **sempre seguido de `onStart()` e deve obrigatoriamente conter uma chamada a `super.onCreate()`**, ou será lançada uma exceção, e a atividade poderá não funcionar. O trecho de código seguinte ilustra a implementação deste método para uma aplicação chamada `acalculator`.

```

package pt.ubi.di.pmd.acalculator;

import android.app.Activity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
import android.util.*;

public class ACalculator extends Activity
{
    Button oButton;
    EditText oTEdit1;
    EditText oTEdit2;
    TextView oTView1;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        oButton = (Button) findViewById(R.id.SUM);
        oTEdit1 = (EditText) findViewById(R.id.number1);
        oTEdit2 = (EditText) findViewById(R.id.number2);
        oTView1 = (TextView) findViewById(R.id.result);
    }
}
  
```

```

oButton.setOnClickListener(
    new View.OnClickListener()
    {
        public void onClick(View view)
        {
            double d1 = (new Double(oEdit1.getText()
                ().toString()).doubleValue());
            double d2 = (new Double(oEdit2.getText()
                ().toString()).doubleValue());
            double sum = d1 + d2;
            oTextView1.setText( sum + " " );
        }
    });
}

```

Há vários detalhes que podem ser enfatizados no exemplo dado, nomeadamente que (i) a classe `ACalculator` estende a classe `Activity`, (ii) que o código re-escreve (`@Override`) o método `onCreate()`, (iii) que um objeto chamado `savedInstanceState`, do tipo `Bundle` é passado como argumento ao método e, (iv), que o método `super.onCreate(Bundle)` é chamado logo no início. Como se pode constatar, esta implementação contém quase todos os passos referidos anteriormente, desde a inicialização do *layout* até à definição das rotinas de tratamento dos eventos nos objetos interativos.

2.4 Método `onStart()`

onStart() Method

O método `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**. Este método é sempre seguido de `onResume()` e deve obrigatoriamente conter uma chamada ao método análogo na sua super-classe (`super.onStart()`).

2.5 Método `onResume()`

onResume() Method

O método `onResume()` é **despoletado quando a atividade está a transitar de um estado invisível ou tapada para o primeiro plano**. Por isso, 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. Este método é sempre seguido de `onPause()`.

2.6 Método `onPause()`

onPause() Method

O método `onPause()` é **despoletado quando a atividade está a perder o foco**. A documentação oficial sugere que **não se faça nada demasiado moroso** no âmbito deste método, dado que o sistema não evolui para a nova atividade enquanto esta não terminar (`return`). Esta função é **normalmente usada para guardar dados persistentes que a atividade esteja a editar**, de

forma a permitir que, se for retomada, os dados que já haviam sido inseridos voltem a ser ajustados. A ideia é também garantir que, caso a atividade tenha de ser destruída, os dados referidos não são perdidos. Este método também concretiza o lugar ideal para **gerir a paragem de animações e outras operações que requeiram recursos de computação**, de forma a **agilizar a transição para a nova atividade**, ou para **fechar recursos externos que são de acesso exclusivo** como a câmara. O armazenamento do estado da aplicação pode normalmente ser feito recorrendo ao método `onSaveInstanceState(Bundle)`.

Este método é normalmente, mas não necessariamente, seguido de `onStop()`. Caso a atividade fique ainda visível, mas em segundo plano (e.g., uma pequena caixa de diálogo está na sua frente), então encontra-se num estado pausado e pode evoluir para `onResume()`, caso seja retomada. Só se a atividade ficar completamente invisível é que o método seguinte é chamado. Note que é possível que o sistema mate processos relativos a atividades que foram pausadas, em caso de necessidade expressiva.

Este método, tal como os anteriores, **deve chamar o seu análogo na super-classe**.

2.7 Método `onStop()`

onStop() Method

O `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, já que não estão a ser usadas de nenhuma forma a partir do `onStop()`. 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`.

Note que **não deve aguardar por este método para guardar o estado da atividade**, dado que este pode nunca vir a ser despoletado. A re-escrita deste método deve conter uma chamada a `super.onStop()`.

2.8 Método `onRestart()`

onRestart() Method

Dado que este método **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()`**, por exemplo. Se estão a ser usados ponteiros dinâmicos para recursos do sistema (e.g., conteúdo de uma base de dados), é aqui que se devem refrescar esses conteúdos. Como antes, deve conter uma chamada para `super.onRestart()`.

2.9 Método `onDestroy()`

`onDestroy()` *Method*

Finalmente, o método `onDestroy()` é **invocado quando a atividade está para terminar normalmente** (i.e., não forçosamente), quer programaticamente, quer por ação do utilizador (que carrega prolongadamente em back, por exemplo). Algumas ações básicas que se devem incluir aqui incluem **a libertação de recursos computacionais, nomeadamente *threads*** que tenham sido criadas no contexto da atividade. Note novamente que **este método pode não ser chamado caso a atividade seja terminada de modo abrupto**, pelo que **não deve conter a implementação de funcionalidades críticas**. A sua implementação deve conter a invocação de `super.onDestroy()`.

Nota: o conteúdo exposto na aula e aqui contido não é (nem deve ser considerado) suficiente para total entendimento do conteúdo programático desta unidade curricular e deve ser complementado com algum empenho e investigação pessoal.