

Introdução à Programação de Aplicações Android

2ª Edição

Apontamentos de Apoio e Guias Laboratoriais de Programação de Dispositivos Móveis

Pedro R. M. Inácio

Copyright © 2021 Pedro R. M. Inácio

Publicado como eBook
www.di.ubi.pt/~inacio

Este trabalho encontra-se licenciado ao abrigo de uma licença *Creative Commons* (CC) Atribuição-NãoComercial 4.0 Internacional. Esta licença determina que o trabalho pode ser partilhado e adaptado, desde que inclua o devido crédito ao autor, uma ligação para a licença e a indicação de que foram feitas alterações, quando aplicável. Qualquer partilha ou adaptação deverá ter fins não-comerciais e ser disponibilizada nos termos da mesma licença. Os termos podem ser consultados na íntegra em http://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.



Segunda edição, datada de outubro de 2021

Sinto que dedicar um livro técnico não sabe ao mesmo que dedicar um livro cheio de poesia. É como oferecer uma bigorna a uma criança. Mas, não tendo poesia melhor que esta, a vós dedico tudo o que faço:

Patrícia, Margarida e Olívia.

Acerca do Autor

Pedro Ricardo Morais Inácio é professor associado do Departamento de Informática da Universidade da Beira Interior (UBI). Leciona unidades curriculares relacionadas com segurança e garantia da informação, programação e simulação assistida por computador, a cursos de licenciatura, mestrado e doutoramento, nomeadamente aos de Engenharia Informática. É coordenador da equipa de resposta a incidentes CSIRT.UBI e instrutor da Academia CISCO @ UBI.

Tem uma licenciatura pré-Bolonha em Matemática Informática e um Doutoramento em Engenharia Informática, obtidos na Universidade da Beira Interior (UBI) em 2005 e 2009, respetivamente. Os trabalhos de Doutoramento foram desenvolvidos no ambiente empresarial da Nokia Siemens Networks Portugal S.A., ao abrigo de uma bolsa de investigação da Fundação para a Ciência e a Tecnologia.

É membro sénior IEEE, associado ACM e investigador do Instituto de Telecomunicações (IT). Os seus interesses de investigação incluem segurança e garantia da informação, simulação assistida por computador, e monitorização, análise e classificação de tráfego de rede. Tem cerca de 50 publicações em livros, revistas e conferências científicas internacionais com revisão por pares. Revê com frequência artigos para revistas IEEE, Springer, Wiley e Elsevier. Foi membro do Comité Técnico de diversas conferências e seminários nacionais e internacionais, como a ACM Symposium on Applied Computing - Track on Networking. É atualmente Editor Associado da revista IEEE Access.

inacio@di.ubi.pt | www.di.ubi.pt/∼inacio | @in4cio - Twitter | LinkedIn

Prefácio

A maior parte do conteúdo deste livro eletrónico foi elaborado durante a preparação da unidade curricular designada por *Programação de Dispositivos Móveis*, incluída no 3º ano do curso de Engenharia Informática da Universidade da Beira Interior. Por esse motivo, o tom da exposição é coloquial em algumas partes, ou até mesmo jocoso, se tal resultar, no entendimento do autor, em prol da pedagogia. É incluído um sumário no início de cada capítulo, o que também prova a adaptação quase direta dos referidos conteúdos para este formato. Os sumários estão em Português e em Inglês, apenas porque são também assim incluídos nas aulas, para ajudar eventuais estudantes de outras nacionalidades, de visita pelo programa Erasmus, a encontrar conteúdo sobre os temas abordados.

O texto está largamente estilizado, e é recorrente a utilização de **negrito** para **realçar várias partes** que o autor pensa serem importantes. Este conteúdo é fornecido como material de apoio ao estudo de uma unidade curricular, concretizando este livro um esforço para colar todo esse conteúdo, deixando alguns dos artefactos que fazem deles apontamentos.

Este trabalho encontra-se em constante desenvolvimento, e apesar de ter sido revisto várias vezes e por várias pessoas antes de ser disponibilizado, pode conter erros ou falhas, que serão resolvidas em iterações futuras. Caso queira reportar algum erro, ou apenas fazer chegar algum comentário ao autor, queira fazê-lo para inacio@di.ubi.pt.

Este livro está dividido em duas partes principais: a primeira parte concatena os conteúdos de apoio às aulas teóricas; enquanto que a segunda compila uma série de guias laboratoriais. A primeira parte é, por isso, de índole mais estático, de leitura e análise mais dura; enquanto que a segunda tem um tom mais ligeiro e interativo. A maior parte dos tópicos abordados na primeira parte encontram um paralelo prático na segunda. Cada guia laboratorial contém diversas tarefas que guiam o executante na d´ireção da implementação de aplicações Android simples, intercaladas com perguntas de escolha múltipla ou direta, cujo objetivo é o de ajudar a estruturar o caminho. Irão perdoar o tom mais brincalhão desta parte em relação à primeira. Sugere-se, claro, a execução prática dos vários guias laboratoriais

Finalmente, aqui fica expresso o agradecimento aos revisores deste documento. Sem nenhuma ordem em particular, para além da alfabética, um *Muito Bem Haja* a: Diogo A. B. Fernandes, Joana Costa, Liliana F. B. Soares, Luís Perez, Miguel Neto, Pedro Tavares e Tiago Roxo.

Conteúdo

I Apontamentos Teóricos

1 Definição de Dispositivo Móvel, Âmbito e Motivação 3

1 Introdução . . . 3
2 Objetivos de Aprendizagem . . . 3
3 Motivação para Estudar Programação para Dispositivos Móveis . . . 4
4 Razões para o Foco na Plataforma Android . . . 5
5 Definição de Dispositivo Móvel . . . 7
6 Ferramentas Úteis . . . 9

2 Desenvolvimento de Interfaces 11

1 Introdução . . . 11
2 Programas Sequenciais . . . 12
3 Interfaces de Utilizador Orientadas por Eventos . . . 14

3 Modelo-Visão-Controlador 21

1 Introdução . . . 21
2 Modelo . . . 22
3 Visão . . . 22
4 Controlador . . . 23

5 Comunicação entre Componentes 23

4 A Plataforma Android 25

1 Introdução 25
2 Pilha de *Software* 25
3 Camada do Núcleo Linux 26
4 Bibliotecas Nativas 26
5 *Framework* Aplicacional 28
6 Camada de Aplicação 29

5 Componentes das Aplicações Android 31

1 Introdução 31
2 Atividade 32
3 Serviço 33
4 Fornecedor de Conteúdos 33
5 Recetor de Difusão 34
6 Processo de Preparação de uma Aplicação Android 35
7 `R.java` e `strings.xml` 37
8 `AndroidManifest.xml` 39

6 Depuração de Aplicações Android 41

1 Introdução 41
2 Dispositivos Virtuais Android 41
3 Logcat 42
4 Inspetor de *Layout* 44
5 Servidor do Monitor de Depuração *Dalvik* 45
6 Monitor de Dispositivos Android 45
7 Funcionalidades Avançadas dos Dispositivos Virtuais Android 45

7 A Componente Atividade 49

1 Introdução 49
2 Pilha de Retrocesso de Tarefas 50
3 Ciclo de Vida de uma Atividade 51
4 Método `onCreate()` 53
5 Método `onStart()` 54

6 Método `onResume()` . . . 55
7 Método `onPause()` . . . 55
8 Método `onStop()` . . . 55
9 Método `onRestart()` . . . 56
10 Método `onDestroy()` . . . 56

8 Intentos 57
1 Introdução . . . 57
2 Instanciação de Intentos . . . 59
3 Intentos Explícitos . . . 61
4 Intentos Implícitos . . . 62
5 Envio de Dados Via Intento . . . 64
6 Obtenção de Resultados Via Intento . . . 65

9 Segurança em Android 67
1 Introdução . . . 67
2 Controlo de Acesso e IDs do Utilizador . . . 68
3 Assinatura Digital da Aplicação . . . 69
4 Pedir Permissões no Manifesto . . . 71
5 Definir Permissões no Manifesto . . . 72
6 Aplicar Permissões a Componentes . . . 74

10 Armazenamento de Dados Persistentes 77
1 Introdução . . . 77
2 Preferências Partilhadas . . . 78
3 Armazenamento Interno – Escrita . . . 80
4 Armazenamento Interno – *Cache* . . . 81
5 Armazenamento Interno – Leitura . . . 82
6 Armazenamento Externo . . . 82

11 Armazenamento de Dados Estruturados 85
1 Introdução . . . 85
2 SQLite . . . 86
3 Linguagem Estruturada de Consultas . . . 87
4 Bases de Dados SQLite em Android . . . 91

5 `SQLiteDatabase` . . . 95
6 Cursores . . . 97
7 Depuração de Bases de Dados . . . 98

12 **A Componente Serviço** **101**
1 Introdução . . . 101
2 Definição de Serviço . . . 103
3 Ciclo de Vida de um Serviço . . . 104
4 Criar um Serviço sem Vínculo . . . 106
5 A Classe `IntentService` . . . 110
6 Criar um Serviço com Vínculo . . . 111
7 Terminar um Serviço . . . 116

13 **Notificações** **119**
1 Introdução . . . 119
2 Notificações via Mensagens Flutuantes . . . 119
3 Notificações na Barra de Estado . . . 121
4 Correr o Serviço em Primeiro Plano . . . 124

14 **A Componente Recetor de Difusão** **127**
1 Introdução . . . 127
2 Registo no Manifesto e Implementação . . . 129
3 Registar Recetores de Difusão Programaticamente . . . 130
4 Enviar Intentos em Difusão . . . 132

15 **A Componente Provedor de Conteúdos** **137**
1 Introdução . . . 137
2 Criar um Provedor de Conteúdos . . . 138
3 Aceder a um Provedor de Conteúdos . . . 143
4 URIs dos Conteúdos . . . 147

16 ***Framework* de Sensores** **149**
1 Introdução . . . 149
2 Sensores em Android . . . 150

3 Constituição da *Framework* de Sensores 152
4 A Classe `Sensor` 155
5 A Interface `SensorEventListener` 157
6 A Classe `SensorEvent` 160
7 Testar a Implementação 161

II Guias Práticos Laboratoriais

17 Programação de Aplicações Android num IDE 165
1 IDE Eclipse, AVD e SDK 166
2 Olá Planeta Terra 168

18 Agora sem Mãos! 171
1 Instalação do Apache Ant 172
2 Explorando o Gradle™ 173
3 Explorar o SDK Android 175
4 Preparação de um Pacote de Instalação Android via Linha de Comandos . . 176

19 Introdução ao Desenvolvimento de Interfaces Android 181
1 Preliminares 182
2 Introdução às Interfaces de Utilizador Orientadas por Eventos 183

20 Consumidores de Eventos e Ciclo de Vida de Atividades 191
1 Rotinas de Tratamento de Eventos 192
2 Registo do Sistema e Ciclo de Vida de uma Atividade 197

21 Intentos Implícitos e Explícitos 203
1 Intentos Implícitos 204
2 Intentos Explícitos 207

22 Filtros de Intentos, Intentos com Retorno e Permissões 215
1 Filtros de Intentos 216
2 Intentos com o Retorno 218

3 Pedir Permissões no Manifesto 223

23 Preferências Partilhadas, Armazenamento Interno e Externo 231

1 Ficheiros Disponibilizados como Recurso do Projeto 231
2 Preferências Partilhadas 236
3 Armazenamento Interno 238
4 Armazenamento Externo 240

24 Bases de Dados SQLite 243

1 Criação de Bases de Dados Locais SQLite 243
2 Inserção de Dados em Bases de Dados SQLite 248
3 Depuração da Base de Dados 251
4 Listagem e Remoção de Registos 254
5 Edição de Registos 261

25 Serviços e Recetores de Difusão 263

1 Componente Serviço 263
2 Serviços e *Threads* 267
3 *UI Thread* e *Handlers* 269
4 Recetores Difusão 272

Lista de Figuras

1.1 Evolução e estimativa das tendências em termos de dispositivos ligados à Internet (fonte Forbes). 4

1.2 Domínio de mercado das plataformas móveis no primeiro quartil de 2014 (fonte *Business Insider*). 5

1.3 *Top* 10 (e *top* 30) das linguagens de programação em 2014, de acordo com a revista Spectrum (fonte *Institute of Electrical and Electronics Engineers* (IEEE) Spectrum). 6

1.4 Representação visual do fenómeno da fragmentação na plataforma Android em 2014 (fonte *Open Signal*). 7

2.1 Exemplo de piada relativo a dificuldades encontradas na utilização de `vi`/`vim`. 14

2.2 Hierarquia dos objetos interativos numa interface gráfica de uma aplicaçao móvel. 17

4.1 Pilha de *software* da plataforma Android. 27

5.1 Processo de compilação simplificado de uma aplicação Android. 35

5.2 Processo de preparação de uma aplicação Android. 37

6.1 Captura de ecrã da ferramenta de depuração da hierarquia de elementos da interface de utilizador em execução. 44

6.2 Captura de ecrã da ferramenta de *profiling* Android Device Monitor. 46

6.3 Captura de ecrã da ferramenta de depuração Android, embutida neste caso no Android Studio. 46

6.4 Exemplo do aspeto da janela de controlos de um dispositivo emulado, ilustrando a panóplia de configurações e ajustes possíveis. 47

7.1 Representação da pilha de retrocesso de atividades mantida pelo Android, bem como da sua evolução ao longo da execução de uma tarefa (obtida de referência em nota de rodapé). 51
7.2 Ciclo de vida de uma atividade. 52

8.1 O mecanismo inerente ao uso de `intents` para despoletar atividades em Android. 59

11.1 A instrução SQL de `SELECT` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_select.html). 89
11.2 A instrução SQL de `INSERT` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_insert.html). 90
11.3 A instrução SQL de `UPDATE` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_update.html). 91
11.4 A instrução SQL de `DELETE` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_delete.html). 91

12.1 Os ciclos de vida de um serviço Android. À esquerda é mostrado o ciclo de vida para serviços criados com `startService()`, enquanto que à direita está o ciclo de vida para o caso em que é criado com `bindService()`. 105

Parte I

Apontamentos Teóricos



1 Definição de Dispositivo Móvel, Âmbito e Motivação

Sumário
Âmbito e motivação para estudo dos temas que a unidade curricular aborda. Definição de dispositivo móvel.

Summary
Scope and motivation for studying the topics covered by this course. Definition of mobile device.

1 Introdução

Em jeito de introdução, este capítulo apresenta os principais objetivos de aprendizagem que se pretendem atingir com estes apontamentos e exercícios práticos, bem como a motivação (como se realmente necessária) para estudar programação para dispositivos móveis. Mais para o fim deste capítulo, define-se dispositivo móvel de uma maneira um pouco mais formal, embora o conceito não se aplique tão linearmente quanto apresentado, nos dias de hoje.

2 Objetivos de Aprendizagem

O objetivo desta unidade curricular é **abordar o desenvolvimento de aplicações nativas para plataformas móveis**, enfatizando detalhes específicos relativos ao *design*, estrutura, recursos e linguagens utilizadas nesse desenvolvimento. Pretende se que o(a) aluno(a) **transporte e aplique conhecimento** previamente adquirido noutras áreas

para a **engenharia e implementação** dessas aplicações. Pretende-se também cativar no(a) aluno(a) **a sensibilidade para problemas** específicos a esta área, nomeadamente no que se refere ao **design e limitações típicas de aplicações móveis**.

No final da unidade curricular o estudante deve ser capaz de:

- **Projetar** e **implementar autonomamente** aplicações para dispositivos móveis;
- **Trabalhar com uma equipa na engenharia e desenvolvimento de *software*** ou sistema direcionados para dispositivos móveis;
- **Utilizar**, com facilidade, **ambientes gráficos de desenvolvimento** integrado ou a **interface de linha de comandos** para desenvolver aplicações móveis;
- Lidar com **detalhes relativos ao armazenamento e comunicações** em dispositivos móveis com facilidade;
- Tirar partido dos **recursos multimédia e sensores** disponibilizados por estes dispositivos nas aplicações que desenvolver.

3 Motivação para Estudar Programação para Dispositivos Móveis

Os **dispositivos móveis**, como *smartphones* e *tablets*, constituem hoje em dia **o meio mais comum e natural para as pessoas interagirem com aplicações e serviços informáticos**. **A procura de programadores com conhecimentos em desenvolvimento de aplicações móveis está**, portanto, **a crescer a um ritmo acentuado**. Contudo, esta competência requer a **mestria** de muitos princípios da **engenharia** e da **informática**, a **aprendizagem dos detalhes específicos** a algumas plataformas móveis e a **capacidade artística** de desenhar interfaces cativantes que respondem simultaneamente ao *como*, *onde* e *porquê* de determinada aplicação móvel.

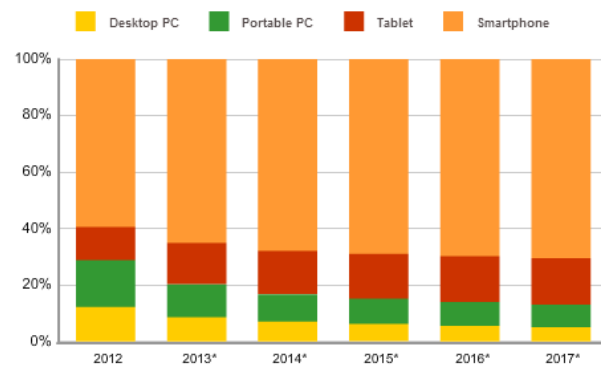


Figura 1.1: Evolução e estimativa das tendências em termos de dispositivos ligados à Internet (fonte Forbes).

A primeira parte do parágrafo anterior é facilmente justificada com recurso a dados atuais e a previsões, como o que é incluído no artigo da Forbes, cujo título sugere que em **2017,**

87% dos dispositivos ligados são *tablets* ou *smartphones*. A figura 1.1, incluída no artigo, da responsabilidade da *International Data Corporation* (IDC), uma firma de análise estatística e inteligência em mercados globais, mostra a evolução das compras dos últimos anos, em termos de dispositivos móveis, e estima as tendências para o futuro próximo. A **aposta nestes dispositivos parece ganha**.

Esta unidade curricular pretende **cobrir os princípios fundamentais da programação**, arquitetura de *software* e considerações subjacentes à experiência do utilizador, bem como dos seus **ambientes de desenvolvimento**. De forma a abordar todos os conceitos, este curso vai envolver a **implementação prática de aplicações para a plataforma Android**.

4 Razões para o Foco na Plataforma Android

Podem-se enumerar pelo menos **4 razões** que justificam a colocação de algum foco no desenvolvimento de aplicações para a plataforma Android. Estas razões não devem, contudo, ser tomadas em detrimento da qualidade de outras plataformas móveis, como o iOS ou Windows Mobile.

Em **primeiro lugar**, **a plataforma Android é gratuita, e é simples encontrar ferramentas também gratuitas e prontamente disponíveis** para desenvolvimento de aplicações. Em **segundo lugar**, esta é atualmente **a plataforma que domina o mercado global em termos de *smartphones* e *tablets***, conforme ilustra o gráfico da figura 1.2, conseguido pela IDC, e analisado num artigo da *Business Insider*. **80% do mercado global** estava a usar Android no primeiro quartil de 2014.

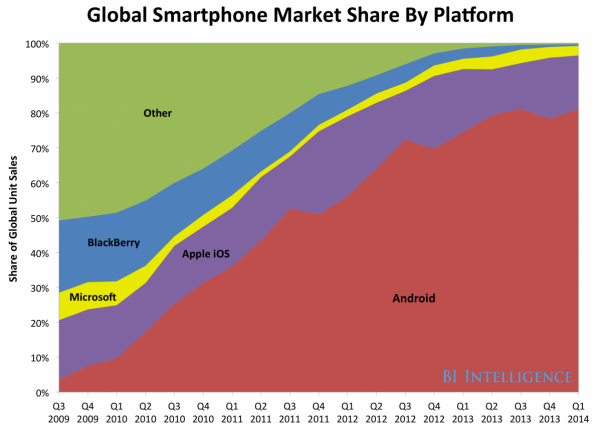


Figura 1.2: Domínio de mercado das plataformas móveis no primeiro quartil de 2014 (fonte *Business Insider*).

Talvez mais como consequência do que como razão, o facto da programação de aplicações nativas para Android ser feita em `Java` também resulta em benefício da escolha feita.

Como demonstra o gráfico da figura 1.3, retirado de um artigo da *Institute of Electrical and Electronics Engineers* (IEEE) Spectrum, esta linguagem de programação **é a**

mais utilizada hoje em dia, seguida de C e de C++, Objective-C, que é usada para programação nativa para iOS, aparece em 16º lugar.

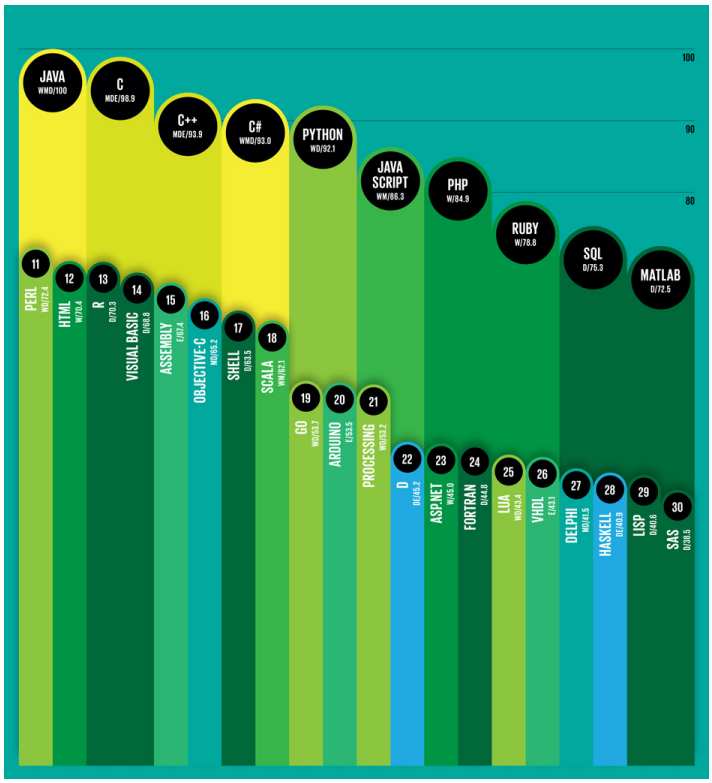


Figura 1.3: *Top* 10 (e *top* 30) das linguagens de programação em 2014, de acordo com a revista Spectrum (fonte *Institute of Electrical and Electronics Engineers* (IEEE) Spectrum).

Por último, e em vez de a tomar como **uma desvantagem**, pode-se dizer que **a enorme fragmentação**, quer em termos de **versões de Sistema Operativo (SO)**, quer em termos de ***hardware* e das suas especificações** (ecrã incluído), fazem do Android **a plataforma ideal para ganhar alguma destreza com a programação para dispositivos móveis**. Os **desafios** associados com a fragmentação **são maiores para esta plataforma** do que para as outras duas concorrentes diretas. Para dar uma ideia deste problema, incluí-se o gráfico da figura 1.4, retirado de um relatório da *Open Signal*, que apresenta, de uma forma gráfica, a enorme quantidade de dispositivos no mercado (cada retângulo é um dispositivo diferente) e de versões do SO em utilização (cada cor corresponde a uma versão do SO), bem como a cota de mercado (a área do retângulo é diretamente proporcional à cota de mercado).

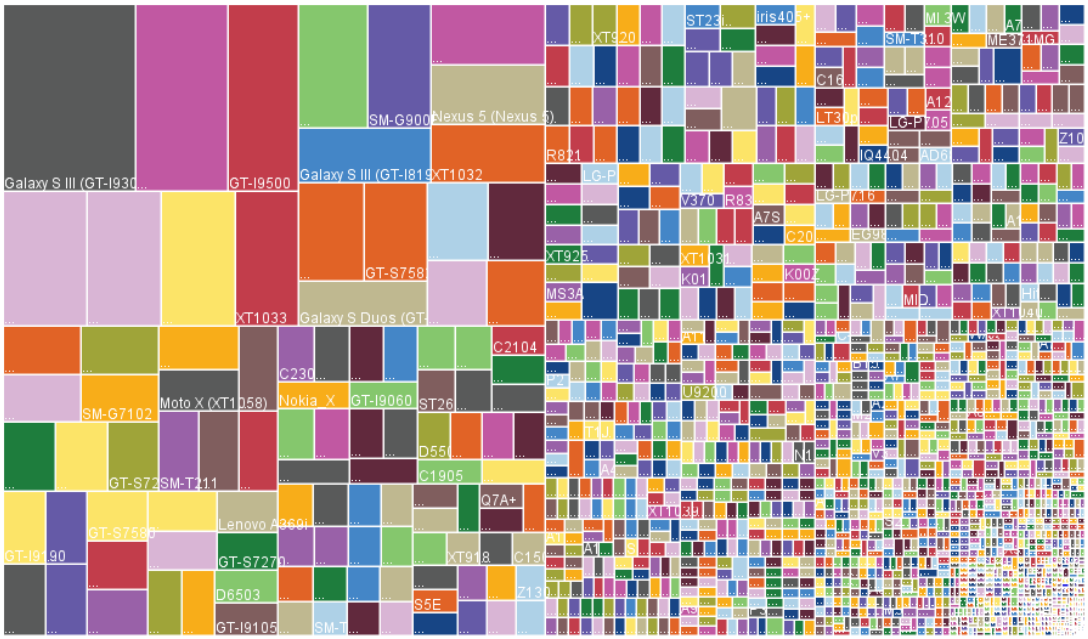


Figura 1.4: Representação visual do fenómeno da fragmentação na plataforma Android em 2014 (fonte *Open Signal*).

5 Definição de Dispositivo Móvel

Hoje em dia é muito fácil encontrar objetos a que chamamos de dispositivos móveis.

De uma maneira geral, pode-se dizer que **um dispositivo móvel**, normalmente designado em inglês por ***handheld***, é **um aparelho eletrónico com as seguintes características**:

Computação – possui a **capacidade de processar e armazenar dados**;

Interatividade com Utilizador – **possuí dispositivos de entrada e saída de dados** embutidos, nomeadamente um ecrã, colunas de som, teclado (ainda que virtual), microfone e câmara fotográfica, entre outros possíveis sensores, que são **usados para interagir com um utilizador**. Dispositivos que não permitem a interatividade com um utilizador não são normalmente considerados dispositivos móveis;

Portabilidade – são sobretudo **definidos pela sua passividade de serem movidos frequente e facilmente**. Qualquer dispositivo móvel deve funcionar consistentemente quer em repouso, quer em movimento, independentemente da proximidade a uma fonte de alimentação ou ligação física à Internet. Os dispositivos móveis têm normalmente baterias recarregáveis que permitem várias

horas em funcionamento sem acesso a uma fonte de alimentação externa;

Dimensões e peso – os dispositivos móveis têm dimensões e peso que permitem a sua portabilidade e incluem normalmente componentes eletrónicos com **maior grau de miniaturização** do que dispositivos análogos não móveis. Algumas definições de dispositivos móveis ditam que apenas os dispositivos que podem ser **utilizados com uma só mão é que devem ser considerados móveis**. Neste caso, computadores portáteis com um tamanho que inviabilize a sua utilização com uma só mão não podem ser considerados dispositivos móveis;

Comunicações sem fios – **possuem normalmente tecnologia que permite a comunicação com outros dispositivos semelhantes, computadores e sistemas** (estacionários), com **redes de computadores ou de telecomunicações, ou com telefones portáteis**. Hoje em dia, estes dispositivos são capazes de acederem à Internet através de redes *Bluetooth* e *Wi-Fi*, e muitos modelos são capazes de aceder a redes celulares (*e.g.*, no caso dos *smartphones*).

Alguns exemplos comuns de dispositivos móveis são: *smartphones*, *Personal Digital Assistants* (PDAs); telemóveis, consolas portáteis, *ultrabooks*, *notebooks* e *netbooks*.

Como será melhor elaborado de seguida, a **área da programação para dispositivos móveis** é uma das que **maior desenvolvimento tem conhecido no passado recente**, sem sinais de abrandamento. Esta área é, hoje em dia, o **foco de intensa investigação**, sofrendo de um dinamismo fora do comum. A **panóplia de ferramentas e tecnologias** que podem ser utilizadas na programação de dispositivos móveis é vasta. Por um lado, diferentes dispositivos móveis têm diferentes formas de aceder às suas funcionalidades, por outro, a **proliferação destes dispositivos deu-se no seio da existência de várias tecnologias** (nomeadamente linguagens de programação) com **provas dadas, diferentes, mas com algumas funcionalidades equivalentes**.

Um dos **maiores desafios** do desenvolvimento de aplicações para dispositivos móveis (daqui em diante designadas por *aplicações móveis*) refere-se **ao teste dessas aplicações durante o seu desenvolvimento**. Ao contrário de aplicações para computadores portáteis, de secretária ou servidores, por exemplo, **as aplicações móveis não são desenvolvidas no mesmo ambiente onde irão ser tipicamente executadas**. Devido a isso, as ferramentas de desenvolvimento dessas aplicações precisam conter **formas de virtualizar esses ambientes** ou **ferramentas de instalação rápida em dispositivos físicos**, embora já existam alguns IDEs para desenvolvimento no próprio dispositivo alvo, com as limitações e vantagens óbvias.

6 Ferramentas Úteis

Como de resto evidenciado antes, a componente prática e parte da componente teórica debruçar-se-ão sobre a **plataforma Android**. Sugere-se, assim, a **instalação do *Software Development Kit* (SDK)** para esta plataforma no computador pessoal que utilizar como recurso de aprendizagem. Como também ficará claro nas aulas práticas, **o uso de emuladores ou de um dispositivo físico** para testes das aplicações será extremamente útil. Para a plataforma em questão existem **diversas soluções** disponíveis **gratuitamente** na Internet (algumas incluídas com, ou que incluem, o SDK).

Em termos de ambientes de desenvolvimento integrados, sugere-se o uso daquele que é atualmente o **mais recomendado** para o efeito, de nome ***Android Studio***, que pode ser obtido de `http://developer.android.com/sdk/installing/studio.html`. Como alternativa, pode indicar-se o **IntelliJ IDEA**, disponível em `https://www.jetbrains.com/idea/download/`, e cuja versão *community* é gratuita. Na verdade, o *Android Studio* é baseado na edição *community* código aberto deste IDE. Em termos de emuladores, pode optar pelos que já vêem incluidos pelo SDK (cada vez mais velozes).

Note, contudo, que **é livre de escolher a plataforma para a qual desenvolve as aplicações móveis pedidas dos trabalhos práticos**, bem como a linguagem de programação que está associada, bem como **o ambiente de desenvolvimento integrado (ou não) que utiliza**.



2 Desenvolvimento de Interfaces

Sumário

Discussão do tema relativo ao desenvolvimento de Interfaces de Utilizador como forma de introduzir conceitos específicos ao desenho de aplicações interativas, partindo das Interfaces de Utilizador para programas sequenciais.

Summary

Discussion of the subject concerning user interface development as an excuse to introduce concepts specific to the design of interactive applications, starting with the user interfaces for sequential programs.

1 Introdução

Um dos mais importantes componentes de aplicações móveis, pelo menos daquelas que interagem com o utilizador, é a *interface do utilizador*. Note-se que **nem todas as aplicações móveis têm de ter uma destas interfaces**, já que algumas são apenas implementadas para **fornecer serviços ou executar tarefas em segundo plano**. A definição deste componente é simples:

> Uma *interface do utilizador* é um programa de computador, ou parte dele, que permite que o utilizador comunique com esse computador ou com outras partes do programa.

As *interfaces de utilizador* têm **evoluído muito** ao longo da história da informática, sendo essa evolução motivada quer **por desenvolvimentos de *hardware* quer de *software*.** De modo a introduzir o *desenvolvimento de interfaces de utilizador utilizando programação orientada por eventos*, talvez seja indicado começar por referir o modelo

de interfaces usado em programas sequenciais. Assim, a próxima subsecção discute o modelo mencionado em último, enquanto que a seguinte discute o anterior.

2 Programas Sequenciais

Os modelos de interação **para programas sequenciais**, baseados em dispositivos de entrada como o teclado, **são bastante simples**. Note-se, neste caso, são os programas que controlam o fluxo de interação de uma forma bastante rígida. Os utilizadores esperam que o programa peça *inputs* para aí interagirem com o mesmo (controlo interno). Em termos conceptuais e usando pseudo-código, o fluxo de execução destes programas pode ser ilustrado da seguinte forma:

```
while(true)
  pedir input
  ler input
  analisa (e processa) input
  toma ação em conformidade
  (eventualmente gera saidas)
```

Um dos **maiores problemas** desta abordagem é que se torna **difícil modelar, do ponto de vista da utilização do programa, formas de captar ações do utilizador com índole substancialmente diferente** (*e.g.*, editar texto e salvar o texto). Dada a sua arquitetura, isso será apenas possível recorrendo a uma **estratificação sequencial das funcionalidades oferecidas**. *I.e.*, o utilizador precisa normalmente percorrer um fluxo modelado por uma árvore até chegar à funcionalidade que pretende, nas folhas da mesma. Por outro lado, **esta abordagem permite sempre que o programa funcione em linha de comandos**, o que pode constituir uma vantagem em muitos casos e para muitos cenários de aplicação, embora não frequentemente para dispositivos móveis. **Alguns Sistemas Operativos** (SOs) funcionam apenas em linha de comandos (*e.g.*, o *Internetwork Operating System* (IOS)), e **a própria *shell* do SO usa esta abordagem**.

Existem muitos exemplos de aplicações bastante complexas que usam este modelo de interação. O editor de texto `vi` concretiza um desses exemplos. **A estratificação é por vezes conseguida através da introdução de *modos***, que permitem que o utilizador **navegue para um determinado conjunto** de ações a partir de determinado ponto de execução. O fluxo de execução desses programas pode ser representado, de uma forma geral, como se mostra a seguir:

```
while(true)
  label 1
  modo 1:
```

```
    while(true)
      pedir input
      ler input
      analisa (e processa) input
      toma ação em conformidade
      if( muda de modo )
        goto 2
      (eventualmente gera saidas)

  label 2:
  modo 2:
    while(true)
      pedir input
      ler input
      analisa (e processa) input
      toma ação em conformidade
      if( muda de modo )
        goto 1
      (eventualmente gera saidas)
  ...
```

Embora o modelo de interação descrito **escale para qualquer conjunto de ações** (**à custa de variáveis de estado e de alguma complexidade programática**), **não é muito intuitivo** e **dá azo a confusões e erros**. O editor de texto `vi`, por exemplo, tem dois modos diferentes: o de *introdução de texto* e o de *entrada de comandos* e, para um principiante, é comum a situação de estar a tentar introduzir texto no modo *entrada de comandos* ou vice-versa; enquanto que num ambiente gráfico essa confusão poderia nem existir. Para além disso, a implementação do modelo **requer** normalmente **a especificação de muitas variáveis de estado** que controlam o caminho percorrido na árvore. A própria forma de como a mudança de modo deve ser programada nem sempre é limpa, na medida em que pode não ser feita sempre da mesma forma (*e.g.*, no `vi`, a mudança de modo para *inserção de comandos* é feita pressionando a tecla `Esc`, enquanto que a mudança para o modo de *inserção* é feita escrevendo `i` e pressionando `Enter`). As dificuldades associadas à utilização de `vi` tem sido inclusive alvo de piadas nas redes sociais, como exemplificado na figura 2.1.

Alguns **dispositivos físicos** também usam esta abordagem por modos na sua interface do utilizador. Por exemplo, os **comandos de televisão, *boxes* e equipamentos multimédia** contêm frequentemente um botão de mudança de modo para cada um dos dispositivos suportados. Note que algumas interfaces de utilizador de aplicações móveis ainda usam esta abordagem, principalmente em dispositivos mais simples, sem ecrã tátil. Por exemplo, alguns telemóveis para idosos contêm funcionalidades bastante avançadas, mas a aplicação de marcação de números de telefone, uma vez acedida, fica a aguardar a inserção do número de telefone pretendido ou o seu cancelamento. No caso de um



Figura 2.1: Exemplo de piada relativo a dificuldades encontradas na utilização de `vi`/`vim`.

smartphone, a mesma aplicação pode simultaneamente aguardar que se pressione prolongadamente o ecrã para despoletar a opção de colar (*paste*) o que estiver no *clipboard*.

3 Interfaces de Utilizador Orientadas por Eventos

Os desenvolvimentos tecnológicos que permitiram a **produção em massa e adoção de ecrãs táteis** constituem pontos chave na história das interfaces de utilizador, contribuindo ainda mais para a solidificação do modelo de interação orientado por eventos. Este modelo, utilizado na maior parte das aplicações móveis modernas, permite disponibilizar um **conjunto maior de funcionalidades em simultâneo e de forma intuitiva, correspondendo ainda a uma simplificação na estrutura do programa**. Neste caso, é **o programa que espera pelo utilizador** (controlo externo), nomeadamente que este lhe forneça um dos possíveis *inputs*.

A **manipulação** destas interfaces pode ser **feita de forma direta**, *i.e.*, para cada ou alguns dos objetos mostrados ao utilizador. Tecnologicamente, tal é conseguido através de **mecanismos de comunicação entre objetos interativos e o sistema de entrada e saída** (*e.g.*, um ecrã tátil[1]), concretizada por eventos:

> No contexto da informática e, particularmente, programação, **um evento corresponde a uma ação ou ocorrência que um programa foi capaz de detetar e tratar**. Um evento tem algumas propriedades que, em última análise, definem o seu tratamento por parte de um programa, nomeadamente **o tipo de acontecimento** (*e.g.*, clique de um dos botões do rato, pressionar de uma tecla), **a posição do cursor ou tecla pressionada**, o **programa ao qual se destina** o evento, etc.

A rotina principal de captação de eventos é da responsabilidade do SO, que

[1] Note que um ecrã tátil é, simultaneamente, um dispositivo de entrada e saída.

a executa constantemente e **coleciona os eventos numa pilha *First In First Out*, assegurando que estes são tratados pela ordem que chegaram**. Note-se que alguns eventos podem ser descartados (*e.g.*, um clique numa área do *desktop* que não tem nenhuma ação associada). De uma maneira abstrata, o modo de funcionamento desta rotina pode ser representada como se mostra a seguir:

```
while(true)
  espera por evento
  envia evento para lista de eventos
  analisa evento e envia para programa
  (se o programa estiver a dormir, acorda-o)
```

Ao receber um evento, o programa fá-lo **chegar à rotina de tratamento** a que corresponde:

```
while(true)
  espera por evento
  if( existir evento )
    recebe e analisa evento
    envia evento para rotina de tratamento
  else
    sleep
```

Note que, no caso da programação orientada por eventos, um programa pode entrar no estado de *sleep* quando não existem eventos, pelo que o Sistema Operativo pode atribuir tempo de processamento a outros programas.

A programação baseada por eventos é normalmente feita recorrendo àquele que é conhecido por ***modelo de delegação de eventos***, **baseado nas entidades *Controlos*, *Consumidores* e *Interfaces*** (de programação). Os ***controlos* serão a fonte do evento** (*e.g.*, um botão), os ***Consumidores* (*Listeners*) correspondem às rotinas de tratamento desses eventos** (*i.e.*, consomem – no sentido de tratarem – essas ocorrências) e as ***interfaces* são a forma normalizada de comunicação entre as duas entidades** antes referidas. Aquando da implementação, **o programador tem de definir os controlos e registar o conjunto de eventos que deseja escutar** para cada um deles, bem como **implementar a interface** (de programação) do evento que pretende escutar.

O código Java seguinte mostra um exemplo simples de como o parágrafo anterior se concretiza em termos de implementação.

```
ckb1 = (Checkbox) findViewById(R.id.chk1);
ckb2 = (Checkbox) findViewById(R.id.chk2);

ckb2.setOnClickListener(new OnClickListener(){
```

```
  @Override
  public void onClick( View v ){
    if( (CheckBox v).isChecked() )
      ckb1.setEnabled(false);
  }
});
```

No trecho declaram-se dois *check buttons*, que estão definidos num ficheiro *eXtensible Markup Language* (XML) e que são automaticamente configurados através do método `findViewByID()`. De seguida, é invocado o método `setOnClickListener` que regista um novo *listener* para o `ckb2`, redefinindo imediatamente o *handler* para o evento clique no rato. Esta redefinição é feita através de uma classe anónima e re-escrita do método `onClick()` (da interface `OnClickListener`).

O código incluído anteriormente pode parecer confuso a partir da quarta linha mas, basicamente, o que se está a fazer a partir desse ponto é condensar as seguintes atividades:

1. Invocar o método `setOnClickListener( p )` com um parâmetro, que tem de ser da classe que implementa a interface referida a seguir;
2. Inicializar um novo objeto da classe `OnClickListener`;
3. Aproveitar para redefinir, imediatamente, o método `onClick()` da interface.

A classe resultante é **conhecida**, no seio do jargão Java, como **classe anónima**. A definição de classes desta forma **é útil em várias situações**, nomeadamente **quando não faz sentido um determinado objeto existir quando o que o invoca não existe**, ou quando se quer **definir parte do comportamento da classe junto do local onde é declarada**.

Note-se que esta forma de programar e construir interfaces de utilizador é **extremamente útil para ambientes gráficos** já que, por um lado, **permite alguma abstração ao programador**, que **define apenas o que deve ser feito para os eventos que lhe interessam** e, por outro, permite **aproveitar o espaço gráfico da aplicação** para transmitir as suas funcionalidades de forma efetiva. As *frameworks* ou **bibliotecas existentes fornecem já a maior parte das classes e métodos necessários**, ficando a faltar a implementação de alguns desses métodos. A **captura e reencaminhamento dos eventos é feito de forma transparente para o programador**, que não precisa de se preocupar com esses detalhes.

É claro que a programação orientada por eventos assenta em vários **pressupostos**. Um dos mais importantes é que **a interface gráfica é composta por vários objetos interativos e elementos cuja localização é conhecida**, e que podem ser **estruturados hierarquicamente em árvore** (*e.g.*, no universo Android é mencionada uma

User Interface Layout[2], enquanto que no universo iOS o mesmo conceito é designado por *View Hierarchy*[3]). A estrutura hierárquica surge natural à abordagem, já que cada objeto interativo irá pertencer a determinada aplicação, que por sua vez corre dentro de uma janela, etc. A profundidade desta hierarquia não costuma ser superior a 5 ou 6 níveis, caso contrário também se torna difícil de gerir e desenhar, para além de **afetar negativamente a *performance* da mesma**.

A figura 2.2 ilustra uma hierarquia de objetos interativos para uma aplicação móvel Android. A decomposição apresentada pode não ser a que foi realmente usada na aplicação, mas será uma aproximação igualmente possível. A hierarquia tem 4 níveis, sendo que na raiz está o ecrã, o maior contentor possível para os diversos objetos. No primeiro nível está a barra de notificações e a atividade em exibição da aplicação Android. Por sua vez, a atividade é composta por uma barra de título, uma grelha e dois botões. A grelha e a barra de titulo são dois contentores, que contêm etiquetas de texto e botões.

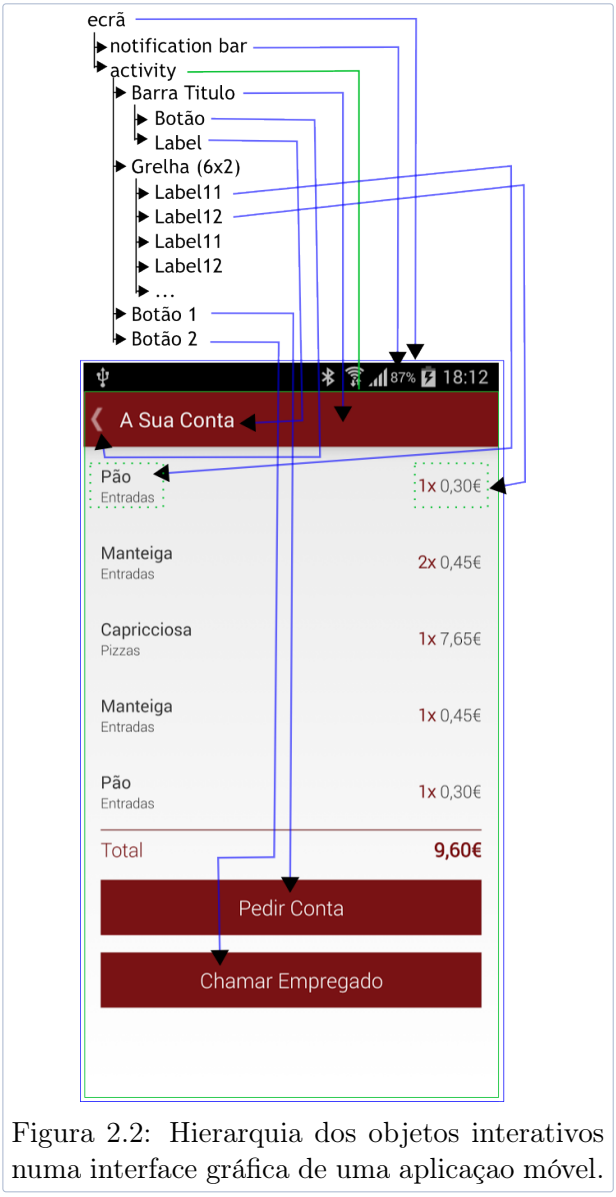


Figura 2.2: Hierarquia dos objetos interativos numa interface gráfica de uma aplicaçao móvel.

Apesar da ilustração ser feita para uma aplicação Android, **a explicação seria semelhante** para a esmagadora maioria das aplicações com interface gráficas, nomeadamente aquelas implementadas para Microsoft Windows, Linux (ambientes gráficos como o Gnome), iOS ou macOS.

Na documentação da especialidade, **os objetos interativos** são sobretudo conhecidos pela designação inglesa ***widgets* ou controlos**. Por serem usados para criar interfaces para humanos, estes objetos têm **uma representação gráfica semelhante ao que conhecemos de outros dispositivos usados no quotidiano**, nomeadamente **botões de pressão ou de on/off**, que mudam de aparência

[2] http://developer.android.com/guide/topics/ui/overview.html
[3] Mais info em https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html

aquando da interação (*e.g.*, ao premir um botão no ecrã, a imagem muda para simular essa ação). **Estes objetos são reutilizáveis e disponibilizados**, tipicamente, **numa biblioteca ou *framework*** preparada para o efeito. No caso do Android, por exemplo, o conjunto de classes que disponibilizam os objetos interativos estão em `android.widgets`[4] e `android.view`[5][6] (entre outras), enquanto que nas versões recentes do iOS é a *framework* UIKit[7] que as providencia.

O **comportamento** das aplicações cuja programação é orientada por eventos **é, portanto, inteiramente definido pelas rotinas de tratamento desses eventos**. Programaticamente, são essas rotinas que definem grande parte do fluxo, lógica e funcionalidades das aplicações. O desenvolvimento da aplicação pode, por isso, **começar pelo planeamento da interface de utilizador e evoluir para a implementação das rotinas de tratamento**. Se revisitar o código Java incluído anteriormente, irá reparar que, no trecho, começa-se por definir os objetos interativos para depois definir o comportamento.

Existem normalmente **dois tipos genéricos** de elementos úteis para o desenho de interfaces de utilizador orientadas por eventos. Os **objetos interativos de entrada ou saída** e **os contentores**. Os **contentores são elementos que podem conter outros contentores ou objetos interativos** como botões ou caixas de texto. Os contentores **permitem organizar o aspeto gráfico da aplicação** (*e.g.*, oferecendo funcionalidades de alinhamento vertical ou horizontal, posicionamento através de deslocamento, etc.).

Os eventos gerados dependem normalmente do contexto do objeto interativo e aqueles que vão ser efetivamente **tratados dependem dos objetivos da aplicação**. Por exemplo, uma aplicação que pretenda imitar um piano pode produzir sons logo que um botão é premido, enquanto que uma que pretenda imitar uma guitarra só produz o som após o botão ser libertado. Alguns dos eventos (e fontes) mais comuns (bem como os seus contextos) usados para construir aplicações móveis podem ser enumerados como se segue:

1. **Botões** de pressão com rato, toque ou caneta, podem gerar eventos de clicados, premidos ou libertados e duplamente clicados;

2. **Teclado** pode gerar eventos de tecla premida ou libertada;

3. **Movimentação do rato ou cursor** pode gerar eventos de entrada e saída de regiões;

[4] http://developer.android.com/reference/android/widget/package-summary.html
[5] http://developer.android.com/reference/android/view/package-summary.html
[6] Na verdade, as classes de `android.widgets` são sub-classes das classes de `android.view`.
[7] Mais em https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/index.html

4. **Os controlos das janelas ou da aplicação** podem gerar eventos de fecho, minimização, redimensionamento, etc.

É comum que os **objetos interativos tenham acesso aos métodos de outros objetos no mesmo nível hierárquico** ou acima. No exemplo com código Java incluído anteriormente, um dos objetos interativos podia alterar as propriedades do outro invocando o método `setEnabled()`. A **comunicação entre objetos interativos** pode, assim, ser feita de **3 formas diferentes**:

1. Através da **manipulação direta de propriedades de outros objetos** (exemplo anterior);
2. **Avisando o elemento imediatamente acima na hierarquia** acerca das alterações a fazer, sendo que este terá acesso a outros elementos do mesmo grau hierárquico;
3. **Gerando outros eventos** (pseudo-eventos, já que não são exatamente gerados por humanos), que são capturados e tratados pelas rotinas dos consumidores registadas pelo objeto interativo destino.

Ainda tomando o código incluído antes como referência, deve ser notado que **o *handler*** (a rotina de tratamento) **recebe o próprio objeto interativo como parâmetro de entrada**. O facto é que quando um evento é gerado, este segue para a pilha de eventos até ser brevemente processado e despachado para o *consumidor* registado por esse objeto, onde poderá dar jeito verificar o estado da fonte do evento.

> O **objeto interativo onde o evento é gerado** é designado **por fonte**.

Para terminar, **interessa compreender bem o fluxo de execução de uma aplicação orientada por eventos**. Para isso, considere o trecho de código seguinte:

```
import android.view.*;
import android.widgets.*;
...
btn = (Button) findViewById(R.id.btn);

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();
  }
});
...
```

O código anterior implementa uma pequena parte de uma aplicação Android em que é **definido um botão e registado um novo *consumidor* para o evento *clique***. Sempre que um utilizador clica no botão (com o rato ou com o teclado):

1. **É capturado** um evento do tipo *onClick* no botão `btn`;
2. O evento do tipo *onClick* **é enviado** para o *consumidor* que lhe corresponde;
3. **É executado** o código que está definido no objeto `OnClickListener`, enviado para dentro do objeto `btn`, no método `onClick`. No caso específico apresentado, seria mostrada uma mensagem numa *widget* `Toast` flutuante com o texto `Clicou no botao!` durante 2 segundos (`Toast.LENGTH_SHORT`).



3 Modelo-Visão-Controlador

Sumário
Introdução à arquitetura para desenvolvimento de aplicações interativas conhecida por *Model-View-Controller*.

Summary
Introduction to the Model-View-Controller software development architecture.

1 Introdução

Como implicitamente sugerido no capítulo anterior, é útil pensar numa aplicação em termos de eventos e de interface de utilizador, tal como é útil poder separar a semântica da aplicação da sua apresentação.

> De um modo simplista e geral, a **arquitetura Modelo-Visão-Controlador** (da designação inglesa *Model-View-Controller* (**MVC**)) é **um modelo de arquitetura para desenvolvimento de *software* que separa a lógica da aplicação, a representação da informação e a interação do utilizador** através da definição de 3 componentes: **o modelo, o controlador e a visão** (discutidos melhor em baixo).

Esta arquitetura de desenvolvimento de *software* foi criada por Trygve Reenskaug **nos anos 70**, enquanto este trabalhava com *Smalltalk* para a empresa Xerox Parc. **O objetivo** de estruturar a aplicação de acordo com o MVC é o de **favorecer a sua escalabilidade e manutenção**, bem como **a portabilidade e a reutilização de código**. A estrutura definida **permite também um maior grau de abstração** por parte de cada um dos elementos da equipa de implementação e desenho. A MVC é **muito popular**

na indústria de desenvolvimento de aplicações Web e móveis, sendo largamente sugerida por gigantes na área. Por exemplo, a Microsoft fornece a *framework* ASP.NET MVC dentro da sua solução .NET[1], enquanto que Apple enfatiza o benefício da sua utilização na implementação de aplicações Cocoa[2].

2 Modelo

O ***Modelo* é o componente central do MVC. Encapsula o estado interno** e consiste na **implementação dos objetos, métodos ou rotinas que capturam o comportamento base da aplicação**. É o modelo que processa os dados ou muda o seu estado. Nas especificações mais conservadoras, **só pode interagir com o componente controlador**, sendo **completamente independente da interface** do utilizador.

Numa implementação de uma aplicação móvel, pode-se pensar no *Modelo* como o conjunto de classes que concretizam os seus objetivos principais. Fazendo uso do que é discutido antes nesta aula, é no *Modelo* que concetualmente devem estar **as implementações dos métodos que, de forma concisa, segura e uniformizada** alteram os dados/informação interna da aplicação. De uma forma geral, é o modelo que contém as classes e métodos para **aceder e manipular bases de dados** ou memória persistente, bem como toda a informação que pode ser exibida ao utilizador através de uma visão. No MVC, **um utilizador nunca interage diretamente com o *Modelo***.

A implementação das classes pertencentes ao *Modelo* da aplicação devem ser **o menos específicas possível em relação à representação dos dados** ou interação com os mesmos, para favorecer a portabilidade do código. A sua **interface de programação deve, contudo, ser clara e consistente** ao longo do período de vida da aplicação (já que isso garante que o controlador pode comunicar corretamente com o *Modelo*).

3 Visão

No MVC, **uma *Visão* consiste numa possível representação dos dados da aplicação**. É no componente *Visão* que se **definem as Interfaces de Utilizador**. Note-se que podem existir **várias representações para o mesmo conjunto de dados, daí a importância em separar este componente da lógica aplicacional**. Por um lado, permite que a equipa de desenvolvimento seja repartida pelas várias áreas, por outro, permite que a **interface do utilizador possa ser desenvolvida em paralelo e melhorada ao longo do tempo**.

A separação entre *Modelo* e *Visão* é cabal em aplicações Web. Por exemplo, é comum

[1] Ver http://msdn.microsoft.com/en-us/library/dd381412%28v=vs.108%29.aspx.
[2] Ver https://developer.apple.com/documentation/uikit/about_app_development_with_uikit.

a implementação destas aplicações utilizando a combinação de tecnologias *HyperText Markup Language* (HTML), *Cascade Style-Sheet* (CSS) e *Hypertext PreProcessor* (PHP) ou JavaScript. Neste caso, a especificação da apresentação da informação é sobretudo definida nos ficheiros HTML e CSS, enquanto que a lógica aplicacional estará em ficheiros PHP. Este facto permite que as aplicações Web possam aparecer de forma diferente para diferentes utilizadores (*e.g.*, o Gmail no *browser*), à custa apenas de um estilo CSS diferente.

No caso das aplicações móveis, essa separação também acontece de forma vincada. Na **plataforma Android**, por exemplo, é permitido que **a interface do utilizador seja definida em ficheiros XML** completamente independentes do código da aplicação escrito em Java. Estes ficheiros estão normalmente na diretoria `app/src/main/res/` (de *resources*) e o seu nome termina tipicamente com a palavra *layout*. O código da aplicação (que concretiza o *Modelo* e o *Controlador*) encontra-se na pasta `app/src/main/java/`.

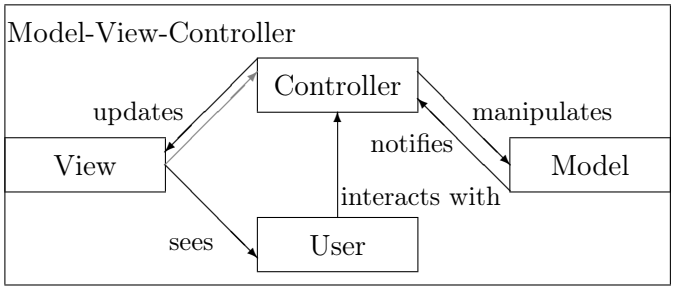
4 Controlador

O ***Controlador* é o intermediário entre o *Modelo*, as *Visões* e o utilizador**. Na verdade, é possível encontrar referências em que se define o *Controlador* como intermediário dos dois outros componentes e não do utilizador, sendo que este apenas interage com o componente *Visão*, onde são gerados os eventos. Contudo, apesar dos eventos poderem ser gerados sobre objetos da *Visão*, **é no *Controlador* que estão as rotinas de captura e reencaminhamento desses eventos**, pelo que também é aceite a figura estática das perspetivas. **São procedimentos do *Controlador* que atualizam os objetos da *Visão* e que despoletam mudanças no estado da aplicação através do envio de instruções para o *Modelo***. Para além disso, é da responsabilidade do *Controlador* a coordenação de tarefas de uma aplicação ou a gestão do ciclo de vida de objetos. As **rotinas de tratamento de eventos** incluem-se, portanto, no componente *Controlador*. Estas rotinas invocam métodos do *Modelo*, que são os que melhor sabem manipular os dados. É boa prática separar o código ou classes que lidam com o modelo (*e.g.*, manipulam bases de dados) do código dos controladores (*e.g.*, podem-se colocar a(s) classe(s) de manipulação de bases de dados num ficheiro, e definir a interação com o utilizador noutros).

5 Comunicação entre Componentes

É possível encontrar diferentes diagramas de comunicação entre os 3 componentes e o utilizador na literatura e *online*. Alguns desses diagramas são diferentes ao ponto de se contradizerem. O diagrama seguinte é, por isso, **uma das abordagens mais consensuais à forma de comunicação entre os 3 componentes** descritos antes e o

utilizador (note que, na figura, as setas → denotam o sentido das comunicações entre componentes). O diagrama tem o componente ***Controlador* ao centro**, evidenciando



o seu papel na comunicação. O **utilizador interage com a aplicação através de uma ou mais *Visões* e do *Controlador***. As *Visões* mostram a informação do *Modelo* ao utilizador, que interage com a interface para manipular essa informação. Essa **interação gera eventos**, capturados pelo *Controlador*, que os processa e envia para as rotinas de tratamento definidas no *Modelo*. O ***Controlador* pode imediatamente atualizar uma ou mais *Visões* aquando da receção de um evento**, ou esperar pelos seus efeitos no *Modelo*. Caso o estado seja alterado, é gerada uma notificação para o *Controlador*, que este usa para despoletar a atualização de uma ou mais *Visões*.

Note que, visto **não existir comunicação direta entre o *Modelo* e as *Visões***, qualquer alteração que precise ser refletida na interface tem de ser comunicada ao *Controlador*. A **notificação contém os dados do *Modelo* necessários à alteração**. A expressão *manipulates*, que dita o fluxo de informação entre o *Controlador* e o *Modelo*, também pode significar a transmissão de dados entre os dois componentes. Como exemplo, considere que um utilizador acabou de inserir uma expressão a ser pesquisada na base de dados, pressionando *Enter* de seguida. O *Controlador* captura o evento de tecla premida e o texto inserido, enviando-o para o *Modelo*. A rotina de processamento (definida no modelo) acede à base de dados, faz a pesquisa, e retorna o novo estado ao *Controlador*, que atualiza a *Visão* respetiva, chegando o resultado aos olhos do utilizador.

O exemplo incluído no último parágrafo pode levar à discussão da **possível existência de comunicação na direção de *Visões* para o *Controlador***, já que o evento que inicialmente despoletava a rotina de pesquisa na base de dados era o clique na tecla *Enter*, e não a inserção do texto. Neste caso, o *Modelo* teria de notificar o *Controlador* que havia informação em falta, e este teria de ir buscar essa informação ao objeto respetivo na *Visão* (daí a seta em cinza claro na figura).



4 A Plataforma Android

Sumário

Discussão da pilha de *software* que define a plataforma Android, bem como das 4 camadas que a compõem.

Summary

Discussion of the software stack that defines the Android platform, as well as of the 4 layers it contains.

1 Introdução

Ultimamente tem-se assistido, contudo, à tentativa de **estender a utilização do Android a outros dispositivos**, nomeadamente ***smart TVs***. A popularidade da plataforma, o facto de ter o código aberto e ser **relativamente simples desenvolver aplicações para a mesma**, tem ditado **uma evolução fora do comum**, tornando-a hoje **num sistema bastante completo e complexo, no sentido também de albergar já um vasto conjunto de componentes**.

2 Pilha de *Software*

De uma maneira geral, pode dizer-se que **a plataforma Android é composta por**:

1. **uma pilha de *software*, com várias camadas**, desenhada para permitir a construção e execução de aplicações móveis;
2. **um kit de desenvolvimento de *software*** (da designação *Software Development Kit* (SDK)); e

3. uma extensa **documentação**.

Esta pilha de *software* foi **desenhada sobretudo, mas não exclusivamente, para dispositivos móveis**, nomeadamente *smartphones* e *tablets*. É composta por **4 camadas** que se estendem desde **o nível do núcleo do SO até às aplicações que o utilizador pode manipular** (*e.g.*, *browser*, atendedor de chamadas ou aplicação de *e-mail*). A pilha de *software*, bem como alguns dos seus componentes e organização costuma ser representada como se mostra na figura 4.1. Cada uma das camadas vai ser alvo de uma breve discussão nas secções seguintes.

3 Camada do Núcleo Linux

A **camada mais baixa** da pilha de *software* compreende **o próprio núcleo do SO**. É esta camada que **fornece os serviços base** de que dependem as restantes funcionalidades de qualquer dispositivo com Android. Por usar **o *kernel* Linux**, disponibiliza muitos dos serviços que nele estão implementados, nomeadamente **uma arquitetura de permissões** (para restringir acesso a recursos), **mecanismos padrão de gestão de memória e de processos**, **suporte a comunicações entre-processos**, **operações de baixo nível de leitura e escrita em ficheiros**, e **comunicações em rede**. É nesta camada que são colocados **os *drivers*** que podem ser adicionados ou incluídos para suporte de **dispositivos de *hardware* adicionais** como câmaras fotográficas, antenas de rádio ou sensores.

Note que **o núcleo Linux usado no Android não é exatamente igual aos núcleos tipicamente usados para *desktops***. O facto é que os dispositivos móveis **têm funcionalidades e necessidades diferentes** dessas máquinas, pelo que tiveram de ser feitas **adaptações**. Por exemplo, o núcleo utilizado no Android tem **os seus próprios gestores de energia e de memória**, mais porque os dispositivos móveis são tipicamente alimentados por baterias e podem ser mais limitados em termos de recursos computacionais. Por exemplo, o ***low memory killer* constitui uma das funcionalidades do gestor de memória**. Outro bom exemplo é o mecanismo de **comunicação entre-processos** do Android (*binder*), que permite que os processos partilhem memória e informação de uma **forma simples e eficiente**.

4 Bibliotecas Nativas

A **segunda camada** (a contar de baixo) **contém as bibliotecas nativas** disponibilizadas pela plataforma, nomeadamente a `bionic libc`, `sqlite` e SSL. Estas biliotecas nativas são normalmente **implementadas em C ou C++**, e estão encarregues de **atividades críticas relacionadas com o desempenho do dispositivo**, como por exemplo

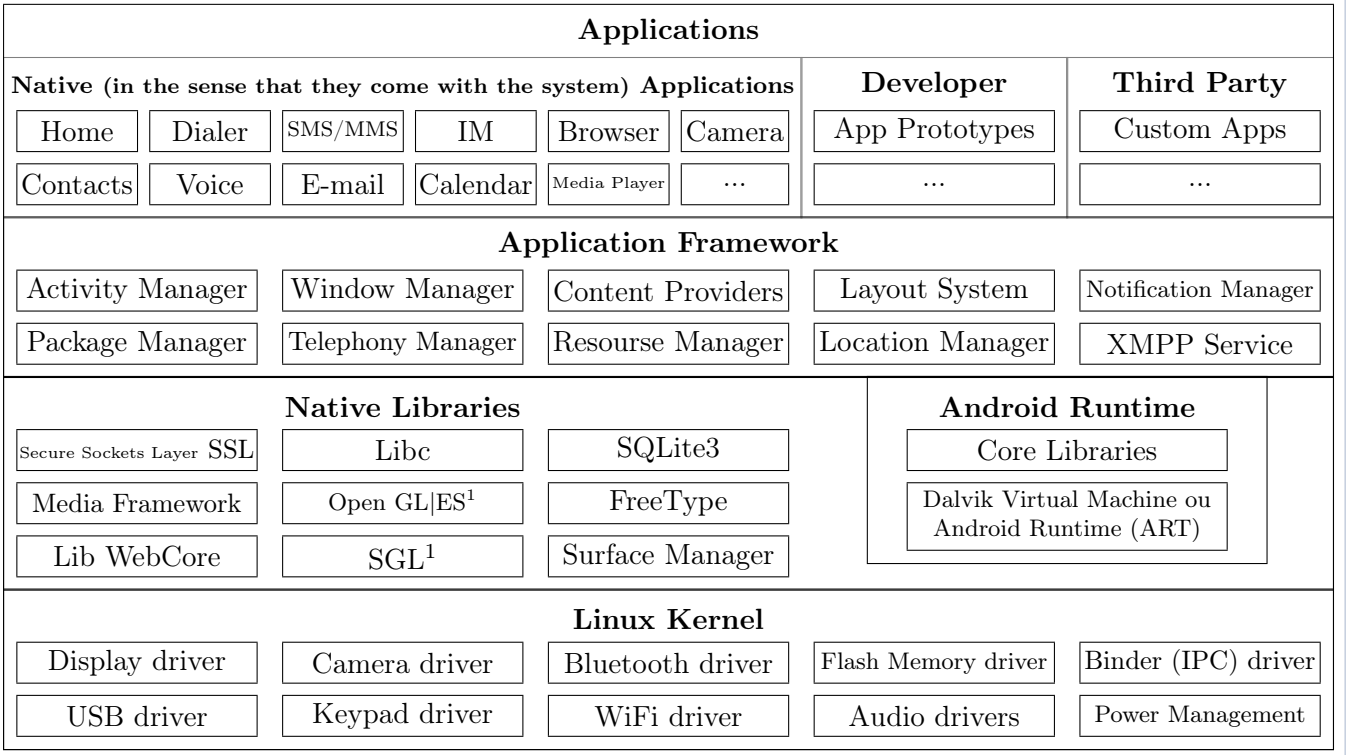


Figura 4.1: Pilha de *software* da plataforma Android.

refrescar o ecrã (feito pelo *Surface Manager*) ou **renderizar páginas web** (da responsabilidade do *LibWebcore*). A biblioteca `sqlite` **permite a gestão de bases de dados** em aplicações Android. **A** `bionic libc` concretiza outra diferença para sistemas Linux padrão, já que esta também **foi adaptada para Android** tendo em conta as particularidades do núcleo e do dispositivo. A biblioteca *Media Framework* disponibiliza o conjunto de funções base para lidar com áudio e vídeo, enquanto que a *Open Graphics Library* (OpenGL) dá suporte a aplicações gráficas de alto desempenho.

É nesta camada que também se inclui **o ambiente de execução virtual** de aplicações Android, composto por **dois componentes principais**: as **bibliotecas Java base** e a **Máquina Virtual (da expressão inglesa *Virtual Machine* (VM)) Dalvik** ou o **ambiente de execução *Android Runtime***. Em versões anteriores à 5.0, a máquina virtual padrão deste sistema era a Dalvik, enquanto que a partir dessa versão as aplicações passaram a executar num ambiente designado por *Android RunTime* e simplesmente conhecido por **ART**.

As aplicações para Android são normalmente implementadas em Java e, para que isso seja possível, é **disponibilizado um conjunto de classes que podem ser prontamente**

[1]ES abrevia *Embedded Systems*, GL abrevia *Graphics Library* e SGL abrevia *Skia Graphics Library*.

utilizadas. Por exemplo, **classes das bibliotecas** `JAVA.*` **ou** `JAVAX.*` contêm *software* para **manipulação de ficheiros ou definições de estruturas**, enquanto que **a biblioteca** `ANDROID.*` contém classes relacionadas com o **ciclo de vida de aplicações** Android, nomeadamente **para criação da interface de utilizador** ou *logs*, etc.

Note que as aplicações móveis para esta plataforma **são escritas em Java ou compiladas para *bytecode* Java**. Podem até ser compiladas usando ferramentas Java comuns (*e.g.*, o `javac`), **mas não correm nas típicas VMs Java**, porque a a Google **desenvolveu a sua própria VM** e ambiente de execução. O formato do código desta máquina é diferente do que corre nas VMs normais. O **processo de implementação e compilação** de uma aplicação Android é normalmente decomposto nos seguintes passos:

1. A aplicação é implementada em Java (ou Kotlin);
2. O código é **compilado para *bytecode* usando um compilador padrão**;
3. O *bytecode*, ainda que pertencente a várias classes, é **traduzido para o formato** `dex` **e compilado para um único ficheiro** (`classes.dex`) por uma ferramenta chamada `dx`;
4. O **código, dados e ficheiros** contendo os mais variados **recursos** relativos à aplicação (*e.g.*, ficheiro com imagens, ícones, *layouts*) **são, por fim, empacotados para um arquivo** com extensão `.apk` por uma ferramenta designada por `aapt`.

A aplicação resultante é **carregada e executada numa VM em ambiente totalmente isolado** (*sandboxed*) de outras aplicações. A execução de cada aplicação corresponde à criação de **uma VM com o seu próprio *User ID***. Este modelo de operação garante que, **durante a execução e em condições normais, uma aplicação não deve conseguir aceder aos dados e ficheiros de outra diretamente**.

Note que, apesar de parecer disruptiva, a escolha da Google deve-se novamente ao facto das aplicações móveis executarem num ambiente diferente dos computadores de secretária. A Dalvik ou o ART **foram desenhados para ambientes com potenciais limitações em termos de recursos**, nomeadamente em termos de memória, bateria e processamento.

5 *Framework* Aplicacional

A *framework* aplicacional **contém *software* ou recursos que as aplicações Android podem necessitar e reutilizar**, como por exemplo ficheiros com imagens de botões ou elementos gráficos (componente *View System*). **Alguns dos componentes** incluídos nesta camada podem ser resumidamente descritos da seguinte forma:

- O **gestor de pacotes** (*Package Manager*) mantém **o registo de todas as aplicações instaladas no sistema**. É o funcionamento deste componente que permite que algumas aplicações encontrem outras e **registem serviços que queiram disponibilizar**;
- O **gestor de janelas** (*Windows Manager*) que **lida com as várias janelas e partes mostradas no ecrã** aquando da utilização de uma aplicação ou, por exemplo, **com as sub-janelas** que são despoletadas pela abertura de um menu;
- O **gestor de recursos** (*Resource Manager*) **manipula os recursos de uma aplicação que não foram compilados**, nomeadamente *strings*, os ficheiros de *layout* ou imagens;
- O **gestor de atividades** (*Activity Manager*), que **coordena e suporta a navegação entre atividades** (um ecrã de interação), bem como o seu **ciclo de vida**;
- Os **provedores de conteúdos** (*Content Providers*) são, na sua essência, **bases de dados que as aplicações usam para guardar ou partilhar informação do sistema** (*e.g.*, a base de dados dos contactos num *smartphone*). O componente com o mesmo nome tem a responsabilidade de os gerir. Note que, normalmente, não é possível aceder aos dados de uma aplicação noutra de uma forma direta. **Os *content providers* constituem uma das formas de agilizar** essa troca;
- O **gestor de localização** (*Location manager*), que disponibiliza **informação acerca do movimento e localização** *Global Positioning System* (GPS);
- O **gestor de notificações** (*Notification Manager*), que **é o componente que controla o conteúdo da barra de notificações**. Esta barra é um dos componentes mais importantes do sistema, porque **fornece uma área que está quase sempre visível para o utilizador** e fornece **um meio para informar** utilizadores acerca de **eventos que possam ocorrer fora do âmbito da atividade** que está desenvolver.

6 Camada de Aplicação

A camada no topo da pilha é onde se **incluem as aplicações que vêm com o sistema** (de fábrica) e as que são **instaladas subsequentemente**, e que podem incluir **aplicações em desenvolvimento** (*debug*) ou **desenvolvidas por terceiros**. O **utilizador final interage com esta camada diretamente**. Em Android, nenhuma das aplicações que vêm com o sistema são de uso obrigatório. É sempre possível construir uma aplicação para determinado fim e usá-la em detrimento da que vem instalada com o sistema ou outras.



5 Componentes das Aplicações Android

Sumário
Descrição dos blocos fundamentais que constituem as aplicações Android e análise do seu processo de preparação e compilação.

Summary
Description of the fundamental components that may be used to compose an Android application and analysis of its building process.

1 Introdução

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 é também **implementado em Java**. Os **4 blocos fundamentais** sobre os quais as aplicações Android são construidas 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 é discutido com um pouco mais de detalhe nas secções seguintes e também adiante neste curso.

2 Atividade

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 **definição**[1], **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 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 construida a interface (`setContentView()`).

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

[1] *E.g.*, ver http://developer.android.com/reference/android/app/Activity.html.

```
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);
    }
}
```

3 Serviço

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 aplicações[2].

4 Fornecedor de Conteúdos

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` e quando 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 efeito[3].

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

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

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 necessária para fornecer ou atualizar os dados de uma determinada aplicação de 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.

5 Recetor de Difusão

O componente **recetor de difusão escuta e processa eventos** na plataforma Android. Na **arquitetura de *software*** para troca de mensagens, conhecida como ***publish-subscribe***, as instâncias destes componentes **tomam o papel de subscritores**. O seu **objetivo é o de permitir que determinada aplicação se registe no sistema como capaz de lidar com determinado evento e, quando esse evento acontece, esta seja chamada pelo sistema operativo para o processar**. **Outras aplicações criam estes eventos** através da definição de 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 de difusão são **componentes importantes em termos de *performance* e funcionalidade** de um sistema Android. São eles que permitem que uma determinada aplicação registe no sistema a sua vontade em receber determinados eventos. **Caso não existisse** um componente com esta forma de funcionamento, cada vez que 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 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.

[4] Ver `http://developer.android.com/reference/android/provider/package-summary.html`.

6 Processo de Preparação de uma Aplicação Android

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 5.1, adaptada da documentação oficial disponível no *Uniform Resource Locator* (URL) `http://developer.android.com/tools/building/index.html`, ilustra o processo de compilação da aplicação de uma forma simplificada. **Durante o processo de preparação da aplicação, alguns ficheiros do projeto são compilados sendo depois empacotados juntamente com outros recursos num arquivo com extensão** `.apk`[5]. O **arquivo**, conforme esquematizado, **contém**: **os ficheiros** `.dex` (que são os ficheiros `.class` compilados para o *bytecode* específico da máquina virtual onde vão correr), **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).

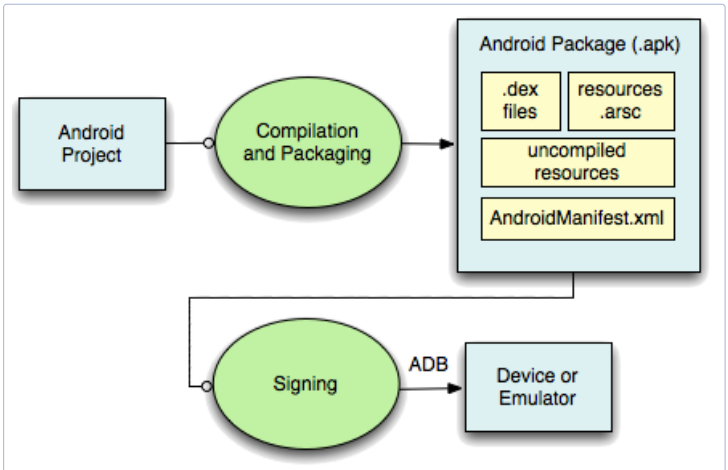


Figura 5.1: Processo de compilação simplificado de uma aplicação Android.

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 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

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

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 certificadora[6].

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, a definição das políticas de privacidade, etc. Em princípio, e caso a aplicação não contenha *malware*, será publicada ao fim de pouco tempo[8].

O processo de preparação da aplicação Android encontra-se ilustrado **com bastante detalhe** na figura 5.2. 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.

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

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

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

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

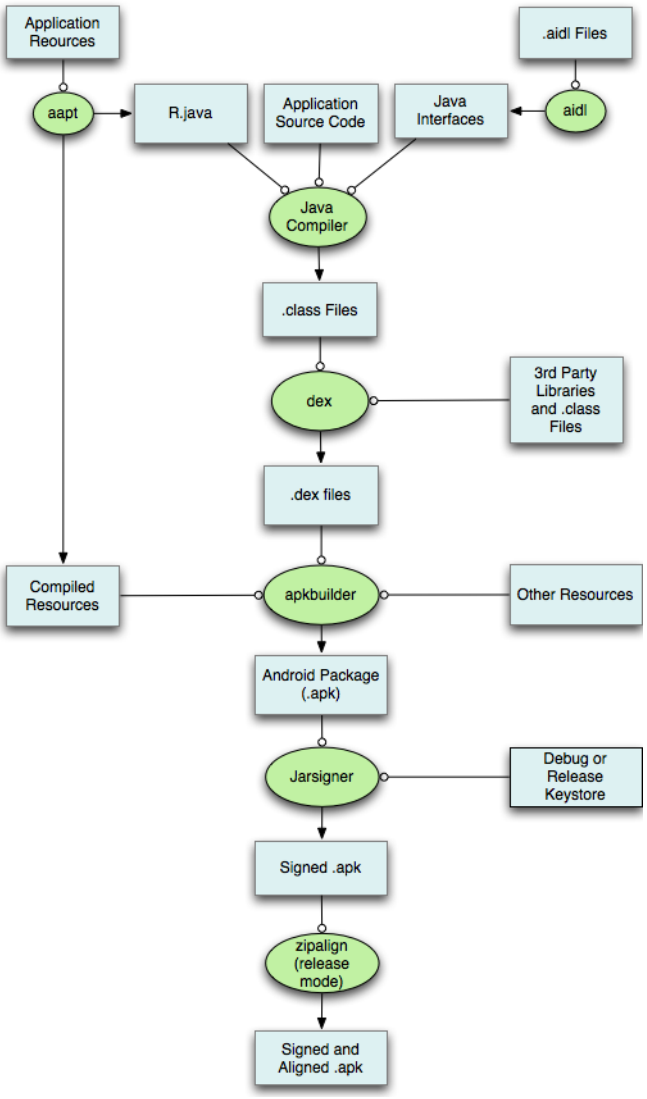


Figura 5.2: Processo de preparação de uma aplicação Android.

7 R.java e 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`

A titulo 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 IDentificador (**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-à-string`**.

8 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 a 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 intentos 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"/>`.



6 Depuração de Aplicações Android

Sumário

Apresentação e discussão de algumas ferramentas para depuração de aplicações Android.

Summary

Presentation and discussion of some of the tools available for debugging Android applications.

1 Introdução

A **depuração** de aplicações pode ser feita recorrendo tipicamente a uma **panóplia de ferramentas**. Uma dessas ferramentas consiste na **definição de pontos de rutura** no código e na **análise do estado da aplicação após paragem nesse ponto**, depois de executada. O ambiente de desenvolvimento integrado Android Studio tem suporte a este tipo de depuração para aplicações Android, semelhante ao que já deve conhecer do desenvolvimento de aplicações em Java. Contudo, nesta secção enumeram-se **outros recursos que podem ser usados para depuração**, mais ligados ao facto das aplicações serem testadas em dispositivos virtuais ou reais, mas que se podem **monitorizar**. Antes de enumerar esses recursos, apresentam-se algumas vantagens e desvantagens da utilização de dispositivos virtuais.

2 Dispositivos Virtuais Android

Conforme já mencionado antes, o desenvolvimento de aplicações Android é dominado pelo uso de emuladores para virtualização de dispositivos móveis. O facto é que, mesmo que se

tenha acesso a um ou mais dispositivos reais, **assegurar que uma aplicação funciona para todos ou parte dos dispositivos** disponíveis no mercado irá requerer, quase seguramente, **o uso de virtualização**. Apesar de lentos no arranque e por ventura na execução, por requererem que a máquina virtual processe, simultaneamente, a emulação do dispositivo móvel e o funcionamento do sistema operativo, **os dispositivos virtuais Android têm a grande vantagem de poderem ser facilmente monitorizados**. As **vantagens** de usar um emulador Android são:

- Em termos **financeiros** – não é necessário comprar um dispositivo móvel real;
- Relativamente à **versatilidade** – o *hardware* pode ser virtual e facilmente configurado (*e.g.*, o tamanho do cartão de memória SD);
- Em termos de **confinamento** – as alterações que forem feitas, *e.g.*, ao sistema, pela aplicação móvel desenvolvida, são confinadas ao dispositivo.

As **desvantagens** de usar um emulador são:

- Em termos de **desempenho** – a emulação é normalmente mais lenta que o uso de um dispositivo real;
- Quanto às **funcionalidades** – algumas funcionalidades não estão disponíveis em emuladores ou podem ser emuladas de forma não satisfatória em alguns casos (*e.g.*, não há *Bluetooth*);
- No que se refere ao **realismo** – mesmo que um emulador esteja próximo de um dispositivo real, no que diz respeito a imitar o seu funcionamento, pode sempre falhar algum detalhe que só é notado após se experimentar em ambiente real (*e.g.*, sensores).

A **plataforma Android disponibiliza** atualmente um **vasto conjunto de ferramentas e serviços de depuração**. Em baixo referem-se apenas alguns desses recursos.

3 Logcat

Um dos recursos mais utilizados na depuração de aplicações é a análise de *logs*. Este não deve, contudo, concretizar o principal recurso utilizado, embora aconteça frequentemente. Aquando da depuração de programas escritos em Java, é comum a utilização do procedimento `System.out.println()` para obter valores de variáveis ou estimar o pontos de falha durante a execução da aplicação na consola.

A **plataforma Android disponibiliza o seu próprio sistema de *logging***, bem como funcionalidades para colecionar e visualizar informação de depuração. Os *logs* de várias

aplicações ou porções do sistema **são colecionadas em pilhas circulares**, que podem depois ser **analisadas ou filtradas através do comando** `logcat`, também fornecido com o *Android Debug Bridge* (ADB). Repare-se que é o facto do sistema de *logging* ser fornecido com a plataforma que permite que **a informação seja produzida e também analisada de uma maneira uniforme** para todas as aplicações. O facto do comando `logcat` estar integrado no `adb` permite que **as mensagens do *log* possam ser lidas em tempo real**, tanto num dispositivo virtual como real (desde que ligado via *Universal Serial Bus* (USB) e com o modo de depuração ativado).

O **comando** que permite aceder ao `logcat` é o seguinte: `$ adb logcat`
Também é possível **executar diretamente o comando** `logcat` **na *shell*** oferecida pelo `adb`, nomeadamente através do encadeamento das seguintes instruções:
`$ adb shell`
`$ logcat`
A informação devolvida pode ser filtrada ou tratada através de opções do comando.

Uma aplicação pode escrever entradas no `logcat` **através da classe** `Log` (`import android.util.log;`). Alguns dos métodos que podem ser usados para esse efeito enunciam-se a seguir, realçando-se, de imediato, a sua **interface simplificada e consistente**:

```
v(String, String) // (verbose)
d(String, String) // (debug)
i(String, String) // (information)
w(String, String) // (warning)
e(String, String) // (error)
```

Cada um dos métodos exibidos antes **aceita duas *strings*** e estão **apresentados por ordem de verbosidade, da maior para a menor**. O primeiro parâmetro (*string*) deve ser uma cadeia de caracteres que **identifica a aplicação ou parte do sistema** que está a escrever no *log* (normalmente designada por *tag*), enquanto que **o segundo é a mensagem em si**. É recomendado que a *tag* seja declarada estaticamente no código, para que seja usada de forma uniforme durante toda a aplicação, como se mostra no excerto de código seguinte:

```
private static final String TAG = "pt.ubi.di.pdm.example";
...
 @Override
    public void onCreate(Bundle savedInstanceState)
    {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      Log.i(TAG,"All fine up to this point!")
      ...
    }
```

Note que **o grau de verbosidade é importante no momento de definir uma entrada** para o `logcat`. Por exemplo, uma entrada do **tipo *verbose*** (`Log.v(...)`)

só deve ser usada na fase de depuração (desenvolvimento) da aplicação, e deve ser **manualmente** removida antes da compilação da versão final da aplicação. As **entradas do tipo *debug*** (`Log.d(...)`) **são compiladas para a aplicação final, mas retiradas durante execução da mesma**, enquanto que **as entradas do tipo *error*, *warning* e *info* são sempre mantidas** (*i.e.*, mesmo aplicações assinadas e distribuídas via *Google Play* podem emitir entradas destas).

4 Inspetor de *Layout*

O SDK Android disponibiliza também **uma ferramenta para depuração e otimização da interface de utilizador**. Esta ferramenta, designada por **Inspetor de *Layout*** (da designação inglesa *Layout Inspector*) e da qual se inclui uma captura de ecrã em baixo (figura 6.1), mostra, de uma maneira **muito intuitiva, a forma como os vários elementos da interface de utilizador estão interligados**. De acordo com o que foi dito anteriormente, a perspetiva é obtida através da representação de **uma árvore deitada e com orientação da esquerda para a direita**, sendo que a ramificação é indicativa de como os **elementos estão contidos dentro de outros**. A ferramenta é **completa ao ponto de permitir que se visualize o conteúdo exibido no dispositivo móvel acedendo** interativamente **a cada um dos nós** que compõem a interface gráfica (*e.g.*, se clicar no nó relativo à barra de título de uma atividade, é mostrada essa barra de título com o aspeto e texto definido).

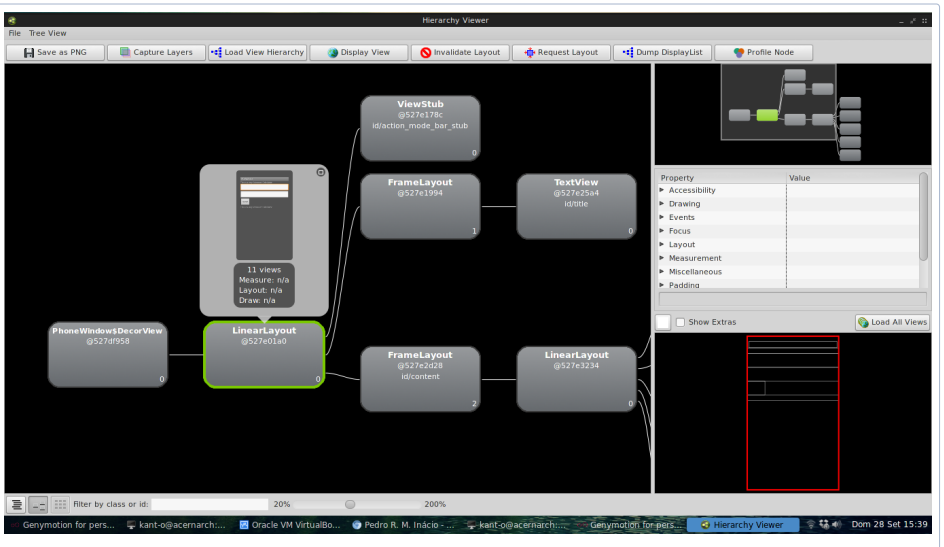


Figura 6.1: Captura de ecrã da ferramenta de depuração da hierarquia de elementos da interface de utilizador em execução.

A ferramenta pode **ser acedida** via menus em **ambientes de desenvolvimento integrado** (*e.g.*, no Android Studio poderá ser acedida via Tools → Layout Inspector). Versões recentes permitem inclusive a navegação pelos elementos da interface **enquanto**

a aplicação está a ser executada num emulador.

5 Servidor do Monitor de Depuração *Dalvik*

O Servidor do Monitor de Depuração *Dalvik*, conhecido pelo acrónimo da sua designação inglesa ***Dalvik Debug Monitor Server*** **(DDMS)** é uma aplicação que congrega, numa só interface gráfica e de uma forma mais uniforme, diversas ferramentas e serviços de monitorização de um dispositivo Android, algumas delas já antes referidas. **Disponibiliza serviços de reencaminhamento de portas, captura de ecrã, informação acerca de *threads*, processos e memória** nos dispositivos ligados ou emulados, **visualização de *logs*** *logcat*, **simulação de chamadas e mensagens** *Short Message Service* (SMSs) e modificação de informação de localização. Para além de um visualizador para o *logcat* e do visualizador de hierarquia, o DDMS **integra** ainda **uma ferramenta para análise dos registos de execução de métodos**[1], perspetivas sobre a atividade de rede, e um explorador de ficheiros com acesso a todos os ficheiros e diretorias do sistema. O DDMS está atualmente integrado no *Android Device Monitor*.

6 Monitor de Dispositivos Android

Versões mais recentes do SDK sugerem utilizar o ***Android Device Monitor*** **(ADM) para monitorização de dispositivos Android** ao invés do DDMS. É possível despoletar esta ferramenta no modo solitário emitindo `$ ./monitor` na diretoria `tools` do SDK (e ilustra-se o seu funcionamento na captura de ecrã incluída na figura 6.2), mas é óbvio que muitas das funcionlidades de depuração e *profiling* já se encontram integradas no próprio Android Studio, conforme se mostra na figura 6.3. O ADM integra o *Layout Inspetor*, o DDMS e muitas outras ferramentas úteis para depuração de aplicações.

7 Funcionalidades Avançadas dos Dispositivos Virtuais Android

Apesar de lentos no arranque, os Dispositivos Virtuais Android são **extremamente ricos em termos de funcionalidades**. Por exemplo, é possível **controlar alguns aspetos simulados** no dispositivo através de janelas de configuração (ver figura 6.4) ou até de **uma ligação *telnet***. O destino da ligação deve ser **a porta atribuída ao dispositivo no** `localhost`, *e.g.*: `$ telnet localhost 5555`

[1] Permite capturar a execução de determinados procedimentos em registos, que podem ser posteriormente analisados.

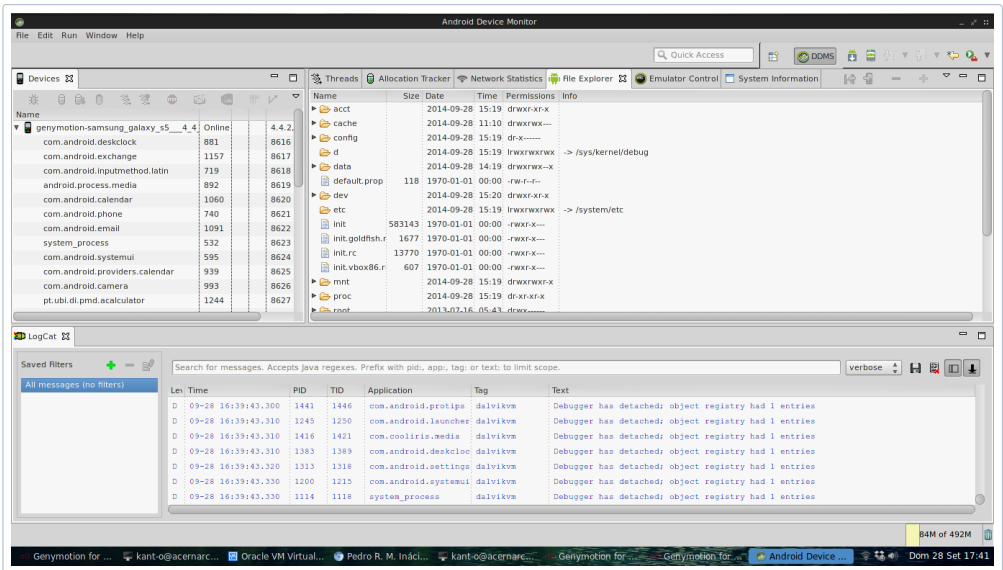


Figura 6.2: Captura de ecrã da ferramenta de *profiling* Android Device Monitor.

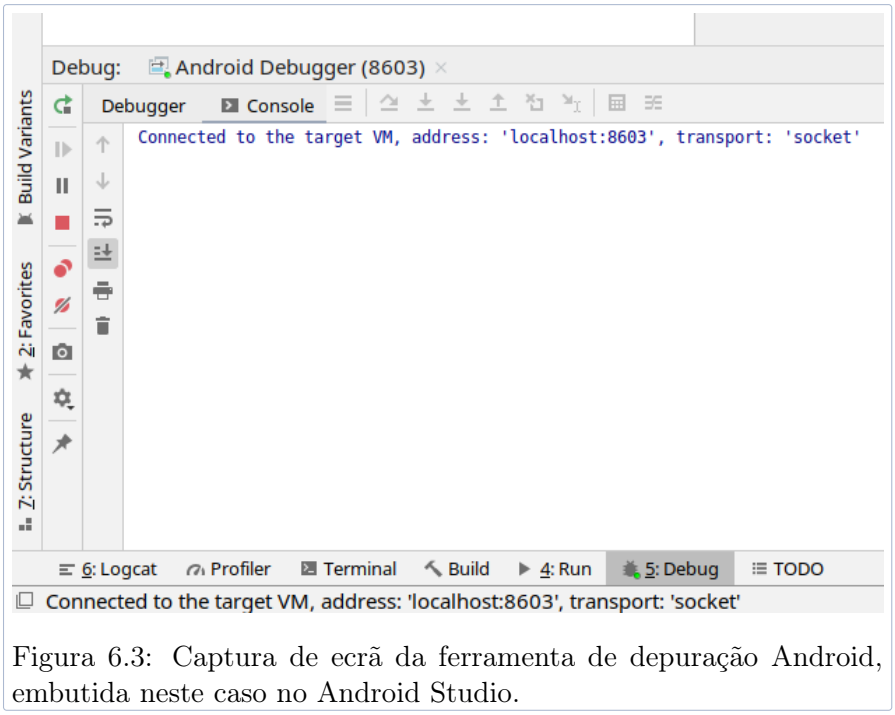


Figura 6.3: Captura de ecrã da ferramenta de depuração Android, embutida neste caso no Android Studio.

Note que é normalmente possível obter a porta onde está o dispositivo à escuta através do comando `$ adb list devices`.

Após obter ligação, podem-se controlar diversos aspetos do dispositivo emulado com **simples instruções** no terminal, *e.g.*:

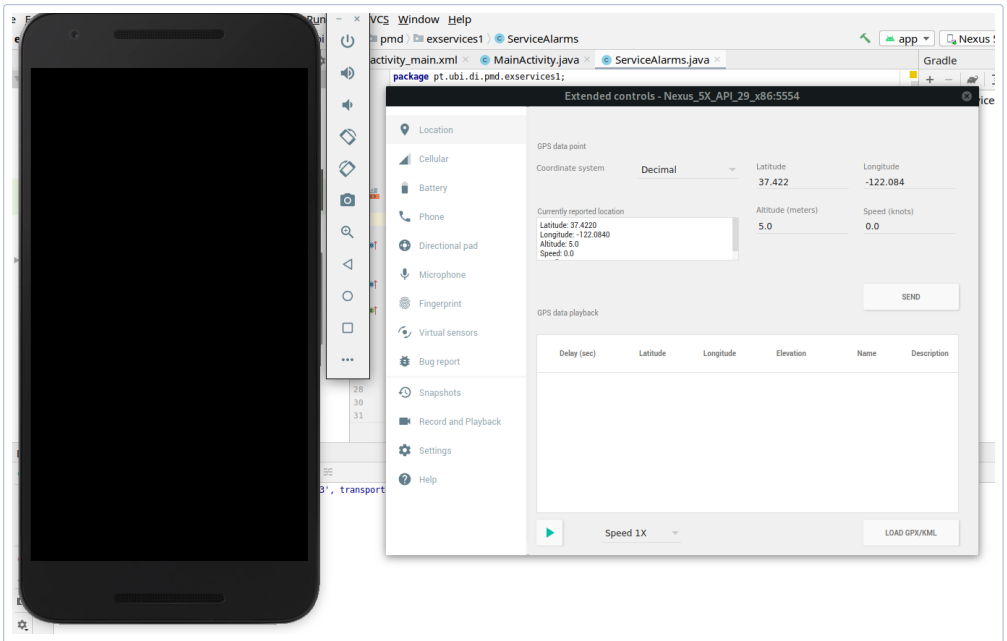


Figura 6.4: Exemplo do aspeto da janela de controlos de um dispositivo emulado, ilustrando a panóplia de configurações e ajustes possíveis.

- Mudar a carga da bateria no emulador para 50%
  `$ power capacity 50` ;
- Simular uma rede *EDGE*[2]
  `$ network speed edge`;
- Enviar uma mensagem SMS para o dispositivo virtual
  `$ sms send 555555555 "ola mundo"`;
- Iniciar uma chamada com o dispositivo virtual
  `$ gsm call 555555555`;
- Ajustar as coordenadas GPS para as da fase VI da Universidade da Beira Interior (UBI)
  `$ geo fix 40.27 -7.50`.

Note que também **é normalmente possível fazer chamadas ou enviar SMSs entre dois dispositivos virtuais** Android a correr ao mesmo tempo no mesmo SO. **O número a marcar** num dos dispositivos **corresponde à porta** *Transmission Control Protocol* (TCP) **onde o outro está à escuta**.

[2]Enhanced Data rates for GSM Evolution. GSM é o acrónimo de *Global System for Mobile Communications*.



7 A Componente Atividade

Sumário

Foco na componente `Activity`, descrevendo com detalhe o ciclo de vida e os mecanismos de gestão das atividades no sistema operativo.

Summary

Thorough discussion of the `Activity` component, namely of their life cycle and of the mechanisms available in the operating system for managing them.

1 Introdução

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 Backstak***[1]). O sistema **gere automaticamente o ciclo de vida** das atividades (ver em baixo), e garante que estas são **devidamente inicializadas, suspendidas ou retomadas, bem como destruídas em caso de falta de memória**.

2 Pilha de Retrocesso de Tarefas

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 7.1 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 suspendida 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.

[1] Ver http://developer.android.com/guide/components/tasks-and-back-stack.html.

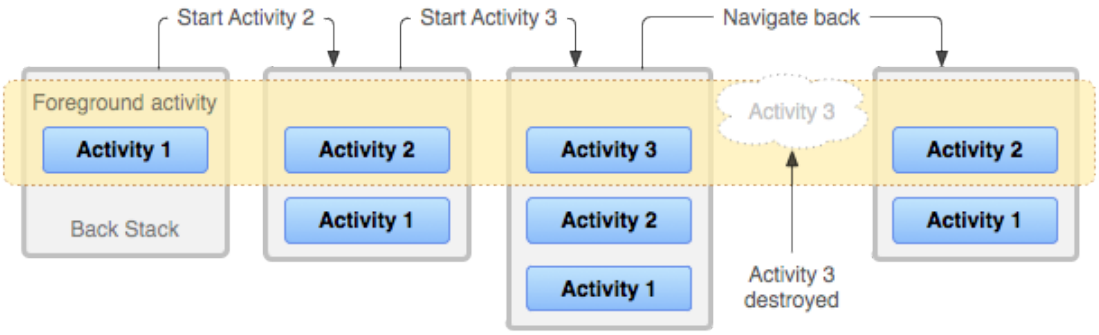


Figura 7.1: Representação da pilha de retrocesso de atividades mantida pelo Android, bem como da sua evolução ao longo da execução de uma tarefa (obtida de referência em nota de rodapé).

3 Ciclo de Vida de uma Atividade

Uma atividade pode estar em **um de 3 estados fundamentais**:

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 7.2 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**

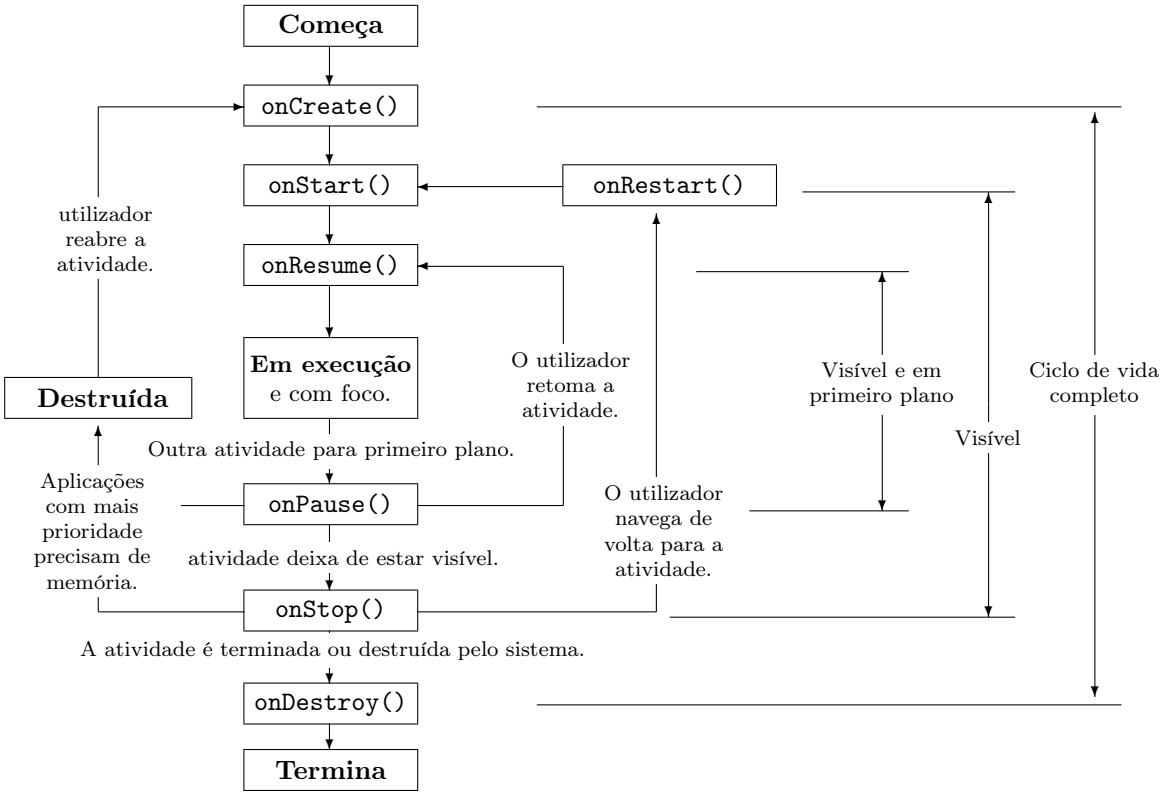


Figura 7.2: Ciclo de vida de uma atividade.

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**.

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.

4 Método `onCreate()`

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`, fornecido 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(oTEdit1.getText().toString())).
                doubleValue();
            double d2 = (new Double(oTEdit2.getText().toString())).
                doubleValue();
            double sum = d1 + d2;
            oTView1.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.

Note-se contudo o seguinte detalhe. Apesar de se estar a fazer o *override* do método `onCreate()`, o facto de ser obrigatório incluir o análogo da super classe indica claramente que, na realidade, o que se está a fazer é a **estender** a funcionalidade já implementada.

5 Método `onStart()`

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()`).

6 Método onResume()

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()`.

7 Método onPause()

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**.

8 Método onStop()

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()`.

9 Método `onRestart()`

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()`.

10 Método `onDestroy()`

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()`.



8 Intentos

Sumário

Discussão de um dos conceitos fundamentais da filosofia de implementação de aplicações Android: intentos. Análise de intentos implícitos e explícitos, bem como da forma como estes podem ser usados para transportar informação e ligar componentes de uma aplicação ou mais.

Summary

Discussion of one of the fundamental concepts of the implementation philosophy for Android applications: the intent objects. Analysis of implicit and explicit intents, as well as of the means that can be used to transfer data between components of one or more applications.

1 Introdução

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

> Na gíria específica do universo Android, um intento (da designação inglesa ***intent***) **é um objeto que encapsula, de uma forma abstrata, a intenção de de-**

terminada componente fazer uma ação. **Dependendo da especificidade** do intento, esta ação **pode ser executada por uma componente bem definida** da mesma aplicação ou de outra, capturada pelo SO e **entregue a uma de várias componentes** que podem fazer essa ação, **ou ser descartada**. Um intento é descartado **caso a componente alvo não exista ou caso não haja componentes capazes de lidar com a mesma**.
Na documentação oficial[a], é descrito como **um objeto mensagem** que pode ser usado para pedir uma ação a outra componente.

[a]Ver, *e.g.*, http://developer.android.com/guide/components/intents-filters.html.

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

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

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

- Os **serviços** (componentes que executam ações em segundo plano) **podem ser executados** (ou executar ações para determinada aplicação) **recorrendo ao método** `startService(Intent)`, que também **aceita o intento** a definir o serviço e eventualmente **alguns dados que este deve processar**. É ainda **possível obter uma ligação** (da classe `ServiceConnection`) **duradoura a um serviço** através de `bindService(Intent,ServiceConnection,int)`, que **permite que um serviço esteja associado à execução** de determinada atividade ou outro serviço,

sendo **terminado quando estes terminarem também** (é útil quando determinado serviço só deve funcionar enquanto a aplicação ou atividade estiver a ser executada). Estes intentos serão discutidos noutro capítulo;

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

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

2 Instanciação de Intentos

A figura 8.1 demonstra como é que, de um ponto de vista abstrato, o mecanismo associado aos intentos deve funcionar. Basicamente, **quando um componente de uma aplicação quer começar outro** componente, **instancia um intento e envia-o para o sistema**, que fica **responsável por identificar, verificar as permissões e**, em caso de encontrar o seu destino e ser permitido, **de o enviar para execução**.

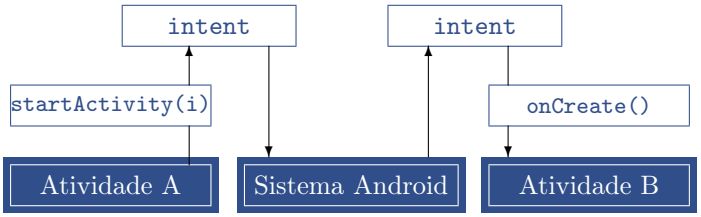


Figura 8.1: O mecanismo inerente ao uso de `intents` para despoletar atividades em Android.

Portanto, para que o mecanismo funcione, o objeto da classe `Intent` **tem de necessariamente transportar alguma informação** que permita ao sistema efetuar essa tarefa, nomeadamente a **ação a efetuar, e o nome ou categoria do componente que deve receber o intento**. Adicionalmente, pode ainda **conter dados** que o componente destino usa para efetuar a ação pretendida. Assim, um intento **pode conter**:

1. O **nome do componente destino** – esta informação é opcional, mas é a que no fundo **distingue intentos explícitos** (caso contenha esta informação) **de implícitos** (caso contrário). O facto é que a plataforma permite que uma componente

determine apenas a ação a ser executada no intento, deixando ao **critério do SO a escolha da componente que a vai fazer**. Esta **escolha** pode adicionalmente **ser baseada na categoria especificada ou nos dados incluídos**. Esta informação é corretamente definida recorrendo a objetos da classe `ComponentName` (ver exemplo em baixo), que podem ser incluídos no construtor do intento ou ajustados através de métodos como `setComponent()`, `setClass()` ou `setClassName()`;

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

3. Os **Dados** – **compostos por um *Uniform Resource Identifier* (URI) e pelo tipo MIME** (*Multi-Purpose Internet Mail Extensions*) **do conteúdo para onde aponta**. O tipo de dados é **normalmente determinado pela ação do intento** (*e.g.*, se a ação for do tipo `ACTION_EDIT`, o URI deve apontar para o documento a editar). Contudo, **é recomendado que o tipo seja sempre explicitamente ajustado**, para que **o intento seja melhor filtrado. Tomar essa opção evita que, por exemplo, um visualizador de imagens seja colocado como opção ao utilizador quando este tenta abrir um ficheiro de música**. O URI e o tipo de dados **podem ser ajustados usando os métodos** `setType(String)` **e** `setData(URI)`, respetivamente. Caso seja necessário ajustar ambos, então deve ser utilizado o `setDataAndType(.,.)`;

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

5. Os **Extras** – que são **pares chave-valor** usados para **transferir dados adicionais** necessários para determinada ação. Tal como algumas ações usam URIs com uma configuração específica, também outras podem fazer uso de dados adicionais.

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

Por exemplo, quando se usa a ação `ACTION_SEND`, podem-se usar os extras com chaves `EXTRA_EMAIL` ou `EXTRA_SUBJECT` para definir o endereço de *e-mail* destino ou assunto da mensagem, respetivamente, já que algumas aplicações de *e-mail* fazem uso dele para preencher automaticamente o endereço do destinatário. Para colocar estes valores em intentos, pode-se **recorrer aos vários métodos `putExtra()`, que aceitam sempre o valor da chave no primeiro parâmetro e o valor a transportar no segundo**. Também se podem **definir todos os extras num objeto da classe `Bundle`**, passando-o depois ao intento;

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

3 Intentos Explícitos

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

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

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

O segundo exemplo mostra também um **intento explícito, mas com destino a uma aplicação diferente da que o instanciou**. Neste caso, o intento é inicializado recorrendo ao construtor vazio, sendo depois **o pacote e o nome da componente ajusta-**

dos através de `new Component(String pkg, String cls)`. Neste caso, o intento deverá culminar na abertura da calculadora que vem por defeito no SO Android.

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

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

4 Intentos Implícitos

Os **intentos implícitos** são aqueles para os quais **não é especificado o nome ou pacote do componente a executar**. Em vez disso, **é declarada uma ação geral a ser desenvolvida** pela componente recetora e eventualmente **uma ou mais categorias** a que esta deve pertencer, bem como **dados adicionais**. Estes intentos são **particularmente úteis** para quando se quer fazer **uso de uma funcionalidade que outra aplicação do sistema possa oferecer**, sem especificar exatamente qual. Por exemplo, uma aplicação pode querer mostrar uma imagem ao utilizador, apesar de não possuir essa funcionalidade. Nesse caso, pode emitir um intento com ação `ACTION_VIEW` e aguardar que o SO lhe localize uma aplicação capaz de lidar com essa ação específica.

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

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

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

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

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

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

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

5 Envio de Dados Via Intento

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

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

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

```
import android.content.Intent;
...
Intent iActivity = new Intent(this,Activity2.class);
iActivity.putExtra("string1","This string is going to Activity2.");
startActivity(iActivity);
```

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

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

6 Obtenção de Resultados Via Intento

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

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

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

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



9 Segurança em Android

Sumário

Introdução ao tema da segurança na plataforma Android, dando especial ênfase à arquitetura de permissões e controlo de acesso sobre a qual elabora.

Summary

Introduction to the security subject, paying special attention to the permissions and access control architecture in which the Android platform elaborates on.

1 Introdução

A **arquitetura de segurança do Android elabora** simultaneamente **em mecanismos que o núcleo do SO Linux disponibiliza** (nomeadamente o sistema de controlo de acesso a ficheiros) **e em mecanismos adicionais que implementa**, baseados sobretudo em filtros[1]. Um dos **alicerces base** da arquitetura consiste na **assunção de que uma aplicação, por defeito, não tem permissão para executar operações que possam ter impacto adverso noutras aplicações, no SO ou para o utilizador**. Esta restrição aplica-se, portanto, a **dados privados** que possam estar no dispositivo (*e.g.*, contactos ou fotos), **à leitura e escrita em ficheiros** pertencentes a outras aplicações, e **aceder a recursos considerados protegidos ou sensíveis**, como redes de comunicação, câmara, etc. É esta assunção que formaliza o conceito de *sandbox*.

Como **cada aplicação** Android executa numa *sandbox*, estas **têm que explicitamente pedir permissões para usar recursos** não fornecidos, de forma nativa, por essa *sandbox*. Estas permissões **são declaradas estatica e explicitamente no manifesto** da

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

aplicação, e **o sistema pede o consentimento ao utilizador** para o usufruto dos recursos **aquando da instalação** ou **durante a execução**. Na verdade, em versões anteriores à 6.0, as permissões teriam de ser **todas aceites** pelo utilizador aquando da instalação, ou a aplicação nem sequer instalava. A partir da versão 6.0, é possível dar permissões durante a execução da aplicação, sendo exibida pelo SO uma caixa de diálogo para esse efeito. Assim, para aplicações desenhadas para a versão 6.0 ou superior, é necessário adicionar código na aplicação que verifique se uma permissão já foi pedida, e que despolete o processo de aceitação em caso negativo.

A documentação oficial define que, no caso do Android, **o conceito de *sandbox* não está intimamente relacionado com a máquina virtual Java**, e que esta não deve ser entendida como a tecnologia que garante a segurança por isolamento. Na verdade, **qualquer tipo de aplicação, tenha ela sido implementada em Java, nativa ou híbrida, é *sandboxed*** através do mecanismo referido em baixo, que emana do núcleo do SO.

2 Controlo de Acesso e IDs do Utilizador

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

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

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

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

Após instalada, e a menos que expressamente indicado em contrário, **todos os dados guardados** por determinada aplicação ficarão, portanto, **associados ao ID que lhe foi atribuído**. Os **métodos** que normalmente são **usados para criar ficheiros em aplicações Android** são o `getSharedPreferences(String, int)`[2], `openFileOutput(String, int)` ou o `openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory)`. Todos esses métodos **aceitam uma *flag*** (um `int`) que determina as **permissões com que os respetivos ficheiros são criados**. Para permitir que outras aplicações acedam aos dados para leitura ou escrita, podem-se **usar os modos** `MODE_WORLD_READABLE` **e** `MODE_WORLD_WRITEABLE`, respetivamente, embora sejam atualmente fortemente desencorajados. Isto provoca a criação de ficheiros com o *User* ID da aplicação, mas com as seguintes permissões:

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

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

3 Assinatura Digital da Aplicação

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

A ferramenta `keytool` **pode ser usada para criar o certificado X.509** e um par

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

de chaves RSA através da combinação de opções seguinte:

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

.

Note que o certificado gerado fica guardado no ficheiro `chaveiro.keystore`, que o tipo de chaves é RSA, que o seu tamanho é de 2048 bits e que a sua validade é de 9150 dias, que pode ser decomposto na multiplicação 366 dias × 25 anos. Este valor foi aqui ajustado para enfatizar que, atualmente, a Google apenas aceita certificados com validade superior a 25 anos. O *alias* é o nome utilizado em baixo para assinar a aplicação.

A preparação da versão *release* da aplicação Android prossegue depois com a compilação da mesma usando o comando:
`$ gradle build` ou `$ ./gradlew build`
e com a assinatura do pacote resultante (`app/build/outputs/app-release-unsigned.apk`) usando a **ferramenta `jarsigner` (v1) ou `apksigner` (v2 e v3), fornecida com o SDK**:

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

.

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

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

O arquivo criado com o comando `$ gradle build` **não está alinhado** (aos bytes), conforme requerido no processo de preparação de uma aplicação Android. Um último passo consiste, portanto, na emissão de um comando parecido com o seguinte, que faz uso da ferramenta `zipalign`:
`$ zipalign -v 4 application.apk application-aligned.apk`. Note que, **para além deste alinhamento, o arquivo não deve sofrer quaisquer outra modificação após ter sido assinado digitalmente**, visto que tal irá invalidar a assinatura. O alinhamento garantirá apenas que os dados não compilados (recursos) começam todos com um alinhamento específico em relação ao início do ficheiro, o que tipicamente provoca uma redução na quantidade de RAM consumida pela aplicação.

4 Pedir Permissões no Manifesto

Por defeito, não são dadas quaisquer permissões a aplicações Android. O sistema **veda acesso a todos os recursos que estão para além daqueles incluídos no projeto ou dos que são criados pela própria aplicação aquando da sua execução**. Isto significa que sem o **pedido explícito de permissões no manifesto**, conforme descrito a seguir, uma determinada aplicação móvel só terá acesso aos ficheiros incluídos na pasta `res` e aos que entretanto criar, que são normalmente guardados num componente de armazenamento interno [3] ou externo. O pedido de permissões é **feito através da colocação de uma ou mais *tags* `<uses-permission>`, cujo atributo `android:name` específica o recurso a que se quer ter acesso**. O elemento `<uses-permission>` está **contido obrigatoriamente no elemento `<manifest>`** (e não em `<application>` ou `<activity>`, pelo que **se aplica a toda a aplicação**. A seguir inclui-se um exemplo de um pedido de permissão para ler o registo de chamadas do sistema[5]:

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

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

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

1. **Definir** a permissão no manifesto, como descrito em cima (sempre necessário);
2. **Verificar** se a permissão já foi dada, recorrendo ao método `checkSelfPermission(String)` (ver trecho de código seguinte); e

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

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

3. **Pedir** a permissão com o método `requestPermissions(String[], int)` caso ainda não tenha sido concedida antes, que funciona de forma assíncrona (retornando imediatamente).

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

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

Quando uma aplicação tenta **aceder a um recurso para o qual não tem permissão** (*e.g.*, porque não a declarou, verificou e pediu), **é normalmente enviada** para o componente em questão **uma** `SecurityException`. Embora essa exceção possa nem sempre ser disparada[4], **problemas de permissões são quase sempre reportados no *log*** do sistema. Se o componente respetivo não estiver preparado para lidar com a exceção, a aplicação pode terminar abruptamente.

5 Definir Permissões no Manifesto

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

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

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

```
    android:label="@string/activity_permission"
    android:description="@string/act_permission_desc"
    android:permissionGroup="android.permission-group.CAMERA"
    android:protectionLevel="dangerous" />
    ...
</manifest>
```

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

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

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

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

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

A seguir mostra-se um exemplo para estes dois atributos, que vem no seguimento do anterior:

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

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

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

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

O output será semelhante a:

```
All Permissions:

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

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

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

6 Aplicar Permissões a Componentes

É possível **aplicar as permissões com bastante granularidade** no SO Android. Depois de definida (e nomeada), uma permissão é **aplicada a determinados componentes ou a toda a aplicação através do atributo**

`android:permission="name_of_the_permission"`.

A **permissão mais específica sobrepõe sempre a menos específica**. *E.g.*, uma permissão aplicada a uma *activity* terá precedência relativamente a uma permissão apli-

cada à *application*. De uma forma breve, a forma de atuação das permissões para os vários componentes de uma aplicação Android pode ser definida da seguinte forma:

- As permissões **aplicadas a uma atividade restringem que componentes é que a podem despoletar**. As permissões são verificadas quando é invocado o método `startActivity()` ou `startActivityForResult()`. Caso as permissões não sejam concedidas, é disparada uma `SecurityException`;
- As permissões **aplicadas a um serviço** (`<service>` *tag*) **restringem também quem pode começar ou associar-se ao mesmo**. São verificadas aquando da invocação de `startService()`, `stopService()` ou `bindService()`. Eventualmente, podem-se verificar permissões durante a execução de um serviço através do método `checkCallingPermission(string)`, em que a *string* indica o nome da permissão que se quer verificar;
- As permissões **aplicadas a Recetores de Difusão são definidas nas *tags* `<receiver>` e restringem quais as aplicações que podem enviar eventos para esses recetores**. Neste caso, **a permissão só é validada após o método `sendBroadcast()` devolver** (*i.e.*, `return`), já que **é o sistema que faz a tentativa de entrega ao recetor**, e não o próprio emissor. Por isso, **o emissor tem de esperar pelo retorno do sistema**. Neste caso, **nunca será levantada uma exceção**. É possível que um recetor forneça também a permissão numa *string* aquando da invocação do método `sendBroadcast()`, para controlar programaticamente os emissores que podem enviar mensagens para o componente. De igual forma, o componente emissor pode especificar a permissão aquando da invocação de `sendBroadcast()`;
- Finalmente, as permissões **aplicadas a fornecedores de conteúdos são definidas na *tag* `<provider>` e são usadas para restringir o acesso a determinado conteúdo**. Contudo, **para estes componentes é possível definir dois atributos** para as permissões que determinam se um aplicação especifica pode **ler** (`android:readPermission`) **ou escrever** (`android:writePermission`) no fornecedor de conteúdos [5]. As **permissões são verificadas na primeira invocação** do provedor de conteúdos. O método `query()` requer permissões de leitura, enquanto que os métodos `insert()`, `update` e `delete()` requerem a permissão de escrita.

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



10 Armazenamento de Dados Persistentes

Sumário
Armazenamento e gestão de dados persistentes em aplicações móveis Android: dados privados, partilhados e estruturados.

Summary
Storage and management of persistent data in Android mobile applications: private, shared and structured data.

1 Introdução

Uma das **funcionalidades mais úteis** para a maior parte das aplicações móveis é a de **gerir e armazenar dados de forma persistente**. O SO Android disponibiliza diversas formas de o fazer, nomeadamente[1]:

1. Um **recurso/classe chamado `SharedPreferences` (preferências partilhadas), para se guardarem dados primitivos em pares chave-valor**;
2. **Armazenamento interno**, para se guardarem dados na **memória persistente do dispositivo**;
3. **Armazenamento externo**, para se guardarem **dados públicos na memória persistente partilhada e externa** (*e.g.*, SDcards);

[1]Este capítulo é sobretudo baseado no `http://developer.android.com/guide/topics/data/data-storage.html` e nas referências principais desta unidade curricular.

4. Bases de dados **SQLite**, para **armazenamento e acesso eficiente de dados estruturados** em bases de dados **privadas**; e

5. Formas de **acesso à rede**, para armazenamento e gestão de **dados remotos**.

É claro que **o tipo** de armazenamento ou recurso específico utilizado **irá depender dos requisitos específicos da aplicação ou funcionalidade implementada**. Por exemplo, se a ideia é guardar as **definições de utilizador ou valores de estado simples** da aplicação, as `SharedPreferences` irão compreender o recurso ideal. Se a ideia for **guardar dados não estruturados** (*e.g.*, textos ou imagens) mas que apenas possam ser **acedidos pela própria aplicação**, o armazenamento interno concretiza uma melhor opção. Por outro lado, se a ideia é guardar dados com formato variado em ficheiros que possam ser partilhados (*e.g.*, fotos), então o armazenamento externo será o alvo ideal. Finalmente, se se pretende **guardar dados com estrutura** e que mais tarde possam ser acedidos de forma eficiente, uma **base de dados relacional** será indicada.

Em baixo, discutem-se os vários recursos/classes enumerados em cima, exceto o último, eventualmente descrito num capítulo adiante. À discussão das bases de dados SQLite será dedicada uma secção, enquanto que os três primeiros serão descritos nas 3 subsecções seguintes.

2 Preferências Partilhadas

A implementação da classe `SharedPreferences`[2] **oferece o *software* necessário para guardar e recuperar dados de tipos Java primitivos** (ou *strings*), como `booleans`, `floats`, `ints`, `longs` ou `Strings`. Estes dados são **guardados em pares chave-valor**, em que **a chave é a *string* (o nome)** que define aquele valor. Apesar deste recurso ser **especialmente útil para guardar as preferências do utilizador** para uma aplicação como, por exemplo, o tamanho da letra ou definições de som, pode ser usado para **armazenar e gerir dados que precisem persistir entre sessões** de utilização de uma aplicação, desde que sejam constituídos pelos tipos mencionados antes[3].

A **instanciação de um objeto** da classe `SharedPreferences` é **feita através da invocação do método** `getSharedPreferences(string, int)` **ou** `getPreferences(int)`. O método referido em primeiro lugar deve ser usado **caso se pretenda dar um nome ao ficheiro de preferências, ou obter o ficheiro previamente guardado com esse nome**. O nome é dado no parâmetro da classe *String*. O método aceita também um inteiro, que especifica o modo como o ficheiro deve ser guardado ou acedido (os modos de um ficheiro são discutidos nas próximas secções). **Caso só se esteja a usar o ficheiro**

[2] Ver http://developer.android.com/reference/android/content/SharedPreferences.html.

[3] Note que pode estender uma classe `PreferenceActivity` (não tratada neste curso) para melhor lidar com um menu de preferências.

por defeito, pode ser usado o método `getPreferences(int)`, que aceita apenas o inteiro que define o modo de acesso ao ficheiro. Ambos os métodos são **fornecidos com o contexto da componente em utilização**. As preferências partilhadas **são ficheiros XML, guardados normalmente na diretoria de dados** da aplicação (dada por `/data/data/nome_pacote_aplicacao`), nomeadamente na subdiretoria `shared_prefs`:

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

Após se instanciar o objeto `SharedPreferences`, é possível **ler qualquer um dos valores** que armazena através de **métodos semelhantes a** `getBoolean(string,boolean)` **ou** `getInt(string,int)`**, que aceitam a chave** que define cada par, e devolvem o respetivo valor. O segundo parâmetro destes métodos é usado por conveniência, e para definir o que a função deve devolver **por defeito caso a referida chave não exista** no ficheiro.

Para **escrever dados nas preferências partilhadas, é preciso obter um segundo objeto do tipo** `Editor` **através do método** `edit()`. Este segundo objeto **disponibiliza vários métodos com prefixo** `put`, nomeadamente `putBoolean(string,boolean)` ou `putInt(string,int)`, que permitem precisamente guardar pares no objeto. Contudo, **estes pares só serão verdadeiramente armazenados no ficheiro** com caráter persistente **depois de se emitir o método** `apply()` **ou** `commit()`.

O pedaço de código seguinte exemplifica a utilização das preferências partilhadas[4]. É mostrado como se pode obter um valor booleano previamente guardado bem como este se pode ajustar após obtenção do editor respetivo. Aparentemente, o valor booleano é guardado quando se invoca o método `exit(View)` (provavelmente despoletado pelo clique num botão), imediatamente antes da atividade terminar.

```
public class SimpleNotes extends Activity {
  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    ...
    // Get the previously stored preferences
    SharedPreferences oSP = getPreferences();
    boolean bRecupera = oSP.getBoolean("recupera", false);
    if( bRecupera )
      ...
  }

  public void exit(View v){
    // Instantiate the Editor object
```

[4]Este código é uma adaptação de um dos exercícios práticos da aula prática 7.

```
        SharedPreferences oSP = getPreferences();
        SharedPreferences.Editor oEditor = oSP.edit();
        oEditor.putBoolean("recupera", true);

        // Make the edit persistent
        oEditor.commit();
        ...
        super.finish();
    }
}
```

3 Armazenamento Interno – Escrita

A segunda forma de armazenamento discutida aqui refere-se a **ficheiros comuns e à memória interna do dispositivo**. Note-se que o entendimento de memória interna e externa pode depender de vários fatores. Normalmente, a **memória externa é a que não vem por defeito com o dispositivo**, ou que se **pode amover**, ou ainda que **pode ser tipicamente acedida por várias aplicações ou utilizadores**.

Para **guardar ficheiros na memória interna, nomeadamente na diretoria** `/data/data/nome_pacote_aplicacao/`, **pode usar-se o método** `openFileOutput(String, int)`, que aceita como parâmetros o **nome do ficheiro e o modo de acesso** com que o ficheiro deve ser guardado ou aberto e devolve um objeto da classe `FileOutputStream`. **Os ficheiros** criados ou editados conforme descrito nesta secção **são eliminados automaticamente pelo sistema aquando da desinstalação** da aplicação.

Tal como para os anteriores (embora não tenha sido dito explicitamente), **o dono dos ficheiros** criados com `openFileOutput(String, int)` **é dado pelo ID da aplicação** que invocou o método. Quando guardados com o modo `MODE_PRIVATE`, estes ficheiros só estão disponíveis para a respetiva aplicação móvel. Existem 4 modos de operação que interessa conhecer:

1. `MODE_PRIVATE` (valor 0), que é **o modo sugerido por defeito, que define que apenas a aplicação que criou o ficheiro ou todas aquelas que com ela partilhem o ID lhe podem aceder**. Caso o ficheiro já exista quando é tentada a abertura com `openFileOutput(String, int)`, este é primeiro eliminado e depois recriado;
2. `MODE_WORLD_READABLE` (valor 1), que define que **o ficheiro pode ser acedido para leitura por qualquer aplicação**;
3. `MODE_WORLD_WRITEABLE` (valor 2), que especifica que **qualquer aplicação pode aceder ao ficheiro** para escrita; ou

4. `MODE_APPEND` (valor 32768), que determina que **a escrita de dados novos no ficheiro deve ser feita no final**, caso este já exista.

Note que, a partir da API 17, **os modos numerados com o 1 ou o 2 são desencorajados**, já que podem constituir problemas graves de segurança.

A **escrita de dados no ficheiro pela aplicação que o criou é possível em qualquer um dos modos** indicados antes, e pode ser simplesmente conseguida através de métodos como o `write(int iByte)`, o `write(byte[])` ou o `write(byte[] buffer, int offset, int count)`. O `FileOutputStream` deve ser terminado invocando o seu método `close()`. O código seguinte mostra como se pode escrever a *string* `Ola mundo!` num ficheiro chamado `ficheiro.txt` numa aplicação Android:

```
FileOutputStream fosFile = openFileOutput("ficheiro.txt", Context.
    MODE_PRIVATE);
fosFile.write("Ola Mundo!".getBytes());
fosFile.close();
```

Repare no grau de abstração e facilidade fornecidos pela API e também que os vários modos estão definidos estaticamente no `Context`, para sua conveniência.

Para além dos que já foram descritos em cima, podem-se **enunciar outros 4 métodos bastantes úteis no que toca a manipulação de ficheiros**:

- `getFilesDir()`, que devolve **o caminho absoluto** da diretoria onde os ficheiros são guardados;
- `getDir(string, int)`, que **cria a diretoria com o nome definido** no primeiro parâmetro, com as permissões de acesso definidas no segundo;
- `deleteFile(string)`, que **elimina o ficheiro** cujo nome é especificado no primeiro parâmetro; e
- `fileList()` que **devolve um vetor de *strings* com o nome de todos os ficheiros já guardados pela aplicação**.

4 Armazenamento Interno – *Cache*

Caso o objetivo seja o de **apenas guardar dados por algum tempo**, pode **recorrer a uma *cache*** que o sistema Android fornece nativamente. A ideia é simplesmente **guardar os ficheiros na subdiretoria** `cache` da diretoria de dados da aplicação. O caminho para esta diretoria pode ser obtida no seio da execução através do método `getCacheDir()`. Quando os **recursos de armazenamento começarem a escassear no sistema, este**

começará a eliminar ficheiros contidos em subdiretorias `cache`, pelo que **não se deve presumir que estes ficheiros estarão sempre disponíveis** e sobrevivam entre uma sessão e a seguinte. É ainda **recomendado** que os programadores **não façam depender do sistema a eliminação dos ficheiros em** `cache`, já que concretiza uma má política. Devem ser implementados métodos na aplicação que, frequentemente ou quando um determinado limite é atingido, limpem ou organizem a referida diretoria.

5 Armazenamento Interno – Leitura

A **leitura do conteúdo de um ficheiro armazenado internamente é conseguida através da instanciação de um** `FileInputStream`, que resulta da invocação do **método** `openFileInput(string)`. O seu único parâmetro é o nome do ficheiro. O método `read (byte[] buffer, int byteOffset, int byteCount)` pode depois ser usado para devolver `byteCount` bytes para o vetor de bytes `buffer`, ou o ficheiro pode ser lido um byte de cada vez recorrendo a `read()`, que devolve um inteiro (representando um byte) e itera o cursor de leitura no ficheiro. Por defeito, a **aplicação procura o ficheiro a abrir na diretoria** `/data/data/nome_pacote_aplicacao/`.

Note que **é possível incluir um ficheiro estático no projeto** da sua aplicação. Nesse caso, deve **guardá-lo numa subdiretoria de** `res` **chamada** `raw`**.** O nome da diretoria informa o empacotador que não deve comprimir este ficheiro. O método `openRawResource(int)` é o que permite abrir ficheiros deste género **para leitura**, devolvendo um objeto da classe `FileInputStream`. O único parâmetro do método é o ID do ficheiro, escrito na forma `R.raw.nome_do_ficheiro`. Note que **não é possível escrever para ficheiros guardados em** `res/raw` (daí a qualificação de estático incluída antes).

6 Armazenamento Externo

Todos os dispositivos Android suportam armazenamento externo partilhado que também pode ser usado para guardar dados em ficheiros de forma persistente. O meio de armazenamento pode ser **um cartão de memória externo** (tal como um cartão *Secure Digital* (SD)) ou concretizada por **uma parte do sistema de ficheiros num dispositivo não amovível (portanto interno)**. A particularidade que melhor distingue este tipo de armazenamento é o facto de **ficar disponível como uma unidade de armazenamento USB** quando o dispositivo móvel é ligado a um computador. Os ficheiros guardados no armazenamento externo **têm permissões de leitura para todos** (*i.e.*, são `world-readable`), e **é possível que um utilizador os possa modificar via outro dispositivo computacional compatível**.

Dadas as características do armazenamento externo, **não deve ser presumido que este estará sempre disponível** ou presente (*e.g.*, um utilizador pode remover um cartão

SD ou ligar o *smartphone* ao computador), **nem que os ficheiros que aí são guardados se mantêm inalterados de uma execução para outra** (um utilizador ou uma aplicação podem alterá-los). Consequentemente, **recomenda-se que qualquer código que manuseie armazenamento externo seja guardado por uma verificação** se este realmente existe ou está disponível. Esta verificação pode ser feita **obtendo o estado do armazenamento com o método** `getExternalStorageState()`, que devolve uma *string*, e **comparando-o com um dos estados predefinidos estaticamente** na classe `Environment`. O código seguinte mostra a implementação de um método que devolve `true` caso o armazenamento externo esteja disponível **pelo menos** (dado pelo operador `||`[5]) para leitura:

```
public boolean isExternalStorageReadable() {
  String state = Environment.getExternalStorageState();
  if (Environment.MEDIA_MOUNTED.equals(state) ||
      Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)){
    return true;
  }
  return false;
}
```

È preciso ter em atenção que, **para uma aplicação ter acesso** ao armazenamento externo, **tem que normalmente pedir essa permissão** no `AndroidManifest.xml`, nomeadamente através da inclusão de **uma das duas linhas seguintes** naquele ficheiro:

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

A exceção aplica-se ao caso em que aplicação só vai aceder a ficheiros criados por si nesse armazenamento e a versão do SO seja superior à **4.4.** Antes da versão indicada, todos os acessos teriam de ser explicitamente pedidos. Note que, **caso apenas sejam necessárias permissões de leitura, então deve ser usado o elemento com nome** `READ_EXTERNAL_STORAGE`. **O outro elemento pede permissões para escrita, e para a combinação escrita/leitura**.

O exemplo seguinte é um pouco mais elaborado que o anterior e contém mais detalhes. O objetivo do excerto de código é copiar o conteúdo do ficheiro `f1.txt`, incluído na pasta `res/raw` do projeto, para o `ficheiro.txt`, que estará alojado no armazenamento externo. A cópia só é tentada depois de ser verificado que a unidade está disponível, nomeadamente através da comparação do seu estado com a *string* `Environment.MEDIA_MOUNTED`. No caso afirmativo, é obtida a diretoria da aplicação na memória externa através de `getExternalFilesDir(null)` e aí criado o ficheiro destino. Depois disso, o objeto que representa o ficheiro é, no fundo, convertido num `OutputStream`, para onde são escritos todos os bytes lidos de `f1.txt`. Note-se que o ficheiro `f1.txt` foi aberto através do método `getResources().openRawResource(R.raw.f1)`, conforme discutido acima:

[5] `||` é o operador lógico *ou exclusivo* em Java.

```
String state = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
  File fFile1 = new File(Environment.getExternalFilesDir(null), "ficheiro.txt");
  OutputStream fosFile = new FileOutputStream(fFile1);
  InputStream fisFile = getResources().openRawResource(R.raw.f1);
  byte[] baBuffer = new byte[fisFile.available()];
  fisFile.read(baBuffer);
  fosFile.write(baBuffer);
  fosFile.close();
  fisFile.close());
}
```

Note que o código incluído antes pode disparar exceções não tratadas no exemplo.

O método `Environment.getExternalStorageState()` devolve todos os estados possíveis do armazenamento externo (*e.g.*, também pode indicar se o dispositivo está atualmente ligado a um computador). Estes estados podem ser **tratados com mais granularidade** para definir vários fluxos para o programa ou para notificar o utilizador em conformidade.

Uma determinada aplicação **pode guardar ficheiros numa diretoria do armazenamento externo que lhe é dedicada**, ou **numa que já tenha sido criada pelo sistema** (*e.g.*, a diretoria das imagens, etc.). Caso a intenção seja guardar algo numa **diretoria de topo, pública e conhecida**, esta pode ser **procurada especificando o seu nome no primeiro parâmetro do método `getExternalStoragePublicDirectory(string)`**. Por exemplo, quando invocada com `getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES)`, o método devolve o caminho qualificado da diretoria pública que contém as imagens no armazenamento externo na forma de um ficheiro.

É preciso saber que, no caso em que os **ficheiros são guardados numa diretoria da aplicação** (*i.e.*, construída via `getExternalFilesDir(null)`), estes **serão eliminados caso a aplicação seja desinstalada**. Por isso, não se devem guardar ou transferir dados privados importantes do utilizador para essa diretoria (*e.g.*, uma música que o utilizador comprou via *smartphone*). Por outro lado, o facto da diretoria pertencer à aplicação **não determina necessariamente que outras aplicações (com as devidas permissões) ou o utilizador não possam manusear** os seus ficheiros. Por fim, **uma diretoria pertencente a uma aplicação não é normalmente rastreada pelo *Media scanner***. Aliás, é possível esconder o conteúdo dessas diretorias a esse programa colocando lá dentro um ficheiro chamado `.nomedia`.



11 Armazenamento de Dados Estruturados

Sumário

Gestão de dados estruturados através do motor de bases de dados com suporte à linguagem *Structured Query Language* (SQL) conhecido por SQLite.

Summary

Management of structured data using the databases engine with support to Structured Query Language (SQL) known as SQLite.

1 Introdução

O capítulo anterior discutia com algum afinco a forma de armazenar e manusear dados persistentes, mas possivelmente sem estrutura, num dispositivo móvel Android. Uma vez tendo acesso ao recurso abstrato encabeçado por **um ficheiro**, é óbvio que já existe forma de guardar dados estruturados, como por exemplo registos telefónicos ou informações de uma rede social. Contudo, **os ficheiros não fornecem, por defeito, formas eficientes e consistentes de pesquisar, aceder, modificar e eliminar dados específicos com estrutura**. Essas funcionalidades e a eficiência estariam **dependentes da inclusão de mecanismos** no código que são **típicos dos chamados sistemas de gestão de bases de dados,** mais conhecidos pela sua designação inglesa *Database Management Systems* (DBMSs).

Os **DBMSs modernos incorporam mecanismos bastante avançados** de manuseamento de dados e **interpretam**, para as bases de dados relacionais, **a linguagem estruturada de consultas**, conhecida sobretudo pelo acrónimo da sua designação in-

glesa *Structured Query Language* (SQL). Estes dois factos (que se conjugam a outros não referidos aqui), **permitem que os programadores** que recorrem a estes sistemas **se possam abstrair da maior parte dos detalhes de como os dados são guardados** (qual o formato), com **questões de eficiência ou regras de consistência, podendo concentrar-se apenas no conteúdo**. **Os dois SOs** para dispositivos móveis **mais populares** do mercado (iOS[1] e Android) **recorrem ao SQLite para armazenamento e gestão de bases de dados relacionais**.

2 SQLite

O **SQLite**[2] é **uma biblioteca de *software* que implementa um motor de bases de dados SQL**. Por construção, o SQLite **não é um DBMS**, mas apresenta **muitas das suas características** através da inclusão dos mecanismos típicos que estes sistemas usam. O SQLite é, atualmente, e também em consequência da sua integração nos SOs que dominam o mercado dos dispositivos móveis, **o motor de bases de dados mais instalado do mundo**, sendo compatível e capaz de interpretar a **norma SQL de 1992**, embora as atualizações que vai sofrendo o tragam **cada vez mais próximo de normas atuais** em alguns aspetos.

O SQLite, atualmente na versão 3.36.0 e por isso tipicamente conhecido por **SQLite3**, foi inteiramente desenvolvido na **linguagem de programação ANSI-C e é código aberto**. Com todas as suas funcionalidades ativas, **não ocupa mais do que 500 Kb**, sendo que **uma utilização normal do motor requer 250 Kb de memória**, o que se apresenta **ideal para dispositivos móveis**, onde podemos ter recursos limitados. É possível obter **toda a implementação** do SQLite **num único ficheiro `.c` designado por `amalgamation.c`** e a documentação técnica define-o, por causa disso, como **um motor auto-contido**. Contrariamente a muitos outros DBMSs, o SQLite **não implementa o modelo Cliente/Servidor** (por isso não existe nenhum servidor) e **não requer qualquer configuração para ser usado**. Ainda assim, **tem total suporte para transações**, garantindo a a sua Atomicidade, Consistência, Independência e Durabilidade (**ACID**):

- **[Atomicidade] Uma transação** (que pode ser constituída por várias operações de leitura e escrita a uma base de dados) é uma unidade atómica de processamento, que **é realizada completamente ou**, simplesmente, **não é realizada**;
- **[Consistência]** A execução de **uma transação preserva a consistência da base de dados**, *i.e.*, cada transação leva a base de dados de um estado consistente para outro;

[1] *E.g.*, ver `https://developer.apple.com/technologies/ios/data-management.html`.
[2] Visitar `http://www.sqlite.org/`.

- **[Isolamento]** As **atualizações feitas por várias transações concorrentes produzem o mesmo efeito que se estas fossem executadas em série**. Por vezes, tal significa que as modificações de uma transação são invisíveis para outras transações enquanto não atinge o estado `COMMITTED`;

- **[Durabilidade]** Se uma transação altera a base de dados e é `COMMITTED`, **as alterações nunca se perdem mesmo que ocorra uma falha** posterior.

Normalmente, **cada base de dados SQLite é guardada num ficheiro** e manuseada por **uma única aplicação de cada vez**. Também é **comum haver apenas uma única base de dados em determinado ficheiro**. Isto significa que o SQLite é **sobretudo usado para bases de dados de âmbito local**. Contudo, são **suportados volumes de dados até 2 TB** e foram incluídos mecanismos que **permitem o acesso concorrente à mesma base de dados** por mais do que uma aplicação. É ainda **fornecida uma *shell***, semelhante ao cliente de muitos DBMSs, que permite **manipular o esquema, consultar e gerir a base de dados** a partir de uma interface de linha de comandos (ver em baixo). Para além das instruções mais comuns SQL, **é também possível definir *triggers* (gatilhos)** para as bases de dados. Note que o facto do SQLite ser implementado em C em poucos ficheiros (ou apenas em um, numa das variantes) permite que este **seja compilado e embutido totalmente no executável de uma aplicação**, favorecendo a sua portabilidade.

3 Linguagem Estruturada de Consultas

SQL é a linguagem de manipulação de bases de dados **relacionais** por excelência. Contudo, como o SQLite vem nativo às principais plataformas, estas também fornecem algum *software* que permite **abstrair o programador da linguagem, embora não totalmente**. Independentemente disso, o conhecimento em SQL é **importante tanto na manipulação dos dados como na depuração de uma aplicação**, já que pode ser necessário consultar ou alterar o esquema de uma base de dados para além da própria aplicação, num dos passos da depuração de um problema. Nesta secção, discute-se muito brevemente a sintaxe de algumas das instruções SQL mais importantes, tanto para o desenvolvimento como para a depuração.

A listagem seguinte mostra **a sintaxe de criação de uma tabela conforme entendida pelo SQLite**. Neste caso, a sintaxe é dada numa **forma totalmente textual**, embora os engenheiros deste motor de bases de dados prefiram uma representação visual, como de resto se mostra em baixo. **As partes** da instrução de criação da tabela **opcionais são indicadas entre parêntesis retos**. Como se pode deduzir, em SQLite é possível criar tabelas temporárias e **definir o seu esquema explicitamente a partir de definições de colunas** (`ColumnDef_1`,...,`ColumnDef_N`) **ou herdá-lo de outras relações a partir de uma instrução de** `SELECT`.

```
CREATE [TEMP] TABLE [IF NOT EXISTS]
[Database.] TableName
[(
    ColumnDef_1, ..., ColumnDef_N, TableConstraints
)
[WITHOUT ROWID]]
[AS SELECT_STMT]
```

A sintaxe inclui uma **opção designada por `WITHOUT ROWID`**. Aquando da criação de tabelas SQLite, é **criada uma coluna especial com um ID**, usada para diversos objetivos pela própria implementação do motor. Este campo é **normalmente invisível** para o utilizador. Caso se queira **evitar explicitamente que a coluna seja criada**, é esta instrução que o vai permitir.

Caso se opte pela **definição das várias colunas, é necessário indicar obrigatoriamente o nome de cada uma**. Opcionalmente, deve ser **indicado o tipo de cada uma** (*e.g.*, `INT`, `VARCHAR`, `FLOAT`, etc.) **e restrições adicionais** (*e.g.*, `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `CHECK` e `FOREIGN KEY`), conforme se mostra a seguir:

```
Column_name [type] [constraint]
```

As **restrições permitem definir regras de integridade e consistência** da base de dados. Finalmente, e ainda no caso de se ter optado pela definição das colunas, é possível especificar **restrições ao nível de toda a tabela ou que se aplicam a mais do que uma coluna no final**, após todos os nomes estarem declarados. Por exemplo, caso se queira definir uma chave primária composta por `ColumnDef_1` e `ColumnDef_2`, tal pode ser conseguido colocando a restrição final `PRIMARY KEY(ColumnDef_1, ColumnDef_2)`.

A **eliminação de uma tabela** em SQLite pode ser conseguida através de uma instrução com a seguinte sintaxe (neste caso, basta indicar o nome da tabela a eliminar após declarar `DROP TABLE`):

```
DROP TABLE [IF EXISTS] [Database.] TableName
```

Recorde que a **SQL é uma linguagem declarativa**, o que basicamente significa que **as instruções definem o que deve ser feito, não como deve ser feito**. Caso as instruções sejam emitidas numa *shell*, estas devem ser normalmente sucedidas de um caracter terminador, que é muito frequentemente o ponto e virgula (`;`).

A figura 11.1 **ilustra a sintaxe da instrução de `SELECT` em SQLite**. A imagem é a mesma que aparece na documentação oficial do SQLite e representa a forma preferida dos seus criadores para a expressar. De facto, a representação é útil, porque **permite construir a instrução seguindo as várias *keywords* iterativamente**. De uma forma básica e direta, pode dizer-se que a palavra `SELECT` pode ser precedida por `WITH RECURSIVE`, utilizado para *queries* em dados estruturados hierarquicamente, e que

é sempre sucedido pela discriminação das colunas que devem ser exibidas, bem como pela palavra `FROM`**, que precede o nome da ou das tabelas para as quais a consulta é feita. Condições** à seleção são **introduzidas pela cláusula** `WHERE`. A **agregação** de dados pode ser conseguida por uma **cláusula de** `GROUP BY` **e restrições a esta agregação são definidas pela cláusula** `HAVING`[3]. A figura enfatiza que **a cláusula de ordenação, a ter de existir, deve ser sempre colocada após todas as outras expressões e restrições, exceto da de limitação** no número de resultados.

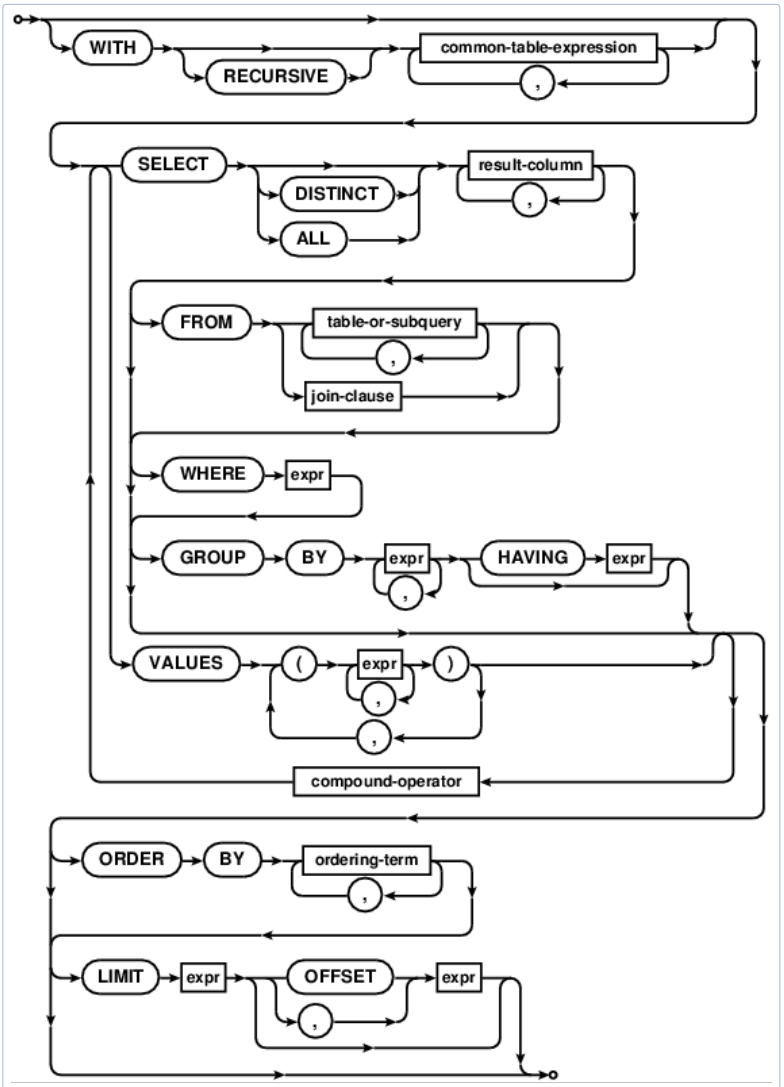


Figura 11.1: A instrução SQL de `SELECT` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_select.html).

[3] A cláusula `HAVING` tem o mesmo efeito de `WHERE`, mas aplica-se a dados agrupados.

Note que a representação mostra os vários caminhos que podem ser seguidos para construir a instrução, evidenciando simultaneamente quais são opcionais ou não. A convergência dos vários caminhos para determinada *keyword* é sinal da sua obrigatoriedade.

As figuras 11.2, 11.3 e 11.4 ilustram a sintaxe das instruções de `INSERT`, `UPDATE` e `DELETE`, respetivamente. No primeiro caso, as instruções são normalmente da forma

```
INSERT INTO Table_Name(col1, ..., coln)
VALUES(val1, ..., valn);
```

enquanto que no segundo e terceiro, as instruções são normalmente parecidas com

```
UPDATE Table_Name
SET col1 = val1, ..., coln = valn
WHERE exp;
```

```
DELETE FROM Table_Name WHERE exp;
```

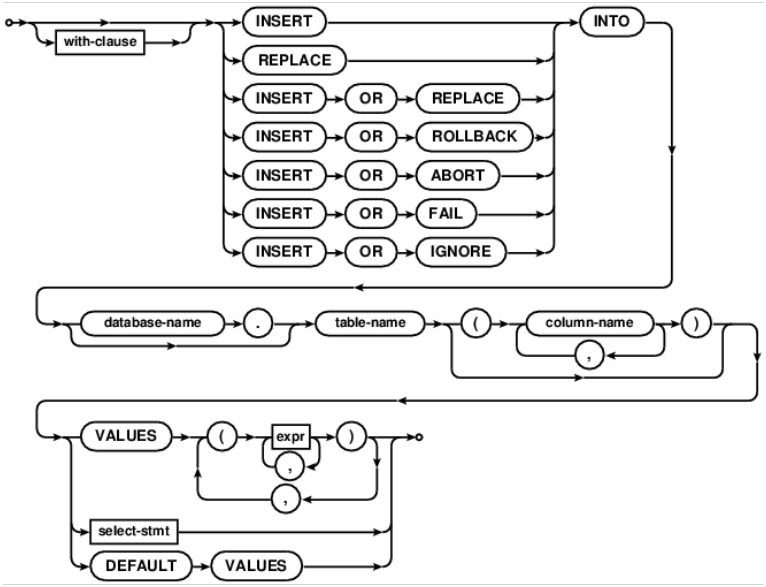


Figura 11.2: A instrução SQL de `INSERT` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_insert.html).

A instrução de `DELETE` pode ser usada para evidenciar a necessidade de **definir chaves primárias com sentido nas relações** de uma base de dados relacional. Note-se que, de um modo geral, a **eliminação de uma determinada linha numa tabela da base de dados só pode ser conseguida se existir uma coluna cujos valores identifiquem univocamente cada linha**. Caso a cláusula `WHERE` não esteja declarada nos últimos dois tipos de instruções, todas as linhas da tabela `Table_Name` serão atualizadas (no caso do `UPDATE`) ou eliminadas (no caso do `DELETE`).

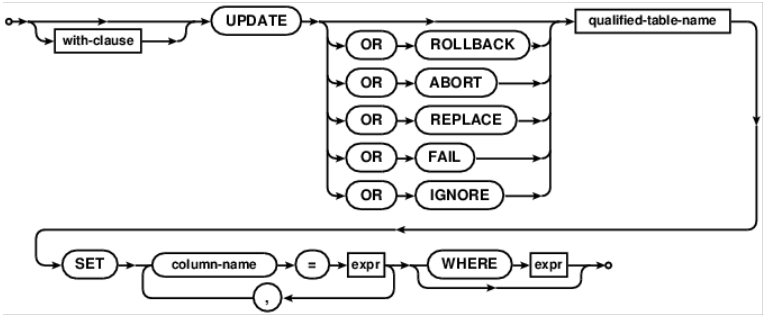


Figura 11.3: A instrução SQL de `UPDATE` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_update.html).

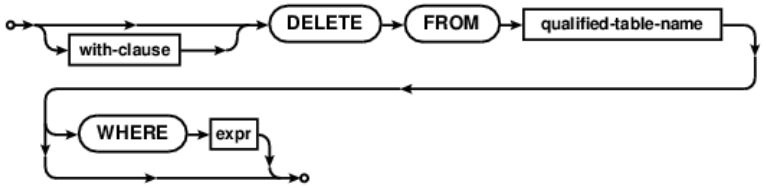


Figura 11.4: A instrução SQL de `DELETE` conforme entendida pelo SQLite3 (retirada de https://www.sqlite.org/lang_delete.html).

4 Bases de Dados SQLite em Android

A utilização de bases de dados SQLite em Android é normalmente conseguida através de ***software*** **nos pacotes** `android.database` **(que contém classes genéricas para lidar com bases de dados) e** `android.database.sqlite` **(que contém classes específicas para manusear bases de dados SQLite)**.

A forma **recomendada para criar ou atualizar** uma base de dados SQLite é através da **declaração e implementação de uma subclasse de** `SQLiteOpenHelper`, que requer o *import* de `android.database.sqlite`. Ao estender essa classe é **também necessário reescrever obrigatoriamente o método** `onCreate()` e, opcionalmente, o método `onUpgrade()`. O método `onCreate()` é apenas **executado da primeira vez que uma base de dados é criada**, ou seja, quando é tentada a abertura de uma base de dados e esta ainda não existe. Assim, **este método constitui o local ideal para colocar as instruções para criação da base de dados**. Em baixo, inclui-se o código Java que exemplifica a criação da base de dados `EngInf`, que apenas irá conter uma única tabela com 3 campos:

```
package pt.di.ubi.pmd.exstorage2;

import android.database.sqlite.SQLiteDatabase;

public class AjudanteParaAbrirBaseDados extends SQLiteOpenHelper {
```

```
  private static final int DATABASE_VERSION = 2;
  private static final String DATABASE_NAME = "EngInf";
  protected static final String TABLE_NAME1 = "Student";
  protected static final String COLUMN1 = "number";
  protected static final String COLUMN2 = "name";
  protected static final String COLUMN3 = "avg";

  private static final String STUDENTS_TABLE_CREATE =
    "CREATE TABLE " + TABLE_NAME1 + " (" +
    COLUMN1 + " INT PRIMARY KEY, " +
    COLUMN2 + " VARCHAR(30), " +
    COLUMN3 + " FLOAT);";

  private static final String STUDENTS_TABLE_DROP =
    "DROP TABLE " + TABLE_NAME1 + ";";

  private static final String STUDENTS_TABLE_TEMP =
    "CREATE TEMP TABLE AlunosAux AS SELECT * FROM " + TABLE_NAME1 + ";";

  private static final String STUDENTS_TABLE_INSERT =
    "INSERT INTO " + TABLE_NAME1 +
    "(" + COLUMN1 + "," + COLUMN2 + ") " +
    "SELECT * FROM " + STUDENTS_TABLE_TEMP + ";";

  AjudanteParaAbrirBaseDados(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }

  @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);
  }
}
```

Note que, no exemplo anterior, são colocados em evidência vários detalhes. Primeiro, é feita **a especificação do *package*, que deve ser igual para todas as classes do projeto**. Depois, é declarado um *import* importante, embora possam ser necessários mais. A implementação da classe começa pela **declaração de *strings* que definem o nome da tabela e das suas colunas, bem como das instruções SQL que permitem criar e atualizar a base de dados**. O **construtor padrão da superclasse é simplificado** através da definição estática de alguns valores pertencentes à base de dados em questão, pedindo apenas **o contexto da aplicação onde é criada**. A **única instrução** que o método `onCreate()` executa, neste caso, é a que cria a tabela `Student`.

O exemplo anterior é, na verdade, um pouco mais complexo do que o que é normalmente encontrado em introduções ao tema. Nele é também exemplificado como se pode fazer o

***upgrade* de uma base de dados**. Considere os dois cenários seguintes:

1. No **cenário 1, um utilizador instala e executa a aplicação pela primeira vez**;
2. No **cenário 2, um utilizador atualiza a aplicação, tendo já utilizado a versão anterior antes**. Considere que a versão anterior da aplicação fazia uso de uma base de dados mais simples, contendo apenas as colunas `number` e `name`.

No cenário 1 e durante a sua primeira execução, a aplicação chama o método `onCreate(SQLiteDatabase)` do `SQLiteOpenHelper` porque nota que a base de dados ainda não existia. Neste caso, o método criaria apenas a tabela `Student` com as 3 colunas.

No cenário 2, e durante a primeira execução após atualização, **o método `onUpgrade(SQLiteDatabase,int,int)` será invocado porque**: (i) **a aplicação nota que a base de dados já existe**; e (ii), **a versão que é passada ao construtor**[4] **é superior à anterior** (considere que a primeira versão da aplicação usava `DATABASE_VERSION = 1;`).

O método `onUpgrade(.,.,.,.)` executa 4 instruções SQL através do método `execSQL(String)`, que submete à base de dados a **instrução** que estiver **definida no parâmetro como uma *String***. Note que a forma de operar desta função faz do **SQL uma linguagem embutida e do Java a linguagem anfitriã**. Visto que a tabela da versão anterior da base de dados continha apenas dois campos, opta-se pelo seguinte procedimento para a sua atualização:

1. Começa-se por **fazer uma cópia temporária** da tabela `Student` para `AlunosAux`;
2. **Elimina-se a tabela** `Student`;
3. **Cria-se a nova tabela** `Student` (que já irá conter as 3 colunas); e
4. Finalmente, **copiam-se os registos anteriores para a nova tabela**, deixando um dos campos a *null*.

O procedimento aplicado é bastante genérico e deve poder ser usado, depois de adaptado, a diversos cenários de atualização. Contudo, visto que a única diferença entre a tabela anterior e a atual é o número de colunas, **o mesmo efeito poderia ter sido conseguido através de uma instrução SQL** `ALTER TABLE ...`

O excerto de código seguinte mostra como é que a classe `AjudanteParaAbrirBaseDados` pode depois ser invocada numa atividade, para além de chamar a atenção para outros detalhes. Basicamente, apenas **é necessário declarar e instanciar um objeto da classe referida, e chamar um dos métodos `getWritableDatabase()` ou**

[4] O número da versão é último parâmetro de `super(Context,String,CursorFactory,int)`.

`getReadableDatabase()`, para garantir que a base de dados é criada (ou atualizada) e aberta. Note que o pacote usado nesta aplicação é o mesmo que foi usado no exemplo anterior[5]:

```
package pt.ubi.di.pmd.exstorage2;

import android.database.sqlite.SQLiteDatabase;
...

public class EnfInfStudents extends Activity {
  private SQLiteDatabase oSQLiteDB;
  private AjudanteParaAbrirBaseDados oAPABD;

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
    oTView1 = (TextView) findViewById(R.id.name);
    oTView2 = (TextView) findViewById(R.id.avg);
    // ... ?other instructions?
    oAPABD = new AjudanteParaAbrirBaseDados(this);
    oSQLiteDB = oAPABD.getWritableDatabase();

    ContentValues oCValues = new ContentValues();
    oCValues.put(oAPABD.COLUMN1,new Integer(12589));
    oCValues.put(oAPABD.COLUMN2,"Pedro");
    oCValues.put(oAPABD.COLUMN3,new Double(10));
    oSQLiteDB.insert(oAPABD.TABLE_NAME1,
      null,oCValues);

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

    oCursor.moveToFirst();
    oTView1.setText(oCursor.getString(0));
    oTView2.setText(""+oCursor.getDouble(1));
  }
  @Override
  protected void onResume(){
    super.onResume();
    oSQLiteDB = oAPABD.getWritableDatabase();
  }
  @Override
  protected void onPause(){
    super.onPause();
    oSQLiteDB.close();
  }
}
```

[5]Note que, por uma questão de legibilidade, não é feita qualquer tentativa de tratar exceções no exemplo apresentado.

O exemplo anterior mostra também que **o método** `getWritableDatabase()`, para além de ser o que despoleta a criação ou abertura da base de dados, também **devolve um objeto da classe** `SQLiteDatabase`**, que pode mais tarde ser usado para consultar ou aceder à base de dados**. Para se fazer uso de tal objeto, é necessário importar `android.database.sqlite`. O método `getWritableDatabase()` **permite o acesso para leitura** (`SELECT`) **ou escrita** (`INSERT`, `DELETE` ou `UPDATE`), **enquanto que o método** `getReadableDatabase()` **apenas permite operações de consulta**.

Repare que o código incluído antes define uma atividade que contém pelo menos duas etiquetas de texto no seu *layout*. Estas duas etiquetas são instanciadas ainda antes da base de dados ser aberta e depois são preenchidas com o nome e média do melhor aluno na base de dados. Pelo meio, é ainda inserido um novo registo na tabela (o aluno com o número 12589, chamado *Pedro* e com a média 10).

O exemplo anterior também ilustra **a utilidade dos métodos** `onPause()` **e** `onResume()`, que ainda não tinham sido implementados até à data nesta unidade curricular. Neste caso, **revelam-se ideais para fechar ou reabrir a base de dados**, respetivamente. Quando a aplicação se preparar para sair de foco ou terminar, **convém fechar o acesso à base de dados**. Caso se volte à aplicação, convém reabrir a ligação, até porque os dados podem ter sido alterados entretanto.

As **bases de dados criadas no contexto de uma aplicação ficam tipicamente guardadas na diretoria de dados** atribuída a essa aplicação, dentro de uma **subdiretoria chamada** `databases`. Isto significa que ao criar uma base de dados chamada `EngInf.db` no contexto de uma aplicação cujo *package* é `pt.ubi.di.pmd.exstorage2`, por exemplo, a base de dados ficará guardada em `/data/data/pt.di.ubi.pmd.exstorage2/databases/EngInf.db`. **Por defeito, esta base de dados só estará acessível, pelo nome, para qualquer classe da respetiva aplicação**, mas não para qualquer outra.

5 SQLiteDatabase

Um objeto da classe `SQLiteDatabase` **representa determinada base de dados e disponibiliza** um conjunto de **métodos de conveniência** para a sua consulta e manipulação. Embora seja necessário algum conhecimento **SQL** para usar esses métodos, alguns deles **não requerem que se usem as *keywords* que lhe são características**. **Alguns dos métodos mais interessantes** neste contexto são:

```
long insert(String table, String nullColumnHack, ContentValues values)
```

que aceita o **nome da tabela** e os **vários valores a inserir** num objeto da classe `ContentValues`. O parâmetro `nullColumnHack` é opcional e usado para indicar o nome de uma coluna que aceite *null* para o caso de querermos inserir uma linha inteira vazia (com `ContentValues` vazio). O método **devolve o ID da linha inserida** ou -1 em caso de erro.

```
int delete(String table, String whereClause, String[] whereArgs)
```

que aceita o **nome da tabela** como primeiro parâmetro, **a cláusula `WHERE` em forma de *String*** no segundo parâmetro, e um conjunto de valores, a substituir pelo caracter `?`, se este for usado na cláusula `WHERE`. O método **devolve o número de linhas afetadas**.

```
int update(String table, ContentValues values, String whereClause, String[]
    whereArgs)
```

cuja descrição é semelhante à anterior. Note que a *String* referente à cláusula `WHERE` deve conter apenas a definição da condição, e não a própria palavra `WHERE`.

```
execSQL(String sql)
```

que **executa instruções SQL que não devolvem resultados** (ideal para instruções de `CREATE`, `ALTER` ou `DROP TABLE`).

```
Cursor rawQuery(String sql, String[] selectionArgs, CancellationSignal
    cancellationSignal)
```

que permite **executar uma instrução de `SELECT` totalmente definida como uma *String*** passada no primeiro argumento. Caso se queira usar pré-processamento e **garantir o saneamento na entrada de dados**, pode-se **definir a *query* com pontos de interrogação (`?`)**, que são depois substituídos pelos valores dados no *array* de *Strings* em segundo lugar. Para o exemplo apresentado antes, o trecho de código seguinte iria obter a(s) linha(s) que contivessem o nome `Pedro`:

```
rawQuery("SELECT * FROM Student WHERE name=?;", String[] {"Pedro"}, null);
```

Esta forma de definir os valores **permite**, por exemplo, **evitar ataques de injeção de código SQL**. Caso o parâmetro pelo qual se se quer consultar a base de dados seja oriunda da interface com o utilizador, e introduzida por ele(a), podia-se correr o risco de se estar a enviar para o motor SQLite uma *query* contaminada e com código malicioso. **Ao colocar um `?` e ao definir as *Strings* à parte, qualquer *input* vindo do utilizador é garantidamente interpretado como uma *String*, e não como código**.

Finalmente, o método

```
Cursor query(String table, String[] columns, String selection, String[]
    selectionArgs, String groupBy, String having, String orderBy, String
    limit)
```

permite também **submeter uma instrução de** `SELECT` à base de dados, mas **especificando cada parte da sintaxe em *Strings* diferentes**. No exemplo apresentado antes, são pedidos todos os registos de alunos ordenados pela média das classificações (`avg`), da maior para a menor. A cláusula de ordenação é definida no penúltimo parâmetro (`avg DESC`). Realça-se a importância de **definir os vários nomes dados às tabelas e colunas estaticamente** na implementação da classe `SQLiteOpenHelper`, já que isso **aumenta a escalabilidade do código e evita erros**, facilitando também a construção das *queries*. Para **consultas mais elaboradas**, pode fazer-se uso da **classe** `SQLiteQueryBuilder`.

6 Cursores

Cada método de consulta à bases de dados **devolve um objeto da classe** `Cursor` (`android.database.Cursor`)[6]. Esta classe vem **resolver aquele que é conhecido como o *problema da impedância***, que se **refere ao facto dos dados devolvidos por uma linguagem como a SQL terem tamanhos que não podem ser estimados à partida**, contrariamente ao que acontece em linguagens procedimentais ou orientadas por objetos. *E.g.*, não se sabe, à partida, quantas linhas a maior parte das *queries* SQL devolvem.

Num determinado momento, e se devidamente inicializado, **um objeto** `cursor` **aponta para uma linha do conjunto de dados devolvido**, **disponibilizando um conjunto de métodos de conveniência para navegar** nesse conjunto de dados. Por exemplo, contém os métodos `getCount()` e `getPosition()`, para saber o número de linhas do conjunto de dados e a linha para que aponta atualmente; ou os métodos `moveToFirst()`, `moveToLast()` ou `moveToNext()` para **navegar para o início, para o fim ou para a próxima linha** do conjunto de dados a que se refere, respetivamente.

O objeto `cursor` facilita também **os métodos que permitem obter os valores de tipos primitivos** (neste caso Java) **para cada uma das colunas** da linha para onde aponta. Estes métodos **assumem normalmente a forma** `get[type](int)`. Por exemplo, `getInt(int)` ou `getString(int)` devolvem o inteiro ou a *String* na coluna cujo índice é dado como parâmetro ao método, respetivamente. **Caso não se saiba o índice da coluna** a que se quer aceder, mas **o seu nome seja conhecido, pode recorrer-se ao método** `getColumnIndex(String)`, cujo primeiro parâmetro define precisamente esse nome, devolvendo -1 em caso de erro, ou o índice (começando em 0) da coluna pretendida.

[6] Ver http://developer.android.com/reference/android/database/Cursor.html.

7 Depuração de Bases de Dados

A **depuração de aplicações móveis** que criem ou manipulem ficheiros ou bases de dados no sistema **passa** necessariamente **por verificar se esses ficheiros existem ou contêm os dados necessários e no formato certo**. No caso do Android, **a ferramenta *Android Monitor***, ou mais especificamente a funcionalidade de **gestor de ficheiros que incorpora**, pode ser **usada para verificar a existência de ficheiros nas diretorias do sistema**. Ver o conteúdo de ficheiros normais também será relativamente simples. Contudo, a consulta ou manipulação do conteúdo de uma base de dados, bem como do seu esquema irá requerer uma ferramenta dedicada para esse efeito.

Existem **aplicações para abrir e navegar em bases de dados SQLite** com interface gráfica e **em modo linha de comandos**, sendo que a última pode revelar-se **interessante por permitir, por exemplo, que se aceda a uma base de dados através da *shell*** fornecida pelo `adb` na plataforma Android. De resto, **o SDK Android já inclui a ferramenta** de linha de comandos `sqlite3` (na diretoria `tools`) e o **macOS X também a disponibiliza de fábrica**. A emissão do comando `sqlite3 nome-ficheiro-bd` conduz o utilizador a uma *shell* para manipulação da base de dados.

Enquanto que, **para o iOS, a depuração de uma base de dados requer que o ficheiro respetivo seja copiado do dispositivo móvel real para o computador**[7], **no sistema Android pode-se abrir a base de dados diretamente no dispositivo** (virtual ou real), **desde que se possuam permissões de acesso** à mesma. O conjunto de passos e *outputs* que concretizam o procedimento a tomar neste caso pode ser estruturado da seguinte forma:

1. Começa-se por **obter informação acerca dos dispositivos móveis** (virtuais ou reais) reconhecidos pelo `adb` com o comando `$ adb devices`. O comando anterior deve produzir um *output* parecido com o que se inclui em baixo

   ```
   emu1-5554
   emu2-5555
   ...
   ```

2. Sabendo o nome do dispositivo alvo, **consegue-se acesso através de uma *Bourne shell*** com o comando `$ adb -s emu1-5554 shell`.
   Note que, caso só exista um dispositivo alvo, o excerto `-s emu1-5554` é desnecessário, já que esta parte apenas define qual o dispositivo para o qual se vai estabelecer a ponte (*bridge*).

3. Finalmente, **executa-se a ferramenta `sqlite3` para acesso e manipulação à base de dados**. O **nome** da ferramenta deve ser **seguido do nome e caminho**

[7] Caso esteja a utilizador um simulador, o ficheiro da base de dados já está disponível na árvore do sistema de ficheiros desse simulador no disco local.

completo da base de dados no sistema, caso se esteja a executar da raiz do sistema de ficheiros. O comando será semelhante ao seguinte:
`$ sqlite3 /data/data/pt.di.ubi.pmd.exstorage2/databases/exdatabase.db`.
Alternativamente, pode navegar até à diretoria que contém a base de dados e simplesmente emitir o comando `$ sqlite3 nome-basedados.db`. Ao entrar na *shell* assim despoletada, deve ser mostrado um `output` parecido com o seguinte

```
SQLite version 3.6.20
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

e a *prompt* será semelhante a `$ sqlite>`

Apesar do seu aspeto modesto, **a ferramenta disponibiliza todas as funcionalidades que permitem a total consulta e manuseamento da informação na base de dados e do seu esquema**. Para além de ser possível a **introdução direta de** todas as **instruções SQL** que interpreta (seguidas de um ponto e vírgula (`;`)), ainda **providencia um conjunto de comandos específicos** e úteis. Estes comandos são normalmente **precedidos por um ponto (`.`) e podem ser listados escrevendo** `.help`. Alguns dos mais interessantes são brevemente descritos na tabela seguinte:

`.backup`	Permite **fazer uma cópia de segurança** de uma base de dados para determinado ficheiro (**se** o ficheiro contiver apenas uma base de dados, **corresponde a fazer uma cópia desse ficheiro**);
`.dump`	**Produz um *script* SQL** com todas as instruções que definem determinada base de dados, bem como o seu conteúdo;
`.load`	Que **carrega o conteúdo de um ficheiro** especificado como parâmetro, e contendo dados devidamente separados por um caracter bem definido, **para uma tabela à escolha**;
`.restore`	**Recupera uma base de dados** a partir de uma cópia construída, *e.g.*, a partir de `.backup`;
`.schema`	**Mostra o conjunto de comandos** `CREATE TABLE` e seus homónimos **que**, no fundo, **definem o esquema** da base de dados;
`.tables`	**Lista o nome de todas as tabelas** da base de dados;
`.exit`	Para **sair** do SQLite.

Note que alguns dos comandos descritos antes aceitam parâmetros de entrada que não foram especificados na tabela, embora alguns tenham sido referidos na descrição.



12 A Componente Serviço

Sumário

Execução de tarefas e programas sem interface de utilizador em dispositivos móveis. Notificações, mensagens flutuantes não intrusivas e intentos pendentes. Comunicações entre processos através de vínculos entre *threads* e de mensagens para o sistema.

Summary

Execution of tasks and programs with no user interface in mobile devices. System notifications, non-intrusive floating messages and pendind intents. Inter-processes communication via binding between threads and system messages.

1 Introdução

Anteriormente, foram discriminados os **4 componentes** que podem ser usados como **blocos de construção de uma aplicação Android**, embora ainda só se tenha discutido com detalhe a **componente *Atividade***. Esta componente é a que **lida com interfaces de utilizador**, sendo por isso **central à maior parte das aplicações** Android. As restantes componentes servem diferentes propósitos, e este capítulo foca-se, sobretudo, na componente *Serviço*.

Note que, atualmente, é comum ao utilizador de um computador **iniciar várias tarefas ao mesmo tempo**, ou **iniciar determinada tarefa e deixá-la a executar em segundo plano** enquanto este se foca noutra atividade. Por exemplo, é comum continuar a navegar na *web* enquanto se descarrega um ficheiro, ou então ler um documento enquanto se ouve uma música. Algumas vezes, inicia-se uma aplicação apenas para colocar determinada ação a correr em segundo plano, atirando essa aplicação para segundo plano logo de seguida. Se recordar a forma de operar das ***atividades*, bem como o seu ciclo**

de vida, irá concluir que este tipo de funcionalidade não pode ser conseguida com essa componente.

As **aplicações Android correm numa máquina virtual Java**. Assim, determinada instância de execução **corresponde apenas a um processo, onde primariamente corre a *main thread*** (por vezes também designada por *user interface thread* **ou *UI thread***). É óbvio que o modelo de execução em que é **apenas usada uma única *thread* não pode responder a todos os cenários de utilização de uma aplicação móvel**. O uso de uma só *thread* **significa que qualquer conjunto de operações é executado de uma forma sequencial**, pelo que **a presença de uma operação lenta irá incorrer numa situação em que a aplicação deixa de responder**, o que é **inadmissível em termos de *user experience* (UX)**, também por construção, em sistemas atuais. Por exemplo, o Android define **o tempo máximo de reação das suas aplicações para os 5 segundos**. Depois disso, o sistema assume controlo, mostrando a **mensagem *Application not Responding* (ANR)** ao utilizador, oferecendo-lhe a hipótese de terminar a aplicação.

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

1. A **classe** `Threads` está disponível em `java.lang.Thread` e pode ser usada para **processamento assíncrono**. Também é possível usar *software* do **pacote** `java.util.concurrent` **para executar tarefas em segundo plano** (*e.g.*, usando classes como `ThreadPools` ou `Executor`). Contudo, conforme já sugerido em cima, **é preciso ter em conta que as *threads* criadas desta forma não podem atualizar a interface de utilizador diretamente**, visto estarem a correr isoladamente da `thread` principal. Para se conseguir esse efeito, **é preciso garantir que se comunica à *thread* principal quais as alterações a fazer**. As classes `Handler` e `AsyncTasks`, que não são discutidas com detalhe aqui, facilitam essa comunicação;

2. **A classe** `Handler` (`android.os.handler`)[1] **permite definir um manípulo na *thread* principal, para o qual se podem enviar mensagens ou código para ser executado** (*e.g.*, via método `post(Runnable)`). Neste caso, **declara-se a tarefa que se quer executar** de forma assíncrona **dentro de uma nova *thread***, e todas as **operações de atualização da interface de utilizador são** definidas dentro de uma classe `Runnable`, **enviadas para serem realizadas pela *thread* principal** através do método `post(Runnable)`. Pode optar-se também por se definir uma mensagem (classe `Message`) a ser tratada na *thread* principal. Neste caso, é necessário reescrever o método `handleMessage(Message)` na *thread* principal, e colocar nele o código que atualiza a interface de utilizador;

[1] Ver http://developer.android.com/reference/android/os/Handler.html.

3. **A classe** `AsyncTask` (`android.os.AsyncTask`)[2] pode ser **usada para criar *threads*** que facilmente comunicam com a *UI thread*, já que **define 4 métodos que podem ser reescritos**, sendo que **alguns correm na *thread* em segundo plano** (*e.g.*, `doInBackground(.)`), **enquanto que outros** (*e.g.*, `onProgressUpdate(.)` ou `onPostExecute(.)`) **correm na *thread* onde o objeto é criado**, que normalmente corresponde à *thread* principal;

4. Finalmente, **a classe** `Service` (`android.app.Service`)[3], definida a seguir.

2 Definição de Serviço

Dado os seus propósitos, o serviço é uma das componentes mais importantes da plataforma.

Em Android, um serviço (`Service`) não é mais do que **uma forma de anunciar o desejo de uma aplicação Android executar uma operação demorada sem interação com o utilizador**, ou então definir uma **forma de fornecer funcionalidades a outras aplicações**, que se podem vincular ao serviço para obter essas funcionalidades.

Convém endereçar algumas confusões que se geram quando se mencionam serviços, nomeadamente que:

- Um serviço **não é um processo separado** e que;
- Um serviço **não é, nem cria, uma *thread*** (separada).

Isto significa que, **a não ser que seja estritamente definido em contrário, um serviço corre no mesmo processo da aplicação**. Também significa que, **caso se queira fazer trabalho demorado** dentro de um serviço, **uma nova *thread*** para lidar com esse trabalho **deve ser explicitamente declarada pelo programador**.

Uma das **características que melhor distingue um serviço** de uma atividade é o facto de **não ter uma interface de utilizador**. Os exemplos mais comuns dados no contexto dos serviços referem-se às **funcionalidades de *download* de ficheiros** ou de **ouvir música**. Para obter uma ideia prática da utilidade dos serviços, considere que um utilizador iniciava o *download* de um ficheiro a partir de uma atividade, e que era a própria atividade que lidava com esse *download* até terminar. Neste caso, e assim que o utilizador saísse da atividade (*e.g.*, para ir para outra aplicação ou para o *home screen*), o *download* era interrompido, porque a atividade era suspensa (`onPause()` → `onStop()`).

[2] Ver http://developer.android.com/reference/android/os/AsyncTask.html.
[3] Ver http://developer.android.com/reference/android/app/Service.html.

Por outro lado, o *download* do ficheiro **não seria interrompido caso as instruções relacionadas com o *download* fossem colocadas num serviço**, ainda que este execute na *thread* principal, porque **o Android deixa o processo da aplicação a correr em segundo plano**.

Enquanto que as *threads* criam módulos de execução separados dentro do mesmo processo, os serviços declaram ao sistema que determinada tarefa deve continuar, ainda que outros processos/aplicações/atividades sejam trazidos para primeiro plano. Os serviços permitem que outros componentes se vinculem a estes para obter funcionalidades e são por vezes usados para *InterProcess Communication* (IPC).

Existem basicamente **dois tipos de serviços**:

1. Serviços **sem vínculo** ou *started*, que são **colocados em execução por outro componente (como uma atividade) através do método** `startService()`. Uma vez iniciados, os serviços sem vínculo **podem correr indefinidamente em segundo plano, mesmo que o componente que os colocou em execução seja destruído. Normalmente**, os serviços deste tipo **fazem apenas uma operação e não devolvem resultados** para a componente que os invocou (*e.g.*, fazem a atualização de uma base de dados). Adicionalmente, depois de cumprirem o seu destino, **o serviço deve auto terminar-se**;

2. Serviços **com vínculo** ou *bound*, que são **colocados em execução ou é criado o vínculo através do método** `bindService()`. Este tipo de serviços providenciam **formas de definir uma interface que permite que outros componentes da mesma aplicação, ou de aplicações diferentes, interajam com o serviço** numa arquitetura cliente-servidor, **enviando-lhe pedidos e obtendo resultados**. É possível que **várias componentes se liguem simultaneamente** a um serviço e este **só sobrevive enquanto tiver componentes vinculadas** (*i.e.*, após perder o último vínculo, o serviço é destruído).

Note que **um mesmo Serviço pode funcionar em modo *started* ou *bound*, dependendo de como é invocado** e dos métodos da sua superclasse que são implementados. Por exemplo, **um Serviço que funcione em modo *started* deve implementar o método** `onStartCommand(.,.,.)`**, enquanto que um Serviço que funcione em modo** `bound` **deve implementar o método** `onBind()` (ver em baixo), mas **não há nada que impeça que ambas sejam implementadas**, para que o serviço possa ser despoletado das duas formas.

3 Ciclo de Vida de um Serviço

A figura 12.1, retirada diretamente da documentação oficial da plataforma Android[4], ilustra os **dois ciclos de vida que uma componente serviço pode percorrer, dependendo do seu tipo** (`started` ou `bound`). Como se pode concluir da observação da figura, **ambos os ciclos de vida invocam os métodos `onCreate()` e `onDestroy()`**, cujo conteúdo é executado quando o serviço é colocado em execução pela primeira vez ou imediatamente antes de ser destruído, respetivamente. Contudo, o método `onCreate()` não recebe, para os serviços, nenhum `Bundle`, como de resto acontece para Atividades.

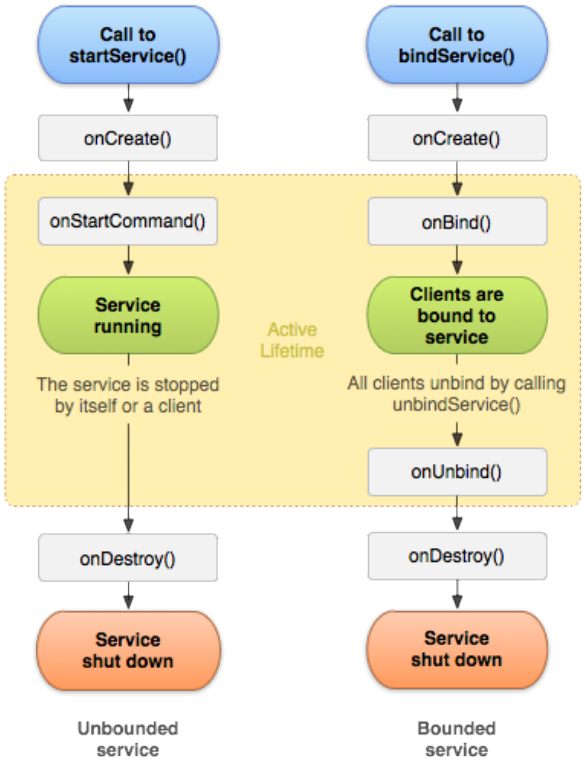


Figura 12.1: Os ciclos de vida de um serviço Android. À esquerda é mostrado o ciclo de vida para serviços criados com `startService()`, enquanto que à direita está o ciclo de vida para o caso em que é criado com `bindService()`.

De modo a deixar claro quais **os parâmetros e retornos** de cada um dos métodos esquematizados na figura, incluem-se, em baixo, **os protótipos** de cada um deles. Enfatiza-se que **não é necessário reescrever todos os métodos** durante a implementação, e que alguns deles **não são**, inclusive, **necessários, dependendo do tipo de Serviço** (*e.g.*, o método `onBind(.)` nunca é invocado para um serviço que funcione sempre sem vínculo).

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

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

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

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

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

[4]Ver http://developer.android.com/guide/components/services.html.

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

O **ciclo de vida** de um Serviço **começa pela invocação de `onCreate()`, onde deve ser feita a configuração inicial da instância do componente**, e pode terminar com a invocação de `onDestroy()`. Por exemplo, caso queira colocar música a correr em segundo plano, é no `onCreate()` que deve colocar o código que declara e configura a *thread* secundária. **O período de atividade** do ciclo de vida de um Serviço **começa com a invocação de um dos métodos `onStartCommand(.,.,.)` ou `onBind()`. A cada um destes métodos é automaticamente entregue o intento que invocou o Serviço** (como não pode deixar de ser, e sendo um Serviço um componente aplicacional Android, este é invocado através de um intento). Contrariamente aos Serviços sem vínculo, **os serviços com vínculo só terminam depois do método `onUnbind(.)` retornar**.

4 Criar um Serviço sem Vínculo

Para efeitos de exemplificação e demonstração da forma de criação de um **Serviço sem vínculo**, considere que queria criar uma aplicação cuja única finalidade era mostrar 10 avisos no ecrã em intervalos de 2 em 2 segundos. Os avisos devem ter o texto `Warning Number 1`, `Warning Number 2`, ..., `Warning Number 10` e ser exibidos mesmo que a atividade principal da aplicação saia de foco, *i.e.*, mesmo que o utilizador passe a utilizar outra aplicação. Para o exemplo seguinte, e para conseguir o objetivo de mostrar as mensagens, faz-se uso de *toasts* (ver adiante). Note que, conforme descrito, o cenário parece ideal para um Serviço. Assuma ainda que o Serviço só é despoletado depois de um botão associado a `onButtonClick()` ser clicado. Aparte alguns detalhes, o código que implementa a atividade principal terá uma implementação semelhante a:

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

import android.app.Activity;
...

public class FloatingAlarms extends Activity {

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
  }
  public void onButtonClick(View v){
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);
  }
}
```

Enquanto que o respetivo serviço poderia ser implementado como se mostra a seguir:

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

import android.app.Service;
...

public class ServiceAlarms extends Service {

  @Override
  public int onStartCommand
  (Intent intent, int flags, int startId) {
    for(int i=1; i<11; i++){
      Toast.makeText
      (this, "Warning #" + i, Toast.LENGTH_SHORT)
      .show();
      Thread.sleep(3000);
    }
    // Se o processo for morto, nao voltar a
    // tentar reinicia-lo (START_NOT_STICKY)
    return START_NOT_STICKY;
  }
}
```

Como se pode ver pelo exemplo, **criar um Serviço em Android envolve importar e estender a classe `Service`, contida no pacote `android.app.Service`** (envolve também declarar o Serviço no manifesto, conforme discutido em baixo). **Pode** então **ser necessário reescrever alguns dos métodos da superclasse**, nomeadamente o `onCreate()`, `onStartCommand()`[5] e `onDestroy()`. Um dos **métodos mais importantes do ciclo de vida é o `onStartCommand()`, onde basicamente devem ser tomadas as providências necessárias para que as tarefas que definem o Serviço sejam desempenhadas**. No exemplo anterior, é neste método que se coloca um ciclo `for` que itera 10 vezes, de 3 em 3 segundos, mostrando uma mensagem em cada uma dessas iterações.

O serviço é começado por outra componente através de um intento. Neste caso, o **intento é explícito** porque na sua instanciação é indicado claramente qual é o seu componente destino, conforme se mostra a seguir:

```
Intent oIntent = new Intent(this, ServiceAlarms.class);
```

Neste caso, **o intento é passado ao método `startService(intent)`**. Após ser despoletado, o Serviço **passa automaticamente pelo método `onCreate()`** (não implementado no exemplo) **e depois para o método `onStartCommand(Intent,int,int)`**.

[5]Em APIs mais antigas (de nível inferior a 5), o método utilizado chamava-se `onStart()`, pelo que pode também implementar este método para efeitos de compatibilidade.

O método mencionado em último **recebe o intento** enviado pela componente que o invocou.

Para que possa **ser visível para o sistema**, o Serviço deve **ser declarado no `AndroidManifest.xml` usando uma *tag* `<service>`** que pode ter vários atributos e sub-elementos, **inclusive o nome (`android:name`) e alguns filtros de intentos** (usando a *tag* `<intent-filter>`). A seguir incluí-se o excerto que declararia o serviço mencionado antes no manifesto da respetiva aplicação:

```
<service android:name="ServiceAlarms">
 <intent-filter>
 <action android:name="pt.ubi.di.pmd.ServiceAlarms.SERVICE" ></action>
 </intent-filter>
</service>
```

Recorde que **os filtros de intentos são usados pelo sistema para encontrar componentes que são capazes de lidar com determinada ação**. No fundo, estes filtros definem capacidades desses componentes, **podendo ser definido mais do que um filtro para cada componente**. Para o exemplo nesta secção, e dado o seu manifesto, pode-se dizer que **o sistema também permitia despoletar este Serviço via um intento implícito** que definisse a ação `pt.ubi.di.pmd.ServiceAlarms.SERVICE`, *i.e.*, via

```
Intent service = new Intent("pt.ubi.di.pmd.ServiceAlarms.SERVICE");
startService(service);
```

Relembre também que os filtros de intentos **permitem especificar, com mais granularidade, quais os intentos que são entregues a determinado componente**. Contudo, **estes filtros apenas só se aplicam a intentos implícitos**, já que os intentos explícitos são sempre entregues ao componente destino.

O método **`onStartCommand(.,.,.)` termina com um retorno** (no exemplo, é devolvido o valor `START_NOT_STICKY`). Há **3 retornos importantes** para este método:

1. O **valor `START_NOT_STICKY` define que caso o serviço seja morto pelo sistema enquanto está no estado ativo, este não deve voltar a tentar criá-lo se puder**. É útil para Serviços que são recomeçados por tarefas agendadas e que mesmo que sejam mortos por falta de memória, serão eventualmente recomeçados quando o agendamento se der;
2. O valor `START_STICKY` **define que o sistema deve tentar a recomeçar um serviço que matou logo que tenha recursos para isso**. Neste caso, há que ter em atenção que **o método `onStartCommand(.,.,.)` é invocado de novo, mas sem o intento** que originalmente o chamou (o parâmetro do intento vai a *null*). Isto significa que é preciso lidar com este facto no próprio código, caso o intento seja preciso para o seu bom funcionamento;

3. O valor `START_REDELIVER_INTENT` é parecido ao anterior, mas **indica ao sistema que este deve guardar o intento** que criou determinado Serviço antes de o matar, **para que possa mais tarde ser utilizado para o recriar** com esse intento.

O exemplo contido no início desta secção **não é**, na realidade, **um bom exemplo** de como se deve criar ou implementar um Serviço. Há pelo menos dois detalhes que justificam esta afirmação:

1. Como o serviço corre na *thread* principal, **colocar a *thread* a dormir corresponde a colocar a interface do utilizador num modo adormecido e que não responde**. Caso o número de segundos seja aumentado de 3 para 5 ou superior, será exibida uma mensagem ANR;
2. **Quando se cria um serviço sem vínculo deve sempre garantir-se que este termina quando o trabalho que tem para fazer também termina**.

O excerto de código seguinte endereça ambos os problemas mencionados antes. Neste exemplo, é **criada uma *thread* que irá executar a tarefa demorada** definida como um objeto da classe *Runnable*. É também **invocado o método `stopSelf()` depois do trabalho estar feito, efetivamente terminando o serviço**.

```
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
      public void run() {
        try {

          for(int i=1; i<11; i++){
            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 reinicia-lo (START_NOT_STICKY)
    return START_NOT_STICKY;
  }
}
```

Recorde ou fique a saber que, **ao ser chamado o método `start()` da classe `Thread` é automaticamente executado o método `run()`**, definido para uma instância da classe `Runnable`. **O método `run()` é executado numa nova *thread***, conforme sugerido por `new Thread(oTask).start();`, enquanto que **o Serviço continua a correr na `thread` principal**.

Note que o código anterior ainda **contém um erro crasso** (serve de lição). A `toast` está a ser invocada dentro de uma *thread* diferente da principal (que é a que trata da interface gráfica), pelo que **não irá ser exibida**.

5 A Classe `IntentService`

No final da secção anterior ficou claro que **é muito frequente ser necessário definir uma nova *thread*** para lidar com tarefas longas que correm em segundo plano. Devido a esse facto, **a plataforma Android disponibiliza a classe chamada `IntentService`** no pacote `android.app`, que **cria**, de forma transparente para o programador, **uma *thread* separada da principal, que executa todos os intentos entregues em `onStartCommand()`**. Na verdade, uma instância desta classe **cria uma pilha de trabalho que permite lidar com vários intentos** que sejam enviados para a componente, de forma sequencial. **É também invocado, por defeito** e automaticamente, **o método `stopSelf()` assim que todas as tarefas tenham sido concluídas**, evitando o problema de este poder ser esquecido. Adicionalmente, **fornece uma implementação por defeito dos métodos `onBind()`** (ver em baixo), para o qual devolve sempre *null*, **e `onStartCommand()`, no qual é enviado o intento para a pilha de trabalho e depois para um método `onHandleIntent()`**.

```
public class ServiceAlarms extends IntentService {
  // E necessario implementar um construtor simples,
  // invocando o super IntentService(String), de
  // forma a dar um nome a thread criada.
  public ServiceAlarms() {
      super("ServiceAlarms");
  }

  @Override
```

```
  protected void onHandleIntent(Intent intent) {
    for(int i=1; i<11; i++){
      Toast.makeText(this, "Warning #" + i, Toast.LENGTH_SHORT).show();
      Thread.sleep(3000);
    }
  }
}
```

Para utilizar esta classe, **um programador apenas tem de reescrever o método `onHandleIntent()` e implementar o construtor que simplesmente invoca o seu análogo da superclasse, fornecendo-lhe um nome para a *thread* que é criada**. O excerto de código anterior exemplifica o que foi dito nesta secção, constituindo uma alternativa ao código que foi mencionado em último na secção anterior.

6 Criar um Serviço com Vínculo

Para muitos casos, é **útil criar um vínculo entre a componente que invoca o Serviço e este último**. Por exemplo, **uma aplicação de música** pode oferecer uma interface de utilizador para controlar se a música está a tocar ou pausada, e colocar um Serviço em segundo plano a tocar essa música, para que o utilizador possa continuar a usar o dispositivo móvel entretanto. Neste caso, a atividade que oferece a interface de utilizador precisa **pedir ao Serviço que pare ou coloque determinada música a tocar**, dependendo dos *inputs* do utilizador. A forma de conseguir esse objetivo é através da **criação de um vínculo que funciona no modelo cliente-servidor, sendo que o Serviço toma o papel de servidor** e a componente que o invoca toma o papel do cliente.

Há **três formas básicas de definir um vínculo** (*binder*) entre as componentes:

1. **Estender a classe `Binder` (`android.os.Binder`), aplicável em situações em que o Serviço é apenas usado pela própria aplicação e corre no mesmo processo do cliente**, o que acontece com frequência. Esta opção **não é aplicável** no caso em que o Serviço é implementado para ser **usado por várias aplicações**. Em baixo elabora-se um pouco mais nesta opção;

2. **Combinar objetos das classes `Messenger` (`android.os.Messenger`) e `Handler` (`android.os.Handler`), em que um manípulo é criado num Serviço, e um conjunto de mensagens a tratar é definido programaticamente no seu código**. As mensagens são enviadas para o serviço através do `Messenger`, **sendo também possível ao cliente criar um destes objetos e enviá-lo via uma mensagem ao Serviço, para que este lhe possa enviar retorno**. Esta é a **forma mais simples de criar IPC**, já que as mensagens enviadas pelos objetos da classe `Messenger` são todas colocadas numa única fila e tratadas numa só *thread*, **evitando problemas de sincronização**;

3. Finalmente, é possível **usar *Android Interface Definition Language* (AIDL) para definir interfaces que determinado Serviço quer expor a clientes**. Estas interfaces estão disponíveis para aplicações diferentes daquela em que a componente é definida. A definição é feita **em ficheiros `.aidl` que são colocados nas diretorias `src/` da aplicação que disponibiliza o Serviço e de todas as que o usam**. Aquando da compilação, as ferramentas do SDK criam uma interface `IBinder` com base no conteúdo do ficheiro `.aidl` e guardam o resultado em `gen/`. As interfaces podem depois ser implementadas no Serviço respetivo e invocadas nos clientes, com instruções semelhantes às seguintes:

```
IRemoteService oRS;
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder
        service) {
        oRS = IRemoteService.Stub.asInterface(service);
    }
```

   Se se optar por esta possibilidade, **é preciso ter em consideração que não é garantido**, por defeito, **que as chamadas aos vários métodos da interface sejam feitas na *thread* principal, pelo que devem ser tomadas precauções relativas a *multi-threading***. O serviço deve ser implementado de forma a ser `thread-safe`[6].

A explicação subsequente irá apenas elaborar na primeira das três formas de criar um vínculo entre um Serviço e outra componente aplicacional Android. Recorde que, para esta opção, o vínculo é interno, *i.e.*, entre componentes da mesma aplicação e que correm no mesmo processo. As condições gerais que permitem a criação do vínculo são os seguintes:

1. **No Serviço, há que instanciar um objeto da classe `Binder`** que:
   - Contém métodos públicos que o cliente poderá chamar; ou
   - Devolve uma instância de outra classe, cuja definição está contida na do Serviço e que contém métodos públicos que o cliente pode invocar; ou
   - **Devolve a própria instância do Serviço atual, permitindo que o cliente invoque os seus métodos públicos**;
2. Ainda no Serviço, há que **devolver a instância da classe `Binder`** no método *callback* `onBind()`;
3. **No cliente, há que receber o `Binder` da função *callback* `onServiceConnected()`, bem como instanciar um objeto local da mesma classe do objeto instanciado no `Binder`, e tirar partido dos métodos públicos** por ele fornecidos.

[6]Ser `thread-safe` significa que a execução concorrente de várias operações que acedem aos mesmos recursos não incorrem em estados de inconsistência no estado da aplicação.

O exemplo seguinte mostra como é que esta forma de criar um vínculo pode funcionar na prática. Começa-se por mostrar a implementação de um Serviço simples chamado `ToastasMistas`, cujo único objetivo é lançar mensagens flutuantes com a expressão `Toastas!`, sempre que o método `dizToastas()` é invocado:

```
...
public class ToastasMistas extends Service {
  private final BinderLocal oBinder = new BinderLocal();

  // Redefinicao da classe Binder
  public class BinderLocal extends Binder {
    ToastasMistas getService() {
      // Devolve a classe do proprio servico
      return ToastasMistas.this;
    }
  }

  @Override
  public Binder onBind(Intent intent) {
    return oBinder;
  }

  // Metodo publico que pode
  // ser usado por um cliente.
  public int dizToastas() {
    Toast.makeText(this, "Toastas!",
      Toast.LENGTH_SHORT).show();
  }
}
```

Note que, tal como mencionado em cima, é estendida a classe `Binder` e tomada a opção de devolver a instância do Serviço em si através da instrução `return ToastasMistas.this;`. Isto significa que o método `dizToastas()` será o que estará disponível na componente que receber o `Binder`.

De seguida mostra-se a implementação da atividade principal, que cria o *layout* da aplicação e que despoleta o Serviço referido antes através do método `bindService(intent, mConnection,int)`:

```
...
public class FloatingToastas extends Activity {
  ToastasMistas oTM;
  boolean bVinculo = false;

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

  @Override
  protected void onStart() {
```

```
    super.onStart();
    // Fazer o vinculo ao Servico ToastasMistas.
    Intent oIntent = new Intent(this, ToastasMistas.class);
    bindService(oIntent, mConnection, Context.BIND_AUTO_CREATE);
  }

  @Override
  protected void onStop() {
    super.onStop();
    // Desfazer o vinculo do Servico.
    if (bVinculo) {
      unbindService(mConnection);
      bVinculo = false;
    }
  }

  // Quando o botao e clicado, o Servico mostra
  // uma mensagem no ecra.
  public void onButtonClick(View v) {
    if (bVinculo) {
      // Pode-se chamar um metodo publico
      // diretamente do Servico.
      oTM.dizToastas();
    }
  }

  // A classe seguinte define os metodos
  // callback que sao passados para bindService().
  private ServiceConnection mConnection =
  new ServiceConnection() {
    @Override
    public void onServiceConnected(
      ComponentName className, IBinder service) {
      // Fazer o cast do binder para
      // ToastasMistas.BinderLocal.
      BinderLocal binder =
            (ToastasMistas.BinderLocal) service;
      oTM = binder.getService();
      bVinculo = true;
    }

    @Override
    public void onServiceDisconnected
                (ComponentName arg0) {
      bVinculo = false;
    }
  };
}
```

Há bastantes detalhes a notar no exemplo anterior. Por exemplo, que **o vínculo com o Serviço é criado apenas em `onStart()` e não em `onCreate()`**. Por outro lado, **o vínculo é destruído em `onStop()`**. Esta forma de proceder **permite que o vín-**

culo (e, neste caso, também o Serviço) **só exista quando a atividade está em execução, libertando recursos quando esta está em segundo plano**. Também é notável que **o despoletar do Serviço** ou a criação do vínculo (se aquele já existir) se faça, como não podia deixar de ser, **através de um intento**. Ao ser invocado o método `bindService(.,.,.)`, são chamados no Serviço os métodos `onCreate()` seguidos de `onBind()`, sendo que este último devolve um `Binder`. Este `Binder` é tratado no **método** `onServiceConnected(.,.)` **do cliente, que é automaticamente despoletado após** `bindService(.,.,.)` **retornar**. Por fim, note que é o facto do Serviço e do cliente estarem na mesma aplicação e processo que permite que o *cast* no método `onServiceConnected` seja possível.

A criação do vínculo é assíncrona. Neste caso, o cliente que invoca `bindService(.)` tem de implementar uma instância da classe `ServiceConnection` e passá-la como parâmetro àquele método. **O método** `bindService(.)` **retorna imediatamente, e o** `IBinder` **só é devolvido**, na realidade, **quando o sistema Android**, eventualmente mais tarde, **invocar o método *callback*** `onServiceConnected()`.

No cliente, e de uma forma resumida, **a criação do vínculo ao Serviço é conseguida através dos dois passos seguintes**:

1. **Instancia-se um objeto da classe** `ServiceConnection` **e reescrevem-se os métodos** `onServiceConnected(.,.)` **e** `onServiceDisconnected(.)`**.** O primeiro método é invocado pelo Android para entrega do `IBinder` devolvido pelo método `onBind()` do Serviço. O segundo método é invocado automaticamente pelo Android quando a ligação ao serviço cai inesperadamente, nomeadamente quando o Serviço parou ou foi destruído. Note que, ao contrário do que se possa pensar, **este método não é invocado quando o cliente usa o método** `unbindService()`;

2. Invoca-se o método `bindService(.,.)`, passando-lhe o intento, a implementação específica de `ServiceConnection` e o contexto do cliente.

Note que apenas as **atividades**, **Serviços** e **provedores de conteúdos** (discutidos adiante) é que se **podem vincular a um serviço**. **Um recetor de conteúdos não o pode fazer**. Nos métodos *callback*[7] `onServiceConnected(.)` deve ser colocada a lógica que, no mínimo, instancia a classe que contém os métodos públicos do Serviço, para que possam ser usados na componente atual.

O **terceiro parâmetro** do método `bindService(.,.,.)` **é uma *flag*** que especifica algumas opções para o vínculo e, portanto pode tomar vários valores, definidos também estaticamente na implementação da classe `Context`. No exemplo apresentado, é usada a ***flag*** `BIND_AUTO_CREATE`, que basicamente indica ao sistema que **o Serviço alvo deve ser despoletado caso não esteja a correr**. Outro exemplo inclui `BIND_IMPORTANT`,

[7] Recorde que os métodos *callback* são aqueles que são automaticamente executados pelo sistema convenientemente quando um método devolve algo.

que indica ao sistema que **o serviço para o qual está a ser criado o vínculo é muito importante** para o cliente e que, portanto, este deve ser **promovido para o mesmo nível de execução em primeiro plano em que o cliente está**.

7 Terminar um Serviço

Já aqui foi referida a importância de **terminar os Serviços quando estes não são necessários**. Cabe ao programador **ter essa tarefa em consideração**, mas com as seguintes **atenuantes**:

- Enquanto que os Serviços **sem vínculo devem ser explicitamente terminados**, quer **dentro do serviço**, após este terminar a tarefa para a qual tinha sido invocado (melhor solução), ou **por outra atividade** que a ele tenha acesso;
- Os serviços **com vínculo não precisam ser explicitamente terminados**, **desde que se garanta que é feita a desvinculação** quando já não são necessários. A razão para esta forma de operar deve-se ao facto **do sistema matar automaticamente (e mais facilmente) os serviços que não têm qualquer vínculo**.

Já foi visto que **um Serviço *started* pode ser terminado num ponto da sua própria execução recorrendo ao método** `stopSelf()`. O mesmo tipo de Serviços pode ser **terminado usando o método** `stopService(Intent)` **a partir de uma componente** que a ele tenha acesso (*e.g.*, a atividade que o despoletou). Como não podia deixar de ser, este método **aceita um intento** para identificar o Serviço a terminar, conforme se mostra no excerto de código seguinte:

```
Intent service = new Intent(this,ServiceAlarms.class);
stopService(service);
```

Para se **desvincular determinada componente de um Serviço recorre-se ao método** `unbindService(.)`, que **aceita o objeto da classe** `ServiceConnection` como parâmetro:

```
unbindService(mConnection);
```

Quando um componente **cliente é destruído pelo sistema, também é automaticamente desvinculado** de qualquer Serviço ao qual tenha sido vinculado. De qualquer forma, é boa prática proceder à desvinculação no código, de forma a **poupar recursos quando o Serviço não estiver a ser usado pela aplicação**. Normalmente, a **vinculação e desvinculação é feita em momentos simétricos do ciclo de vida** da componente cliente, nomeadamente (para a componente atividade):

- Se a interação com o serviço só precisa acontecer enquanto **uma atividade está visível**, deve fazer-se a vinculação e desvinculação **nos métodos `onStart()` e `onStop()`, respetivamente**;
- Caso se queira que a **atividade receba respostas do Serviço mesmo enquanto está parada em segundo plano**, então o Serviço pode ser **vinculado no método `onCreate()` e desvinculado em `onDestroy()`**.

Note que **não é muito eficiente fazer a vinculação ou desvinculação a um Serviço nos métodos `onPause()` e `onResume()`**, principalmente porque isto causaria a **invocação dos métodos de *callback* a cada transição**, inclusive durante a apresentação de pequenos diálogos em que se desfoca a interface atual, causando algum **impacto na *performance* do sistema**. É preciso também ter em conta que, quando uma atividade se desvincula de um Serviço, este pode ser destruído pelo sistema, se não estiver ligado a mais nenhum componente. Caso a próxima atividade (*e.g.*, da mesma aplicação) quiser aceder a esse mesmo Serviço, este tem de ser recriado, enquanto que se não tivesse sido destruído, este passo não era necessário novamente.



13 Notificações

Sumário
Notificações, mensagens flutuantes não intrusivas e intentos pendentes.

Summary
System notifications, non-intrusive floating messages and pendind intents.

1 Introdução

Dado um Serviço não possuir uma interface, **é comum usar mensagens conhecidas por *toasts* ou notificações na barra de status para informar o utilizador de determinadas operações** efetuadas por este componente. Enquanto que **uma *toast* aparece na superfície da perspetiva atual, desaparecendo pouco tempo depois, uma notificação na barra de *status* é caracterizada por um ícone que fica residente até que a barra seja expandida e a mensagem que lhe está associada seja lida**. Opcionalmente, o utilizador **pode ainda atuar sobre essa mensagem** através de uma ou mais ações que lhe são fornecidas na própria barra expandida (na gaveta). As **notificações** na barra de status são, portanto, **ideais para aquelas tarefas que correm em segundo plano mas que necessitam que utilizador aja quando terminam** (*e.g.*, quando é feito o *download* do ficheiro).

2 Notificações via Mensagens Flutuantes

Na gíria Android, uma ***toast* é um objeto interativo** (um *widget*) direcionado à **exibição de mensagens curtas ao utilizador**. Estas mensagens aparecem numa caixa

de diálogo flutuante durante um período de tempo, desaparecendo depois disso de uma forma suave. Estas mensagens **podem ser criadas de uma forma muito simples recorrendo à classe** `Toast` (`android.widget.Toast`)[1].

O trecho de código seguinte mostra como se pode despoletar uma mensagem do tipo discutido. No exemplo, o objeto `oToast` é **primeiro instanciado** (o construtor recebe o contexto da aplicação que a despoleta), **depois configurado com a mensagem** (através de `setText(CharSequence)`) **e com o tempo de exibição** (através de `setDuration(int)`), sendo por fim **dada a ordem de exibição** no ecrã por `oToast.show()`:

```
import android.widget.Toast;
...
Toast oToast = new Toast(this);
oToast.setDuration(Toast.LENGTH_LONG);// 3.5 seconds
oToast.setText("This is a toast message.");
oToast.show();
...
```

Note que o método `setDuration(int)` aceita um inteiro que, até à data, **apenas pode tomar um de dois valores possíveis**: `LENGTH_SHORT`, que equivale a uma exibição de **2 segundos**, e `LENGTH_LONG`, que equivale a uma exibição com uma duração de **3.5 segundos**. Estes dois valores estão **definidos estaticamente na classe** `Toast`, para evitar enganos, mas correspondem apenas aos valores 0 e 1, respetivamente.

Da leitura do código anterior, fica **claro que o ciclo de vida do objeto está confinado às operações de instanciação de um objeto, configuração e exibição da mensagem**. Assim, sugere-se o **uso de um dos dois métodos estáticos definidos na classe** `makeText(.,.,.)`**, que permitem combinar todas as operações numa única instrução**, conforme se mostra a seguir:

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

Note que ambos os métodos estáticos referidos **têm o mesmo nome, sendo que apenas o tipo de parâmetro intermédio muda nas assinaturas** e ambos devolvem um objeto da classe `Toast`. O método `makeText(context, CharSequence, int)` aceita o contexto da aplicação que a despoleta, bem como a mensagem a apresentar e o período, enquanto que `makeText(context, redID, int)` aceita um `ID` de um recurso, caso a mensagem a exibir esteja definida numa `strings.xml`, ao invés da própria *string*. Repare também no método `show()`, invocado no final da instrução. Sem esse método, a mensagem não é mostrada.

Se despoletada por um serviço, a *view* é mostrada sobre a interface de utili-

[1] http://developer.android.com/reference/android/widget/Toast.html

zador da aplicação atualmente em execução (seja a que lhe corresponde ou não). Contudo, **a mensagem nunca chega a receber foco**, pelo que o utilizador pode continuar a interagir com a aplicação. A ideia destas mensagens é precisamente **serem o menos intrusivas possível**, configurando o meio ideal para mostrar informação que pode ser útil (*e.g.*, para avisar que terminou o *download* de um ficheiro).

3 Notificações na Barra de Estado

Uma **notificação** na barra de status é um objeto **algo elaborado**, principalmente **porque permite interação e porque tem de necessariamente transmitir uma ideia ao utilizador da forma mais eficiente possível**. Para tornar a criação de notificações mais simples, **a plataforma fornece já outro objeto** (`NotificationCompat.Builder`) **para construir estas notificações**, sendo apenas necessário passar-lhe alguns elementos, antes de invocar o método `build()`, que debita já um objeto da classe pretendida. Os **elementos necessários** são os seguintes:

- O nome do recurso que aponta para **um ícone** (definido através de `setSmallIcon()`);
- Um título para a notificação (definido através de `setContentTitle()`);
- **Uma pequena descrição** (definido através de `setContentText()`).

O pequeno excerto de código seguinte mostra precisamente a criação de uma notificação simples recorrendo aos métodos e objetos referidos antes:

```
1  package pt.di.ubi.pmd.exservice;
2
3  import android.app.Service;
4  import android.app.NotificationManager;
5  ...
6
7  public class ServiceAlarms extends Service {
8    // A proxima variavel define um numero que
9    // identifica a notificacao e que pode ser
10   // usado para mais tarde a atualizar.
11   private int iID = 100;
12
13   @Override
14   public int onStartCommand
15   (Intent intent, int flags, int startId) {
16   ...
17   NotificationCompat.Builder oBuilder =
18     new NotificationCompat.Builder(this)
19     .setSmallIcon(R.drawable.icone_notificacao)
20     .setContentTitle("Floating Alarms")
21     .setContentText("O Servico dos alarmes terminou. Carregue aqui para o
           reiniciar!");
22
23   NotificationManager oNM = (NotificationManager)
24     getSystemService(Context.NOTIFICATION_SERVICE);
25   oNM.notify(iID, oBuilder.build());
26 }
```

Note que **o Java permite** que a invocação dos 3 métodos referidos antes seja feita de forma **muito compacta**. No exemplo anterior, mostra-se que, ao mesmo tempo que se cria o objeto da classe `Builder`, são **imediatamente chamados os 3 métodos de forma contígua**, sem pontuação a indicar o final de cada método (à exceção do último), delineando apenas o seu início com um ponto. Na verdade, **passamos a ter uma só instrução terminada com o último ponto e virgula**. As linhas 17, 18, 19, 20 e 21 são, por isso, equivalentes às seguintes:

```
NotificationCompat.Builder oBuilder =
  new NotificationCompat.Builder(this);
  oBuilder.setSmallIcon(R.drawable.icone_notificacao);
  oBuilder.setContentTitle("Floating Alarms");
  oBuilder.setContentText("O Servico dos alarmes terminou. Carregue aqui para o
      reiniciar!");
```

A **notificação é criada através do método `build()`, sendo imediatamente passada ao gestor de notificações do sistema** na última linha de código útil do `onStartCommand()` através do método `notify(.,.)`[2]. **A instância particular do gestor de notificações utilizada pelo sistema é obtida através da utilização do método do contexto `getSystemService(String)`, cujo parâmetro determina o serviço Android que se quer obter. Muitos dos gestores disponíveis no sistema são obtidos desta forma.**

[2] O primeiro parâmetro de `notify(.,.)` é um `int` que identifica a notificação e que pode ser usado para a cancelar remotamente.

Para além dos três elementos referidos antes, podem ser opcionalmente especificados outros. Na verdade, **é sempre uma boa ideia especificar pelo menos uma ação**[3] **para quando o utilizador clica na notificação** na gaveta onde são apresentadas. Repare que **o ideal será inclusive redirecionar o utilizador para uma atividade** onde possa interagir com a aplicação ou explorar melhor a mensagem (*e.g.*, quando recebe uma notificação de *novo e-mail*, clicar na mesma leva-o para a aplicação de *e-mail*). **Para se conseguir esse efeito, tem de se construir um objeto conhecido por *Intento Pendente*** (*i.e.*, um objeto da classe `PendingIntent`), conforme exemplifica o trecho de código seguinte:

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

import android.app.Service;
import android.app.PendingIntent;
import android.app.NotificationManager;
...

public class ServiceAlarms extends Service {
  private int iID = 100;
  private int iReqCode = 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!");

  // Intento explicito para uma determinada atividade.
  Intent iMain = new Intent(this, FloatingAlarms.class);
  PendingIntent iPImain = getActivity(this, iReqCode, iMain, PendingIntent.
      FLAG_ONE_SHOT)

  oBuilder.setContentIntent(iPImain);
  NotificationManager oNM =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

  oNM.notify(iID, oBuilder.build());
}
```

No código incluído antes ilustra-se a criação de um intento que redireciona para a atividade principal da aplicação. Este intento é explicito, **encapsulado num intento pendente através de `PendingIntent iPImain = getActivity(.,.,.,.)`, e colocado na notificação que vai ser lançada (ver `oBuilder.setContentIntent(iPImain);`).** Note que o intento toma **a qualificação de pendente porque simplesmente não é executado imediatamente, mas sim enviado para outra aplicação** (neste caso é enviado para o *NotificationManager*). Quando um utilizador carrega na mensagem

[3] Podem especificar-se mais do que uma ação para determinada notificação.

da notificação, este intento é despoletado. Repare que, **como o intento pendente é criado com o contexto da aplicação, quando este é despoletado por outro componente, o sistema executa-o como se viesse da aplicação original, com as suas permissões**.

Um dos problemas que ainda aqui não foi referido, mas que deve merecer a consideração do programador é o facto da **pilha de retrocesso de atividades poder ficar inconsistente pela inserção de um intento pendente numa notificação**. Para obter uma ideia do problema, considere, por exemplo, que recebia uma notificação de nova mensagem de *e-mail* enquanto navegava no *browser*. Ao clicar na notificação, era redirecionado para a atividade de visualização do *e-mail*. Caso não sejam tomadas medidas em contrário, a pilha de atividades passaria a conter a atividade do *e-mail* no topo, imediatamente precedida da do *browser*. O botão *back* do dispositivo móvel redirecionaria então o utilizador da atividade de visualização de *e-mail* para o *browser*, quando o ideal seria redirecioná-lo para a atividade principal da aplicação e depois para o ecrã *home* do sistema. **Para se resolver este problema, é necessário definir a hierarquia de atividades da aplicação no manifesto e dinamicamente especificar a pilha de retrocesso da atividade despoletada através de uma instância da classe** `TaskStackBuilder`. Este detalhe não é, contudo, discutido no âmbito das aulas.

4 Correr o Serviço em Primeiro Plano

Devido ao facto dos **Serviços correrem normalmente em segundo plano**, estes **concretizam tipicamente bons candidatos à destruição quando o sistema precisa de recursos** (*e.g.*, para mostrar uma página grande num *browser*). **Para se evitar** que determinado Serviço seja destruído tão facilmente em caso de necessidade, **pode-se definir um Serviço de primeiro plano**. Estes Serviços estão **normalmente associados a algo que o utilizador está ciente** (*e.g.*, ouvir música) e que este não quer que seja facilmente terminado pelo sistema.

Para colocar um Serviço em primeiro plano **recorre-se ao método** `startForeground(int, Notification)`**, que aceita dois parâmetros**:

1. **Um inteiro**, que identifica univocamente a notificação (tem significado local); e
2. **Uma notificação**, que fica residente na barra de estado até que o serviço seja terminado. Esta notificação é colocada numa secção *Ongoing* da gaveta de notificações, o que significa que a notificação não pode ser limpa enquanto o serviço não terminar ou seja removido de primeiro plano.

O método `startForeground(.,.)` **deve ser colocado na implementação do Serviço respetivo, exatamente no ponto a partir do qual se quer que este execute em primeiro plano**. Antes de este poder ser invocado, a notificação e (opcionalmente)

um intento pendente devem ser devidamente instanciados e configurados. O exemplo seguinte dá uma ideia de como usar estes recursos:

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

import android.app.Service;
import  android.app.PendingIntent;
import android.app.NotificationManager;
...

public class ServiceAlarms extends Service {
  static final int iID = 1;

  @Override
  public int onStartCommand
  (Intent intent, int flags, int startId) {
    ...
    NotificationCompat.Builder oBuilder =
      new NotificationCompat.Builder(this)
      .setSmallIcon(R.drawable.notification_icon)
      .setContentTitle("Super Alarm System")
      .setContentText("Super alarm system is running!")
      .setWhen(System.currentTimeMillis())
      .setOngoing(true);

    Intent oIntent = new Intent(this, ExampleActivity.class);
    PendingIntent oPI = PendingIntent.getActivity(this, 0, oIntent, 0);

    oBuilder.setContentIntent(oPI);
    startForeground(iID, oBuilder.build());
    // O codigo seguinte corre em primeiro plano
    ...
    stopForeground(true);
  }
...
```

Para remover um Serviço do primeiro plano de execução, **deve recorrer-se a `stopForeground()`. Terminar o Serviço também tem o mesmo efeito**. Este método **aceita um booleano que define se a notificação introduzida por `startForeground()` deve ser retirada** (no caso de ser `true`) **ou não** (no caso de ser `false`).



14 A Componente Recetor de Difusão

Sumário
Criação de recetores de eventos que atuam ao nível do sistema ou de uma aplicação móvel: a componente recetor de difusão.

Summary
Creation of event receivers acting at the system or mobile application level: broadcast receiver component.

1 Introdução

Um *recetor de difusão* (adaptação da designação inglesa *Broadcast Receiver*) é **um componente das aplicações móveis Android que permite que estas se registem para receber eventos provenientes do sistema ou de outra aplicação**. Estes **eventos** são também **representados por intentos**. Contudo:

- Os **intentos usados no contexto de inicialização de Serviços ou atividades são entregues** pelo sistema **a um único componente**, pelo que não podem ser usados para avisar, por exemplo, diversas aplicações acerca de determinado evento. Os **intentos enviados em difusão chegam a todas as aplicações com recetores registados para receber esses eventos**;

- Os **intentos** que são **lançados em difusão são diferentes** dos que são usados para iniciar atividades ou Serviços, pelo que **nunca atingem filtros de componentes destes dois componentes**;

- Por outro lado, os **intentos usados por atividades ou Serviços para despoletar outras componentes do mesmo género nunca chegam aos filtros dos recetores de difusão**.

Este componente é bastante importante no sistema Android, visto **permitir avisar um conjunto de aplicações**, registadas para o efeito, **acerca da ocorrência de determinado evento** no sistema ou aplicação **de uma forma eficiente**. É, por exemplo, este componente que permite que determinada aplicação reaja à receção de uma chamada de voz ou ao facto da bateria estar com pouca carga. Nestes casos, o sistema difunde eventos direcionados a determinado filtro de intentos, e todas as aplicações com um recetor registado ficam avisadas de que esse evento está a acontecer, ou já aconteceu.

A emissão de um intento em difusão **pode ser feita por qualquer aplicação do sistema**. Ainda assim, há que ter em conta que **a receção de alguns intentos em difusão pode requerer o pedido de algumas permissões no manifesto**. Caso este recurso não existisse, teria de ser simulado através da multiplicação e envio de intentos para todas as aplicações que o desejassem receber.

Algumas das ações definidas como padrão na classe `Intent` para difusão são mostradas na tabela seguinte. Intentos representando estas ações são **normalmente enviados pelo sistema Android** (*i.e.*, por várias das suas aplicações e gestores):

`ACTION_TIMEZONE_CHANGED`	O fuso horário foi mudado;
`ACTION_BOOT_COMPLETED`	Terminou o **arranque** do sistema;
`ACTION_PACKAGE_ADDED`	Foi **adicionada uma nova aplicação**;
`ACTION_PACKAGE_REMOVED`	Foi **removida uma aplicação**;
`ACTION_BATTERY_CHANGED`	O estado da **bateria** do dispositivo mudou;
`ACTION_POWER_CONNECTED`	O dispositivo foi **ligado à corrente elétrica**;
`ACTION_POWER_DISCONNECTED`	O dispositivo foi **desligado da corrente elétrica**;
`ACTION_SHUTDOWN`	O dispositivo está **prestes a ser desligado**.

Dos exemplos apresentados em cima, e que concretizam alguns dos mais típicos intentos difundidos pelo sistema Android pode, por exemplo, dizer-se que a ação `ACTION_PACKAGE_ADDED` é enviada em difusão pela aplicação nativa *Package Manager* para enfatizar o facto de que são as aplicações que geram estes intentos, e que qualquer aplicação o pode fazer, desde que tenha uma razão para isso (*e.g.*, avisar outras que determinados dados já estão disponíveis). Outros eventos mais específicos, mas relativos ao sistema, são originados pelos respetivos gestores caso a funcionalidade a que se referem esteja ativa. Por exemplo, caso o dispositivo móvel suporte telefonia, o respetivo gestor (`TelefonyManager`) também estará disponível, e difundirá intentos da sua responsabilidade, nomeadamente o `DATA_SMS_RECEIVED_ACTION`[1], caso seja recebida uma nova *Short*

[1] A definição de intentos de difusão disponíveis para alguns dos gestores é feita na implementação de algumas classes que a eles dizem respeito. Por exemplo, a ação `DATA_SMS_RECEIVED_ACTION` está definida em https://developer.android.com/reference/android/provider/Telephony.Sms.Intents.html.

Message Service (SMS).

Existem **duas formas de registar recetores**:

- **Estaticamente**, a partir da inclusão de uma *tag* `<receiver>` **no manifesto da aplicação**;
- **Dinamicamente**, *i.e.*, **programaticamente**, no **código da implementação de um dos outros componentes de determinada aplicação**. Esta opção faz **uso do método** `registerReceiver()`, disponível no contexto da aplicação.

2 Registo no Manifesto e Implementação

Quando definido **estaticamente**, o recetor de difusão é **registado aquando do arranque do sistema Android ou logo que um pacote é instalado**. O excerto de XML seguinte mostra a declaração estática de um recetor chamado `ReceiverForAlarms`, acompanhado pela definição de um filtro de intentos que *filtra* (ou, neste caso, *captura*) o evento `BOOT_COMPLETED`. Isto significa que este recetor é despoletado logo que o sistema termina o arranque:

```
<receiver
  android:name="ReceiverForAlarms" android:priority="100">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
  <intent-filter>
    <action android:name="pt.di.ubi.pmd.exservice.ServiceAlarms"/>
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</receiver>
```

Note que **há recetores para certo tipo de eventos que não podem ser declarados estaticamente** e que alguns requerem um pedido de permissões para funcionar. Por exemplo, o recetor definido antes precisa de uma permissão para aceder aos eventos que solicita, nomeadamente de:

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

O facto de alguns recetores só poderem ser definidos dinamicamente **deve-se sobretudo a questões de *performance* do sistema**. Os recetores, **se em grande número ou incidência, podem degradar a *performance***, já que **são invocados cada vez que o evento** que escutam **é despoletado** no sistema. Por exemplo, enquanto que o evento de `BOOT_COMPLETED` só é despoletado depois do arranque, os eventos do tipo `ACTION_TIME_TICK` são despoletados a **todos os minutos**. Neste caso, o ideal é que uma aplicação só esteja à escuta por esse evento, se dele necessitar, enquanto está em

execução. Por este motivo, o evento referido em último não pode ser capturado por recetores definidos estaticamente.

O excerto de código seguinte mostra, para já, a forma de **implementar um recetor de difusão**. De um modo sucinto, pode dizer-se que **tal é conseguido através da extensão da classe `BroadcastReceiver`, que requer importar `android.content.BroadcastReceiver`, e da reescrita do método `onReceive(.,.)`**:

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

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class ReceiverForAlarms extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // Este recetor faz parte do mesmo
    // pacote que o ServiceAlarms.
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);
  }
}
```

Quando um evento/intento representando uma das ações definidas nos filtros é despoletado, **o sistema Android entrega esse intento a todos os recetores registados** (e autorizados) **através da invocação do método referido**. Note que **o ciclo de vida de um recetor de difusão é unicamente determinado por esse método**. Conforme ilustrado, o componente Recetor de Difusão **afigura-se ideal para despoletar Serviços ou Atividades aquando da ocorrência de determinado evento**. No código incluído antes, mostra-se inclusive como é que o Serviço de alarmes, dado como exemplo na aula anterior, pode ser iniciado aquando do arranque do sistema, já que o recetor está definido para esse evento, e a sua implementação do método `onReceive(.,.)` define um intento explícito para o Serviço.

3 Registar Recetores de Difusão Programaticamente

Na secção anterior foi enfatizado o facto de **alguns recetores de difusão não poderem ser declarados estaticamente por motivos de *performance*** do sistema (*i.e.*, usá-los dessa forma pode levar a uma degradação da experiência do utilizador ou da capacidade de resposta do dispositivo de uma maneira geral). Por isso, a plataforma disponibiliza **formas para registar ou eliminar um registo de um recetor de difusão programaticamente**, mais especificamente através dos métodos:

- `registerReceiver(.,.)`, que **aceita o objeto recetor de difusão** a registar e **o objeto filtro de intentos** a que este deve ficar à escuta; e

- `unregisterReceiver(.)`, que **aceita apenas o objeto recetor de difusão cujo registo deve ser eliminado**. Note que este objeto (recetor de difusão) é normalmente implementado num ficheiro à parte, conforme também já ilustrado na secção anterior.

De modo a facilitar o entendimento desta forma de registar recetores de difusão, considere o seguinte excerto de código, acompanhado pelo seguinte cenário: considere que queria que qualquer serviço no sistema Android fosse capaz de disparar o serviço `ServiceAlarms` (discutido na aula anterior) através de um intento em difusão para o filtro `pt.di.ubi.pmd.exservice.ServiceAlarms`. Esta funcionalidade específica já era conseguida pela inclusão do filtro no manifesto exibido na secção anterior, é claro, mas assuma, adicionalmente, que queria que o recetor apenas estivesse disponível enquanto o utilizador estivesse a utilizar a atividade `FloatingAlarms`. Neste caso, o recetor de difusão deveria ser registado no método `onResume()`, sendo o seu registo anulado (preferencialmente) no método `onPause()`:

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

import android.app.Activity;
import android.content.IntentFilter;
...

public class FloatingAlarms extends Activity {
  private final ReceiverForAlarms oMyReceiver =
        new ReceiverForAlarms();

  @Override
  protected void onResume(){
    super.onResume();
    IntentFilter oIF = new IntentFilter("pt.di.ubi.pmd.exservice.
        ServiceAlarms");
    registerReceiver(oMyReceiver, oIF);
  }

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
  }

  @Override
  protected void onPause(){
    super.onPause();
    unregisterReceiver(oMyReceiver);
  }

  public void onButtonClick(View v){
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);
  }
}
```

Note que, no excerto de código anterior, e para além dos métodos de interesse para esta secção, é também utilizada uma classe que ainda não havia sido referida antes. **A classe `IntentFilter`, disponível em `android.content`, permite a definição dinâmica** (*i.e.*, programaticamente) **de filtros de intentos**, sendo obrigatória a sua utilização aquando do registo de um recetor de difusão, já que constitui o segundo argumento de `registerReceiver(.,.)`. Para terminar esta parte da discussão, resta dizer que, em baixo, mencionar-se-á como é que outra atividade ou Serviço pode enviar um intento para o recetor com este filtro.

4 Enviar Intentos em Difusão

A **API 21 recomenda apenas duas formas de enviar intentos em difusão**, embora anteriormente fossem disponibilizadas mais. Essas duas formas/métodos são:

- Intentos em difusão **não ordenados, enviados através do método `sendBroadcast(.)`, que aceita o intento** a enviar como parâmetro de entrada. Este tipo de intentos **é enviado para o sistema Android, que os entrega a todos os recetores** que estejam registados para os receber, sem nenhuma ordem em particular. Isto significa também que **este método (`sendBroadcast()`) é assíncrono, *i.e.*, retorna imediatamente à componente** que o chamou depois de ser entregue ao sistema. Como tal, **não permite que sejam recebidos resultados da execução dos recetores ou abortar determinado intento** enviado em difusão;

- Intentos em difusão **ordenados, enviados através do método `sendOrderedBroadcast(.,.)`, que aceita o intento** a enviar **e uma *String*** como parâmetros de entrada. **Estes tipos de intentos são enviados para o sistema Android, que os entrega a todos os recetores que estejam registados para os receber**, mas **por uma ordem em particular**, que pode ser **definida no manifesto através do atributo `android:priority="integer"` na *tag* `receiver`**. Este método **também é assíncrono** relativamente à componente que a invocou **mas permite**[2], por exemplo, **que um recetor ajuste dados que são enviados com o intento para outros recetores, antes que estes recebam o intento**. **Também permite que determinado recetor aborte o intento antes de este chegar a outros recetores com prioridade inferior**.

[2] Basicamente, este facto significa que a componente que invocou o método continua a executar normal e imediatamente após o ter emitido, já que o sistema retorna imediatamente, embora os vários recetores fiquem a executar em paralelo para além da aplicação ou componente.

Os dois métodos referidos antes estão **incluídos na classe `Context`, por conveniência**. Esta classe **engloba muitos dos métodos fulcrais para o desenvolvimento de aplicações móveis**, sendo **uma das maiores**, cuja **importância** é assim de fácil justificação[3]. A classe contém **mais métodos com assinaturas semelhante às anteriores**, nomeadamente um `sendBroadcast(.,.)` que aceita também uma *String* para as permissões necessárias do recetor, e um `sendBroadcastAsUser(.,.,.)`, que permite enviar um intento em difusão em nome de determinado utilizador/aplicação, ou um `sendOrderedBroadcast(.,.,.,.,.,.)` com 7 argumentos, para **um maior controlo sobre determinada difusão**. O método referido em último permite receber retorno através de um truque. Na verdade, um dos argumentos indica um recetor do mesmo intento que é enviado, e o SO trata-o como sendo o último a recebê-lo (portanto, pode receber o retorno que outro lhe coloque).

O pequeno trecho de código seguinte mostra como uma Atividade, numa outra aplicação diferente da anteriormente referida, pode enviar um intento em difusão para o recetor que foi registado antes:

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

import android.app.Activity;
...

public class ExActivity extends Activity {

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);
  }

  public void onButtonClick(View v){
    Intent oIntent = new Intent("pt.di.ubi.pmd.exservice.ServiceAlarms");
    sendOrderedBroadcast(oIntent, null);
  }
}
```

Por curiosidade, atente nos dois detalhes seguintes: (i) o registo do recetor no ficheiro `AndroidManifest.xml` (incluído já nesta aula) contém um atributo `android:priority`, que só foi discutido nesta secção; (ii) **o intento** criado no método `onButtonClick()` **é implícito, mas aponta diretamente para o filtro** que foi definido naquele manifesto, sendo enviado usando o método `sendOrderedBroadcast()`. Assim sendo, e dada a prioridade definida no manifesto, e a menos que outra aplicação defina uma prioridade maior, **este intento será entregue a esse recetor, antes de ser entregue a outros** que tenham o mesmo filtro. Por outro lado, **como o segundo parâmetro de `sendOrderedBroadcast(.,.)` está a `null`, os recetores registados para este intento não precisam ter pedido qualquer permissão** para o poder receber. O pa-

[3] Ver http://developer.android.com/reference/android/content/Context.html.

râmetro da permissão pode ser usado para definir também quais os recetores que podem receber determinado intento.

Considere ainda que a implementação do recetor de difusão continha uma linha adicional com a **invocação do método** `abortBroadcast()`, conforme se mostra a seguir:

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

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class ReceiverForAlarms extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // Este recetor faz parte do mesmo
    // pacote que o ServiceAlarms.
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);

    // Note a seguinte linha de codigo:
    abortBroadcast();
  }
}
```

Neste caso, e **só porque o intento foi enviado com o método `sendOrderedBroadcast()`, este recetor podia cancelá-lo**, impedindo que outros recetores (com menos prioridade) o recebessem. Note que quando é usado `sendBroadcast()`, todos os recetores registados para determinado intento o recebem, independentemente da ordem. Isto significa que **intentos enviados com `sendBroadcast()` não podem ser abortados**.

Caso fosse necessário propagar resultados entre recetores, e assumindo que estes eram invocados pelo sistema depois deste receber um intento enviado com `sendOrderedBroadcast()`, poder-se-iam usar os métodos `getResultData()` e `setResultData(.)`, entre outros, para obter e adicionar dados que eram transmitidos com o evento, como se mostra a seguir:

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

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class ReceiverForAlarms extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // Este recetor faz parte do mesmo
    // pacote que o ServiceAlarms.
    Intent oIntent = new Intent(this, ServiceAlarms.class);
    startService(oIntent);

    // Note as seguintes linhas de codigo
```

```
    String sData = getResultData();
    sData = sData + " Another receiver!";
    setResultData(sData);
  }
}
```

O excerto de código anterior mostra como um determinado recetor (chamado `ReceiverForAlarms`) iria processar uma *String* vinda de outro recetor anterior, antes de a enviar para o próximo. Neste caso, iria usar o método de conveniência `getResultData()`, que devolve uma *String*, adicionando-lhe uma nova frase `Another receiver!` antes de voltar a ajustar aquele parâmetro. Caso houvessem 3 recetores a receber o referido intento, o último a recebê-lo iria obter uma `String` semelhante a:

`Another receiver! Another receiver! Another receiver!`.

O registo de recetores de difusão é uma funcionalidade **algo delicada do sistema**, principalmente porque pode constituir **um ponto de entrada (leia-se vetor de ataque) para determinada aplicação**. Se o recetor estiver **acessível para todas as aplicações, pode ser despoletado, por exemplo, por uma aplicação maliciosa**[4], que pode tentar subverter o fluxo normal da aplicação através da manipulação do intento ou com outras estratégias. Estes **problemas resolvem-se combinando vários recursos e definições**, nomeadamente através do ajuste de parâmetros no manifesto (*e.g.*, ajustando atributo `android:exported`). Contudo, **caso os eventos em difusão sejam apenas dirigidos a objetos do mesmo processo, pode recorrer ao gestor** `LocalBroadcastManager`, que lhe ajuda a **registar recetores e a enviar intentos em difusão de âmbito local**. Neste caso, **intentos externos ao processo não chegam aos recetores assim declarados, e os intentos gerados não chegam a outros recetores do sistema**. Para comunicações locais, esta solução é **muito mais eficiente** que a que foi antes discutida.

[4]Mais sobre este assunto em http://developer.android.com/reference/android/content/BroadcastReceiver.html#Security.



15 A Componente Provedor de Conteúdos

Sumário

Acesso e manipulação de conteúdos partilhados. Definição de um provedor de conteúdos no âmbito de determinada aplicação móvel.

Summary

Access to, and manipulation of, shared contents. Definition of a content provider in the scope of a given mobile application.

1 Introdução

Os **Provedores de Conteúdos constituem o quarto tipo de componentes Android** discutidos neste curso. É um componente deste tipo que **permite gerir o acesso a um repositório de dados de uma determinada aplicação**. A utilização de um Provedor de Conteúdos é **sobretudo direcionada a aplicações diferentes daquela que o criou**, já que essa tem acesso direto ao que é disponibilizado. O componente é fulcral para o objetivo em questão, já que **permite controlar, de uma forma consistente e central**, o acesso a recursos que, de outra forma, deveriam ser privados. **Oferece uma interface padrão para aceder aos dados que lida automaticamente com comunicação entre-processos e segurança** no acesso aos dados.

É comum que **uma aplicação que implemente um Provedor de Conteúdos também forneça uma *User Interface* (UI) para esses conteúdos**, já que faz sentido que seja essa aplicação a maior interessada em lidar com todos os detalhes dos dados que fornece. No exemplo incluído neste capítulo mostra-se como se pode construir um

Provedor de Conteúdos para uma base de dados que guarda os filmes favoritos de um utilizador. Este exemplo foi explorado numa aula prática anterior, e a respetiva aplicação continha formas de ver, aceder, inserir, atualizar ou eliminar dados da base de dados, embora sem aceder ao Fornecedor de Conteúdos.

A funcionalidade deste componente é concretizada, na verdade e por desenho, **por dois módulos de *software* diferentes** (os **Fornecedores de Conteúdos** em si, e os **clientes desses Fornecedores**), disponíveis em **classes diferentes**. A discussão seguinte aborda, por isso, os seguintes tópicos:

1. **Como criar e declarar um Provedor de Conteúdos** em determinada aplicação Android;
2. **Como aceder a um Provedor de Conteúdos disponível numa outra aplicação**.

2 Criar um Provedor de Conteúdos

A criação de um provedor de conteúdos é tido como **um processo algo elaborado**, principalmente porque deve **comportar uma fase de análise da real necessidade de o implementar, a modelação e estruturação dos dados e a implementação/declaração** propriamente dita. Após se decidir que deve ser efetivamente criado um Provedor de Conteúdos, **os passos estritamente necessários à sua criação são** os seguintes:

1. **Modelar a forma como os dados são armazenados e implementar a lógica necessária para os criar**. Os fornecedores de conteúdos podem disponibilizar **dados na forma de ficheiros** ou **dados estruturados**, normalmente armazenados em bases de dados relacionais SQLite;
2. **Definir uma String que determina, de forma unívoca no universo Android, o Provedor de Conteúdos** a implementar. Esta `String` é tipicamente chamada de **autoridade (*authority*)**, e a documentação oficial sugere que seja usado **um esquema parecido ao que é usado para definir domínios e recursos na Internet**, **mas de forma reversa**. *E.g.*, enquanto que para o site do Departamento de Informática se usa `di.ubi.pt`, no caso das aplicações Android e dos Provedores de Conteúdos usa-se `pt.ubi.di.nome_applicacao` e `pt.ubi.di.nome_aplicacao.nome_provedor`. Este esquema é, na maioria das vezes, suficiente para garantir que a autoridade é única à escala global.
3. **Implementar a classe** `ContentProvider` (disponível no pacote `android.content`), **reescrevendo alguns dos seus métodos** (`onCreate()`, `query()`, `insert()`, ...),

como de resto é costume para outras componentes de aplicações Android, como as Atividades ou Serviços;

4. **Declarar o novo componente no manifesto Android**, bem como **definir as permissões necessárias** que outra aplicação que lhe quer aceder deve ter.

De modo a cristalizar melhor os passos enunciados em cima, discute-se a seguir **um exemplo** de uma implementação de um Provedor de Conteúdos. Note que este exemplo elabora (e introduz algumas alterações) ao que já foi feito numa das aulas práticas anteriores. A ideia da aplicação e do respetivo Provedor é lidar e **disponibilizar uma lista de filmes favoritos de determinado utilizador**. O exemplo e discussão serão focados na disponibilização de conteúdos sob a forma de **dados estruturados guardados numa base de dados SQLite, que é a mais comum em aplicações Android**. Esta explicação pode ser generalizada, em algumas partes, para a disponibilização de ficheiros. Contudo, quando se lida com ficheiros, o retorno dos Fornecedores de Conteúdos é constituído por *handlers* para esses recursos, e não por `Cursores`.

Conforme sugerido no passo 1, a primeira parte da implementação de um Provedor de Conteúdos deve ser focada na **modelação dos dados e na lógica necessária para os criar/atualizar**. O primeiro trecho de código mostra a implementação de uma classe que estende a `SQLiteOpenHelper`, conforme já discutido numa aula anterior. Conforme irá notar, são declaradas **algumas variáveis de conveniência relativas à base de dados**, nomeadamente a *String* contendo a instrução **SQL embutida** para criação da única tabela da base de dados. Note que este exemplo é básico, visto que apenas irá referir uma única tabela, para facilitar a explicação. Para exemplos mais elaborados, sugere-se a consulta da documentação em http://developer.android.com/guide/topics/providers/content-provider-creating.html.

Relativamente ao código que já foi disponibilizado antes, o seguinte excerto de código difere apenas no nome da chave primária, que aqui se chama `_ID` e no facto do nome da base de dados ser enviado como um parâmetro no construtor da classe, em vez de estar definido estaticamente como uma *String* no início da classe:

```
package pt.ubi.di.pmd.exstorage2;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;

public class AjudanteParaAbrirBD extends SQLiteOpenHelper {
  private static final int DB_VERSION = 1;
  protected static final String
          TABLE_NAME = "Movie";
  protected static final String COL1 = "_ID";
  protected static final String COL2 = "name";
  protected static final String COL3 = "year";

  private static final String CREATE_MOVIE =
```

```
    "CREATE TABLE " + TABLE_NAME +
    " ("+COL1+" INTEGER PRIMARY KEY,  "
        +COL2+" VARCHAR(50), "
        +COL3+ " INT);";

  public AjudanteParaAbrirBD(Context context, String DB_NAME) {
    super(context, DB_NAME, null, DB_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(CREATE_MOVIE);
  }
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
      {
    db.execSQL("DROP TABLE " + TABLE_NAME + ";");
    db.execSQL(CREATE_MOVIE);
  }
}
```

É **útil definir a chave primária das tabelas da base de dados afetas a um Fornecedor de Conteúdos com o nome** `_ID`, já que **alguns objetos permitem carregar o conteúdo de um** `Cursor` **para uma** `View` (*e.g.*, `ListView`) **automaticamente**, assumindo que um campo com o nome `_ID` é fornecido.

Tendo a criação e atualização da base de dados assegurada, passa-se para a implementação do Provedor de Conteúdos propriamente dito, ilustrado de seguida:

```
package pt.ubi.di.pmd.exstorage2;

import android.content.ContentProvider;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.net.Uri;

public class ProvedorFavFilmes extends ContentProvider {
  // Sera necessario um handle para a
  // base de dados a abrir:
  private AjudanteParaAbrirBD oAPABD;

  // O nome da base de dados:
  private String DBNAME = "FavoriteMovies";

  // O objeto que ira permitir
  // aceder a base de dados:
  private SQLiteDatabase oSQLiteDB;

  private String sAuthority =
    "pt.ubi.di.exstorage2.provedor";

  @Override
```

```
  public boolean onCreate() {
    // Criar um objeto do tipo AjudanteParaAbrirBD.
    // Este metodo devolve muito rapido porque a
    // base de dados apenas e aberta quando
    // oSQLiteDB.getWritableDatabase() for invocado:
   oAPABD = new AjudanteParaAbrirBD(
            getContext(),
            DBNAME
        );
        return true;
    }

  // Implementa o metodo query do Provedor:
  @Override
  public Cursor query(
      Uri uri,
      String[] projection,
      String selection,
      String[] selectionArgs,
      String sortOrder){
    // Abrir a base de dados para leitura:
    oSQLiteDB = oAPABD.getReadableDatabase();
    Cursor oCursor = oSQLiteDB.query(
      "Movie", projection, selection, selectionArgs,
      null, null, sortOrder);
    return oCursor;
  }

  // Implementa o metodo insert do Provedor:
  @Override
  public Uri insert(Uri uri, ContentValues oCValues) {
    // Inserir aqui, eventualmente, algum codigo
    // relacionado com a gestao de erros, etc.

    // Abrir a base de dados:
    oSQLiteDB = oAPABD.getWritableDatabase();
    int iId = (int) oSQLiteDB.insert(oAPABD.TABLE_NAME, null, oCValues);
    oSQLiteDB.close();
    return new Uri(sAuthority+"/"+iId);
  }
}
```

Conforme se mostra em cima, **é necessário importar e estender a classe** `android.content.ContentProvider`. Neste caso, é também necessário **importar as classes** `SQLiteDatabase` (já discutida anteriormente) **e a classe** `Uri`**, disponível no pacote** `android.net`. Relembre que o acrónimo URI **expande para Uniform Resource Identifier** e que, portanto, **é este objeto que permite identificar, de forma única, um determinado recurso num determinado domínio**.

O código incluído anteriormente reescreve apenas dois métodos da classe `ContentProvider` (embora pudessem ser implementados mais):

- **O método `onCreate()`, que deve ser usado para inicializar o Provedor de Conteúdos**, nomeadamente para inicializar **um *handler* para a abertura da base de dados**, se for o caso. Este método é **executado na *thread* principal** logo que a aplicação que o contém é iniciada e, por isso, **não deve conter operações demoradas**. Por exemplo, **pode conter a instanciação do objeto que ajuda a criar ou atualizar a base de dados, mas não deve conter a chamada para a abertura dessa base de dados** (*e.g.*, não deve conter `getWritableDatabase()`). As operações mais demoradas, como a abertura da base de dados para escrita, deve ser feita apenas quando for necessária, nomeadamente dentro de métodos como `insert()`, `delete()` ou `update()`. Este **método devolve `true` caso o Provedor tenha sido criado com sucesso, e `false` no caso contrário**;
- O **método `insert(.,.)`** que, neste caso, apenas se limita à abertura da base de dados, e à **inserção dos valores provenientes do cliente deste Provedor** (contidos em `oCValues`). Note que o método **`insert(.,.)` aceita um `Uri` como primeiro parâmetro e também devolve um objeto desta classe**. O `Uri` recebido deve **apontar especificamente para o recurso que se quer utilizar no âmbito do `INSERT`, enquanto que o devolvido deve apontar para a linha inserida**. Neste caso, o `Uri` de entrada não é utilizado no âmbito do método, porque apenas existe uma tabela onde inserir dados, mas este podia ser, por exemplo, **parecido com `content://pt.ubi.di.exstorage2.provedor/Movie`**. Já **o de saída seria semelhante a `content://pt.ubi.di.exstorage2.provedor/Movie/2`, em que o 2 apontava para a linha específica criada pelo `INSERT`** dentro da tabela `Movie`.

O último passo consiste em declarar o componente no ficheiro `AndroidManifest.xml`. A seguir inclui-se um exemplo do que poderia ser o conteúdo deste ficheiro para a aplicação discutida em cima:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
 xmlns:android="http://schemas.android.com/apk/res/android"
 package="pt.ubi.di.pmd.exstorage2"
 android:versionCode="1"
 android:versionName="1.0">

<uses-sdk android:minSdkVersion="8"
  android:targetSdkVersion = "21" />

<permission android:name="pt.ubi.di.pmd.perm-provider"
  android:label="Permission for Movies Provider"
  android:description="@string/act_permission_desc"
  android:protectionLevel="dangerous" />

<application android:label="@string/app_name"
             android:icon="@drawable/ic_launcher">
  <activity android:name="FavoriteMovies"
            android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
```

```
    </intent-filter>
  </activity>

  <provider android:name="ProvedorFavFilmes"
    android:authorities="pt.ubi.di.pmd.exstorage2.provedor"
    android:permission="pt.ubi.di.pmd.perm-provider" />
</application>
</manifest>
```

Repare nos detalhes seguintes:

- O segundo *tag* incluído no elemento `manifest` corresponde à **definição de uma permissão nova**, chamada `pt.ubi.di.pmd.perm-provider`;
- Dentro do elemento `application` é **definido um elemento** `activity` **e um elemento** `provider`, cujo nome é `ProvedorFavFilmes`;
- Dentro da *tag* `provider` é **definida a *authority* deste Provedor de conteúdos**, representada pelo URI `pt.ubi.di.pmd.exstorage2.provedor`. **Não se podem declarar Provedores de uma forma dinâmica**, pelo que têm de ser obrigatoriamente declarados no `AndroidManifest.xml`; e
- **A permissão** referida no primeiro item desta lista **é aplicada ao provedor através do atributo** `android:permission`. Ao usar este atributo está imediatamente a definir a **permissão para aceder ao Provedor de Conteúdos tanto para leitura, como para escrita**. Caso quisesse **diferenciar** o tipo de permissões necessárias para os dois casos, podia primeiro declará-las com nomes diferentes e depois **usar os atributos** `android:readPermission` **e** `android:writePermission` para obter esse tipo de granularidade.

Relembre-se que, ao serem declaradas as permissões necessárias para aceder a determinado componente de uma aplicação Android, o utilizador tem de a aceitar e fornecer explicitamente à aplicação que a requisita aquando da sua instalação. A aplicação anterior não está a requisitar permissões, mas sim a declará-las. Isto significa que serão as aplicações que quiserem utilizar o Provedor que as terão de pedir.

3 Aceder a um Provedor de Conteúdos

Uma aplicação pode **aceder aos dados de determinado Provedor de Conteúdos recorrendo a um objeto da classe** `ContentResolver`. Note que **este objeto assumirá o papel de cliente no modelo de comunicação** entre a aplicação que disponibiliza o conteúdo e aquela que o recebe. **Para cada método disponível no cliente, será despoletado um método equivalente no servidor** (Provedor), **sendo os resultados entregues ao primeiro**. O `ContentResolver` oferece assim, na aplicação

cliente, **os métodos básicos de obtenção** (*i.e.*, `query()`) **e edição** (*i.e.*, `insert()`, `update()` e `delete()`) **do conteúdo**. É inclusive usado o **acrónimo CRUD para referir o conjunto de funcionalidades disponibilizadas**, abreviando ***C**reate*, ***R**etrieve*, ***U**pdate* e ***D**elete*. Repare que **os dois objetos executam em diferentes processos** (pertencentes a cada uma das aplicações), mas **lidam automaticamente com as comunicações** entre os mesmos.

O acesso a um Provedor de Conteúdos requer tipicamente o **preenchimento das seguintes condições**:

1. **Pedido de permissões** para o Provedor específico no `AndroidManifest.xml`;
2. **Conhecimento prévio** (ou forma de obter essa informação em tempo de execução) **do URI** do Provedor de Conteúdos;
3. **Instanciação de um objeto da classe `ContentResolver`, seguida da utilização dos métodos** que esta disponibiliza para acesso aos conteúdos. Opcionalmente, pode fazer-se **diretamente uso dos métodos estáticos para acesso a esses conteúdos**.

A **obtenção da informação acerca das permissões** necessárias **ou do URI** do Provedor de Conteúdos **é conseguida normalmente através da consulta da documentação do próprio Provedor**. Por exemplo, **o sistema Android já fornece uma panóplia de Provedores de Conteúdos para uso em aplicações**, nomeadamente para acesso aos registos das chamadas (`content://call_log/calls`), contactos (`content://contacts/people`), *bookmarks* (`content://browser/bookmarks`), etc. Os URIs respetivos são mencionados na documentação oficial, sendo que, por vezes, **também estão disponíveis via variáveis estáticas e públicas nas classes que implementam os Provedores**. Por exemplo, a classe `android.provider.UserDictionary` contém uma *String* chamada `CONTENT_URI` com essa informação específica.

Nesta secção faz-se referência a uma nova aplicação Android que irá fazer uso do Provedor de Conteúdos implementado antes. Considere que a aplicação tinha como objetivo mostrar não só os filmes favoritos de um utilizador, mas também as músicas. Caso a aplicação `FavoriteMovies` estivesse instalada, esta nova, e melhorada, aplicação com nome `MoviesAndMusique` **iria tentar importar os filmes que o utilizador já tivesse colecionado, através do Provedor**. Para já, **é necessário pedir as permissões necessárias no manifesto** Android. O conteúdo do ficheiro seria, assim, algo semelhante a:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
 xmlns:android="http://schemas.android.com/apk/res/android"
 package="pt.ubi.di.pmd.otherapp"
 android:versionCode="1"
 android:versionName="1.0">
```

```
<uses-sdk android:minSdkVersion="8"
  android:targetSdkVersion = "21" />

<uses-permission android:name="pt.ubi.di.pmd.perm-provider" />

<application android:label="@string/app_name"
             android:icon="@drawable/ic_launcher">
  <activity android:name="MoviesAndMusique"
            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>
```

Note que o manifesto incluído antes pede a permissão `pt.ubi.di.pmd.perm-provider`, sem a qual não poderia aceder ao Provedor. De seguida mostra-se a implementação da atividade `MoviesAndMusique`, que irá **tentar obter os valores da tabela `Movie` e exibi-los imediatamente numa `ListView`**, tudo na função `onCreate()`:

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

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

import android.net.Uri;
import android.database.Cursor;
import android.content.ContentValues;

import android.view.View;
import android.widget.ListView;
import android.support.v4.widget.SimpleCursorAdapter;

public class MoviesAndMusique extends Activity
{
  private Uri oUriProvider =
    new Uri("pt.ubi.di.pmd.exstorage2.provedor");
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Cursor oCursor = getContentResolver().query(
    oUriProvider,
    new String[]{"name","year"},
    null,  // Criterio de selecao
    null,  // Parametros do criterio de selecao
    null); // Ordem

    if ( (null == mCursor ) || (mCursor.getCount() < 1)) {
       // Caso nao sejam devolvidos resultados
       TextView  oTV = (TextView) findViewById(R.id.TV);
       oTV.setText("No data available!");
    } else {

      ListView oLV = (ListView) findViewById(R.id.lv);
```

```
        SimpleCursorAdapter oAdptr = new SimpleCursorAdapter(
          this,           // Contexto da aplicacao
          R.layout.line, // Um XML a definir uma linha
          oCursor,        // O cursor com o resultado
          new String[]{"name","year"},    // Nomes das colunas
          new int[]{R.id.ED1, R.id.ED2}, // IDs das views
          0);             // Flags opcionais

        // Ajusta o adaptador a respetiva widget
        oLV.setAdapter(oAdptr);
      }
    }

    public void onINSERTclick( View v ){
      ContentValues oCValues = new ContentValues();
      EditText oED1 = (EditText) findViewById(R.id.name);
      EditText oED2 = (EditText) findViewById(R.id.year);
      oCValues.put("name", oED1.getText().toString() );
      oCValues.put("year", new Integer( oED2.getText().toString() ));
      Uri iId = getContentResolver().insert(oUriProvider,oCValues);
    }
}
```

O excerto de código anterior contém bastantes detalhes de interesse:

1. Em primeiro lugar, contém **dois *imports* para as classe dos pacotes `android.widget` e `android.support.v4.widget`** nunca referidas anteriormente. **A `ListView` é um contentor que permite mostrar itens numa lista com orientação vertical com *scroll***. Esta `ListView` **pode ser automática e dinamicamente preenchida a partir de um cursor, usando um adaptador**, *e.g.*, da classe `SimpleCursorAdapter`;

2. Em segundo lugar, é **instanciado um objeto da classe `Uri`** no início da implementação da Atividade `MoviesAndMusique`, **que aponta para o Provedor de Conteúdos** definido antes. Este objeto é **depois usado nos métodos `query()` e `insert()`**;

3. Em terceiro lugar, **é feita uma tentativa de obter os dados do Provedor de Conteúdos através do método `query()`, que funciona de modo muito semelhante ao que é usado para aceder a uma base de dados SQLite, embora o primeiro parâmetro seja agora um URI**, e não o nome de uma tabela. Na verdade, em muitos casos, **este URI deve apontar, de forma unívoca para a tabela a que se quer aceder** (neste caso, como só existe uma tabela, basta apontar para o Provedor). Note ainda que **o método `query()` devolve um objeto da classe `Cursor`**;

4. Em quarto lugar, é feita uma verificação se o cursor está vazio ou é nulo, ajustando uma mensagem no *layout* caso isso aconteça;

5. Caso contrário, **é feita uma tentativa de colocar o conteúdo do cursor devolvido pela *query* numa** `ListView`, conforme referido em cima;

6. Por último, é também exibida a implementação do método `onINSERTclick()`, que permite **fazer uma inserção no Provedor de Conteúdos através do método** `insert(Uri,ContentValues)`.

Note que ficam ainda alguns detalhes por especificar, nomeadamente na implementação do Provedor. Na verdade, **não foram implementados os métodos para atualização ou eliminação de registos**, tal como **não foi programada lógica para lidar com URIs mais específicos** (*e.g.*, um URI que especificasse o nome da tabela e o número da linha a devolver numa *query*). Adicionalmente, não foi mostrado o conteúdo de todos os XML usados mas, para clarificar a forma de atuação do adaptador entre a `ListView` e o `Cursor`, mostra-se a seguir o conteúdo do ficheiro `line.xml`, que define como cada linha da lista deve ser desenhada:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <EditText
      android:layout_width="0dp"
      android:layout_height="fill_parent"
      android:gravity="left"
      android:id="@+id/ED1"
      android:layout_weight="2"
    />
    <EditText
      android:layout_width="0dp"
      android:layout_height="fill_parent"
      android:gravity="center"
      android:id="@+id/ED2"
      android:layout_weight="1"
    />
</LinearLayout>
```

4 URIs dos Conteúdos

Os URIs usados no contexto de aplicações Android e, mais concretamente, no contexto de Provedores de Conteúdos, **são recursos muito poderosos**, apesar de não serem alvo de uma análise muito elaborada nas secções anteriores. Normalmente, e neste contexto, **um URI pode ser usado para determinar desde o Provedor de Conteúdos até à linha da tabela que se quer obter**. Um URI pode ser **decomposto nas seguintes**

partes: `<standard_prefix>://<authority>/<data_path>/<id>`. Cada uma das partes pode então ser descrita da seguinte forma:

- A parte `<standard_prefix>` **determina o protocolo ou tipo de recurso** (*e.g.*, para os Provedores de Conteúdo é sempre `content`, para recursos na *World Wide Web* (WWW) é normalmente `http`);
- **A** `<authority>` **determina unicamente o recurso** a que se quer aceder;
- **A** `<data_path>` **especifica a parte do recurso** a que se quer aceder, *e.g.*, nomes de tabelas ou páginas específicas;
- **A parte do** `<id>` **é normalmente utilizada para transportar parâmetros para a parte do recurso especificado**. Em Provedores de Conteúdos é comum incluir um número nesta parte do URI que identifica a linha que se quer obter, atualizar ou eliminar para determinada tabela mencionada no `<data_path>`.

A título de exemplo, pode indicar-se o URI `content://contacts/people/13`, que aponta para o contacto número 13 guardado na tabela `people` do Fornecedor de Conteúdos `contacts`. O URI `content://call_log/calls/1` (que aponta para outro Fornecedor do sistema Android) aponta para a primeira chamada no registo de chamadas, guardado numa tabela que provavelmente se chama `calls`.

O **tratamento do URI num Provedor de Conteúdos significa decompô-lo em várias partes, e lidar com essas partes** (nomeadamente com o nome do recurso e com os parâmetros) **caso a caso**. A plataforma fornece algum *software* que facilita este tratamento, nomeadamente através da **classe** `UriMatcher`[1].

[1] Ver http://developer.android.com/guide/topics/providers/content-provider-creating.html.



16 *Framework* de Sensores

Sumário

Acesso a sensores em dispositivos móveis. Discussão da *framework* que abstrai esse acesso na plataforma Android, mais especificamente através das classes `SensorManager`, `Sensor` e `SensorEvent`, bem como da interface `SensorEventListener`.

Summary

Access to sensors in mobile devices. Discussion of the framework that abstracts the access to sensors in the Android platform, namely through the `SensorManager`, `Sensor` and `SensorEvent` classes, as well as via the `SensorEventListener` interface.

1 Introdução

A **maior parte dos dispositivos móveis de hoje integram sensores** que **medem o movimento, a localização, a orientação e vários parâmetros ambientais, como a temperatura ou a humidade**. Os valores devolvidos por estes sensores podem, aparte algumas limitações, **ser usados no âmbito de aplicações móveis**, de modo a tornar a **experiência mais pessoal**, **transparente**, **simples e integrada** para o utilizador:

- **Pessoal**, porque determinada aplicação pode **mostrar dados que têm a ver com o ambiente** onde o utilizador está (*e.g.*, uma aplicação mostra os restaurantes que estão na redondeza) **ou com o que o utilizador está a fazer** (*e.g.*, uma aplicação faz sugestões para abrandar ou acelerar o ritmo quando o utilizador está a correr);
- **Transparente**, porque determinada aplicação pode **ajustar automaticamente**

o *layout* ou decidir fluxos de execução de acordo com alguns valores dos sensores (*e.g.*, estiver frio, uma aplicação de sugestão de compras pode escolher não mostrar roupa para tempo quente nesse dia);

- **Simples e integrada**, porque determinada aplicação pode usar alguns sensores para **tornar a navegação mais intuitiva ao utilizador** (*e.g.*, um jogo de corridas pode usar o sensor de orientação para permitir a condução do veículo, evitando a utilização de botões para o mesmo efeito);

2 Sensores em Android

A **plataforma Android**[1] **suporta três categorias principais de sensores**, estruturadas da seguinte forma:

1. **Sensores de Movimento**, que **incluem os sensores que medem forças de aceleração e rotacionais em três eixos**. É nesta categoria que são **incluídos os acelerómetros, giroscópios, sensores de gravidade e os rotacionais**;
2. **Sensores de Ambiente**, que **incluem os sensores que medem parâmetros de ambiente**, como a temperatura, a pressão do ar, a humidade ou a iluminação. Aqui se **incluem os barómetros, fotómetros e termómetros**;
3. **Sensores de Posicionamento**, que medem/identificam **a posição física dos dispositivos, e que incluem o *Global Positioning Sensor* (GPS), sensores de orientação e os magnetómetros**.

Como seria de esperar, a plataforma Android permite **um acesso bastante simplificado** a todos os sensores disponíveis no dispositivo móvel a **partir da *framework* de sensores Android**. Esta *framework* **contém classes e interfaces** que podem ser usadas para **adquirir os valores em estado bruto** desses sensores, bem como **outras funcionalidades**, nomeadamente **registo de recetores de eventos** para sensores em particular.

Na verdade, a explicação referente aos sensores constitui também uma boa oportunidade para referir (e relembrar) **a forma padrão** de como bastantes funcionalidades da plataforma Android podem ser conseguidas. Já antes foi referido o facto do SO executar automaticamente **vários gestores, que mais não são do que aplicações que vêm nativas com o sistema**, situadas na parte da ***framework* aplicacional da pilha da plataforma** (ver aula 3). **Não é tipicamente possível aceder diretamente a um**

[1] A explicação subsequente é sobretudo inspirada na documentação oficial da plataforma Android sobre este assunto, nomeadamente na discussão contida no URL `http://developer.android.com/guide/topics/sensors/sensors_overview.html`.

sensor ou, de uma forma geral, a qualquer recurso para o qual haja um gestor disponível. O procedimento elabora em:

1. **Declarar um objeto da classe do gestor** pretendido na aplicação (mas **não instanciá-lo** diretamente);
2. **Pedir** a instância do gestor **ao sistema**;
3. **Aceder a funcionalidades disponibilizadas pelo gestor**, como obter valores, enviar mensagens, registar ou eliminar o registo para um recetor.

Note que este procedimento já foi antes discutido, aquando da menção ao gestor de notificações. Nesse caso, obtinha-se uma instância do gestor pedindo-o ao sistema através do método `getSystemService(Context.NOTIFICATION_SERVICE)`, **fornecido com o contexto**. Depois, era invocado um dos métodos do gestor para lhe entregar uma mensagem (nomeadamente o método `notify()`):

```
...
NotificationManager oNM = (NotificationManager)
  getSystemService(Context.NOTIFICATION_SERVICE);
oNM.notify(iID, oBuilder.build());
```

O uso dos sensores será, portanto, semelhante.

No caso específico dos sensores, a *framework* fornece as seguintes funcionalidades:

- Permite **determinar que sensores estão disponíveis** no dispositivo;
- Permite **obter as capacidades de cada sensor**, tal como o fabricante, os requisitos de energia, resolução e alcance;
- Permite **adquirir os dados em bruto e definir**, em alguns casos, **a cadência com que os dados devem ser adquiridos**; e
- Permite **registar ou eliminar o registo de *listeners* para determinado tipo de eventos** relacionados com sensores, e forma a detetar mudanças nos seus valores.

Os sensores disponíveis num dispositivo com Android **podem ainda ser divididos em dois tipos principais**, embora tal facto não mude a forma como estes são utilizados, **nomeadamente *hardware-* e *software-based sensors***. No primeiro caso, os valores devolvidos pelo sensor são **derivados diretamente de um dispositivo físico**, enquanto que, para o segundo caso, **os valores são derivados de um ou mais dispositivos físicos, podendo ser tratados antes de devolvidos**. O **acelerómetro ou magnetómetro são exemplos de sensores de** `hardware`, enquanto que **o sensor gravitacional constitui um exemplo do segundo**. Note que **nem todos os**

dispositivos integram todos os sensores suportados pela plataforma, e que novos sensores podem vir a ser adicionados com o tempo. Por exemplo, **a maior parte dos dispositivos móveis atuais contêm um giroscópio**, mas **poucos integram um termómetro ou um barómetro**.

3 Constituição da *Framework* de Sensores

A *framework* de sensores é constituída por três classes principais e uma interface, disponibilizadas no **pacote** `android.hardware`:

1. A `SensorManager` (Gestor de Sensores), que é **a classe que encapsula os métodos que podem ser usados para interagir com o gestor em execução** no sistema, e que **constitui o ponto de partida para aceder e listar sensores**, ou registar escutas para os mesmos;

2. A `Sensor`, que é **a classe que permite criar uma instância de um sensor específico** no âmbito da aplicação, e que **fornece os vários métodos que podem ser usados para verificar as características do sensor** instanciado. Enquanto que **uma instância da classe referida no ponto anterior permite verificar se determinado sensor está disponível no sistema, é esta classe que permite**, por exemplo, **verificar qual é a marca ou versão de determinado sensor**, ou a energia (em mA) que este utiliza, entre outras. Note que **não é um objeto da classe `Sensor` que irá permitir obter valores** dos sensores;

3. A `SensorEvent`, que é **a classe que permite**, no fundo, **obter valores em estado bruto (*raw*) de determinado sensor**. Um evento desta classe **inclui informação como os valores, o tipo de sensor que gerou esses dados, a precisão e o momento** (selo temporal) em que foram adquiridos;

4. Finalmente, a `SensorEventListener` é **a interface que pode ser implementada para automaticamente receber eventos quando os valores ou a precisão dos sensores mudam**.

Para já, fica claro que, para poder usar sensores numa aplicação, é **primeiro necessário instanciar o gestor de sensores**, que mais não seja para testar se o sensor a utilizar está disponível ou não. O excerto de código seguinte mostra precisamente a forma de instanciar o `SensorManager` e de **apurar se determinado sensor está ou não disponível** no dispositivo móvel onde a aplicação está instalada, **durante a própria execução**:

```
package pt.di.ubi.pmd.exsensors1;

import android.app.Activity;
import android.hardware.SensorManager;
```

```
import android.hardware.Sensor;
import android.widget.Toast;

public class SensorActivity extends Activity {
  private SensorManager oSM;

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

    oSM = (SensorManager)
      getSystemService(Context.SENSOR_SERVICE);
    if(oSM.getDefaultSensor(Sensor.TYPE_LIGHT) != null){
      Toast.makeText(
        this,
        "You HAVE a light sensor!",
        Toast.LENGTH_SHORT
        ).show();
    }
    else{
      Toast.makeText(
        this,
        "You DO NOT HAVE a light sensor!",
        Toast.LENGTH_SHORT
        ).show();
    }
  }
}
```

Repare nos **dois *imports* necessários** à compilação correta da aplicação e também na forma de instanciar um `SensorManager`, dado por:

```
private SensorManager oSM;
...
oSM = (SensorManager)
    getSystemService(Context.SENSOR_SERVICE);
```

A aplicação exemplificada antes, se executada, mostra uma mensagem *toast* a dizer `You HAVE a light sensor!` caso este sensor esteja disponível; ou `You DO NOT HAVE a light sensor!` no caso contrário.

Caso se queira **obter uma lista de todos os sensores disponíveis no sistema** em que a aplicação está instalada, pode recorrer-se ao **método** `getSensorList(Sensor.TYPE_ALL)`, conforme se mostra a seguir, que devolve um objeto da classe `List`:

```
List<Sensor> lDS = mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

A **declaração da lista anterior faz uso de *genéricos* em Java**, que basicamente **permitem comunicar ao compilador o tipo dos objetos que esta classe (`List`) vai conter**. Significa que sempre que um objeto for obtido desta lista, *e.g.*, fazendo `lDS.get(1)`, **já não será necessário fazer *cast* do mesmo para que este seja considerado um objeto da classe `Sensor`**. Note que uma `List` pode albergar objetos de quaisquer classe, ou até vários objetos de classes diferentes simultaneamente (o que não é o caso). No exemplo dado, já é sabida a classe dos objetos que irá guardar, pelo

que é **mais seguro e cómodo defini-los imediatamente recorrendo à notação que usa parêntesis angulares**.

Atente ainda no código da Atividade mostrada antes, nomeadamente na linha com `oSM.getDefaultSensor(Sensor.TYPE_LIGHT)`. **O método `getDefaultSensor(.)` devolve (ou tenta devolver) um objeto da classe `Sensor`**, embora não tenha sido usado explicitamente no exemplo. **Caso** o sensor específico **não exista, o método devolve `null`, caso exista um ou mais, o método devolve o primeiro que encontrar**. Nas ocasiões em que existe mais do que um sensor de determinado tipo (*e.g.*, sensor de luz), pode usar-se o método `getSensorList()`, como antes, mas especificando o tipo, em vez de os requisitar a todos:

```
List<Sensor> lDS = mSensorManager.getSensorList(Sensor.TYPE_LIGHT);
```

A documentação oficial do Android constituirá sempre o melhor recurso para se obter a ideia de quais os sensores suportados pela plataforma[2]. Contudo, para referência, inclui-se a seguir **uma lista de alguns dos tipos de sensores mais importantes e disponíveis atualmente**:

- `TYPE_ACCELEROMETER`, *hardware-based*, que **mede a aceleração em m/s^2 aplicada ao dispositivo em <u>três</u> eixos** (x, y, z) e que pode ser usado para **detetar movimento, vibrações**, etc.;
- `TYPE_AMBIENT_TEMPERATURE`, *hardware-based*, que **mede a temperatura ambiente** em graus Celsius (^{o}C);
- `TYPE_GRAVITY`, *hardware-* ou ***software**-based*, que **mede a força da gravidade** em m/s^2 aplicada a cada um dos eixos (x, y, z), e que também pode ser usado para detetar movimento, vibrações, etc.;
- `TYPE_GYROSCOPE`, *hardware-based*, que **mede o rácio de rotação em rad/s para cada um dos <u>três</u> eixos** (x, y, z), e que pode ser usado para **detetar movimento** em termos de rotação;
- `TYPE_LIGHT`, *hardware-based*, que **mede a intensidade da luz no ambiente (iluminação)** em lx, e que pode ser usado, por exemplo, para **controlar a luminosidade do ecrã**;
- `TYPE_LINEAR_ACCELERATION`, *hardware-* ou ***software**-based*, que mede aceleração em m/s^2 aplicada a cada um dos eixos (x, y, z), mas **excluindo a força da gravidade**;
- `TYPE_MAGNETIC_FIELD`, *hardware-based*, que **mede o campo eletromagnético do ambiente** em que o dispositivo se insere para **cada um dos eixos** (x, y, z);
- `TYPE_PRESSURE`, *hardware-based*, que mede **a pressão do ar** em mbar;

[2] Ver http://developer.android.com/reference/android/hardware/Sensor.html.

- `TYPE_PROXIMITY`, *hardware-based*, que mede **a proximidade de um objeto**, em cm, relativamente ao local onde o sensor físico está colocado (normalmente está colocado na parte superior do ecrã). Este sensor é, por exemplo, usado por **aplicações que querem saber se o telefone está junto ao ouvido** ou não (*e.g.*, a aplicação de gestão de chamadas de voz pode fazer uso desta funcionalidade para **desativar o ecrã tátil**, evitando que sejam pressionadas funcionalidades acidentalmente);
- `TYPE_RELATIVE_HUMIDITY`, *hardware-based*, que **mede a humidade relativa do ar no ambiente** em que o dispositivo móvel se insere, **em percentagem (%)**. Os valores devolvidos por este sensor são tipicamente **usados para determinar o ponto de orvalho**.

Repare que **os vários tipos de sensores estão definidos, por comodidade, como *Strings* estáticas na classe** `Sensor`.

4 A Classe `Sensor`

O método `getDefaultSensor(Sensor.TYPE_LIGHT)`, já incluído no exemplo da secção anterior, devolve uma instância da classe `Sensor` que pode ser atribuída, caso o objeto haja sido declarado antes. O exemplo seguinte mostra precisamente essa atribuição, elaborando no que já foi discutido, e adicionando mais detalhes discutidos em baixo:

```
package pt.di.ubi.pmd.exsensors1;

import android.app.Activity;
import android.hardware.SensorManager;
import android.hardware.Sensor;
import android.util.Log;

public class SensorActivity extends Activity {
  private SensorManager oSM;
  private Sensor oL;

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

    oSM = (SensorManager)
      getSystemService(Context.SENSOR_SERVICE);
    oL = oSM.getDefaultSensor(Sensor.TYPE_LIGHT);
    if(oL != null){
      String sName = oL.getName();
      String sVendor = oL.getVendor();
      float fP = oL.getPower();
      int iVersion = oL.getVersion();
      Log.i("SENSORACTIVITY",
        "Sensor found! Specs - Name=" + sName +
        " Vendor=" + sVendor +
        " Power=" + fP +
```

```
            " Version=" + iVersion
            );
        }
        else{
          Log.e("SENSORACTIVITY","Sensor not found!");
        }
    }
}
```

Como poderá reparar, desta feita, o objeto `oL` foi declarado como sendo da classe `Sensor`, e privado à `SensorActivity`, para depois lhe ser atribuído o *output* do método `getDefaultSensor(.)`. Para melhorar a legibilidade do código e poupar espaço, neste exemplo é usada a classe `Log`, ao invés da `Toast`. Caso esta aplicação seja executada num dispositivo Android com sensor de luminosidade, o código guardado por `(oL != null)` é executado, sendo impressos no *logcat* algumas das especificações do sensor, nomeadamente **a *String* que concretiza o seu nome, o nome do fabricante, a versão e a energia que o sensor consome**. Conforme prometido, é **o objeto da classe `Sensor` que permite obter estas informações, mas não os valores em bruto que são por ele medidos**.

Outros dois métodos que a classe disponibiliza e que podem ser **úteis em determinados contextos são**:

- `getMaxDelay()`, que **devolve o tempo máximo, em microssegundos, entre dois eventos do sensor** (apenas definido para sensores que fazem medições de forma contínua);

- `getResolution()`, que **devolve a precisão do sensor na unidade para a qual está definido**.

Enquanto que a **utilidade desta classe** ainda não está suficientemente explicita neste ponto, é possível elaborar um pouco mais neste aspeto imediatamente. O facto é que **as aplicações podem ser otimizadas para diferentes tipos de sensores ou até para fabricantes, bem como para a precisão de cada um**. Assim, pode ser **útil obter estes dados em tempo de execução, e redirecionar o fluxo de acordo** com os mesmos, e antes de usar os seus valores. Para concretizar esta discussão com um exemplo, pode imaginar-se uma aplicação que precisa saber se o utilizador agita o telemóvel em determinada situação. Este evento pode ser detetado por um sensor de gravidade (melhor solução) ou por um acelerómetro, e o código terá de ser diferente na presença de um ou de outro, pelo que convém testar qual deles está disponível. Em alguns casos, a versão ou fabricante de determinado sensor, bem como a sua precisão, podem também ser cruciais para os objetivos da aplicação. *E.g.*, a experiência de utilização de um jogo de condução que use o sensor de rotação será tanto melhor quanto mais preciso este for.

5 A Interface `SensorEventListener`

Até aqui, a explicação focou-se na obtenção do gestor de sensores e na verificação da existência e características destes últimos. Esta secção elabora em como se podem efetivamente obter valores dos mesmos.

A **obtenção de valores de sensores requer sempre que se registe um *Listener* para o sensor do tipo desejado** (*e.g.*, `TYPE_LIGHT`) **através do método `registerListener()`, providenciado pelo gestor de sensores** (*i.e.*, pelo `SensorManager`). O *Listener* que é registado **é um objeto de uma classe que obrigatoriamente implementa a interface `SensorEventListener`**, cuja definição determina também que **os métodos `onAccuracyChanged()` e `onSensorChanged()` estejam concretizados neste objeto**. De uma forma geral, pode estruturar-se o procedimento da seguinte forma:

1. **Declaram-se os objetos** das classes `SensorManager` e `Sensor`;
2. **Obtém-se a instância do Gestor de Sensores** a correr no sistema conforme discutido acima;
3. **Obtém-se a instância do sensor pretendido**, através do método `getDefaultSensor(int)`, também como já foi discutido antes;
4. **Cria-se uma nova classe que implemente a interface `SensorEventListener` ou**, alternativamente, **define-se que a própria Atividade implementa esta classe** (neste caso, os métodos são definidos dentro da classe que estende a `Activity` – ver exemplo seguinte);
5. **Implementam-se os dois métodos da interface `SensorEventListener`**, nomeadamente o `onAccuracyChanged()` e o `onSensorChanged()`;
6. **Instancia-se um novo objeto da classe referida** (se o objeto for a própria Atividade, este passo não é necessário);
7. Usa-se o método `registerListener()`, disponível no objeto da classe `SensorManager`, para **registar o consumidor** (*Listener*).

Para além dos passos enunciados, não deve ser esquecido que é também necessário importar todas as classes necessárias, bem como a interface.

O método `registerListener(.,.,.)` aceita 3 parâmetros de entrada: **um objeto da classe que implementa `SensorEventListener`, o objeto da classe `Sensor` e um inteiro, que determina a frequência com que o sistema deve tentar entregar os eventos do sensor a esta aplicação**. Este último valor é meramente indicativo, já que os eventos podem depois ser entregues mais ou menos rápido, embora o sistema tente

responder com a celeridade pedida. Este método **devolve verdadeiro caso o registo tenha sido bem sucedido, e falso no caso contrário**.

Os **dois métodos da interface** `SensorEventListener` são os que irão conter **a lógica computacional que permite atuar sobre os valores do sensor**, e **é o próprio sistema Android que os invoca** aquando da entrega dos valores ou alterações nos sensores:

- **O método** `onAccuracyChanged(.,.)` **recebe dois parâmetros e é invocado sempre que a precisão do sensor sofre alteração** (*e.g.*, por ter sido calibrado). O primeiro parâmetro é um objeto da classe `Sensor`, e define o próprio sensor em que se deu a alteração, enquanto que **o segundo parâmetro é um inteiro que informa a nova precisão**. O inteiro pode ser um dos 3 valores definidos estaticamente na classe `SensorManager`, nomeadamente `SENSOR_STATUS_ACCURACY_HIGH`, `SENSOR_STATUS_ACCURACY_LOW` ou `SENSOR_STATUS_ACCURACY_MEDIUM`;
- **O método** `onSensorChanged(.)` **recebe apenas um parâmetro e é invocado quando o sensor reporta novos valores**. Em algumas APIs, este método também é invocado quando o *Listener* é registado, para se obter uma leitura imediata dos valores. **O parâmetro é um objeto da classe** `SensorEvent`, já discutido anteriormente. Normalmente é possível **obter os valores em bruto do sensor acedendo ao *array*** `values` **desse** `SensorEvent` (*e.g.*, `event.value[0]` corresponde ao valor medido no eixo dos xx no sensor giroscópio).

De maneira a concretizar melhor o procedimento desrito antes, inclui-se, em baixo, um exemplo de uma aplicação Android com uma única Atividade que implementa os métodos da interface na própria Atividade, após o qual se elabora nos detalhes mais importantes para o entendimento deste assunto:

```
package pt.di.ubi.pmd.exsensors2;

import android.app.Activity;
import android.hardware.SensorManager;
import android.hardware.Sensor;
import android.hardware.SensorEventListener;
import android.hardware.SensorEvent;
import android.util.Log;

public class SensorActivity extends Activity implements SensorEventListener
    {
  private SensorManager oSM;
  private Sensor oL;
  private boolean bMessage = false;

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

```
    oSM = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    oL = oSM.getDefaultSensor(Sensor.TYPE_LIGHT);
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    Toast.makeText(
        this,
        "Accuracy has changed!",
        Toast.LENGTH_SHORT
        ).show();
  }

  @Override
  public final void onSensorChanged(SensorEvent event) {
    float lux = event.values[0];
    // 0.27 -- 1.0 lux Full moon on a clear night
    if( ( lux < 1.0 ) && (lux > 0.27))
      if( !bMessage ){
        Toast.makeText(
            this,
            "What a beautiful full moon!",
            Toast.LENGTH_SHORT
            ).show();
        bMessage = true;
      }
  }

  @Override
  protected void onResume() {
    super.onResume();
    mSensorManager.registerListener(this, oL, SensorManager.
        SENSOR_DELAY_NORMAL);
  }

  @Override
  protected void onPause() {
    super.onPause();
    oSM.unregisterListener(this);
  }
}
```

O funcionamento desta aplicação é simples de explicar. A Atividade regista um *Listener* para o sensor do tipo `TYPE_LIGHT`. **Na primeira vez em que** os valores da luminosidade no sensor mudam para o intervalo entre 0.27 e 1.0, a aplicação lança uma mensagem `Toast` a dizer `"What a beautiful full moon!"`. Para evitar que esta mensagem apareça mais vezes, uma variável de controlo (`bMessage`) é então colocada a `true`, impedindo que tal se repita.

Note que **foi decidido**, neste exemplo, **que seria a própria Atividade** `SensorActivity`

a implementar a interface `SensorEventListener`. Assim, **os dois métodos da interface estão definidos logo a seguir ao** `onCreate()`, e **os métodos** `registerListener(.,.,.)` **e** `unregisterListener(.)` **aceitam**, como parâmetro, **a *keyword* `this`, que lhes injeta a própria classe que os invoca**. Dado que a classe `SensorActivity` não pode ser, dado o seu objetivo, abstrata (`Abstract`), esta **é obrigada a implementar ambos os métodos da interface**, ainda que pudesse deixar um deles vazio. Por exemplo, a implementação do método `onAccuracyChanged()` seguinte era válida:

```
@Override
public final void onAccuracyChanged(Sensor sensor, int accuracy) {
}
```

Também de enfatizar são os locais onde o *Listener* é registado ou onde o seu registo é eliminado. No exemplo, a primeira operação acontece no método `onResume()`, enquanto que a segunda acontece no ponto simétrico ao primeiro, *i.e.*, no método `onPause()`. **É importante que a aplicação contenha os métodos tanto para ativar, como para desativar os sensores nos locais certos, e recomendado que esta os desative quando não forem necessários**, já que **o seu funcionamento consome energia**. Na verdade, **o Android não desativa automaticamente os sensores quando uma aplicação é pausada ou mesmo quando o ecrã é desligado**, já que estes podem ser necessários por alguma aplicação. No exemplo anterior, e visto que não necessitarmos do sensor enquanto a aplicação está pausada ou parada, o seu registo é eliminado no método `onPause()`.

O **método de eliminação de registo** ainda não havia sido mencionado anteriormente:

```
private SensorManager oSM;
  ...
@Override
protected void onPause() {
  super.onPause();
  oSM.unregisterListener(this);
}
```

O `unregisterListener(.)`, também providenciado pela classe `SensorManager`, **aceita, como único parâmetro de entrada, o objeto que implementa a interface** `SensorEventListener` (que, em cima, era a própria Atividade, logo o uso da *keyword* `this`).

6 A Classe `SensorEvent`

A classe `SensorEvent` **não disponibiliza qualquer método para além dos que herda da sua superclasse** (`Object`), contudo, **todos os seus atributos são públicos**, e é dessa forma que os disponibiliza à chegada, no método `onSensorChanged()`. Os **4**

atributos dos objetos desta classe são:

- `accuracy`, um inteiro (`int`) que determina a **precisão** da captura;
- `sensor`, um objeto da classe `Sensor` que identifica **o sensor onde se deu a captura**;
- `timestamp`, um inteiro (`long`) que representa **o momento, em nanossegundos**, em que se seu a captura;
- `values`, um vetor de decimais (`floats`) com **os valores da captura**.

O **tamanho e conteúdo do vetor de valores depende do tipo de sensor que está a ser usado**, e convém verificar a documentação para se saber quantos valores esperar. Para o sensor da luminosidade, por exemplo, o tamanho do vetor é 1, enquanto que para o rotacional ou gravitacional teria um tamanho de 3.

Convém referir ainda a forma como o sistema define os três eixos para sensores que medem valores para 3 dimensões. **O sistema define como o eixo dos xx como o que segue a linha horizontal do ecrã** (*i.e.*, o lado mais pequeno quando o dispositivo móvel está em modo retrato, ou o lado maior quando este está em modo paisagem); define **o eixo dos yy como aquele que segue a linha vertical do ecrã** (*i.e.*, o lado maior quando o dispositivo móvel está em modo retrato, ou o lado menor quando este está em modo paisagem); e, finalmente, define **o eixo dos zz como sendo aquele que aponta para o céu quando o dispositivo está deitado com as costas viradas para o chão**.

7 Testar a Implementação

Note que **não é normalmente boa ideia testar as aplicações que utilizem sensores em emuladores Android**, já que **estes não simulam**, tipicamente ou em toda a plenitude, **os *outputs* que aqueles produzem** de uma forma satisfatória. Neste caso, **o ideal será testar a aplicação num dispositivo físico** ou, opcionalmente, certificar-se previamente que o dispositivo virtual que vai utilizar está **apetrechado com simuladores de sensores** adequados ao seus testes.

Parte II

Guias Práticos Laboratoriais



17 Programação de Aplicações Android num IDE

Sumário

Introdução ao ambiente de desenvolvimento integrado para aplicações Android. Discussão de diversos conceitos e termos do jargão da área da programação para dispositivo móveis. Criação e teste de um emulador. Criação de uma aplicação para Android e breve abordagem inicial à anatomia de uma aplicação.

Summary

Introduction to the integrated development environment for Android applications. Discussion of several terms of the jargon of the area of programing for mobile devices. Creation and test of an emulator. Creation of an application for Android and brief initial approach to the anatomy of an application.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o IDE Eclipse apedrechado do *plugin* para desenvolvimento de aplicações Android, bem como com o respetivo SDK, uma versão e uma imagem da plataforma Android[1]. Alternativamente, pode tentar a maior parte das tarefas com o Android Studio™[2] e com (ou sem) o Genymotion[3].

[1] Aquele que é conhecido por ADT *Bundle* pode ser descarregado de `http://developer.android.com/sdk/index.html`.

[2] Mais em `http://developer.android.com/sdk/installing/studio.html`.

[3] Mais em `http://www.genymotion.com/`.

1 IDE Eclipse, AVD e SDK

Tarefa 1

A sua primeira tarefa consiste em executar o IDE Eclipse. Após procurar aplicação através do menu iniciar, e chegar à conclusão de que foi tempo perdido, abra o explorador de ficheiros e procure a pasta relativa ao eclipse na raiz (`C:\`).

O arranque da aplicação pode levar alguns momentos, pelo que considere responder às seguintes questões entretanto.

Q1.: O que significa, para si, IDE?

I _____

D _____

E _____

Agora sem ir consultar a Internet: **Q2.: e o acrónimo SDK, é abreviatura de que?**
☐ *Simple Developement Kit* ☐ *Simple Delivery Kit* ☐ *Software Developement Kit*
☐ *Super Donkey Kong* ☐ *Sandakan Airport.*

Q3.: Qual das seguintes opções define corretamente SDK?
☐ É o nome dado a um aeroporto doméstico de Sandakan, cidade natal de Son Goku.
☐ É o nome de um jogo de video, desenvolvido pela Rare para a consola Super Nintendo.
☐ É um conjunto de ferramentas de desenvolvimento de software que permitem a criação de aplicações para um dado pacote, sistema de software, plataforma de hardware, computador, consola de jogos de vídeo, sistema operativo ou outra plataforma de desenvolvimento semelhante.

Tarefa 2

O desenvolvimento de aplicações para dispositivos móveis requer, normalmente, a utilização de emuladores desses dispositivos em software. A sua próxima tarefa consiste, por isso, na criação, execução e teste de um desses emuladores, neste caso em particular, para o SO Android. Antes de prosseguir, procure definir o acrónimo *Android Virtual Device* (AVD):

A _____ V _____ D _____

Dentro do ambiente Eclipse, expanda o menú Window e procure a opção AVD Manager. Na janela que resultar da ação anterior, selecione o botão New... para criar um novo emulador. Ser-lhe-ão pedidas algumas informações e escolha de opções. No nome (`name`),

coloque `Android Emulator 1` e na API alvo coloque a mais recente (*e.g.*, `Android 4.2`). Analise com calma, mas deixe a maior parte das outras configurações por defeito. Depois de configurar o emulador, carregue no botão Create AVD. Finalmente, selecione o dispositivo virtual que acabou de criar (que deve agora aparecer na lista) e execute-o, carregando em Start. O emulador vai demorar a arrancar, por isso seja bastante paciente, e considere responder às seguintes questões enquanto espera.

Q4.: Qual a versão mais recente já em utilização do SO Android?
☐ Android 2.2 ☐ Android 2.3 ☐ Android 3.1 ☐ Android π ☐ Android 3.2
☐ Android 4.4 ☐ Android 5 ☐ Android 5.1 ☐ Android 6

Q5.: Por curiosidade, qual a versão mais recente, e atualmente em utilização, do iOS?
☐ iOS 1 ☐ iOS 2 ☐ iOS 3 ☐ iOS 4 ☐ iOS 5 ☐ iOS 6 ☐ iOS 7 ☐ iOS 8
☐ iOS 9 ☐ iOS 10 ☐ iOS 11 ☐ iOS 12 ☐ iOS XP ☐ iOS ∞

Q6.: Já agora, escreve-se iOS ou IOS?
☐ E isso interessa?
☐ Escreve-se, obrigatoriamente, iOS.
☐ Escreve-se, obrigatoriamente, IOS.

Q7.: Há algum SO chamado IOS (com o 'i' capitalizado)?
☐ Pois existe, que engraçado!
☐ Claro que não, seria estranho existirem dois SOs diferentes com o mesmo acrónimo.

Q8.: Qual é o esquema utilizado pela Google™ para nomear as várias versões do sistema operativo Android?
☐ A designação de cada versão do SO é constituída por dois substantivos começados pela mesma letra e cujo primeiro é o nome de um animal os vários nomes seguem o alfabeto.
☐ A designação de cada versão do SO é constituída por vários substantivos e faz sempre lembrar um doce. A primeira letra da designação segue o alfabeto.
☐ A designação de algumas versões do SO é constituída por vários substantivos e faz sempre lembrar um doce. A primeira letra da designação segue o alfabeto.
☐ A designação de cada versão do SO é constituída pela palavra Android seguida de um número com uma casa decimal.

Q9.: Como se chama a versão mais recente, e em utilização, do SO Android?

Q10.: Qual é o núcleo base, sobre o qual o SO Android foi construído?
☐ Unix ☐ MSDOS ☐ Linux

Q11.: Por curiosidade, qual é o núcleo base do iOS?

☐ Unix ☐ MSDOS ☐ Linux

Tarefa 3

Depois do seu emulador arrancar experimente algumas das suas aplicações, nomeadamente a calculadora. Procure também aceder às suas definições e saber qual a versão do SO no emulador está a utilizar. **Q12.: A versão do emulador coincide com a que definiu anteriormente, aquando da sua criação?**
☐ Sim, coincide. ☐ Que estranho. Não coincide.

Depois de experimentar o emulador, desligue-o, fechando a janela.

2 Olá Planeta Terra

O objetivo desta parte do guia é construir e correr uma aplicação Android sem escrever qualquer trecho de código.

Tarefa 4

No Eclipse, selecione o menu File, seguido de New, e depois por Project. Na caixa de diálogo que se abre, selecione a pasta `Android`, seguido de `Android Application Project`, pressionando o botão Next no fim.

Na próxima janela ser-lhe-ão pedidas várias informações e configurações. Chame à sua aplicação `OlaMundo`, e ao seu projeto também. Note que o nome que deu à aplicação é aquele que eventualmente irá aparecer ma *Play Store*, caso a publique. O nome do projeto é apenas usado locamente e, portanto tem de ser único no seu âmbiente de trabalho. É possível que o Eclipse preencha alguns dados automaticamente, à medida que faz as suas configurações. No `Build SDK` escolha a API mais recente suportada e no SDK mínimo escolha API 8. De seguida escolha um ícone ao seu gosto ou deixe ficar o que lhe é apresentado por defeito. Escolha ainda a opção de criar uma `BlankActivity`. O nome desta atividade pode ser `OlaMundoActivity`. Deixe a maior parte das opções por defeito (*e.g.*, `Navigation Type` deve estar ajustado para `None`). O campo `Title`, que é o texto que aparece na barra de título da aplicação, pode ser `Aplicação Olá Mundo`. No final clique em Finish.

Tarefa 5

Pode executar a aplicação num dispositivo verdadeiro ou no emulador que à pouco experimentou. Para isso, selecione o menu Run e a opção Run. Se tudo correu bem deverá

aparecer uma janela com opções de execução, onde deverá selecionar Android Application e pressionar o botão OK.

O Eclipse deve arrancar o AVD, instalando a aplicação automaticamente. Prepare-se para aguardar um algum tempo novamente. Quando a aplicação executar, feche-a carregando no Esc, e volte a executá-la, selecionando-a na lista de aplicações. Enquanto espera considere executar as tarefas seguintes e responder às questões que contêm.

Tarefa 6

Expanda a pasta do projeto que acabou de criar no `Project Explorer`, que deve estar na parte esquerda da interface. Verifique que foram criadas várias pastas para esta simples aplicação.

Q13.: Onde está contido o código JAVA da aplicação?
☐ Na pasta `bin`. ☐ Na pasta `res`. ☐ Na pasta `src`. ☐ Na pasta `diplomatica`.
☐ Na pasta `JAVA`.

Q14.: Em que pasta está contido o ficheiro `AndroidManifest.xml`?
☐ Na pasta `bin`. ☐ Na pasta `res`. ☐ Na pasta `src`. ☐ Na pasta `pinterest`.
☐ Na pasta `JAVA`.

Tarefa 7

Clique duas vezes no ficheiro `AndroidManifest.xml`. Se tudo correu bem, deve ter à sua frente um editor específico para ficheiros *AndroidManifest*.

Q15.: Quais das seguintes informações pode definir neste ficheiro?
☐ A versão da aplicação. ☐ O *layout* da aplicação. ☐ O nome da aplicação.
☐ As permissões e acessos. ☐ A API mínima. ☐ A API alvo.

Q16.: O que significa o X do acrónimo XML?
☐ *eXtensible* ☐ *Cross(X)* ☐ *Language* ☐ *HyperteXt* ☐ *Xmen* ☐ Linux

Tarefa 8

A última tarefa desta aula pede-lhe que abra o ficheiro `AndroidManifest.xml` com um editor de texto. Com base no que vê, responda às questões que se seguem.

Q17.: Qual o elemento raiz deste XML?
☐ `application` ☐ `manifest` ☐ `android` ☐ `activity`
☐ O que é um *elemento raiz*?

Q18.: Qual o elemento pai de `activity`?
☐ `application` ☐ `manifest` ☐ `android` ☐ `activity`
☐ O que é um *elemento pai*?

Q19.: O elemento `activity` tem algum elemento filho?

☐ Sim, tem, nomeadamente _____.
☐ Não, não tem.

Q20.: De que forma é que é descrito o nome do pacote (*package*) no manifesto da aplicação?
☐ Em forma de elemento. ☐ Em forma de atributo.
☐ Em forma de dados/texto. ☐ Essa informação não está lá!
☐ Ahh, está lá assim escrito: `package="com.example.helloword"`. Era isto?

Q21.: De acordo com o `manifest`, por quantas atividades é constituida a aplicação que acabou de criar?
☐ $-\sqrt{-1}$. ☐ 0. ☐ 1. ☐ 2. ☐ 3. ☐ π.

O ficheiro `AndroidManifest.xml` deve conter alguns atributos definidos com `@`, nomeadamente o icone e o tema da aplicação. Deve encontrar sequências de caracteres parecidas com `@drawable/ic_launcher` ou `@string/app_name`. **Q22.: O que é que isso significa? Consegue dizer exatamente o tema ou o *path* (caminho) do icone no disco?**[4]
☐ Sim, consigo: _____

_____

☐ Não, nem isso consigo.

Nota: nesta primeira aula foi feita uma introdução ao tema da programação para Android usando apenas o ambiente gráfico. Contudo, é possível que nas próximas aulas seja mais focada a possibilidade de programar em ambiente de linha de comandos, de modo a exercitar a destreza sem distrações.

[4] Os 5 centimos da praxe estão associados a esta questão.



18 Agora sem Mãos!

Sumário

Exploração das ferramentas fornecidas com o *Software Development Kit* (SDK) para Android. Exploração da ferramenta de linha de comandos para automatização do processo de compilação de aplicações conhecida por Gradle™. Introdução ao processo de preparação e instalação de aplicações para a plataforma Android, recorrendo para isso a ferramentas da linha de comandos.

Summary

Analysis of the tools provided with the Software Development Kit (SDK) for Android. Analysis of the library and tool for automatizing the build process of applications known as Gradle™. Introduction to the process of building and installing of applications for the Android platform, resorting to command line interface tools to emphasize the steps of the process.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Instalação do Apache Ant

Tarefa 1

Logo que a sua máquina estabilize após a autenticação, abra o Android Studio™ e crie um projeto básico que irá usar para testes adiante. Considere as seguintes sugestões aquando da configuração inicial desse projeto:

- Nome da aplicação/projeto — `Testingradle`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.testingradle`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja perguntado, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `Main Activity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `main_activity.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade.

Q1.: Para efeitos de registo, em que pasta fica o projeto criado pelo comando anterior?

Tarefa 2

Aproveite ter o Android Studio™ aberto para imediatamente inicializar um dispositivo virtual Android (AVD) seguindo o menu Tools > (Android >) AVD Manager ou através do ícone disponível na barra de ferramentas.

Nota: caso não tenha nenhum dispositivo virtual definido, crie um. Pode consultar o guia laboratorial 1 para mais informação sobre este assunto.

2 Explorando o Gradle™

Tarefa 3

Por vezes, é necessário configurar algumas variáveis de sistema (que guardam o caminho de diretorias importantes em memória) quando se instalam SDKs, como o caso do SDK Android. Muitas instalações integradas escondem estas configurações. Esta tarefa consiste em verificar se ao menos a variável `ANDROID_HOME` (uma das variáveis configuradas automaticamente pela instalação do Android Studio™) está bem definida. Para isso:

1. Aceda ao *Android Studio™* e à ferramenta *SDK Manager* (disponível tipicamente em *Tools*);

2. Abra um terminal (Linux) ou consola (Windows) e escreva
   `$ echo $ANDROID_HOME` (Linux); ou
   `$ echo %ANDROID_HOME%` (Windows);

3. Verifique se a diretoria que é impressa na linha de comandos é igual à que está definida na ferramenta *SDK Manager*.

Caso as configurações difiram, proceda à configuração correta da variável de sistema através de um comando semelhante a um dos seguintes:
`$ export ANDROID_HOME=raiz_do_Android_SDK` (Linux)
ou `$ set ANDROID_HOME=raiz_do_Android_SDK` (Windows).

Atente na última instrução. **Q2.: O que é que esta instrução está a fazer exatamente?**
☐ Está a definir a variável `ANDROID_HOME`.
☐ Está a colocar o número 7 (`set`) na variável `PATH`.
☐ Está a especificar o valor da variável `ANDROID_HOME`.

Note que, em Windows, a definição de uma variável só é válida no contexto da consola em que é feita. Precisa definir a variável **em todas** as consolas que tiver de usar.

Tarefa 4

Se fez as tarefas todas corretamente já não deve precisar mais do Android Studio™; por isso, encerre-o.

Tarefa 5

Abra um *browser* Web, dirija-se até `https://gradle.org`, e navegue neste site para obter alguma da informação que lhe permite responder às questões seguintes. Encontre maneira de fazer o *download* do `zip`[1] de instalação do `gradle` e descomprima-o para uma pasta à sua escolha. Considere também navegar até à diretoria do Android Studio™, onde deve encontrar uma pasta chamada `gradle`.

Q3.: Qual é a versão mais recente desta ferramenta de compilação?

☐ Versão 1+ ☐ Versão 2+ ☐ Versão 3+ ☐ Versão 4+ ☐ Versão 5+
☐ Versão 6+ ☐ Versão 7+ ☐ Versão 8+ ☐ Versão X

Q4.: Em que linguagem é que o Gradle™ foi implementado?

☐ Go Language ☐ ANSI C ☐ C++ ☐ JAVA
☐ Groovy ☐ Swift ☐ Python ☐ Kotlin

A ferramenta `gradle` foi especialmente desenhada para *construir* aplicações Java. **Q5.: É possível usá-la para automatizar o processo de compilação e construção para outras linguagens?**

☐ Sim, é.
☐ Sim, é, mas é muito chato e requer muita adaptação.
☐ Não, não é.

Q6.: Na expressão "construir uma aplicação Android" (e *grosso modo*), o que é que lhe parece significar a palavra *construir*?

☐ Ato ou efeito de colocar uns tijolos em cima uns dos outros, interligados tecnologicamente por massa.
☐ Compilar o código fonte.
☐ Compilar o código fonte, juntar e enumerar recursos, e empacotar binários, recursos e descrições, para além de outros artefactos.
☐ Construir é, neste contexto, algo filosófico, e significa *edificar*, no sentido de fazer o bem.

Q7.: *(Just for the sake of it)* Qual o símbolo do *gradle*?

☐ Um batráquio. ☐ Um elefante. ☐ Um crocodilo. ☐ Um Broncossauro?

Tarefa 6

Abra um terminal (Linux) ou uma consola (Windows – *e.g.*, carregue no menu iniciar seguido de *run* (tecla R), escreva `cmd` e pressione *Enter*). Navegue até à pasta onde antes colocou o `gradle` usando comandos como `$ cd C:`, `$ cd ..`, `$ cd DIR_NAME` ou `$ dir`. Deve chegar a um ponto em que a estrutura interna da diretoria é semelhante ao que se apresenta a seguir:

[1] `https://gradle.org/next-steps/?version=6.6.1&format=all`

```
gradle
  +----- NOTICE, LICENSE, etc.
  +----- bin
  +----- docs
  +----- initi.d
  |
  +----- _________ // < PREENCHER!
  |
  +----- media
  |
  +----- _________ // < PREENCHER!
  |
  +----- _________ // < PREENCHER!
```

Q8.: Em que diretoria é que está guardado o comando `gradle`?
☐ `bin` ☐ `lib` ☐ `etc` ☐ `java`

Q9.: O que é que contém a diretoria `lib`?
☐ Contém binários relativos a comandos disponibilizados pela ferramenta.
☐ Contém ficheiros `.jar` relativos à implementação do Gradle™.
☐ Contém o código JAVA relativo à implementação do Gradle™.
☐ Contém uma coleção de livros raros, antes incluídos na Biblioteca de Alejandria.
☐ Contém a documentação da ferramenta.

3 Explorar o SDK Android

Para a parte restante deste guia vai precisar de ferramentas disponibilizadas pelo SDK Android. Assim, sugere-se que navegue até à diretoria que alberga essas ferramentas e a explore. Em princípio, essa diretoria estará dentro da pasta pessoal do utilizador (*e.g.*, em `Users\Aluno\AppData\Local\Android\`), dentro de uma diretoria denominada de `sdk`. Preencha a árvore seguinte e responda às questões que se lhe seguem com base no que observar durante esta análise.

```
Sdk
  +--- build-tools
  |
  +--- _________________ // < PREENCHER
  |
  +--- fonts
  |
  +--- patcher
```

```
|
+--- _________________ // < PREENCHER
|
+--- ____________-_______ // < PREENCHER
|
+--- skins
|
+--- sources
|
+--- system-images
|
+--- ._________________ // < PREENCHER
|
+--- _________________ // < PREENCHER
|
+--- .knownPackages
```

Q10.: Em que pasta está a ferramenta `adb`?

☐ `build-tools` ☐ `system-images` ☐ `temp` ☐ `platforms`
☐ `android` ☐ `etc` ☐ `platform-tools`

Q11.: Em que pasta pode ser encontrada a ferramenta `sdkmanager`?

☐ `build-tools` ☐ `system-images` ☐ `temp` ☐ `platforms`
☐ `android` ☐ `tools` ☐ Não está cá. ☐ `tools\bin`

Q12.: É possível encontrar uma *shell* para bases de dados SLQLite3 em alguma das diretorias antes enunciadas?

☐ Sim, é, nomedamente na diretoria _____.
☐ Não, não é!

4 Preparação de um Pacote de Instalação Android via Linha de Comandos

Tarefa 7

Apesar de já ter criado um projeto Android para testes antes, é normalmente útil averiguar antecipadamente qual a versão da plataforma Android destino para essa aplicação. É ainda mais útil identificar as versões dos sistemas para os quais foram descarregadas imagens e configurados AVDs, que pode ser eventualmente conseguido de forma eficiente através de um comando do gestor de AVD (*Android Virtual Device*). Talvez a resolução das duas questões seguintes ajude a executar a tarefa de obter essa informação:

Q13.: Em que pasta pode ser encontrada a ferramenta `avdmanager`?
☐ Na mesma pasta que a ferramenta `sdkmanager`.
☐ Na pasta `tools\bin`
☐ Na pasta *Mr. Bean*.

Q14.: Qual dos seguintes comandos é que lhe mostra quais as versões disponíveis no seu sistema?
☐ `$ ./avdmanager list targets`
☐ `$ ./avdmanager listing`
☐ `$ ./gradle tasks`
☐ `$ ./avdmanager, please show me the available targets!  Pretty please!`
☐ `$ ./gradle list`

Q15.: Como traduziria o comando para Português?

Verifique se tem a versão da API 25 do Android disponível e anote o `id` da versão mais recente.

Q16.: Já agora, e revisitando o tema da secção *Preliminares*, acha que é possível colocar um dispositivo virtual a correr sem abrir o Android Studio™ ou o AVD *Manager*?
☐ Não.
☐ Sim, emitindo uma reza poderosa à base de sais do mar negro.
☐ Sim, mas não acumulo os conhecimentos necessários para o fazer para já!
☐ Sim. Navegando até à diretoria onde estão as ferramentas do Android SDK e usando o comando `emulator`:

```
$ cd android-sdk/emulator
$ ./emulator -avd NOME_DISPOSITVO
```

☐ Sim. Navegando até à diretoria onde estão as ferramentas do Android SDK e usando o comando `android avd`:

```
$ cd android-sdk/tools
$ ./android avd
```

Nota: use o comando `$ ./avdmanager list avd` para ver os dispositivos virtuais disponíveis.

Tarefa 8

Neste momento e se seguiu bem o guia laboratorial, o código fonte e estrutura base de uma aplicação Android, usando o *template* do Gradle™, devem estar criados. O próximo passo consiste, naturalmente, em compilar o código. A ferramenta `gradle` é alimentada por um ficheiro que lhe fornece algumas das informações necessárias ao processo de compilação. **Q17.: Como se chama esse ficheiro?**

Dica: deverá encontrá-lo algures na pasta do projeto...
☐ `build.gradle` ☐ `xml.xml` ☐ `fetch.gradle` ☐ `build.xml`
☐ Sei lá agora como é que o ficheiro se chama!

Compile o projeto, situando-se na sua raiz na consola e emitindo o seguinte comando :
`$ gradlew assembleDebug` ou (alternativamente)
`$ gradlew assembleDebug lint`

Caso não tenha sido bem sucedido, pode dever-se a alguns *bugs* detetados e reportados pelo Lint, que é uma ferramenta de reporte de falhas no Android. Se pretender analisar o relatório desta ferramenta, considere aceder ao seguinte ficheiro via terminal:
`$ notepad.exe app\build\reports\lint-results.xml`.

Se tudo correr bem na execução desta tarefa deve ter conseguido compilar o seu projeto com o comando de compilação do Gradle™. Um projeto Android é compilado para um arquivo `apk`. **Q18.: O que abrevia exatamente `apk`?**

Q19.: Em que sub-diretoria é que o arquivo resultante foi colocado?
☐ `build` ☐ `gen` ☐ `res` ☐ `src` ☐ `libs`

Q20.: Como se chama o arquivo resultante?
☐ `HelloWorld.apk` ☐ `Android.apk` ☐ `Apk.apk` ☐ `app-debug.apk`

Todas as aplicações que instala num dispositivo com SO Android real têm obrigatoriamente de vir na forma de um arquivo `.apk`. **Q21.: Acha que é possível instalar esta aplicação de teste num dispositivo comum com SO Android?**
☐ Sim, claro. ☐ Hummm... algo me diz que não!

Note que acima foi dito que o projeto Android é compilado para um arquivo. **Q22.: O que é um arquivo?**
☐ É um ficheiro compactado.
☐ É o mesmo que uma diretoria.
☐ É um ficheiro composto por outros ficheiros e meta-dados.

Tarefa 9

Apesar do guia já estar extenso, o número de comandos utilizado para compilar um projeto Android foi apenas um. A próxima fase consiste em testar a aplicação num dispositivo real ou virtual. Para isso, abra outra consola e navegue até à diretoria do SDK, nomeadamente à sub-diretoria que contém a ferramenta `adb` (caso não tenha definido anteriormente essa diretoria na `PATH` do ambiente). Uma vez no local certo, emita o comando incluído a seguir: `$ adb devices`
Se tem o dispositivo virtual a correr, deve vê-lo listado na consola. Caso não haja nenhum

listado, precisa certificar-se de que o colocou a correr, conforme sugerido no início deste guia.

Note que, se tiver um dispositivo físico com SO Android ligado ao computador por USB, este também deve aparecer listado embora, por vezes, seja necessário ativar algumas funcionalidades no SO.

Q23.: Quais das seguintes palavras fazem parte da expansão do acrónimo ADB?

☐ Android ☐ *Asian* ☐ *Alien* ☐ *Development* ☐ *Debug*
☐ *Determined* ☐ *Ball* ☐ *Bridge* ☐ Berlinde.

Tarefa 10

Para instalar a aplicação, e com a *path* da *prompt* ainda dentro da diretoria `platform-tools` (dentro de `sdk`), emita o comando seguinte:

```
$ adb install C:\Users\aluno\PASTA-COM-PROJETOS \Testingradle\app\build\
outputs\apk\debug\app-debug.apk
```

Note que a aplicação que criou está numa versão de *debug*, o que significa que está assinada digitalmente com um certificado auto-assinado criado pelo SDK aquando da sua instalação. As versões *debug* funcionam apenas em dispositivos cujo SO Android tenha sido especificamente configurado para aceitar aplicações em modo *debugging*. Os dispositivos virtuais criados durante estas aulas já têm essa configuração. Se quiser testar uma versão de *debug* num dispositivo físico comum, terá provavelmente de configurá-lo antes de poder instalar a aplicação. Em versões do SO modernas (*e.g.*, a partir da 4.2), pode ligar o modo *debugging* através de um procedimento parecido ao seguinte:

1. No dispositivo móvel com SO Android, navegue até às *Definições de Sistema* (*System Settings*) e depois até *Acerca do Telemóvel* ou *Acerca do Tablet* (*About Phone* ou *About Tablet*), ou semelhante;
2. Deslize pelas opções até chegar a *Número de Compilação* (*Build Number*) e toque sobre essa opção **7** vezes;
3. Se tudo correu bem, deve ter sido emitida uma mensagem a informar que tem acesso ao menu de programador. Para ativar ou desativar o modo *debugging*, pode seguir até ao menu de *Opções de Programador* (*Developer Options*) dentro das *Definições do Sistema* (*Settings*).

Tarefa 11

Mude o foco para o dispositivo virtual (ou o real, se o estiver a usar) e verifique que a aplicação está instalada e a funcionar.

Tarefa 12

Como exercício inicial simples, considere mudar o nome da aplicação de `Testingradle` para `FizTudoViaTerminal`, recompilá-la e voltar a instalá-la. Note que, para **re**instalar uma aplicação através do `adb`, deve adicionar a *flag* `-r` ao comando de instalação.

Nota: caso tenha de editar algum ficheiro, use o *notepad* (`$ notepad.exe NAME_OF_FILE`).

Tarefa 13

Altere o texto que aparece na atividade principal da sua aplicação para `Olarilolela`. Para fazer esta tarefa, talvez seja útil alterar o ficheiro `activity_main.xml`, que deve estar dentro da diretoria `src\main\res\layout` do seu projeto. Não se esqueça de voltar a compilar e reinstalar a aplicação.

Tarefa 14

Experimente emitir o comando `$ adb logcat`. **Q24.: Qual o resultado deste comando?**

- ☐ É exibida a imagem de um gato em arte ASCII.
- ☐ É exibido um *log* do sistema virtualizado que se está a usar para testar a aplicação.



19 Introdução ao Desenvolvimento de Interfaces Android

Sumário

Introdução à programação e desenvolvimento de interfaces de utilizador orientadas por eventos para aplicações Android. Implementação de uma aplicação que simula uma calculadora com as operações mais básicas como forma de estudar a criação de consumidores para objetos interativos e o ficheiro *eXtended Markup Language* (XML) que define o desenho da interface de utilizador.

Summary

Introduction to event-based programming and development of user interfaces for Android applications. Implementation of an application that simulates a calculator with the most basic operations, with the excuse to introduce the creation of listeners for widgets and the eXtended Markup Language (XML) file that defines the design of the user interface.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Preliminares

O guia laboratorial 2 elabora nos passos necessários à criação e compilação (*build*) de projetos de aplicações para a plataforma Android via linha de comandos. Esta abordagem, apesar de não comportar algumas das facilidades oferecidas por ambientes de desenvolvimento integrados, nomeadamente ambientes de edição da interface de utilizador *What You See Is What You Get* (WYSIWYG), permite conhecer em maior profundidade os detalhes de implementação de uma aplicação Android, mas requer que, após instalação do Android Studio™, se atualize e instalem as várias ferramentas do SDK Android[1]. Depois do sistema estar devidamente configurado, 4 passos são suficientes para criar um projeto Android, gerar o ficheiro `.apk` e instalar a aplicação num dispositivo (virtual ou real):

1. Inicializar o dispositivo móvel virtual ou ligar um real ao computador[2];
2. Gerar o projeto através do Android Studio™;
3. Compilar o projeto com a ferramenta Gradle™, emitindo o comando `$ gradlew assembleDebug` na raiz do projeto;
4. Instalar a aplicação com um comando semelhante a `$ adb install -r path\NomeApp-debug.apk`.

Tarefa 1

Como já vem sendo habitual, a primeira tarefa consiste em iniciar um AVD. Para isso, pode emitir o comando `$ emulator`, incluído na pasta `emulator` do SDK, e lançar um AVD. Caso não exista nenhum AVD configurado, crie um[3]. O ideal será um emulador de uma versão igual à 6.0 do SO.

Q1.: Qual das seguintes combinações lista todos os AVDs disponíveis com o comando `$ emulator`?

☐ `$ emulator -list-avds` ☐ `$ emulator -avd list`

Tarefa 2

Verifique se as variáveis `ANDROID_HOME` e `PATH` estão devidamente definidas com comandos parecidos com:

```
$ echo %ANDROID_HOME%
```

[1] Ver https://developer.android.com/studio/intro/update.
[2] Se o dispositivo for real, tem de ter a opção de depuração ativada.
[3] O guia laboratorial 1 contém uma breve discussão acerca deste assunto.

```
$ echo %JAVA_HOME%
$ echo %PATH%
```

Caso as variáveis já estejam devidamente definidas (ou se usar sempre o Android Studio™), passe para a secção seguinte. Caso contrário, precisa de as definir antes de avançar, com comandos semelhantes a:

```
$ set JAVA_HOME=RAIZ do JAVA JDK
$ set ANDROID_HOME=C:\installation location\sdk
$ set PATH=%PATH%; %JAVA_HOME%; %ANDROID_HOME%\tools;
%ANDROID_HOME%\emulator; %ANDROID_HOME%\platform-tools
```

Nota: Para mais detalhes consultar o guia laboratorial 2.

2 Introdução às Interfaces de Utilizador Orientadas por Eventos

O grande objetivo deste guia laboratorial é o de introduzir a programação e o desenvolvimento de interfaces de utilizador orientadas por eventos para aplicações móveis. Antes de prosseguir, procure a definição de *interface* (no contexto da programação) e use o espaço seguinte para incluir essa definição:

Q2.: Consegue aperceber-se da diferença entre uma interface de utilizador e uma interface entre duas componentes de um programa?
☐ Não. :_(☐ Agora que penso nisso... sim, consigo.

Tarefa 3

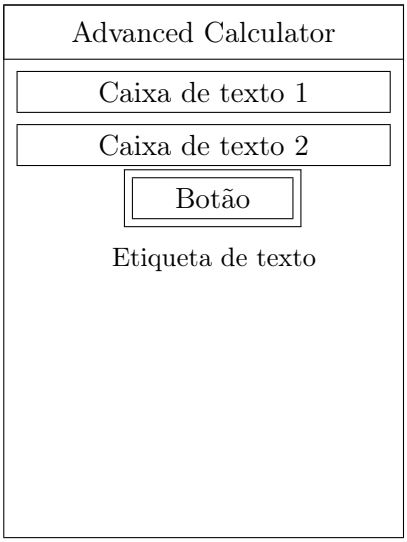
Usando o Android Studio™, crie e teste um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `AdvancedCalculator`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.advancedcalculator`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;

- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Tarefa 4

Depois de se certificar que o esqueleto base não apresenta problemas, mude o visual da aplicação. A tarefa consiste em adicionar duas caixas de texto editáveis seguidas, um botão e uma etiqueta de texto. Considere que vai criar uma calculadora que apenas sabe somar dois números reais. As duas caixas de texto servirão para a introdução dos dois números a somar, enquanto que o resultado aparecerá na etiqueta de texto após o botão ser pressionado. A figura seguinte esquematiza o visual pretendido para a aplicação.



Para conseguir mudar o *layout* da aplicação, abra e edite, com o bloco de notas (`$ notepad.exe nome-do-ficheiro.xml`) ou com o Android Studio™ (separador com o `.xml`, e depois pestana *Text*), o ficheiro XML que contém a definição desse *layout* para a aplicação principal. **Q3.: Em que local pode o ficheiro mencionado ser encontrado?**

☐ `src\main\res\layout` ☐ `libs\layout` ☐ `gen\layout` ☐ `src\main\res`
☐ Em casa, a descansar.

Quando abrir o ficheiro, vai notar que já existe pelo menos uma etiqueta de texto definida. Faça 4 cópias do elemento que a define no XML e altere o nome de três desses elementos para `EditText` (dois desses elementos) e `Button`. Mude o valor da propriedade `android:text` do primeiro elemento para `"This is a very Advanced Calculator!"`. No final deve ter 5 elementos definidos com a seguinte ordem:

- 1 `TextView`;
- 2 `EditText`;
- 1 `Button` (já agora, deve ajustar o texto do botão para o símbolo `+`); e
- 1 `TextView`.

Depois de fazer as alterações mencionadas antes, compile e teste a aplicação. **Q4.: Funcionou?**
☐ Sim senhor. Está tudo a rolar.
☐ Ehh... quer se me dizer... Instalar até instalou, mas está tudo assim: uma salganhada.
☐ Não... deve ter havido algum problema.

Deve verificar que os vários objetos se encontram sobrepostos. Isto deve-se a dois fatores: (i) os diferentes objetos não têm identificadores (IDs) diferentes; (ii) não são definidas corretamente as **restrições** de posicionamento dos objetos. Note que cada um dos objetos devia estar por baixo do anterior.

Para resolver a questão dos IDs, volte a abrir para edição o referido ficheiro de *layout* e adicione a propriedade `android:id="@+id/nome_do_recurso"` a cada um dos elementos que adicionou anteriormente (e à etiqueta de texto que já lá estava). Os IDs a atribuir à etiqueta de texto, às duas caixas de texto, ao botão e à etiqueta de texto devem ser, respetivamente:

- `titulo`;
- `number1`;
- `number2`;
- `sum`;
- `result`.

Agora, é necessário escolher a **restrição** *(constraint)* a usar em cada um dos objetos. **Q5.: Qual das seguintes deverá usar?**
☐ `app:layout_constraintBottom_toBottomOf`
☐ `app:layout_constraintLeft_toLeftOf`

☐ `app:layout_constraintRight_toRightOf`
☐ `app:layout_constraintTop_toBottomOf`

Independentemente do que respondeu antes, note que cada objeto definido no XML deve, no final deste exercício, ter uma definição semelhante à seguinte (com as devidas adaptações):

```
<Button
   android:id="@+id/sum"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text=" + "
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/number2"
   />
```

Q6.: Não pergunta ao Prof. pelo significado do `@` e do `+` na definição da propriedade `android:id`?
☐ Não, porque já sei tudo.
Use o espaço em baixo para dizer o que significam os dois símbolos:

`@` _____

_____

`+` _____

_____

Tarefa 5

Se tudo correu bem até este passo, deve ter conseguido desenvolver uma aplicação minimalista (e com uma só atividade), já com um design semelhante ao que foi pedido, e sem ter de escrever ou manipular uma única linha de código Java. **Q7.: Estes factos vão de encontro ao que foi referido em relação à arquitetura de software MVC?**
☐ Isto não tem nada a ver com a arquitetura MVC.
☐ De facto, isto parece ir ao encontro daquela arquitetura de desenvolvimento de *software* na medida em que as vistas sobre os dados são definidas em XML.
☐ De facto, isto parece ir ao encontro daquela arquitetura de desenvolvimento de *software* na medida em que as vistas sobre os dados são definidas em Java.
☐ De facto, isto parece ir ao encontro daquela arquitetura de desenvolvimento de *software* na medida em que o controlo está separado do código da aplicação.
☐ De facto, isto parece ir ao encontro daquela arquitetura de desenvolvimento de *software*, já que o código está perfeitamente separado das vistas.

Volte para o separador de edição de código (ou procure e abra para edição o ficheiro

contendo o código fonte) da única atividade da aplicação que criou. **Q8.: Já se sabe que estará dentro da diretoria `src`, mas qual é o caminho completo do ficheiro a partir da raiz da aplicação?**

☐ `src/ACalculator.java` ☐ É um comboio desgraçado.
☐ `app/src/main/java/pmd/di/ubi/pt/acalculator/...`

Depois de abrir o ficheiro para edição no notepad (ou no Android Studio™), adicione os dois `imports` seguintes:

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

Q9.: Consegue encontrar a linha de código que *carrega* o *layout* da aplicação a partir do ficheiro?

☐ Sim, é a linha com a instrução

```
super.onCreate(savedInstanceState);
```

☐ Sim, é a linha com a instrução

```
setContentView(R.layout.activity_main);
```

☐ Não, não consigo.

Note que, para poder manipular, alterar ou aceder a valores contidos nos objetos interativos (*widgets*) mencionados na tarefa anterior, precisa de os instanciar na forma de objetos (Java) no código. Para isso, considere declarar os 4 objetos seguintes **antes** do método `onCreate()` (mas depois de declarar a classe `MainActivity`):

```
EditText oTEdit1;
EditText oTEdit2;
Button oButton;
TextView oTView1;
```

Dentro do método `onCreate()`, pode instanciar estes objetos através de instruções semelhantes a:

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

Q10.: Por que é que é que cada vez que declara a função `findViewById(...)` necessita de colocar o nome de uma classe de um objeto entre parêntesis antes?

☐ Só para ter a certeza de que o objeto devolvido é da classe desejada.
☐ Se não se colocar nada entre parêntesis antes de chamar a função, o Java é como que fica chateado comigo... e debita erros, e assim.
☐ O protótipo da função mencionada obriga a que seja chamada desta forma.
☐ A função devolve sempre um objeto mais geral, da classe `View`, que precisa ser convertido para a subclasse mais específica através de um *cast*.

Tarefa 6

Note que o método `findViewById( ID )` aceita o `ID` (de IDentificador) do objeto interativo definido no ficheiro XML de *layout*. Estes IDs já foram anteriormente definidos. Repare que se se está a usar código semelhante a `R.algo`, é porque existe uma classe chamada `R` algures no seu projeto. A sua implementação deve estar num ficheiro chamado `R.java` (porque o Java costuma obrigar a que o nome dos ficheiros tenha o mesmo nome das classes neles implementadas). **Q11.: Agora sem olhar (!), consegue encontrar este ficheiro?**
☐ Sim, algures em `build/generated/source ...`
☐ Sim, em `build/generated/source/r/debug ...`
☐ Sim, algures dentro da pasta `src`.
☐ Sim, em `build/generated/not_namespaced.../debug/r/...`
☐ Sim, algures dentro da pasta `gradle`.

Aproveite o facto de estar a editar o ficheiro XML para explorar algumas particularidades do *layout* através da alteração das propriedades dos elementos. Por exemplo, experimente mudar a propriedade `android:layout_width="wrap_content"` do **botão** para `android:layout_width="match_parent"`. **Q12.: Qual o efeito desta alteração na interface de utilizador da aplicação?**
☐ O botão passa a estar na horizontal, em vez de estar na vertical.
☐ O botão passa a estar na vertical, em vez de estar na horizontal.
☐ O botão passa a estar obliquo, grande maluco!
☐ O botão passa a ter um tamanho que se coaduna com o tamanho do texto que contém.
☐ O botão passa a ocupar o máximo do ecrã na horizontal.

Tarefa 7

Adicione um consumidor (*Listener*) ao objeto botão de forma a definir que, após este ser clicado, a operação por ele representada é efetuada sobre os dois inputs nas caixas de texto (`oTEdit1` e `oTEdit2`) e o resultado é mostrado na etiqueta de texto `oTView1`. O excerto de código seguinte, onde falta uma instrução, pode ser útil na resolução desta tarefa.

```
oButton.setOnClickListener(
```

```
new View.OnClickListener()
{
  public void onClick(View oView)
  {
    double d1 = (new Double(oTEdit1.getText().toString())).doubleValue();
    double d2 = (new Double(oTEdit2.getText().toString())).doubleValue();
    double dSum = d1 + d2;
    // FALTA INSTRUCAO
  }
});
```

Q13.: Que nome se dá à classe criada pela instrução `new View.onClickListener()`?
☐ Classe objeto. ☐ Classe local. ☐ Pseudo classe. ☐ Classe anónima.

Tarefa 8

Teste a aplicação novamente (note que será ideal testar a aplicação repetidamente e sempre que adiciona uma nova funcionalidade).

Tarefa 9

Seja minucioso(a) e procure limar algumas arestas antes de prosseguir:

1. Faça com que o botão e ambas as caixas de texto ocupem o ecrã todo na horizontal;
2. Coloque um número nas caixas de texto;
3. Arranje forma de as caixas de texto apenas aceitarem números;
4. Coloque a palavra `0.0` na caixa de texto onde vai aparecer o resultado da operação.

☐ Minucioso(a), eu?!! Agora a chamar-me nomes!

Tarefa 10

Implemente as 3 operações que faltam para obter uma calculadora básica (*i.e.*, subtração, multiplicação e divisão).

Tarefa 11

Aproveite o facto da aplicação já aceitar 2 valores para sobre eles efetuar um cálculo para implementar as funções de logaritmo do `number1` na base `number2` e a de potência (`number1` elevado a `number2`).

Tarefa 12

Esta tarefa é só para os mais corajosos: arranje forma de atualizar a etiqueta de texto com o resultado **da soma** dos dois números sempre que for introduzido um número na segunda caixa de texto. Note que a etiqueta deve ser atualizada à medida que o número (potencialmente constituído por vários dígitos) é inserido. **Sugestão:** os procedimentos enunciados a seguir devem ser-lhe úteis na execução desta tarefa.

```
oTEdit2.addTextChangedListener(
  new TextWatcher(){
   public void afterTextChanged(Editable s) {
    ...
  }
```

Tarefa 13

Adicione um botão que permita copiar o resultado de uma operação para o *clipboard*. O conteúdo do URL http://developer.android.com/guide/topics/text/copy-paste.html pode ser-lhe útil para a execução desta tarefa.

Q14.: Já agora, só por curiosidade, o que significa URL?

U _____ R _____ L _____



20 Consumidores de Eventos e Ciclo de Vida de Atividades

Sumário

Análise de formas complementares de implementação de consumidores de eventos. Exercícios para análise do ciclo de vida de uma Atividade numa aplicação Android, bem como para a prática do uso do registo de sistema disponibilizado pelo Android e conhecido por `logcat`.

Summary

Analysis of alternative ways of implementing event Handlers. Exercises to analyze the lifecycle of an Android application Activity, as well as for practicing the usage of log system provided by Android and known as `logcat`.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Rotinas de Tratamento de Eventos

O guia laboratorial anterior sugeria a criação de uma aplicação que imitasse uma simples calculadora. Também foi sugerido que associasse um consumidor (*listener*) para cada objeto interativo do tipo botão, definindo-lhe uma rotina de tratamento do evento *clique no rato*, para as várias operações aritméticas pedidas. A ideia desta parte do guia é a de mostrar outras formas de implementar as rotinas de tratamento de eventos ou de associar um evento do tipo *clique* a uma dessas rotinas.

Q1.: Só para relembrar: qual é o pacote que deve importar para usar objetos interativos como o botão?

☐ `android.swing.*` ☐ `android.util.*` ☐ `android.widget.*` ☐ `android.view.*`

Q2.: Por curiosidade, qual é a classe pai (ou super classe) da classe `Button`?

☐ `java.lang.Object` ☐ `android.widget.Button`
☐ `android.view.View` ☐ `android.widget.TextView`

Q3.: Já agora, como se chama o cunhado do `Button`?

☐ ? ☐ ??? ☐ ??????

Tarefa 1

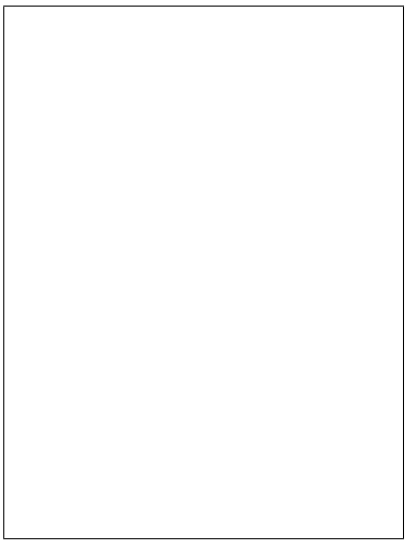
Juntamente com este guia laboratorial são disponibilizados dois arquivos com extensão `.zip`. Comece por descarregar e descompactar o arquivo `AdvCalculator2.zip`. Depois disso, considere analisar o ficheiro XML aí contido e que define a interface de utilizador de uma aplicação chamada `AdvCalculator2`, transcrito também para aqui por comodidade:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
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"
  tools:context=".ACalculator">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/textview1"
    android:text="This is a very Advanced Calculator!"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
  <EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=" "
```

```
    android:id="@+id/number1"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/textview1" />
  <EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=" "
    android:id="@+id/number2"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/number1" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="+"
    android:id="@+id/SUM"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/number2"
    app:layout_constraintLeft_toRightOf="@+id/SUB" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="-"
    android:id="@id/SUB"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toLeftOf="@id/SUM"
    app:layout_constraintTop_toBottomOf="@id/number2" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="x"
    android:id="@+id/MUL"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/SUM"
    app:layout_constraintLeft_toRightOf="@+id/DIV" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="/"
    android:id="@id/DIV"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toLeftOf="@id/MUL"
    app:layout_constraintTop_toBottomOf="@id/SUM" />
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Resultado"
    android:id="@+id/result"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/DIV" />
</android.support.constraint.ConstraintLayout>
```

Use o espaço em baixo para desenhar a interface de utilizador que vai ser gerada no dispositivo sem executar a aplicação, *i.e.*, deduza-a da análise do XML incluído antes:



Tarefa 2

Usando o Android Studio™, crie e teste um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `AdvCalculator2`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.advcalculator2`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Tarefa 3

Procure os ficheiros `activity_main.xml` e `MainActivity.java` dentro da pasta `src/main/...` e substitua-os pelos dois ficheiros que vinham no ficheiro `.zip` que descarregou. **Não se esqueça de** compilar e testar o projeto no dispositivo virtual Android que deve ter inicializado antes.

Q4.: A interface de utilizador que desenhou em cima é semelhante à que realmente apareceu?
☐ É igualzinha!
☐ Ups, os botões não ficaram bem como eu os tinha colocado.
☐ Ups, o resultado não ficou bem como eu tinha pensado que ia ficar. Vou já ver o que é que compreendi mal.

Q5.: Experimentou os botões?
☐ Não sabia que era para experimentar.
☐ Não, não experimentei, mas vou experimentar.
☐ Experimentei, e não faziam nada, mas deviam fazer.
☐ Experimentei, e não faziam nada (como não podia deixar de ser).

Tarefa 4

Note que parte do código Java que implementa a aplicação está comentado (por estar incompleto). Estes trechos de código permitem que seja definido **apenas um** objeto consumidor `onClickListener` e implementada uma só vez a rotina de tratamento `onClick()` para quando os botões são clicados (em vez de um para cada *widget*). Contudo, **é preciso tratar cada botão individualmente dentro da rotina**, conforme já é sugerido. Note que falta código (muito pouco) para que o trecho de código comentado esteja completo.

Complete o código corretamente de modo a que os cliques nos botões produzam o resultado esperado. No final compile e certifique-se de que a aplicação funciona corretamente. **Sugestão:** use os IDs dos vários recursos nos vários casos do `switch`.

Se fez tudo bem, deve ter usado uma referência à classe Java chamada `R`. **Q6.: Onde está implementada esta classe? Por outras palavras, onde está o ficheiro `R.java`?**
☐ Algures em `build/generated/source/r/...`
☐ Algures em `build/generated/not_namespaced.../debug/r/...`
☐ Em `build/generated/source/buildConfig`
☐ Este ficheiro é aquele que permite obter `Riot Points` no League of Legends.
☐ Não faço a mínima ideia, não encontro nenhum ficheiro `R` ...

Abra o ficheiro `R.java`. **Q7.: A que conceito se refere a instrução `R.id`?**

☐ A uma classe estática.
☐ A um atributo estático.
☐ A uma atividade.
☐ A um método.

Q8.: Em que situação é conveniente editar este ficheiro?
☐ Sempre que se quer adicionar um identificador novo a um *widget*.
☐ Só no dia de São Nunca à tarde.
☐ Sempre que se adicionam atividades.
☐ Antes de compilar.
☐ Depois de comer.
☐ Durante o sono.

Q9.: Afinal, o que são os IDs que utiliza no código para referenciar alguns recursos de uma aplicação Android?
☐ São *strings*. ☐ São números inteiros. ☐ São números reais. ☐ São *cheesy* bytes.

Tarefa 5

Na plataforma Android, existem outras formas de associar um método consumidor a um botão, pelo menos para o evento `onClick`. Uma dessas formas consiste em indicar o nome do método que vai tratar o evento no ficheiro XML de *layout*. Pode encontrar mais informação acerca deste assunto na Web, nomeadamente em `http://developer.android.com/reference/android/widget/Button.html` mas, basicamente, esta associação faz-se definindo mais um atributo no elemento botão com a seguinte sintaxe:

```
android:onClick="CALLBACK_METHOD_NAME"
```

Esta tarefa consiste, portanto, na alteração do ficheiro de *layout* do projeto *AdvCalculator2* de modo a que o botão de soma despolete o método `somar(View v)` que também já foi implementado no ficheiro `MainActivity.java`, **para sua conveniência**.

Q10.: De uma maneira geral, esta abordagem parece-lhe mais simples que as que já foram estudadas até aqui?
☐ De facto, parece-me mais simples (legível). ☐ É-me indiferente.

O método que acabou de definir como *Callback* tem um parâmetro de entrada do tipo `View`. **Q11.: Este parâmetro é mesmo necessário?**
☐ É sim. Sem ele, o *callback* não funciona.
☐ Não, neste caso não é preciso.
☐ É sim, e pode levar mais parâmetros adicionais.

Tarefa 6

Depois de fazer a modificação necessária, compile e teste a aplicação, verificando que funciona.

Q12.: Por curiosidade, em que versão da API Android é que os botões foram disponibilizados?
☐ Na API 1 ☐ Na API 5 ☐ Na API 10 ☐ Na API 19 ☐ Na API 21 ☐ Na API 22
☐ E como é que hei-de saber isso?

2 Registo do Sistema e Ciclo de Vida de uma Atividade

A segunda parte deste guia laboratorial tem como objetivo estudar o ciclo de vida das atividades.

Tarefa 7

Usando o Android Studio™, crie e teste um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `ActivityLifeCycle`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.activitylifecycle`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Tarefa 8

Descarregue para o seu sistema o arquivo `ActivityLC.zip`, também disponibilizado com este guia laboratorial. Procure o ficheiro `MainActivity.java` dentro da pasta `src/main/...` e substitua-o pelo ficheiro que vem no arquivo `.zip` que descarregou. **Não se esqueça de** compilar e testar o projeto no dispositivo virtual Android que deve ter inicializado antes.

Tarefa 9

Irá notar que há código comentado e em falta no ficheiro `MainActivity.java`. Os locais onde falta código estão marcados com:

```
// FALTA CODIGO
```

Retire os escapes para comentário e complete o código que falta de modo a que a aplicação escreva entradas no registo do sistema (`logcat`). As mensagens a aparecer devem ser semelhantes a:
`onCreate() method was called`,
`onStart() method was called`,
...
`onDestroy() method was called`.
A *tag* a utilizar deve ser `ALC`. Já agora, estime se as questões seguintes ajudam na execução desta tarefa.

Note que vai precisar de importar a classe que permite escrever entradas no registo do sistema. **Q13.: Como se chama a classe que lhe permite fazer isso?**
☐ `LogCat` ☐ `Log` ☐ `Logarithm` ☐ `Miau` ☐ `Register`

Q14.: Qual o pacote que terá de importar para poder usar a classe mencionada antes?
☐ `Java.util.*` ☐ `android.app.Activity` ☐ `android.util.*` ☐ `Java.View.*`

Tarefa 10

Depois de alterar o código, compile, instale e teste a aplicação. Se estiver a funcionar, volte a fechá-la (garanta mesmo que fica fechada). Caso contrário, procure resolver o problema.

Tarefa 11

Coloque o `logcat` a correr usando o comando `$ adb logcat`. Caso esteja a observar demasiados registos no *output* do programa, considere aplicar um filtro para a *tag* `ALC`.
Q15.: Como é que se aplicam filtros no `logcat` via linha de comandos?
☐ Combinando a opção `-f` com `ALC`. ☐ Combinando a opção `-e` com `ALC`.
☐ Combinando a opção `-s` com `ALC`. ☐ Combinando a opção `-i` com `ALC`.

Certifique-se de que está a conseguir ver as entradas que aplicação faz no registo, executando e terminando a aplicação, e verificando se aparecem no `logcat`.

Tarefa 12

Observe os fluxos possíveis de uma aplicação Android esquematizados na figura central do URL http://developer.android.com/reference/android/app/Activity.html (também incluída na aula teórica 4). **Q16.: Consegue simular um fluxo normal (completo) de execução da aplicação no seu `logcat`?** *I.e.*, o fluxo `onCreate()` ⟶ `onStart()` ⟶ `onResume()` ⟶ `onPause()` ⟶ `onStop()` ⟶ `onDestroy()`
☐ Claro que consigo. ☐ Não consigo...

Use o espaço seguinte para descrever claramente quais os passos que tomou para conseguir simular este fluxo:

1. _____

_____

2. _____

_____

3. _____

_____

4. _____

_____

5. _____

_____

Tarefa 13

Simule de seguida o fluxo definido por `onCreate()` ⟶ `onStart()` ⟶ `onResume()` ⟶ `onPause()` ⟶ `onStop()` ⟶ `onRestart()` ⟶ `onStart()`

Use o espaço seguinte para descrever claramente quais os passos que tomou para conseguir simular este fluxo:

1. _____

_____

2. _____

_____

3. _____

_____

4. _____

_____

5. _____

_____

Tarefa 14

A documentação da plataforma Android diz que é possível que uma aplicação seja terminada sem passar pelos métodos `onStop()` e `onDestroy()`. Simule no `logcat` o fluxo definido por `onCreate()` ⟶ `onStart()` ⟶ `onResume()` ⟶ `onPause()` ⟶ `onStop()` ⟶ `onCreate()` ⟶ `onStart()` ⟶ `onResume()` ⟶ ...

Use o espaço seguinte para descrever claramente quais os passos que tomou para conseguir simular este fluxo:

1. _____

_____

2. _____

_____

3. _____

_____

4. _____

_____

5. _____

_____

Tarefa 15

A documentação da plataforma, nomeadamente o diagrama referido em cima também sugere que é possível passar de `onPause()` para `onResume()`. **Q17.: Consegue simular o fluxo definido a seguir no seu `logcat`?**
`onCreate()` ⟶ `onStart()` ⟶ `onResume()` ⟶ `onPause()` ⟶ `onResume()` ⟶ ...
☐ Canja de galinha.
☐ Esta é impossível com a aplicação fornecida.
☐ Este está a oferecer resistência, mas penso que vou conseguir.

Use o espaço seguinte para descrever claramente quais os passos que tomou para conseguir simular este fluxo:

1. _____

_____

2. _____

_____

3. _____

_____

4. _____

_____

5. _____

_____

Q18.: Por curiosidade, o que é que acontece quando muda a orientação (*e.g.*, de vertical (*portrait*) para horizontal (*landscape*)) do dispositivo, em termos do ciclo de vida de uma atividade?

☐ O facto é que, em termos de ciclo de vida, a atividade parece não ter sofrido qualquer alteração no processo.

☐ A atividade parece ter sido terminada e depois reiniciada!

☐ A atividade parece ter sido colocada em pausa e depois resumida.

☐ A atividade foi simplesmente terminada.

☐ A atividade passou a ser radioatividade.



21 Intentos Implícitos e Explícitos

Sumário

Implementação de simples aplicações que demonstram o funcionamento dos intentos, os objetos que abstraem a forma como as várias componentes de uma aplicação Android comunicam, e que permitem a definição do fluxo dessas aplicações. São abordados intentos implícitos e explícitos, bem como a transmissão de dados a partir da atividade que invoca o intento, para aquela que é invocada.

Summary

Implementation of simple applications that depict the functioning of intents, which are the objects that abstract the means by which several components of an Android application communicate, and that enable the definition of a flow in such applications. This laboratory guide handles implicit and explicit intents, as well as the transmission of data between the invoking and the invoked activities.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Intentos Implícitos

O objetivo desta aula é introduzir o estudo e praticar os objetos que, na plataforma Android, permitem a comunicação e transição entre componentes de uma ou mais aplicações. Estes objetos são conhecidos por `Intents` (intentos) porque refletem a definição de uma operação que ainda precisa ser feita. Esta secção foca-se num dos 2 tipos de intentos: os implícitos.

Tarefa 1

Usando o Android Studio™, crie e teste um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `Implicit`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.implicit`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 2

Edite o ficheiro `main.xml` de forma a incluir o botão especificado a seguir no *layout* (mas **não** compile já a aplicação):

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
```

```
    android:text="Send Message"
    android:layout_gravity="center"
    android:onClick="sendMessage"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>
```

Faça também por **eliminar** a etiqueta de texto que vem por defeito em projetos criados com o Android Studio™.
Observe as propriedades do botão e responda às questões seguintes. **Q1.: Qual a aparência deste botão, em termos de altura?**
☐ Vai ocupar metade do ecrã.
☐ Vai ocupar o ecrã de cima a baixo.
☐ Vai ocupar apenas o suficiente para acolher o texto `Send Message`.

Q2.: Em termos de disposição, onde se vai situar este botão?
☐ À esquerda. ☐ Ao centro. ☐ À direita. ☐ Em cima. ☐ Ao meio. ☐ Em baixo.

Q3.: Só da análise do XML, diria que precisa de implementar algum método na atividade principal para que a aplicação funcione bem?
☐ Humm... diria que não.
☐ Humm... parece-me que sim, nomeadamente o método com protótipo

---

Tarefa 3

Procure e abra para edição o ficheiro contendo o código fonte da atividade principal (`MainActivity`). Coloque o código Java incluído a seguir nesse ficheiro:

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

import android.app.Activity;
import android.os.Bundle;
import android.view.*;
import android.content.Intent;
import android.widget.*;

public class MainActivity extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);
    }

    public void sendMessage(View v){
        Intent iSendMsg = new Intent(Intent.ACTION_SEND);
        iSendMsg.putExtra(Intent.EXTRA_TEXT,"You should type the message
            here!");
        iSendMsg.setType("text/plain");
        // If one wants to make sure that an activity will resolve this
            intent,
        // the following comment should be removed
        // if( iSendMsg.resolveActivity(getPackageManager()) != null )
        startActivity(iSendMsg);
    }
}
```

No código incluído em cima, são importados vários pacotes, nomeadamente o `android.widget.*`. **Q4.: Este *import* é mesmo necessário?**
☐ Sim, é mesmo necessário. ☐ Não percebo por que é que lá foi incluído.

Q5.: E o *import* de `android.content.Intent`, é necessário?
☐ Indubitavelmente.
☐ Não percebo por que é que lá foi incluído.

Q6.: Em que pacote/classe é que o método `setContentView()` está definido?
☐ `import android.app.Activity;`
☐ `import android.os.Bundle;`
☐ `import android.view.*;`
☐ `import android.content.Intent;`
☐ `import android.widget.*;`

Antes de evoluir para a próxima tarefa, analise o código e responda ao seguinte: **Q7.: O que é que esta aplicação supostamente deve fazer?**
☐ Enviar uma SMS.
☐ Enviar um e-Mail.
☐ Abrir o *browser*.
☐ Enviar uma mensagem para o Facebook.
☐ Abrir a calculadora.
☐ Nenhuma das anteriores.
☐ Todas as anteriores, incluindo a que diz: nenhuma das anteriores.

Tarefa 4

Compile, instale e teste o projeto que criou. Confirme se o aspeto da aplicação se coaduna com o que havia respondido anteriormente relativamente ao botão.

Q8.: Por que é que o *intent* que foi criado no âmbito desta parte do guia laboratorial é conhecido por intento implícito?

☐ Porque a ação que vai produzir é desconhecida.
☐ Porque a ação que vai ser feita foi totalmente definida.
☐ Porque a atividade que vai tratar da ação especificada não é conhecida à partida.
☐ Porque a atividade que vai tratar da ação especificada é conhecida e definida no código.
☐ Porque a *intent* tem um parâmetro adicional configurado com o método `putExtra()`.

Q9.: Só para que fique registado, o que é que acontece quando pressiona o botão *Send Message*?

☐ É aberta uma calculadora (que estranho!).
☐ É mostrada uma caixa de diálogo onde é dada a hipótese de escolher qual a aplicação para onde o fluxo vai evoluir. A atividade onde estava fica desfocada no fundo.
☐ É aberta a aplicação que envia SMSs.
☐ É aberta a aplicação que envia e-mails.
☐ É aberta a aplicação que mostra os mapas.
☐ A aplicação é terminada e um erro é mostrado no ecrã.

2 Intentos Explícitos

Esta parte do guia laboratorial é focada em intentos explícitos.

Tarefa 5

Usando o Android Studio™, crie e teste um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `Explicit1`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.explicit1`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);

- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 6

Modifique o *layout* da aplicação que é criado por defeito de modo a conter o botão seguinte:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Start Calculator"
    android:onClick="startCalculator"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Faça também por **eliminar** a etiqueta de texto que vem por defeito em projetos criados com o Android Studio™.

A definição do botão incluída parece sugerir que o botão vai estar centrado na interface gerada pela atividade. **Q10.: Esta assunção parece-lhe válida?**

☐ Assim à primeira vista, não vejo porque não, mas já sei que o Prof. cá mete estas questões só para nos dar a volta.
☐ Assim à primeira vista, não vejo porque não.
☐ Não. Esta assunção é errada.

Justifique a sua resposta anterior, para quando depois for estudar, cá estar tudo explícito:

Tarefa 7

Dada a definição anterior do botão, deve ser necessária a implementação de uma função de tratamento do evento `onClick`. Procure e abra para edição o ficheiro que implementa

a atividade principal (`MainActivity.java`) e declare essa função. **Q11.: Só mesmo por curiosidade, como se chama a classe que implementa a atividade principal?**

☐ Não sei, terei de abrir primeiro o ficheiro `.java`.
☐ Chama-se `main` e estende a classe `Activity`.
☐ Então: tem de se chamar, obrigatoriamente, `MainActivity`.
☐ Chama-se pelo nome.

No corpo dessa função, coloque o seguinte:

```
Intent iCalc = new Intent();
iCalc.setAction(Intent.ACTION_MAIN);
iCalc.setComponent(new ComponentName("com.android.settings","com.android.
    settings.Settings"));
startActivity(iCalc);
```

Dado o código incluído antes, e comparando com os *imports* que o ficheiro que implementa a atividade principal já contém, selecione, em baixo, os nomes dos pacotes que ainda precisa incluir no ficheiro `MainActivity.java`:

☐ `import android.app.Activity;`
☐ `import android.os.Bundle;`
☐ `import android.view.*;`
☐ `import android.content.Intent;`
☐ `import android.content.ComponentName;`

Tarefa 8

Compile, instale e teste a aplicação criada anteriormente.

Q12.: Só para que fique registado, o que é que acontece quando pressiona o botão *Start Calculator*?

☐ É mesmo aberta uma calculadora! E esta, hein?
☐ É mostrada uma caixa de diálogo onde é dada a hipótese de escolher qual a aplicação para onde o fluxo vai evoluir. A atividade onde estava fica desfocada no fundo.
☐ É aberta a aplicação que gere as definições do sistema operativo (que estranho).
☐ É aberto um *browser* com dois *tabs* que imediatamente são direcionados para os dois links especificados no código.
☐ A aplicação é terminada e um erro é mostrado no ecrã.

No código incluído em cima está uma linha de código com o seguinte trecho `ComponentName("com.android.settings", "com.android.settings.Settings")`.

Q13.: O que é que significam as duas *strings* neste trecho?

☐ São dois *links* que o Android abre no *browser* configurado por defeito.
☐ A primeira é o nome do pacote da aplicação de gestão definições que vem de fábrica com o Android; a segunda é o nome da atividade principal dessa aplicação.

☐ A primeira é o nome da atividade principal da aplicação de gestão de definições que vem de fábrica com o Android; a segunda é o nome do pacote dessa aplicação.
☐ Estas duas strings definem sem sombra de dúvida o componente para onde a intenção aponta.

Q14.: Quando usada como um adjetivo, como é o caso em `intento explicito`, a palavra *explícito* deve ou não ser acentuada?
☐ Deve ser. ☐ Não deve ser.

Procure no dicionário (*e.g.*, `http://www.priberam.pt/DLPO`) a definição da palavra antes discutida, e inclua-a no espaço seguinte:

Q15.: A atividade desenvolvida envia dados para a atividade recetora do intento?

☐ Não. ☐ Sim, nomeadamente: _____

Tarefa 9

Até este ponto, todas as aplicações desenvolvidas eram constituídas por apenas uma atividade. A parte restante deste guia sugere a criação de uma atividade adicional, que irá inclusive receber dados da atividade que a invocar. Pode construir um projeto de raiz ou simplesmente alterar o anterior, sendo a descrição seguinte orientada para a alteração daquele que acabou de ser feito.

Comece por criar um novo ficheiro ficheiro chamado `Activity2.java` dentro da diretoria `src\main\java\pt\ubi\di\pmd\explicit1`. Este ficheiro terá o código que implementa esta segunda atividade. Abra o ficheiro com um editor de texto (*e.g.*, `notepad++`), e considere colocar no seu conteúdo uma implementação semelhante à que se segue:

```
package pt.ubi.di.pmd.explicit1;

import android.app.Activity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;
import android.content.Intent;

public class Activity2 extends Activity {
    TextView oTV;
```

```
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main2);
        oTV = (TextView) findViewById(R.id.textview1);

        Intent iCameFromActivity1 = getIntent();
        oTV.setText(iCameFromActivity1.getStringExtra("string1"));
    }
    public void endActivity(View v){
        finish();
    }
}
```

Q16.: O que é que pode concluir da observação do código?

☐ Pouca coisa.
☐ Vá, uma coisa ou outra, mas nada de relevante.
☐ Uii, há ali uns detalhes...
☐ Que esta atividade parece inicializar um objeto `Intent` através do método `getIntent()` e não a partir de um construtor.
☐ Que é instanciada uma `TextView` e depois preenchida com algo que vem dentro da `Intent`, nomeadamente numa variável geral com designação `string1`.
☐ Que, já agora, parece relevante referir que parece haver ali um método que não é usado, provavelmente por ser uma função *callback* para um evento *onClick* definido no XML.
☐ Que, ainda neste contexto, parece interessante mencionar que há ali um outro método que nunca tinha visto (`finish()`). Vou já ver o que faz...

Q17.: Qual o ficheiro XML que define o *layout* desta segunda atividade?

☐ E como é que eu vou saber isso?
☐ `main.xml`
☐ `main2.xml`
☐ `layout.xml`
☐ É um ficheiro que ainda não existe.

Tarefa 10

Note que depois de definir uma nova atividade, é necessário criar o seu *layout* (*i.e.*, um ficheiro XML) e declará-la no `AndroidManifest.xml`. Esta tarefa consiste, portanto, em adicionar **no local correto** a seguinte linha no ficheiro de manifesto:

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

Tarefa 11

Crie o ficheiro XML `main2.xml` na pasta _____

e faça por este conter os elementos incluídos a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout 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"
    tools:context=".Activity1">

   <TextView

        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="End Activity"
        android:onClick="endActivity"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toBottomOf="@id/textview1"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Q18.: Da análise do código que antecede o XML, falta algum atributo em algum elemento incluído em cima?

☐ Sim, falta... e vou já tratar disso.
☐ Parece-me que falta, mas não sei o que fazer.
☐ Não falta nada.

Tarefa 12

De seguida, há que modificar a atividade principal de maneira a que aponte para a que criou recentemente. Comece pelo ficheiro `activity_main.xml`, alterando-o de forma a conseguir que a atividade principal mostre mais um botão com o texto `Start Second Activity`. Quando clicado, este botão deve despoletar a rotina `startActivity2()`.

Provavelmente, será necessário adicionar algo semelhante ao seguinte **(não se esqueça de definir o `id` do botão anterior...)**:

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Second Activity"
        android:onClick="startActivity2"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toBottomOf="@id/button1" />
```

Tarefa 13

Passe agora para o ficheiro `MainActivity.java`, **adicionando** o método seguinte no **local correto**:

```
public void startActivity2(View v){
  Intent iActivity =
         new Intent(this, Activity2.class);

  iActivity.putExtra("string1",
  "If you were able to do this, you are FABULOUS!");

  startActivity(iActivity);
}
```

Q19.: Como é que se instancia um intento para uma atividade local?

☐ Através do construtor `Intent(Context, Class)`, que aceita o contexto da classe atual e o nome do componente destino.

☐ Através do método `putExtra("Class", "Target")`, em que a primeira string define o nome da classe atual e a segunda o nome da componente destino.

☐ Através do método `startActivity(Intent)`.

Tarefa 14

Finalmente, compile, instale e teste a aplicação. **Q20.: Funcionou?**

☐ Como não podia deixar de ser. ☐ Não percebo o porquê de tanto alarido.

Use o espaço seguinte para relatar o funcionamento da aplicação, discutindo, por exemplo, a proveniência da mensagem que é mostrada na segunda atividade.



22 Filtros de Intentos, Intentos com Retorno e Permissões

Sumário

Estudo do funcionamento dos filtros de intentos através de uma aplicação móvel capaz de lidar com a ação de partilha. Implementação de uma aplicação móvel simples que exemplifica a forma como se pode despoletar uma atividade com devolução de resultados. Introdução aos fornecedores de conteúdo e aos mecanismos de acesso a recursos protegidos de um sistema Android.

Summary

Study of the functioning of intent filters via the development of an application capable of dealing with the share action. Implementation of a simple mobile application exemplifying how an activity can be started and return results to its caller. Introduction to content providers and to the mechanisms to access protected resources in Android.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Filtros de Intentos

Tarefa 1

Depois de se certificar que tem um dispositivo virtual a correr e que tem uma plataforma alvo superior à 4.2 disponível, crie e teste (usando o Android Studio™) um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `exIntentFilters`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.exintentfilters`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 2

Procure a forma de especificar que a atividade que criou no âmbito do projeto anterior é capaz de lidar com a ação de partilha (*i.e.*, `ACTION.SEND`). Use as seguintes questões para o guiar no processo.

Q1.: Onde (em que ficheiro) é que se criam os filtros de intentos?

☐ Em `res\layout\activity_tolog.xml`
☐ No `AndroidManifest.xml`
☐ No código Java da atividade respetiva.
☐ Em ficheiros de sistema.
☐ Em todas as outras aplicações que querem fazer partilha.

Q2.: Dentro de que elemento é que deve ser colocado o filtro de intentos no ficheiro XML?

☐ Dentro do elemento `manifest`.
☐ Dentro do elemento `activity`.
☐ Dentro do elemento `activity`, mas fora do elemento `manifest`.
☐ Em tudo quanto é lado!
☐ Dentro do elemento `linear_layout`.
☐ Dentro do elemento `caotic_layout`.

Q3.: Qual dos seguintes excertos de código XML define corretamente o filtro de intentos?

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

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

☐
```
<intent-filter>
 <action android:name="android.intent.action.DIAL"/>
 <category android:name="android.intent.category.DEFAULT"/>
 <data android:mimeType="audio/*"/>
</intent-filter>
```

☐
```
<intent-filter>
 <action android:name="android.intent.action.EDIT"/>
 <category android:name="android.intent.category.DEFAULT"/>
 <data android:mimeType="image/png"/>
</intent-filter>
```

Depois de se certificar que respondeu corretamente às questões anteriores, faça as alterações necessárias no ficheiro correto. Compile, instale e teste a sua aplicação.

Tarefa 3

Repare que, quando é despoletada uma intenção de partilha, é comum transportar dados para a atividade invocada através de `putExtra(Intent.EXTRA_TEXT,string)`. Abra o ficheiro com o código fonte Java para a única atividade da aplicação. Adicione as instruções que achar necessárias para que esta atividade seja capaz de escrever no `logcat` (`Log.i()`) o conteúdo que é enviado por outras aplicações no *Extra* `Intent.EXTRA_TEXT`. A `TAG` a ser mostrada no *log* deve ser `INTFILT`.

Q4.: Acha que vai precisar de alguns dos *imports* seguintes?
☐ `import android.util.Log;` ☐ `import android.content.Intent;`
☐ Vou, sim senhor. ☐ Não preciso de nenhum destes *imports*.

Tarefa 4

Compile, instale e teste a aplicação. Neste caso, para a testar, vai precisar de:

1. Encontrar uma aplicação que permita partilhar conteúdo (*e.g.*, o *browser*);
2. Arranjar forma de observar o `logcat` (o que, a esta altura do campeonato nacional, já não deve ser problema).

Q5.: Das opções seguintes, quais concretizam formas de ver o `logcat`?
☐ `$ adb logcat` ☐ `$ android logcat`
☐ Android monitor. ☐ Aplicação `logcat` no emulador.
☐ Emitir o comando `$ adb shell` seguido de `$ logcat`.

Sugestão: para testar a aplicação, abra o *browser* no emulador e navegue até um site. Depois, carregue no botão ⋮ do canto superior direito, e escolha `Share...`. **Q6.: A sua aplicação aparece na caixa de diálogo de seleção seguinte?**
☐ Pois aparece, que engraçado. ☐ Ehh... não?!

Certifique-se que o *output* que definiu na atividade principal aparece, de facto, no `logcat`.

2 Intentos com o Retorno

Anteriormente, foi explorado como se podiam usar intentos para transportar dados entre componentes de uma aplicação Android, no sentido da que define o intento para a que o recebe. Esta parte do guia laboratorial foca-se no fluxo inverso da comunicação, e na definição, à cabeça, de que a componente (uma atividade) invocada através de um intento deve devolver um resultado.

Tarefa 5

Comece pela criação, compilação, instalação e teste de um novo projeto Android através do Android Studio™ com as seguintes especificações:

- Nome da aplicação/projeto — `exWithResult`;

- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.exwithresult`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 6

A atividade principal deve ter duas caixas de texto, um botão e uma etiqueta de texto em baixo:

- As **caixas de texto** servem para colocar o primeiro e o último nome de um utilizador.
- O **botão** deve dizer `Concat Names!`, e redirecionar para uma nova atividade, cuja única função será concatenar os dois nomes e devolver esse resultado para a primeira atividade.
- A **etiqueta de texto** em baixo deve estar vazia ao início, mas mais tarde deve conter a concatenação dos dois nomes colocados nas caixas de texto, após esta operação ser feita pela segunda atividade.

Q7.: Qual o método que permite que a atividade principal chame a segunda de forma a que esta lhe devolva um resultado?

☐ `beginActivity(Intent,int);`
☐ `startActivity(Intent,int);`
☐ `beginActivityForResult(Intent,int);`
☐ `devolveUMresultadoPoça(Intent,int);`
☐ `startActivityForResult(Intent,int);`

Q8.: Qual a função do segundo parâmetro (um inteiro (`int`)) referido na questão anterior?

- ☐ Este segundo parâmetro serve para transportar erros da componente que invoca para a que é invocada.
- ☐ Este segundo parâmetro serve para transportar erros da componente que é invocada para a que invoca.
- ☐ Este segundo parâmetro não serve para nada. É ridículo.
- ☐ Este segundo parâmetro deve ser igual a 1 caso se esteja à espera de retorno, e 0 no caso contrário.
- ☐ Este segundo parâmetro serve para identificar um pedido de retorno em particular. *I.e.*, se forem lançados vários intentos com pedido de retorno, é este inteiro que permite identificar uma resposta específica.

Considere o seguinte excerto de código para o conteúdo do seu ficheiro `activity_main.xml` (**preencha** corretamente os espaços marcados):

```
<TextView
    PREENCHER
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Enter your first and last name:"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<EditText
    android:id="@+id/fName"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="First Name"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/explanation" />

<EditText
    android:id="@+id/lName"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Last name"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/fName" />

<Button
    android:id="@+id/PREENCHER"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Concat Names!"
    android:onClick="goToSecond"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/lName" />

<TextView
    android:id="@+id/concat"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Empty at first..."
    app:layout_constraintLeft_toLeftOf="parent"
```

```
        app:layout_constraintRight_toRightOf="parent"
        PREENCHER="@id/button" />
```

Tarefa 7

Combine toda a informação reunida na secção anterior, e codifique o método `goToSecond(View view)` de modo a que o botão redirecione para a atividade `ProcessNames.class` quando o botão `Concat Names!` é clicado. Não se esqueça de enviar, no intento respetivo, as duas *strings* com o conteúdo das caixas de texto, em variáveis chamadas `name1` e `name2`.

Tarefa 8

Crie uma segunda atividade com nome `ProcessNames`. Esta atividade deve ter apenas um botão que diz `Done. Go Back!`. Ao ser clicado, o botão deve forçar a que a aplicação volte para a primeira atividade, devolvendo já a concatenação dos dois nomes numa string chamada `concatNames`.

Não se esqueça de declarar a nova atividade no `AndroidManifest.xml`, para que esta seja visível para o sistema. O ficheiro de *layout* desta nova atividade terá apenas um botão, cuja definição deve ser semelhante à seguinte:

```
<Button
  android:id="@+id/button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Go Back!"
  android:onClick="goBack"
  app:layout_constraintLeft_toLeftOf="parent"
  app:layout_constraintRight_toRightOf="parent"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintBottom_toBottomOf="parent"/>
```

Já agora, e também para ajudar, a função que trata o evento de clique no botão `Done. Go Back!` é o seguinte:

```
public void goBack(View v);
```

Tarefa 9

Note que deve fazer o processamento dos dois nomes no método `onCreate(Bundle)`, sendo que apenas no método `goBack()` é que deverá instanciar o intento que devolve o

resultado para a atividade principal, imediatamente antes da instrução para terminar a atividade. O conteúdo do método `onCreate(Bundle)` será semelhante a:

```
public String sConcat;
@Override
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main2);
  Intent iCameFromMain = getIntent();
  sConcat = iCameFromMain.getStringExtra("name1") + " " + iCameFromMain.
      getStringExtra("name2");
}
```

Enquanto que o método `goBack(View v)` será parecido com:

```
public void goBack(View v){
  Intent iResult = new Intent();
  iResult.putExtra("concatNames",sConcat);
  setResult(RESULT_OK, iResult);
  super.finish();
}
```

Q9.: Qual é a função que ajusta o resultado de novo para a primeira atividade?
☐ `super.finish();`
☐ `setResult(RESULT_OK,iResult);`
☐ `iResult.putExtra("concatNames",sConcat);`
☐ `Intent iResult = new Intent();`

Q10.: O resultado é enviado no <u>mesmo</u> intento que despoletou a segunda atividade?
☐ Sim, é.
☐ Não, é enviado num novo intento.
☐ Não, é enviado através de um canal de comunicação criado para o efeito.

Tarefa 10

Esta parte do guia sugeriu um fluxo de implementação que começava pela atividade principal, que coleciona dois nomes, e avançava para a segunda atividade, que processa esses nomes. A ideia é retornar à atividade principal e exibir a concatenação dos nomes na etiqueta de texto ao fundo. Para isso, precisa implementar um último método na `MainActivity.java`. **Q11.: Qual é o método que é automaticamente despoletado quando uma atividade retorna um valor àquela que a invocou?**
☐ `onResume();`
☐ `onActivity(int, int, Intent);`
☐ `onResult(int, int, Intent);`
☐ `onActivityResult(int, int, Intent);`
☐ `onResume(int, int, Intent);`

O método acima mencionado deve ficar com um conteúdo semelhante ao seguinte (note

que falta especificar corretamente o nome e os parâmetros do método):

```
@Override
protected void method(int, int, Intent){
 if( ( iReqCode == iRequest_code ) &&
     ( iResultCode == RESULT_OK  ) ){

   TextView oVT = (TextView) findViewById(R.id.concat);
   oVT.setText( iResult.getStringExtra("concatNames") );

  }
}
```

Q12.: Qual o objetivo da primeira condição do `if`?

☐ O objetivo é verificar se o resultado foi efetivamente devolvido ou se houve algum erro no processo.

☐ O objetivo é verificar se o intento que foi devolvido diz respeito ao que foi despoletado.

☐ O objetivo é verificar se o resultado está de acordo com o que a atividade estava à espera.

☐ O objetivo é verificar se a segunda condição verifica o que a primeira faz e vice-versa.

Q13.: Qual o objetivo da segunda condição do `if`?

☐ O objetivo é verificar se o resultado foi efetivamente devolvido ou se houve algum erro no processo.

☐ O objetivo é verificar se o intento que foi devolvido diz respeito ao que foi despoletado.

☐ O objetivo é verificar se o resultado está de acordo com o que a atividade estava à espera.

☐ O objetivo é verificar se a segunda condição verifica o que a primeira faz e vice-versa.

Tarefa 11

Compile, instale e teste a aplicação tantas vezes quantas forem necessárias para conseguir os objetivos desta parte do guia.

Q14.: Ao todo, quantos ficheiros precisou de editar para conseguir a aplicação desejada?

☐ 0. ☐ 1. ☐ 2. ☐ 3. ☐ 4. ☐ 5. ☐ O rácio dourado.

3 Pedir Permissões no Manifesto

O próximo desafio consiste na implementação de uma aplicação, com uma única atividade, que mostre informação da última chamada de voz feita a partir do dispositivo móvel emulado (ou real).

Tarefa 12

Comece por aceder à aplicação de chamadas do emulador ou dispositivo real. Verifique o registo de chamadas e, caso este esteja vazio, faça uma chamada para o número 123456789. No final, verifique se o registo de chamadas já contém entradas. Eventualmente, pode tentar colocar mais entradas no registo, nomeadamente mediante a abertura de um segundo emulador, e do estabelecimento de chamadas entre os dois. Uma outra alternativa consiste em fazer `telnet` para o emulador alvo (ver aula 2).

Q15.: Como é que se pode fazer uma chamada de um emulador para outro[1]?
☐ Abrindo a aplicação de chamadas num dos emuladores, digitando o número da instância do segundo emulador (dada por `$ adb devices`) e executando as chamadas.
☐ Telefonando para a assistência da Google, dizendo que está a testar uma aplicação e que precisa que ponham o seu emulador a tocar.
☐ Marcando o número 127000001 (que corresponde ao endereço IP do *localhost*).

Uma terceira opção para simular chamadas concretiza-se pela utilização das funcionalidades acopladas ao próprio dispositivo virtual (nomeadamente clicando em `...`). **Q16.: Qual o nome do separador específico que permite fazer chamadas e enviar mensagens de voz?**

Q17.: Existe algum separador para gravação do ecrã do dispositivo virtual?
☐ Ahh, que giro! ☐ Não, mas dava jeito se houvesse!

Q18.: Quantos dedos podem ser simulados para o sensor de impressões digitais?
☐ Essa funcionalidade não existe. Quase me apanhou com esta!
☐ 1 dedo ☐ 2 dedos ☐ 3 dedos ☐ 4 dedos ☐ 5 dedos
☐ 6 dedos ☐ 7 dedos ☐ 8 dedos ☐ 9 dedos ☐ n dedos

Tarefa 13

Usando o Android Studio™, crie e teste um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `exPermissions`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a

[1] É assumido aqui que ambos os emuladores são fornecidos com o SDK.

`pt.ubi.di.pmd.expermissions`);

- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, **mínimo** API 23;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 14

Note: compile, instale e teste o projeto que criou só para se certificar de que está tudo bem até esta parte.

Tarefa 15

Mude o ficheiro XML que define o *layout* da atividade principal de forma a que esta passe a exibir uma etiqueta de texto no centro do ecrã, tanto na vertical como na horizontal. Especifique um identificador (`id`) a esta etiqueta. O `id` deve ser `lc`.

Q19.: Das seguintes, qual concretiza a linha que define corretamente o id da etiqueta de texto?

☐
```
android:id="@id/lc"
```

☐
```
android:id="@+id/lc"
```

☐
```
android:id="lc"
```

Tarefa 16

Altere o código da atividade principal de maneira a que a etiqueta de texto definida antes mostre todas as informações disponíveis no registo do sistema para a última chamada efetuada ou recebida.

Para esta tarefa, vai precisar de um componente Android conhecido como provedor de conteúdos. Estes componentes serão objeto de estudo adiante. Contudo, para já, fica a ideia de que estes componentes permitem a partilha de informação entre várias aplicações, nomeadamente as de sistema. Estes provedores de conteúdos são tipicamente acedidos através da especificação de URIs. Procure *online* qual é o URI que deverá permitir aceder ao registo de chamadas do dispositivo móvel e escreva-o a seguir:

Ordene (*i.e.*, coloque números nos espaços respetivos) os passos/instruções seguintes de forma a que reflitam o seu raciocínio/abordagem para conseguir o efeito desejado:

____ Cristalizar o *layout* através do método

```
setContentView(R.layout.main);
```

____ Instanciar a etiqueta de texto com uma instrução parecida com

```
TextView oTV = (TextView) findViewById(...);
```

____ Instanciar um objeto `Uri` com o URI do provedor de conteúdo relativo ao registo de chamadas. A instrução é semelhante a

```
Uri uriCalls = Uri.parse(...)
```

____ Instanciar um objeto `Cursor` que permite navegar pela informação disponível na base de dados acedida através de uma instrução semelhante a

```
Cursor curCalls = getContentResolver().query(uriCalls,null,null,
    null,null);
```

____ Declarar uma string com o texto `"Last Call:"`

```
String sInfo = "Last Call: ";
```

____ Movendo o cursor para a última linha dos dados, e mostrar o conteúdo de alguns campos da linha para onde está a apontar, através de um conjunto de instruções parecidas com as seguintes:

```
if(curCalls.moveToLast())
  sInfo += curCalls.getString(curCalls.getColumnIndex(CallLog.Calls.
      NUMBER))
        + "\nDuration: " + curCalls.getString(curCalls.getColumnIndex
            (CallLog.Calls.DURATION));
oTV.setText(sInfo);
```

Q20.: Quais dos seguintes pacotes interessa importar para a atividade principal desta aplicação?

☐ `android.app.Activity;`
☐ `android.widget.TextView;`
☐ `android.content.intent`
☐ `android.provider.CallLog;`
☐ `android.os.Bundle;`
☐ `android.net.Uri;`
☐ `android.database.Cursor;`

Tarefa 17

Depois de se assegurar que o código está minimamente completo, compile, instale e teste a aplicação. **Q21.: Funcionou?**

☐ O Professor já ouviu a expressão *às mil maravilhas?* ☐ Não funcionou.

Q22.: O registo de chamadas é considerado como um recurso a proteger no sistema Android?

☐ Pelos vistos não. ☐ Pelos vistos sim.

Q23.: De que forma é que o sistema responde a pedidos a recursos protegidos?

☐ Dando acesso imediato e total.
☐ Dando acesso parcial.
☐ Negando todo o acesso, a não ser que a permissão para aceder ao recurso tenha sido declarada no `AndroidManifest.xml`.
☐ Negando todo o acesso, a não ser que a permissão para aceder ao recurso tenha sido declarada no `AndroidManifest.xml`, e o utilizador tenha especificamente dado essa permissão aquando da instalação da aplicação.

Tarefa 18

Adicione o elemento incluído a seguir no `AndroidManifest.xml` como filho direto de `manifest`:

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

Volte a preparar e a instalar a aplicação. **Q24.: Desta vez já funcionou?**
☐ Tudo nos trinques. ☐ Ainda não. *Reset!*

Tarefa 19

A partir da API 23 e caso seja necessária uma permissão perigosa, **é necessário verificar programaticamente se a permissão já foi dada ou não**, para além de declarar as permissões no manifesto. A próxima tarefa consiste em adicionar as três linhas de código seguintes à sua implementação (mesmo antes de `TextView oTV = (TextView) findViewById(...);`). Note que pode ser necessário fazer alguns ajustes ao código para fechar o bloco corretamente:

```
1 if( checkSelfPermission(Manifest.permission.READ_CALL_LOG) != 0 )
2     // 0 = PackageManager.PERMISSION_GRANTED
3     requestPermissions(new String[]{Manifest.permission.READ_CALL_LOG},1);
4 else {
5 ...
6 }
```

Q25.: O que é que fazem as três linhas de código sugeridas nesta tarefa?
☐ Só estorvam!
☐ A linha 1 verifica se as permissões já foram dadas; a linha 3 pede as permissões.
☐ A linha 3 verifica se as permissões já foram dadas; a linha 1 pede as permissões.
☐ A linha 2 é um comentário. Dah! (Mas explica o significado do 0 dentro do `if`. Incrível!)

Q26.: Por que é que o método `requestPermissions` leva um *array* de *strings*?
☐ Porque se fosse só uma *string* (em vez de um *array*), estávamos apenas no nível normal, e eu sou nível *LE-GEN-DA-RY*!
☐ Porque posso ter que pedir mais do que uma permissão simultaneamente.

Esta é para fazer em casa. **Q27.: O método `requestPermissions(.,.)` é síncrono ou assíncrono?**
☐ É o que? ☐ É síncrono. ☐ É assíncrono.

Esta é para fazer em casa. **Q28.: Qual o significado do segundo parâmetro (um `int`) no método `requestPermissions(.,.)`?**



23 Preferências Partilhadas, Armazenamento Interno e Externo

Sumário

Estudo de como se podem usar vários tipos de armazenamento para dados persistentes em dispositivos com Android, com foco no recurso de preferências partilhadas, armazenamento interno e externo.

Summary

Study concerning the usage of the several storage options for persistent data, provided by devices with Android, with focus on the shared preferences, internal and external storage resources.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Ficheiros Disponibilizados como Recurso do Projeto

O objetivo deste guia laboratorial é o de construir uma aplicação móvel Android totalmente funcional, partindo de parte do que já foi aprendido antes e adicionando mecanismos, recursos e funcionalidades relacionados com o armazenamento de dados persistentes (não estruturados). O objetivo principal é que esta aplicação permita escrever e guardar notas de uma forma muito simples.

Tarefa 1

Depois de se certificar que tem um dispositivo virtual a correr e que tem uma plataforma alvo superior à 4.2 disponível, crie e teste (usando o Android Studio™) um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `exStorage1`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.exstorage1`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 2

Crie a subdiretoria `raw`, dentro da diretoria `res`. Dentro dessa subdiretoria crie depois um ficheiro chamado `instructions.txt` com o seguinte texto:

```
To use this app, start writing your note down, and exit or save anytime
you want.
1. If you exit, any partial note will be saved automatically but not
sent anywhere.
2. When you feel the note is complete, just save it and send it to your
e-mail.
It will be made available in your external storage also.
```

O ficheiro de texto anterior contém as instruções de utilização da aplicação que vai construir, e o seu conteúdo irá ser mostrado sempre que for relevante.

Tarefa 3

Compile a aplicação e procure saber se este ficheiro (`instructions.txt`) é automaticamente mapeado no ficheiro `R.java`.

Q1.: Ainda se lembra em que diretoria é que este `R.java` fica alojado?
☐ Já me esqueci...
☐ Claro que lembro! Humm... Mas só vou verificar para ter a certeza. Fica em:

_____

☐ Sim, lembro! Fica em: _____

Q2.: O ficheiro é mapeado no ficheiro `R.java`?
☐ Não, não é.
☐ Sim, é, ficando com o identificador `R.raw.instructions`.
☐ Sim, é, ficando com o identificador `R.id.instructions`.

Q3.: O identificador para o qual é mapeado o ficheiro tem a ver com o seu nome?
☐ Sem dúvida. ☐ Nem por isso.

Tarefa 4

A aplicação a construir será muito simples. A funcionalidade que oferece é apenas a de permitir que um utilizador escreva notas e as guarde no armazenamento externo, ou as transmita via uma ação de partilha. A aplicação será constituída por uma única atividade que deve conter os seguintes objetos interativos (*widgets*):

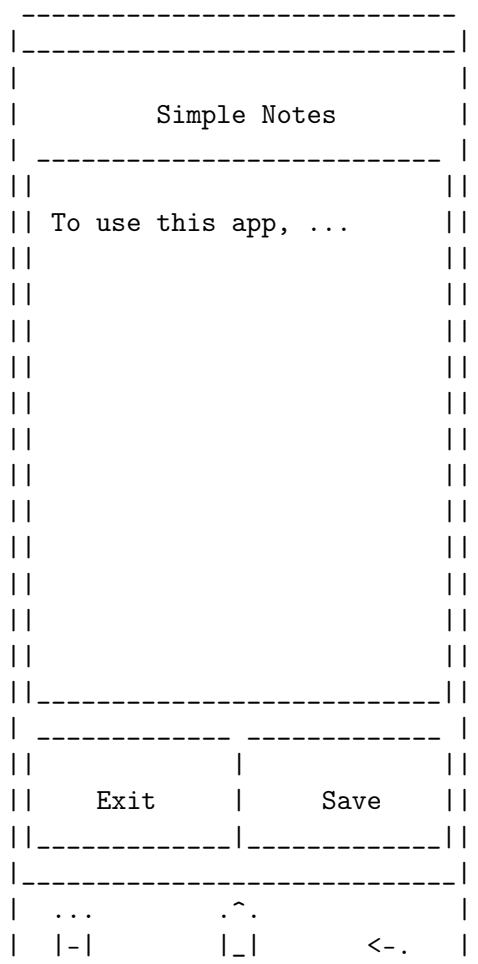
- Uma etiqueta de texto (`TextView`) com o nome da aplicação (`Simple Notes`) centrada ao cimo;
- Uma caixa de texto (`EditText`), que deve ocupar todo o espaço que não esteja ocupado por mais nenhum objeto;
- Um botão com o texto `Exit`, que deve poder ser usado para sair da aplicação; e
- Um botão `Save`, cuja funcionalidade é a de salvar a nota, introduzida na caixa de texto no armazenamento externo, e sair da aplicação.

Tome as providências que achar necessárias para que a atividade principal da aplicação tenha o aspeto mostrado a seguir. Note que deve procurar a forma de **forçar a que**

caixa de texto ocupe todo o espaço deixado livre pelos outros objetos, e que também os dois botões devem ocupar, horizontalmente, todo o espaço disponível.

Q4.: A definição da largura e da altura dos objetos interativos é obrigatória nas versões mais recentes do Android?
☐ Sim, é; caso contrário dá erro de compilação!
☐ Sim, é...
☐ Não, não é.

Q5.: O que é que acontece se definir a largura ou a altura de um *widget* a zero (0) (*e.g.*, com `android:layout_width="0"`)?
☐ Esse *widget* não se vê por ter altura e largura 0.
☐ Esse *widget* passa a ocupar o ecrã todo.
☐ Esse *widget* ocupa sempre o máximo que consegue nessa dimensão tendo em conta outros elementos gráficos e as suas restrições.
☐ Esse *widget* fica com altura a mais e largura a menos ou vice-versa.



```
|_____________________________|
```

Tarefa 5

Já que está a editar o ficheiro XML que define o *layout* da atividade principal, aproveite para definir os métodos `onClick` para os dois botões e um identificador para a caixa de texto. Especifique os métodos `exitNotSave` e `sendSave` para os botões `Exit` e `Save`, respetivamente, e o identificador `etext` para a caixa de texto, *i.e.*, adicione as seguintes linhas nos locais corretos:

```
android:id="@+id/etext"
```

```
android:onClick="exitNotSave"
```

e

```
android:onClick="sendSave"
```

Tarefa 6

Espera-se que a aplicação tenha o seguinte comportamento:

1. Quando é iniciada pela primeira vez, deve mostrar as instruções na caixa de texto;
2. Se um utilizador escrever uma nota na caixa de texto e sair usando o botão `Exit`, a aplicação deve guardar o conteúdo que está na caixa de texto, mas apenas para poder retomar o estado da próxima vez que o utilizador voltar à aplicação;
3. Se o utilizador carregar no botão `Save`, a nota deve ser guardada num ficheiro do armazenamento externo, a aplicação deve sair e, quando voltar a entrar, devem ser novamente mostradas as instruções (como se tivesse sido feito um *reset*).

Para já, implemente o código que lhe permita mostrar o conteúdo do ficheiro `instructions.txt` na caixa de texto. Se estiver com tempo, considere ainda a seguinte questão. Pelo texto, parece ser possível ler de ficheiros que são colocados na subdiretoria `res/raw`. **Q6.: Também é possível escrever nesses ficheiros?**

☐ Claro. Por que não?
☐ Sim, é, mas com muito jeitinho.
☐ Só a própria aplicação é que pode escrever nesses ficheiros.
☐ Não, não é possível escrever nesses ficheiros, principalmente porque estarão dentro do arquivo `apk` aquando da sua execução.

Tarefa 7

Compile, instale e teste a aplicação tantas vezes quantas forem necessárias para conseguir o objetivo da tarefa anterior. Resolva os vários problemas que for encontrando, nomeadamente relacionados com exceções de leitura e escrita em ficheiros, com alguma pesquisa.

Nota: caso precise capturar e tratar exceções, considere escrevê-las no *log* do sistema.

Q7.: Quais os pacotes que necessitou incluir para concluir esta parte do guia?
☐ `import android.os.Environment;`
☐ `import android.content.SharedPreferences;`
☐ `import java.io.InputStream;`
☐ `import java.io.FileInputStream;`
☐ `import java.io.FileOutputStream;`
☐ `import java.io.IOException;`
☐ `import java.io.File;`
☐ `import android.widget.EditText;`
☐ `import android.view.View;`
☐ `import android.util.Log;`
☐ Não havia lá mais?

2 Preferências Partilhadas

Note que será necessário guardar, de alguma forma, e entre utilizações da aplicação, se determinada nota já foi guardada de forma persistente ou não (*i.e.*, se é necessário mostrar as instruções ou a nota anteriormente começada). Para isso, vamos fazer uso do recurso chamado `SharedPreferences`.

Tarefa 8

Considere analisar o seguinte excerto de código Java e incluí-lo, completando-o, no método `onCreate(Bundle)`:

```
SharedPreferences oSP = getPreferences(0);
if( !oSP.getBoolean("recover",false) ){
// Code to populate the text box with the
// instructions in the instructions.txt
// file.
}else{
// Code to inicialize the text box with the
// text: "This functionality has not been
// implemented yet."
```

```
}
```

Q8.: Para que serve o inteiro no método `getPreferences(int)`?
☐ Este inteiro define qual o ficheiro de preferências a abrir (os nomes dos ficheiros de preferências são dados por `numero.xml`).
☐ Este inteiro define a quantos ficheiros de preferências vai aceder.
☐ Este inteiro define o modo de acesso ao ficheiro.
☐ Este inteiro define quantas variáveis vão ser acedidas ou guardadas no ficheiro de preferências.
☐ Este inteiro é sempre igual a `0xff` na API 255.

Q9.: Em que diretoria é que o ficheiro das preferências partilhadas é normalmente guardado?

A função `getBoolean(string, boolean)` aceita uma *string* e um `boolean`. **Q10.: Para que serve o `boolean`?**
☐ Este valor deve ser `false` quando queremos obter o valor que está guardado com a chave `recover`; e `true` quando queremos substituir esse valor.
☐ Este valor deve ser `true` quando queremos obter o valor que está guardado com a chave `recover`; e `false` quando queremos substituir esse valor.
☐ Este valor é devolvido de novo pela função `getBoolean()` caso a chave-valor não exista no ficheiro de preferências.
☐ Esta variável é usada para definir qual é o tipo primitivo da variável que se quer obter.

Q11.: É possível guardar tipos complexos (*e.g.*, objetos) nas `SharedPreferences`?
☐ Não. Só tipos simples primitivos.
☐ Sim, pode-se guardar tudo tudo o que quisermos exceto, talvez, dados estruturados. Esses não! Mas de resto podemos guardar tudo.

Tarefa 9

Compile, instale e teste a aplicação. Não avance antes de se certificar de que tudo está bem até esta parte do guia.

Q12.: Quais os pacotes que necessitou incluir para concluir esta parte do guia (para além dos que já tinha assinalado antes)?

☐ `import java.io.File;`
☐ `import android.view.View;`
☐ `import android.os.Environment;`
☐ `import android.content.SharedPreferences;`
☐ `import java.io.FileOutputStream;`
☐ `import java.io.FileInputStream;`

Tarefa 10

Implemente os dois métodos que tratam o evento de clique nos dois botões definidos. Num dos métodos (`exitNotSave`) deve colocar código que permita colocar a variável `recover` a `true`. No outro (`sendSave`), deve colocar código que permita ajustar a variável `recover` a `false`.

Q13.: Qual é a classe do objeto que lhe permite ajustar os valores guardados nas preferências partilhadas?
☐ `SharedPreferences` ☐ `Editor` ☐ `Adjuster` ☐ `Property`
☐ `Activity` ☐ `putMethod()`

Q14.: Qual ou quais os nomes dos métodos que lhe permitem guardar, de facto, as alterações que estiver a introduzir nas preferências partilhadas?
☐ `forFact()` ☐ `Save()` ☐ `Commit()` ☐ `Apply()` ☐ `Reply()` ☐ `Undo()`

Q15.: Para que serve o método `Undo()`, enunciado na questão anterior?
☐ Para refazer uma determinada ação no objeto.
☐ Para desfazer uma determinada ação no objeto.
☐ Este método não existe. *Busted*!

Tarefa 11

Compile, instale e teste a aplicação. Note que deve testar ambos os botões e verificar se a aplicação já exibe o comportamento esperado.

3 Armazenamento Interno

Anteriormente, foi dito que, caso o utilizador carregasse no botão Exit, qualquer nota que estivesse na caixa de texto deveria ser salva temporariamente, para que quando voltasse, esta ainda persistisse na caixa de texto. Para conseguir este efeito, faça uso do armazenamento interno. Esta secção foca-se, portanto, na implementação de duas funcionalidades diferentes:

1. Aquela que permite guardar o conteúdo da caixa de texto num ficheiro;
2. Aquela que permite restaurar o conteúdo desse ficheiro para a caixa de texto.

Tarefa 12

Foque-se na implementação do método `exitNotSave(View v)`. Recorde quando é que este método é executado:
☐ Nunca é executado.
☐ Quando a aplicação vai para segundo plano.
☐ Quando carregamos no botão `Exit`.
☐ Logo após a execução do método `onCreate()`.

Considere simplesmente copiar o código seguinte para o método mencionado em cima:

```
try{
  FileOutputStream fosFile = openFileOutput("savednote.txt",0);
  EditText oET = (EditText) findViewById(R.id.etext);
  fosFile.write( oET.getText().toString().getBytes() );
  fosFile.close();
}catch(IOException e){ Log.v("SIMPLENOTES","FILE IO PROBLEM"); }
```

Q16.: O que faz o código incluído antes?
☐ Abre e fecha um ficheiro.
☐ Abre um ficheiro, escreve algo nesse ficheiro, e depois fecha o ficheiro.
☐ Abre um ficheiro, lê o seu conteúdo e escreve-o na caixa de texto.

Q17.: Qual o significado do número 0 no método `openFileOutput(string, int)`?
☐ Que o ficheiro aberto não pode ser fechado.
☐ Que o ficheiro é criado de novo com 0 bytes.
☐ Que o ficheiro é criado no modo privado, o que significa que só a aplicação é que lhe pode aceder.
☐ É um erro. Não se pode abrir um ficheiro para leitura especificando um 0 no segundo parâmetro.

Q18.: Lembra-se de ter implementado, no método `exitNotSave(View v)`, um editor para as preferências partilhadas?
☐ Sim, lembro. ☐ Não, não lembro.

Quando implementou essa parte, deve ter colocado o método `commit()` ou o `apply()` no método que agora está a completar. **Q19.: Pense bem: onde é que faz mais sentido colocar esses métodos?**
☐ Dentro do bloco `try{...}catch{...}`
☐ Antes do bloco `try{...}catch{...}`
☐ Depois do bloco `try{...}catch{...}`
☐ Como estou na dúvida, meto em tudo quanto é lado.

Tarefa 13

Note que falta implementar parte do código no método `onCreate(Bundle)`, nomeadamente aquela secção que restaura o conteúdo de uma nota inacabada na caixa de texto. Use o que já aprendeu até aqui para completar esta parte do código. **Sugestão: use a linha de código incluída a seguir.**

```
FileInputStream fisFile = openFileInput("savednote.txt");
```

Q20.: Em que diretoria do sistema de ficheiros Android é que o ficheiro `savednote.txt` é guardado?

Q21.: É possível verificar a existência do ficheiro usando o `Android Monitor`?
☐ Olha! Boa ideia! ☐ Não, não é possível.

Tarefa 14

Compile, instale e teste a aplicação. Saia e entre várias vezes da aplicação carregando no botão `Exit` e alterando o conteúdo da caixa de texto, para se certificar de que o que implementou nesta parte do guia está correto.

4 Armazenamento Externo

As duas funcionalidades que estão em falta dizem respeito ao botão `Save`.

Tarefa 15

No método referido antes, implemente a parte do código que permite guardar a nota escrita na caixa de texto num ficheiro do armazenamento externo. O ficheiro deve chamar-se `note.txt`, e a diretoria **pública** onde guarda o ficheiro pode ser, por exemplo, a das imagens (não faz muito sentido), que costuma estar sempre disponível em emuladores ou dispositivos com Android. Em princípio, vai precisar das seguintes instruções:

```
File path  = getApplicationContext().getExternalFilesDir(Environment.
    DIRECTORY_PICTURES);
File fFile1 = new File(path, "note.txt");
```

```
FileOutputStream fosFile = new FileOutputStream(fFile1);
...
```

No excerto de código anterior, o objeto `path` é declarado como um `File` (um ficheiro).
Q22.: Isto faz sentido?
☐ Em Linux faz todo o sentido, visto que tudo, inclusive as diretorias, são ficheiros.
☐ Não faz sentido nenhum. O Prof. devia ter arranjado outro nome para a variável.
☐ O método `getExternalStoragePublicDirectory(...)` não devolve um `File`, mas sim um `Directory`.

Q23.: Lembrou-se de colocar o `commit()` no local certo da função?
☐ Se não fosse o Prof., não sei o que seria de mim.
☐ Atão não lembrei?

Q24.: Precisou de importar pacotes adicionais para esta parte do guia laboratorial?
☐ Sim, nomeadamente o(s) pacote(s):

☐ Não, já tinha tudo o que precisava.

Q25.: Tem fechado todos os ficheiros que tem aberto?
☐ Eh... estava só a guardar essas partes assim mais para o fim da implementação...
☐ Eh... sim sim, tenho fechado tudo.

Tarefa 16

Compile, instale e teste a aplicação. Saia e entre várias vezes da aplicação carregando no botão `Save` e verificando que o ficheiro `note.txt` é gerado ou reescrito com o conteúdo correto.

Nota: pode eventualmente revelar-se útil a instalação de um gestor de ficheiros no emulador (para navegar pelo sistema de ficheiros, nomeadamente pelo armazenamento externo). Há várias formas de instalar pacotes num emulador (*e.g.*, usado `$ adb install...`). Uma dessas formas consiste em fazer o *download* do arquivo `.apk` via *browser* do próprio emulador, instalando-o depois de o selecionar na pasta *downloads*. Neste âmbito, talvez deva considerar uma visita a http://www.appsapk.com/es-file-explorer/.

Tarefa 17

Finalmente, note que também foi pedido que fornecesse a possibilidade do conteúdo da caixa de texto ser enviada por *e-mail* ou SMS. Isto deve acontecer ao mesmo tempo que a nota é guardada no ficheiro do armazenamento externo, *i.e.*, ao ser pressionado o botão `Save`. Esta última tarefa consiste em implementar esta funcionalidade (usando um `ACTION_SEND`). No final, teste a aplicação.



24 Bases de Dados SQLite

Sumário

Desenvolvimento de uma pequena aplicação para manipulação de uma base de dados local. Análise e exploração da *shell* SQLite3 fornecida com o SDK Android.

Summary

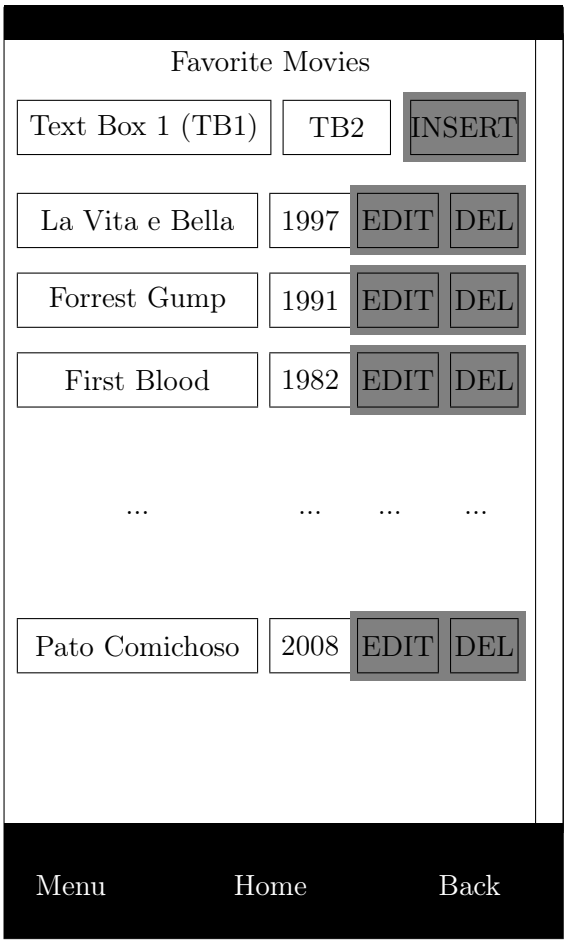
Development of a small application for manipulating a local database. Analysis and exploration of the SQLite3 shell provided with the Android SDK.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Criação de Bases de Dados Locais SQLite

Os objetivos deste guia são construir uma aplicação simples para criação e manipulação de uma base de dados local, e ganhar alguma agilidade na depuração de problemas relacionados com bases de dados SQLite. A aplicação a construir deverá não só permitir visualizar os dados de uma base de dados relativa a filmes, como também permitir a inserção, edição e eliminação de registos. No final da execução deste guia, a aplicação desenvolvida deve ter um aspeto semelhante ao que se ilustra na figura seguinte:



Tarefa 1

Inicialmente, a descrição centra-se na criação da base de dados. Comece por criar e testar (usando o Android Studio™) um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `exStorage2`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.exstorage2`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;

- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 2

Crie um novo ficheiro de código Java para implementar o ajudante de criação de bases de dados. Este ficheiro deve ser criado, por exemplo, na diretoria `src\main\java\pt\ubi\di\pmd\exstorage2`. O nome da classe que irá ajudar a criar a base de dados será `AjudanteParaAbrirBD`, pelo que o nome do ficheiro deve estar a condizer com a resolução tomada. O conteúdo desse ficheiro deve ser algo semelhante ao seguinte:

```
package pt.ubi.di.pmd.exstorage2;

import android.database.sqlite.SQLiteDatabase;
import android.content.Context;

public class AjudanteParaAbrirBD
  extends SQLiteOpenHelper {
  private static final int DB_VERSION = 1;
  private static final String
          DB_NAME = "FavoriteMovies";
  protected static final String
          TABLE_NAME = "Movie";
  protected static final String COL1 = "id";
  protected static final String COL2 = "name";
  protected static final String COL3 = "year";

  private static final String CREATE_MOVIE =
    "COMPLETAR COM A INSTRUCAO SQL CORRETA";

  public AjudanteParaAbrirBD(Context context) {
    super(context, DB_NAME, null, DB_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(CREATE_MOVIE);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db,
```

```
    int oldVersion, int newVersion) {
    db.execSQL("DROP TABLE "
            + TABLE_NAME + ";");
    db.execSQL(CREATE_MOVIE);
  }
}
```

Q1.: O que tem a dizer acerca da afirmação seguinte?
É necessário declarar a classe `AjudanteParaAbrirBD` *no* `AndroidManifest.xml` *para que esta possa ser usada por uma componente da aplicação.*
☐ Esta afirmação é estapafúrdia. ☐ Esta afirmação está correta.

Note que o código anterior tem uma instrução SQL em falta. Complete-a no seu ficheiro de código (recorra aos valores das *Strings* estáticas) e use o espaço seguinte para a anotar, para referência futura. Quando estiver a construir esta instrução, considere que:

- os nomes dos filmes não terão mais do que 50 caracteres, mas que alguns nomes se podem repetir, inclusive no mesmo ano;
- por causa do que foi dito anteriormente, vai precisar de um `id` para cada filme, que será chave primária da tabela.

Q2.: Já agora, o que significa o S do acrónimo SQL?

Significa _____.

Q3.: Que nome se dá às instruções SQL usadas assim no contexto de outra linguagem de programação?
☐ *Embedded SQL* ☐ *Included SQL* ☐ *Host Language*
☐ *Ghost SQL* ☐ *SQLlite* ☐ NoSQL

Q4.: Ainda no contexto da questão anterior, que qualificação se dá à linguagem Java?
☐ Linguagem pai. ☐ Linguagem embutida. ☐ Linguagem anfitriã.
☐ Linguagem hotel. ☐ Linguagem hospedeira. ☐ Linguagem convidada.

Q5.: De acordo com o código incluído antes, o que é que acontece à base de dados durante um *upgrade*?
☐ Nada.
☐ A base de dados é destruída e recriada.

☐ A única tabela da base de dados é eliminada e depois recriada. Todos os registos que lá estavam são mantidos.
☐ A única tabela da base de dados é eliminada e depois recriada. Quaisquer registos anteriores presentes na base de dados são perdidos.
☐ A base de dados é reescrita (`@Override`).
☐ A base de dados ganha um bilhete só de ida para o `stderr`.

Tarefa 3

Depois de compilar de novo o código e de se certificar de que não há erros a tratar, procure entender o seguinte excerto de código. Compare a sua implementação da atividade principal com o código seguinte, e procure completá-la com as instruções que efetivamente criam e abrem a ligação à base de dados:

```
package pt.di.ubi.pmd.exstorage2;

import android.database.sqlite.SQLiteDatabase;
...

public class ActivityMain extends Activity {
  private SQLiteDatabase oSQLiteDB;
  private AjudanteParaAbrirBD oAPABD;

  @Override
  protected void onCreate(Bundle state){
    super.onCreate(state);
    setContentView(R.layout.main);

    oAPABD = new AjudanteParaAbrirBD(this);
    oSQLiteDB = oAPABD.getWritableDatabase();
  }
  @Override
  protected void onResume(){
    super.onResume();
    oSQLiteDB = oAPABD.getWritableDatabase();
  }
  @Override
  protected void onPause(){
    super.onPause();
    oAPABD.close();
  }
}
```

Q6.: Falta algum *import* em alguma das classes implementadas antes?
☐ Não falta absolutamente nada.
☐ Falta importar `android.database.sqlite.SQLiteOpenHelper` no ficheiro `AjudanteParaAbrirBD.java`

☐ Falta importar `android.database.sqlite.SQLiteOpenHelper` no ficheiro `MainActivity.java`.
☐ Falta importar a classe `AjudanteParaAbrirBD` no ficheiro `MainActivity.java`.

Tarefa 4

Prepare, instale e execute a aplicação. Use o *Device File Explorer* do Android Studio™ (normalmente no lado direito em baixo) para se assegurar de que a base de dados foi realmente criada. **Q7.: Qual a diretoria onde as bases de dados são criadas?**

Q8.: A que tipo de armazenamento é que a diretoria que indicou antes pertence?
☐ Armazenamento Android.
☐ Armazenamento de bolachas.
☐ Armazenamento interno.
☐ Recursos do projeto.
☐ Armazenamento em série.
☐ Armazenamento para o inverno.
☐ Armazenamento externo.
☐ Preferências partilhadas.

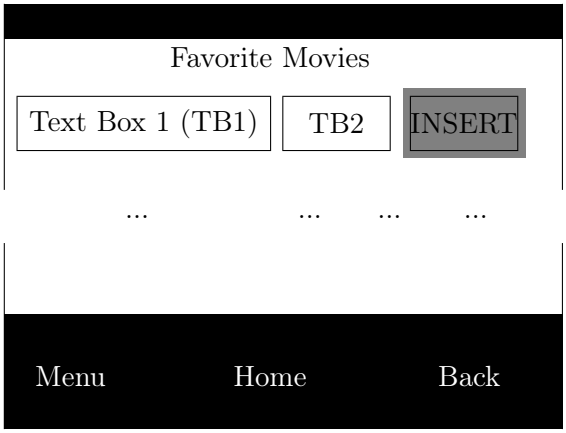
Q9.: Foi criado mais algum ficheiro juntamente com a base de dados?
☐ Que estranho: sim, foi!
☐ Não. Qu'ideia!

2 Inserção de Dados em Bases de Dados SQLite

A aplicação a desenvolver é minimalista, mas com bastante funcionalidade, porque terá de lidar não só com a exibição dos registos da base de dados, como também com a inserção edição e eliminação de valores. Comece por se focar na secção superior da interface de utilizador, que é a que irá permitir introduzir valores na base de dados.

Tarefa 5

A parte superior da interface de utilizador da única atividade da aplicação é composta por uma etiqueta de texto centrada numa linha, e por duas caixas de texto e um botão na outra linha, conforme se representa a seguir:



Esta tarefa consiste em editar o ficheiro XML que define o *layout* da atividade de modo a que se pareça com a que foi ilustrada antes. Note que a segunda linha, que mostra as caixas de texto e o botão, deve ocupar toda a largura. Neste caso, terá de utilizar um `ScrollView` para obter o resultado pretendido. Vai também precisar de definir um `ConstraintLayout`:

```
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent" >

  <androidx.constraintlayout.widget.ConstraintLayout
  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"
    tools:context=".MainActivity">

  <TextView
    android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Favorite Movies"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"/>

  <EditText
    android:id="@+id/name"
    android:text="Movie Name"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    app:layout_constraintHorizontal_weight="3"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toLeftOf="@id/year"
    app:layout_constraintTop_toBottomOf="@id/text" />
```

```
  <EditText
    android:id="@+id/year"
    android:text="year"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    app:layout_constraintHorizontal_weight="1"
    app:layout_constraintLeft_toRightOf="@id/name"
    app:layout_constraintRight_toLeftOf="@id/INSERT"
    app:layout_constraintTop_toBottomOf="@id/text" />

  <Button
    android:id="@+id/INSERT"
    android:text="ADD"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintHorizontal_weight="1"
    android:onClick="onINSERTclick"
    app:layout_constraintLeft_toRightOf="@id/year"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/text"/>

  </androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
```

Q10.: O que pode dizer acerca do tamanho da primeira caixa de texto, em relação aos outros dois objetos interativos da mesma linha, depois de definir o *layout* seguindo as indicações do guia?

☐ Que ficou com o mesmo tamanho que a outra caixa e que o botão.
☐ Que ficou com o dobro do tamanho da outra caixa e do botão.
☐ Que ficou com o triplo do tamanho da outra caixa e do botão.
☐ Que o largura da segunda linha ficou dividida em 5, sendo 3 desses pedaços são ocupados pela primeira caixa de texto.

Tarefa 6

Implemente o método `onINSERTclick(View)`, que é despoletado pelo clique no botão `INSERT`. Siga a sugestão providenciada a seguir:

```
public void onINSERTclick( View v ){
  ContentValues oCV = new ContentValues();
  EditText oED1 = (EditText)findViewById(R.id.name);
  EditText oED2 = (EditText)findViewById(R.id.year);
  oCV.put(oAPABD.COL2, oED1.getText().toString() );
  // FALTA INSTRUCAO
  oSQLiteDB.insert(oAPABD.TABLE_NAME, null ,oCV);
}
```

Q11.: Que classes (adicionais) vai precisar de importar para que o código

compile corretamente?

- ☐ `android.app.Activity;`
- ☐ `android.database.sqlite.SQLiteDatabase;`
- ☐ `android.content.ContentValues;`
- ☐ `android.os.Bundle;`
- ☐ `android.view.View;`
- ☐ `android.widget.EditText;`

Tarefa 7

Compile, instale e teste a aplicação (na medida do possível). Aproveite para inserir um ou dois filmes da sua preferência usando as caixas de texto disponíveis.

Experimente, já agora, colocar uma palavra na caixa de texto referente ao ano, e verifique se a aplicação se queixa ou não desse facto.

3 Depuração da Base de Dados

Tarefa 8

Nota: esta tarefa requer a utilização de um emulador Android ou de um dispositivo móvel onde possua privilégios de administração (*i.e.*, acesso `root`). Como anda não foi implementada forma de mostrar registos da base de dados na atividade (apenas de os inserir), o ideal será recorrer a outros meios para verificar se a aplicação está a funcionar corretamente. Para isso, sugere-se que use a ferramenta `sqlite3`, fornecida com o SDK e nos emuladores Android. Considere as seguintes questões antes de evoluir para a tarefa em si.

Q12.: Em que diretoria do SDK está o executável que lhe permite abrir uma *shell* SQLite3 na sua máquina?

- ☐ `platform-tools`
- ☐ `system-images`
- ☐ `sqlite-tools`
- ☐ `tools`
- ☐ `build-tools`
- ☐ `ferramentas-back-and-decker`

Q13.: Qual o comando que lhe permite ver quais os dispositivos que estão disponíveis via `adb`?

- ☐ `$ adb list devices`
- ☐ `$ adb devices`
- ☐ `$ abc defg hijklmnop qrstuv wxyz`
- ☐ `$ adb please show devices`
- ☐ `$ adb mobile phones`

Tarefa 9

Construa o comando `adb` que lhe permite aceder à *shell* do emulador, e emita-o na sua consola (considere que tinha dois dispositivos ligados):

`> adb` _____

Tarefa 10

Uma vez na *shell*, navegue até à diretoria onde a base de dados deve estar guardada e emita o comando:

```
$ sqlite3 FavoriteMovies
```

Nota: caso obtenha erros de permissões, experimente correr o comando seguinte, mas só depois de descobrir para que serve:

```
$ run-as pt.ubi.di.pmd.exstorage2
```

Tarefa 11

A *shell* disponibilizada pelo comando `sqlite3` integra uma série de comandos interessantes. **Q14.: Qual é o comando que permite ver a descrição de todos os outros?**

☐ `.aid` ☐ `HELP` ☐ `SOS`
☐ `.description` ☐ `.help` ☐ `rainbow dash`

Q15.: Agora que já tem forma de ver todos os comandos disponíveis, qual é o que permite ver todas as tabelas contidas na base de dados?

☐ `.databases` ☐ `SELECT TABLES FROM DATABASE` ☐ `.tables`
☐ `SHOW TABLES` ☐ `.show`

Q16.: Verificou se a tabela `Movies` foi criada, de facto, na base de dados em análise?

☐ Verifiquei e sim, existe! ☐ Verifiquei e não, não existe!
☐ Não verifiquei e sim, existe! ☐ Não verifiquei e sim, não existe!
☐ Não verifiquei e não, existe!

Engendre e emita a instrução que lhe permite verificar se existem registos na base de dados, nomeadamente aqueles que inseriu via aplicação. Use o espaço seguinte para guardar a instrução, para referência futura:

_____

_____

Tarefa 12

O objetivo desta tarefa é o de ficar a conhecer um pouco melhor o SQLite. Para isso, considere os seguintes desafios e questões.

Os autores do SQLite definem-no da seguinte forma:
*SQLite is a software library that implements a **self-contained**, **serverless**, **zero-configuration**, **transactional** SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.*

Procure saber os significados das palavras realçadas a negrito no excerto anterior.

self-contained: _____

_____

serverless: _____

_____

zero-configuration: _____

_____

transactional: _____

_____

Antes de continuar, responda às seguintes questões:
Q17.: Em que linguagem foi escrito o SQLite?

_____

Q18.: Em quantos ficheiros distintos se encontra a implementação do SQLite, e qual a razão principal que levou os seus programadores a fazer a implementação desta forma?

Está implementado em _____ ficheiros, e foi assim implementado para que _____

Q19.: O SQLite permite acesso concorrente?
☐ Essa é boa! Então depois de discutirmos a simplicidade do sistema e o facto de ser *serverless*, ainda queremos que permita acesso concorrente à mesma base de dados por várias aplicações?
☐ Permite.

Q20.: Qual das seguintes opções concretiza uma situação para a qual o SQLite não garante as propriedades ACID de uma transação?
☐ Uma falha do programa;
☐ Uma falha do sistema operativo;
☐ Uma falha de energia.

Q21.: Como se sai do SQLite?

Q22.: É possível obter as instruções que definem a criação das tabelas?
☐ Não, não é.
☐ Sim é, nomeadamente através do comando `.tables`.
☐ Sim é, nomeadamente através do comando `.schema`.
☐ Sim é, nomeadamente através do comando `.creates`.
☐ Sim é, nomeadamente através do comando `.description`.

4 Listagem e Remoção de Registos

Tal como ilustrado na primeira figura que define o *layout* da aplicação, a parte central da atividade é composta por um conjunto de *widgets* que se repete tanta vezes quantas necessário para mostrar toda a informação contida na tabela `Movie`. Este conjunto vai repetir-se um número de vezes igual ao número de linhas na tabela, eventualmente ultrapassando o limite vertical do ecrã para mostrar informação. Essa possibilidade vai ter de ser, assim, acomodada.

Tarefa 13

A próxima tarefa consiste em adicionar uma nova *View* ao ficheiro de *layout* com o conteúdo sugerido a seguir (adicione o excerto de código seguinte na última linha do `ConstraintLayout` que já tem definido no XML, *i.e.*, imediatamente antes do fecho da *tag* `</ConstraintLayout>` existente):

```
<LinearLayout
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  app:layout_constraintTop_toBottomOf="@id/INSERT"
  android:id="@+id/llsv">

</LinearLayout>
```

Q23.: O que é que faz uma `ScrollView`?

☐ Uma `ScrollView` é um contentor especial, que oferece a funcionalidade de *scrolling* sempre que o seu conteúdo ultrapassa os limites do ecrã.

☐ Uma `ScrollView` é um contentor igual aos outros, que pode ser usado para fazer a separação de resíduos.

☐ Uma `ScrollView` é um contentor como os outros, que oferece a funcionalidade de *scrolling* sempre que o seu conteúdo ultrapassa os limites do ecrã.

Q24.: Nota algum detalhe estranho no excerto de código adicionado?

☐ Não.

☐ Sim, o `LinearLayout` interno não contém qualquer objeto.

☐ Sim, o `ScrollView` não pode conter um `ConstraintLayout`.

Q25.: Pode colocar objetos interativos, *i.e.*, *widgets*, (*e.g.*, um botão) diretamente dentro de um contentor do tipo `ScrollView`?

☐ Sim, sem problemas.

☐ De facto, a documentação parece sugerir que não, que dentro de um `ScrollView` só pode ser colocado outro objeto do tipo *contentor*.

Tarefa 14

Vai ser preciso criar um *layout* para cada uma das linhas que apresenta a informação para os filmes na base de dados. Cada uma destas linhas é composta por duas caixas de texto e dois botões (EDIT e DEL), dispostos na horizontal. A primeira caixa de texto, por se referir ao nome do filme, deve ocupar o dobro do que ocupa cada um dos outros objetos. Para manter a complexidade baixa, propõe-se que crie um novo ficheiro de *layout* só para definir cada uma das linhas mencionadas. O ficheiro deve chamar-se `line.xml`, e deve ser colocado na mesma diretoria que o `activity_main.xml`. O seu conteúdo deve ser o que se apresenta a seguir:

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

```
  <EditText
    android:layout_width="0dp"
    android:layout_height="fill_parent"
    android:gravity="left"
    android:id="@+id/ED1"
    android:layout_weight="2" />
  <EditText
    android:layout_width="0dp"
    android:layout_height="fill_parent"
    android:gravity="center"
    android:id="@+id/ED2"
    android:layout_weight="1" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:gravity="center"
    android:text="EDIT"
    android:id="@+id/EDIT"
    android:onClick="onEDITclick" />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:gravity="center"
    android:text="DEL"
    android:id="@+id/DEL"
    android:onClick="onDELclick" />
</LinearLayout>
```

Note que cada um dos *widgets* do *layout* anterior tem um ID definido, e que os últimos dois botões especificam métodos de *callback*.

Tarefa 15

Compile o projeto, instale e teste a aplicação antes de prosseguir.

Tarefa 16

A implementação da parte de listagem dos resultados de uma consulta à base de dados é uma das mais exigentes que já foi tentada nas aulas práticas, talvez não pela sua complexidade, mas por ser necessária a adição de objetos interativos programaticamente, que ainda não foi tentada antes. O racional elabora nos seguintes passos:

1. Faz-se uma consulta (`SELECT`) à base de dados para obter todos os registos;
2. Para cada linha devolvido pelo resultado, adiciona-se um novo conjunto de *widgets* ao `LinearLayout` que está vazio dentro da `ScrollView > ConstraintLayout`.

Para ajudar neste processo, sugere-se uma abordagem faseada, em que o código é dado gradualmente.

Comece por **adicionar** as duas instruções seguintes ao método `onCreate()` da sua atividade:

```
LinearLayout oLL = (LinearLayout) findViewById(R.id.llsv);
Cursor oCursor = oSQLiteDB.query(oAPABD.TABLE_NAME, new String[]{"*"} ,null
    , null, null, null, null,null);
```

Use o espaço seguinte para indicar os *imports* que também precisa adicionar ao seu código para poder compilar o projeto com sucesso:

Procure agora explicar o que faz cada uma das linhas anteriores:

Linha 1:

Linha 2:

Como medida de precaução, pode considerar compilar o projeto, para saber se não foram introduzidos erros ou se falta algum *import*.

Tarefa 17

Adicione agora, e após as duas linhas anteriores, o excerto de código seguinte:

```
boolean bCarryOn = oCursor.moveToFirst();
while( bCarryOn ){
  // FALTA CODIGO
  bCarryOn = oCursor.moveToNext();
}
```

Note que faltam instruções no excerto anterior, mas deve conseguir compilar o projeto de qualquer forma. **Q26.: O que faz o excerto de código anterior?**

- ☐ Coloca o `Cursor` a apontar para a linha imediatamente antes da primeira, movendo-o de seguida para a primeira linha.
- ☐ Coloca o `Cursor` a apontar para a primeira linha, e itera-o até chegar ao fim da tabela de resultados.
- ☐ O código contém um erro, e coloca a aplicação num *loop* interminável.

Q27.: O que é que devolve o método `moveToNext()`?
☐ Verdadeiro caso tenha conseguido mover-se para a próxima linha, e falso caso contrário;
☐ Verdadeiro caso tenha chegado ao fim da tabela, e falso caso contrário;
☐ Falso caso tenha chegado ao fim da tabela, e verdadeiro caso contrário.

Tarefa 18

Para terminar, e no local onde falta o código no ciclo `while` anterior, considere colocar agora as seguintes linhas de código Java, respondendo depois às questões que se lhe seguem:

```
LinearLayout oLL1 =
    (LinearLayout) getLayoutInflater()
    .inflate(R.layout.line, null);
oLL1.setId(oCursor.getInt(0)*10+4);

EditText oED1 =
    (EditText) oLL1.findViewById(R.id.ED1);
oED1.setId(oCursor.getInt(0)*10+2);
oED1.setText(oCursor.getString(1));

EditText oED2 =
    (EditText) oLL1.findViewById(R.id.ED2);
oED2.setId(oCursor.getInt(0)*10+3);
oED2.setText(oCursor.getInt(2)+"");

Button oB1 =
    (Button) oLL1.findViewById(R.id.EDIT);
oB1.setId(oCursor.getInt(0)*10+1);

Button oB2 =
    (Button) oLL1.findViewById(R.id.DEL);
oB2.setId(oCursor.getInt(0)*10);

oLL.addView(oLL1);
```

Q28.: O que faz o método `inflate()`, presente na primeira linha de código do excerto anterior?
☐ Incha o `LinearLayout` até este não caber mais no ecrã, ficando a rebentar pelas costuras.
☐ Lê o conteúdo do recurso dado por `R.layout.line`, convertendo-o para um objeto da aplicação.
☐ Lê o conteúdo do recurso dado por `line.xml`, convertendo-o para um objeto da aplicação.

Q29.: O que fazem todas as linhas de código que contêm o método `setId(.)`?

☐ Na verdade mudam o identificador de cada objeto interativo a que se referem, de maneira a refletir a chave primária da tabela *Movie*.
☐ Na verdade deixam o identificador de cada objeto interativo a que se referem tal como estava.
☐ Na verdade colocam todos os identificadores dos objetos a que se referem com o mesmo valor, *i.e.*, o valor da chave primária da linha atual.

Considere que, a determinado momento, o cursor `oCursor` estava sobre a linha com `id` igual a 5. **Q30.: Qual o ID atribuído a cada um dos objetos interativos representados?**

`oLL1` _____

`oED1` _____

`oED2` _____

`oB1` _____

`oB2` _____

Repita o exercício anterior, mas agora para quando o cursor `oCursor` está sobre a linha com `id` igual a 12:

`oLL1` _____

`oED1` _____

`oED2` _____

`oB1` _____

`oB2` _____

Q31.: Considera que o esquema que o Prof. engendrou para os IDs dos *widgets* é efetivo?
☐ E como é que podia dizer o contrário, tratando-se do Prof.?
☐ De facto, este esquema garante que todos os *widgets* ficam com um ID único na aplicação!
☐ Não, porque há situações em que os IDs se podem repetir, nomeadamente:

_____

_____

Q32.: O que faz a última linha de código do excerto anterior (*i.e.*, `oLL.addView(oLL1);`)?

☐ Elimina todos os *widgets* redefinidos dentro do `while` ao `LinearLayout` que estava (inicialmente) vazio.
☐ Coloca o conteúdo do `oCursor` nas caixas de texto.
☐ Adiciona todos os *widgets* redefinidos dentro do `while` ao `LinearLayout` que estava (inicialmente) vazio.
☐ Adiciona um novo objeto do tipo `View` à aplicação.

Tarefa 19

Compile o projeto e instale a aplicação. Adicione vários filmes à base de dados através da funcionalidade que já havia implementado numa parte anterior deste guia laboratorial, ou através da *shell* SQLite3. Entre e saia da aplicação várias vezes entre inserções à base de dados, para estimar se a funcionalidade antes implementada está a funcionar ou não.

Tarefa 20

Foque-se agora na funcionalidade associada ao botão DEL. Para eliminar algo de uma base de dados SQLite, basta fazer uso do método `delete()`, que aceita o nome da tabela, a *String* que define a cláusula `WHERE` e, eventualmente, os parâmetros de entrada desta última. Para o ajudar nesta tarefa, sugere-se a utilização do seguinte excerto de código, onde falta apenas um parâmetro, que deverá preencher:

```
public void onDELclick ( View v ){
  oSQLiteDB.delete(
    // FALTA PARAMETRO,
    "ROWID="+v.getId()/10,
    null);

  LinearLayout oLL1 =
      (LinearLayout) findViewById(v.getId()+4);
  ((LinearLayout) oLL1.getParent()).removeView(oLL1);
}
```

Q33.: O que fazem as últimas duas linhas de código do excerto de código anterior?
☐ A primeira linha cria uma instância do contentor a retirar e a segunda linha retira-o do seu elemento pai.
☐ A primeira linha cria uma instância do contentor que contém o elemento a retirar e a segunda linha retira-o do seu elemento pai.
☐ A primeira linha cria um filho, a segunda cria uma filha, e não se entendem.

Tarefa 21

Prepare, instale e teste a aplicação.

5 Edição de Registos

Tarefa 22

Note que a funcionalidade de edição ainda não foi implementada, mas deve elaborar num racional semelhante ao que já foi feito para a eliminação e inserção de registos. A tarefa final deste guia consiste então na codificação da parte em falta, nomeadamente do método *callback* `onEDITclick(View)`.



25 Serviços e Recetores de Difusão

Sumário

Desenvolvimento de uma pequena aplicação móvel que utiliza 3 componentes disponíveis na plataforma Android para exercitar os conceitos associados aos Serviços e Recetores de Difusão desta plataforma, bem como dos conceitos e características das *Threads*.

Summary

Development of a small mobile application that uses 3 components available in the Android platform with the objective of practicing concepts related with Services and Broadcast Receivers, as well as the concepts and characteristics of Threads.

Pré-requisitos:

Algumas das tarefas enunciadas a seguir requerem o acesso a um sistema com o Android Studio™ e com o SDK Android, bem como com o Gradle™ instalados ou, alternativamente, com permissões para instalação e configuração do IDE, *kit* e ferramenta. Serão suficientes permissões para criar diretorias e ficheiros num disco local e para configurar variáveis de sistema, nomeadamente a *path*. É necessário ter acesso a uma versão e imagem da plataforma Android ou a um dispositivo físico com o sistema operativo e com a opção de *debug* ativa. É igualmente necessário ter um compilador Java instalado.

1 Componente Serviço

Este guia laboratorial tem como objetivo explorar a componente Serviço de aplicações Android e a segmentação de tarefas em *threads*. Também procura dar a conhecer a *User Interface* (UI) *Thread*, que é aquela que domina o fluxo de execução por defeito numa

aplicação Android.

O guia está estruturado de forma a construir, de forma faseada, e através de tentativa e erro, uma só aplicação móvel. De forma a atingir os vários objetivos propostos ou a enfatizar alguns aspetos, alguns exercícios levam, contudo, a aplicação para concretizações erradas, que depois são solucionadas. Sugere-se, assim, que vá guardando as várias versões da aplicação, ou que faça aplicações diferentes quando achar pertinente.

Tarefa 1

Comece por criar e testar (usando o Android Studio™) um projeto padrão de uma aplicação Android sobre o qual vai elaborar nas próximas tarefas. Considere as seguintes sugestões aquando da preparação do projeto:

- Nome da aplicação/projeto — `exServices1`;
- Domínio — `pmd.di.ubi.pt` (nome do pacote deve então ser semelhante a `pt.ubi.di.pmd.exservices1`);
- Escolha a linguagem de programação Java (em vez de Kotlin). Caso seja explicitamente pedido, desative o suporte para C++;
- Deve ser um projeto para *smartphone* ou *tablet*, mínimo API 21;
- Com uma `Empty Activity` chamada `MainActivity`;
- Se essa opção for pedida explicitamente, peça para gerar o ficheiro de *layout* (o nome do ficheiro de *layout* deve ser `activity_main.xml`);
- Caso seja explicitamente pedido ou estiver disponível, retire qualquer suporte de retrocompatibilidade. Caso contrário, substitua `AppCompatActivity` por `Activity` na classe `MainActivity.java`.

Note que os nomes sugeridos antes devem ser seguidos com rigor, já que deles depende, por vezes, o funcionamento bem sucedido da aplicação a ser desenvolvida.

Tarefa 2

Altere o *layout* da atividade principal de modo a que esta contenha apenas uma etiqueta de texto e um botão, ambos centrados no ecrã. A etiqueta de texto deve dizer `Floating Alarms Application` e o botão deve dizer `Start Service`. Quando pressionado, o botão deve despoletar o método `onButtonClick()`.

Tarefa 3

Prepare, instale e execute a aplicação.

Q1.: A aplicação está a funcionar?
☐ Sim, funciona na perfeição, e já mostra alarmes e tudo!
☐ Sim, funciona, mas o botão ainda não faz nada...
☐ Não há nada que funcione!

Tarefa 4

Crie um novo ficheiro `.java` na diretoria `app\src\main\java\pt\ubi\di\pmd\exservices1` com o nome `ServiceAlarms.java`. Dentro desse ficheiro, coloque o seguinte excerto de código:

```
package pt.ubi.di.pmd.exservices1;

import android.content.Intent;
// MISSING IMPORTS

public class ServiceAlarms extends Service {

  @Override
  public int onStartCommand
  (Intent intent, int flags, int startId) {
      Toast.makeText(
        this,
        "Service Started!",
        Toast.LENGTH_SHORT)
      .show();
    return START_NOT_STICKY;
  }

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }
}
```

Faltam alguns *imports* no excerto de código anterior. **Q2.: Assinale, nas opções seguintes, quais são esses *imports* (e complete o código de acordo com a resposta).**
☐ `import android.app.Service`
☐ `import android.os.IBinder`
☐ `import android.util.Log`
☐ `import android.view.View`

Q3.: Em que pacote é que pode ser encontrada a classe `Toast`?
☐ Na classe `import java.lang`.

☐ Na classe `import android.view`, como de resto não havia outra hipótese.
☐ Na classe `import android.widget`, e faz todo o sentido que assim seja.
☐ Na classe `import android.os`, como não podia deixar de ser!

Analise o código anterior atentamente. **Q4.: Era mesmo preciso reescrever o método `onBind()` para o Serviço em questão?**
☐ Neste caso, este método não está lá a fazer nada, porque até devolve `null`. Portanto: não!
☐ Na verdade, sim. Eu até experimentei, e não compila se o método não estiver lá.
☐ À campeão: sim, é preciso.
☐ À campeão: não, é totalmente desnecessário.

Q5.: Dada a implementação do Serviço, acha que este vai ser um `started service` ou um `bound service`?
☐ Nem sei por onde começar a procurar a resposta a isto.
☐ Um *started service*, *i.e.*, um Serviço sem vínculo.
☐ Um *bound service*, *i.e.*, um Serviço com vínculo.

Tarefa 5

Implemente o método `onButtonClick()` na atividade principal de forma a que esta inicie o Serviço implementado na tarefa anterior. Sugestão:

```
import android.content.Intent;
import android.view.View;
...
public void onButtonClick(View v){
  Intent oIntent = new Intent(this, ServiceAlarms.class);
  startService(oIntent);
}
```

Tarefa 6

Prepare, instale e execute a aplicação. **Q6.: O botão funciona?**
☐ Não funciona, e não consigo perceber a razão por detrás desse facto.
☐ Sim, funciona!
☐ Ahhhh... esqueci-me de declarar o Serviço no `AndroidManifest.xml`!

Tarefa 7

Caso ainda não o tenha feito, declare o Serviço que criou no `AndroidManifest.xml`, colocando o seguinte excerto de XML dentro do elemento `application`:

```
<service android:name="ServiceAlarms">
 <intent-filter>
 <action android:name="pt.ubi.di.pmd.ServiceAlarms.SERVICE" ></action>
 </intent-filter>
</service>
```

No final, teste de novo a aplicação e certifique-se de que ficou a funcionar. **Q7.: Qual o artefacto que demonstra que o serviço ficou realmente a correr?**

☐ Não há artefacto, e eu não sou o Indiana Jones.

☐ A aplicação desaparece (porque o serviço começou a correr em segundo plano).

☐ O artefacto é a presença de uma *toast* quando carrego no botão, e que é despoletada no serviço, não na atividade...

Q8.: Já agora, acha que, para as funcionalidades que já foram enunciadas e implementadas, é necessária a definição do filtro de intentos incluído no código anterior?

☐ É estritamente necessária, já que sem essa definição, o Serviço não fica bem declarado no manifesto.

☐ É estritamente necessária, já que sem essa definição, o Intento que despoleta o Serviço na atividade principal não iria funcionar.

☐ Penso ser desnecessário, visto que ainda não vi qualquer funcionalidade que precisasse desta definição.

2 Serviços e *Threads*

Tarefa 8

Coloque as linhas de código incluídas em baixo depois da emissão da mensagem `Toast` implementada no `ServiceAlarms.java`:

```
try{
  Thread.sleep(7000);
}catch( InterruptedException oIE ){
  Log.e("SERVICEALARMS", "Interrupted Exception!");
}
```

Provavelmente vai precisa de importar as seguintes classes:

```
import java.lang.Thread;
import java.lang.InterruptedException;
import android.util.Log;
```

Q9.: Quanto tempo fica a *thread* adormecida após inclusão das linhas de código anteriores?

☐ 1 segundo. ☐ 7 segundos. ☐ 7 milissegundos. ☐ 1000 segundos.
☐ 7000 segundos. ☐ 7000 milissegundos. ☐ 1000 minutos. ☐ 7000 minutos.
☐ 7000 horas.

Tarefa 9

Prepare, instale e teste a aplicação. **Q10.: Ao carregar no botão, aparece a mensagem `Service Started!`?**

☐ Não aparece... e o meu dispositivo ou aplicação parecem ter *encravado*.
☐ Não aparece... e aplicação parece ter *encravado*, sendo possível, contudo, evoluir para o *Home Screen* com um pouco de paciência.
☐ Não aparece... e no fim de um bocado, aparece uma mensagem a perguntar se quero terminar a aplicação. Má onda!
☐ Aparece sem problemas.

Tarefa 10

Repita as últimas duas tarefas, desta feita, colocando a interrupção da *thread* antes da implementação da mensagem `Toast`.

Q11.: Qual o comportamento da aplicação quando a executa e carrega no botão `Start Service`?

☐ A aplicação tem exatamente o mesmo comportamento de antes.
☐ A aplicação bloqueia (*e.g.*, o botão fica selecionado e não me deixa mexer em nada), mas no fim de 7 segundos, aparece a mensagem `Service Started!` no ecrã!
☐ Desta vez, a aplicação já não *encrava*.

Q12.: Consegue encontrar alguma entrada ANR no `logcat`?

☐ Nem sei o que significa ANR (e sinceramente, nem quero saber)!
☐ Consigo encontrar mas não sei o que significa e, agora tenho uma consulta, pelo que também não posso ir ver do que se trata, mas não fica esquecido.
☐ Consigo encontrar e sei o que significa. Até as galinhas dizem: imp-imp-imp; impecável.

Q13.: Dado o que observou, qual das seguintes afirmações lhe parece ser a mais correta?

☐ A *thread* que adormeceu é a mesma *thread* que trata da interface do utilizador.
☐ A *thread* que adormeceu é diferente da *thread* que trata da interface do utilizador.

Q14.: Qual lhe parece ser a explicação para a diferença entre comportamentos que observou?

☐ Aparentemente, se adormecer a *thread* logo depois de tentar mostrar a mensagem, esta não é mostrada no ecrã, e quando a aplicação recupera já é tarde demais para a mostrar.
☐ Aparentemente, se adormecer a *thread* logo depois de tentar mostrar a mensagem, esta não é mostrada no ecrã, por ter sido cancelada na colocação da *thread* a dormir.
☐ Aparentemente, se adormecer a *thread* antes de tentar mostrar a mensagem, esta não é mostrada no ecrã, e quando a aplicação recupera já é tarde demais para a mostrar.

Q15.: Que nome se dá à *thread* que, entre outras responsabilidades, trata da interação com o utilizador?

---

3 *UI Thread e Handlers*

Note que a ideia principal desta aplicação é implementar um Serviço que, depois de despoletado, emite um alarme em forma de `Toast`, de 5 em 5 segundos, mas que claramente não bloqueie a aplicação. Para isso, possivelmente, terá de recorrer a *threads*.

Tarefa 11

Edite o ficheiro `ServiceAlarms.java` e substitua o código do método `onStartCommand()` pelo seguinte:

```
new Thread(){
  public void run(){
    for(int i = 0; i < 10; i++){
      Toast.makeText(
          ServiceAlarms.this,
          "Service Started!",
          Toast.LENGTH_SHORT)
      .show();

      try{
        Thread.sleep(5000);
      }catch( InterruptedException oIE ){
        Log.e("SERVICEALARMS", "Interrupted Exception!");
      }
    }
  }
}.start();
```

Q16.: Assim só de olhar para o código, acha que esta abordagem vai funcionar?
☐ Tudo parece indicar para que sim.

☐ Vê-se logo que não vai funcionar, nomeadamente porque

Tarefa 12

Compile, instale e teste a aplicação. **Q17.: A aplicação funciona?**

☐ A aplicação é interrompida se carregar no botão `Start Service`, provavelmente porque apanhou a exceção e escreveu no *Log*. Tenho de verificar o *Log*.

☐ A aplicação é interrompida se carregar no botão `Start Service`, provavelmente porque estou a invocar um método para a *UI thread* onde não devia... De qualquer forma, o Android pede desculpa, pelo que não há problema!

☐ A aplicação é interrompida porque a *thread* principal é adormecida demasiado tempo...

☐ A aplicação funciona nos trinques.

Tarefa 13

Existem várias formas de resolver o problema apontado antes. Na verdade, o problema principal (o de estarmos a adormecer a *thread* principal) já está tratado, faltando apenas arranjar forma de, quando necessário, executarmos o método de criação da mensagem `Toast` na *UI Thread*. Para tal, sugere-se a utilização de um `Handler`, que é um objeto que **fica associado à *thread* onde é criado**, permitindo que sejam enviadas mensagens ou objetos `Runnable` para essa *thread* a partir de outras. Em baixo deixa-se uma sugestão para implementação do Serviço em questão:

```
package pt.ubi.di.pmd.exservices1;

import android.app.Service;
import android.content.Intent;
import android.widget.Toast;
import android.os.IBinder;

import java.lang.Thread;
import java.lang.Runnable;
import java.lang.InterruptedException;
import android.util.Log;
// IMPORT MISSING!

public class ServiceAlarms extends Service
{
  Handler oHandler;
  @Override
  public int onStartCommand
  (Intent intent, int flags, int startId) {
```

```
    oHandler = new Handler();
    new Thread(){
      public void run(){
        for(int i = 0; i < 10; i++){
          Runnable oRun = new Runnable(){
            @Override
            public void run(){
              Toast.makeText(
                ServiceAlarms.this,
                "Service Started!",
                Toast.LENGTH_SHORT)
              .show();
            };
          };
          oHandler.post(oRun);

          try{
            Thread.sleep(5000);
          }catch( InterruptedException oIE ){
            Log.e("SERVICEALARMS", "Interrupted Exception!");
          }
        }
      }
    }.start();
    return START_NOT_STICKY;
  }

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }
}
```

Q18.: Qual o *import* que falta no excerto de código anterior?

☐ `import android.util.Handler` ☐ `import android.Handler`
☐ `import android.lang.Handler` ☐ `import android.os.Handler`

Observe o código atentamente. **Q19.: A que *thread* é que o *oHandler* está associado?**

☐ À *UI thread*. ☐ À *thread* secundária. ☐ *Star Wars: Episode I - The Phantom Thread*

Q20.: Quanto tempo dura uma `Toast.LENGTH_SHORT`?

☐ Aproximadamente 1 segundo.
☐ Cerca de 2 segundos.
☐ Mais do que 1,5 segundos, mas menos de 1,8 segundos.
☐ Mais ou menos 3,5 segundos.
☐ Menos do que -2 segundos, mas mais de 3 segundos.

Tarefa 14

Compile, instale e teste a aplicação. **Q21.: Funcionou?**
☐ Às mil maravilhas. ☐ Nem por isso.

Nota: caso tenha feito tudo como deve ser, a aplicação deve ter funcionado. Caso contrário, tome as providências que achar necessárias para resolver os problemas encontrados antes de avançar neste guia.

4 Recetores Difusão

Considere que pretendia que o seu Serviço de alarmes fosse ativado cada vez que o dispositivo móvel (do tipo *smartphone*) recebia uma SMS, independentemente de este ter sido ou não despoletado a partir da atividade principal.

Tarefa 15

Modifique a aplicação que implementou antes de modo a que esta também contemple um recetor de difusão de eventos `android.provider.Telephony.SMS_RECEIVED`. Para conseguir este objetivo, considere as seguintes sugestões:

- Implemente um *BroadcastReceiver*, estendendo (e importando) a classe `android.content.BroadcastReceiver`;
- Reescreva o método `onReceive(Intent)` de forma a que este comece o Serviço pretendido;
- Declare o novo recetor de difusão no manifesto Android, como filho do elemento `<application>` (não se esqueça de declarar também os filtros que lhe permitem receber o evento pretendido, nomeadamente `android.provider.Telephony.SMS_RECEIVED`);
- Declare as permissões que necessita para aceder àquele evento em particular, nomeadamente

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

Tarefa 16

Resolva todos os problemas que eventualmente encontrar. Compile, instale e teste a aplicação num emulador Android ou num dispositivo real que suporte a receção de SMSs

(um *tablet* não deverá servir para esta tarefa). Tome as medidas que achar necessárias para se certificar que o recetor está a funcionar. Por exemplo, num emulador, irá precisar de simular a chegada de uma SMS. Tal funcionalidade pode ser conseguida via `Android monitor`.