Universidade Federal Fluminense – UFF
Escola de Engenharia – TCE
Curso de Engenharia de Telecomunicações – TGT

Programa de Educação Tutorial – PET
Grupo PET-Tele

Tutoriais PET-Tele

Tutorial sobre Programação Android

(Versão: A2019M03D06)

Autor: Rafael Duarte Campbell de Medeiros

Tutor: Alexandre Santos de la Vega

Niterói – RJ

Janeiro / 2019

Apresentação do Tutorial

Este tutorial pretende introduzir o aluno ao mundo da programação de aplicativos para *Android*, apresentando informações básicas sobre a plataforma, o ambiente de desenvolvimento e os principais elementos do projeto. Pretende, também, pontuar as principais dificuldades encontradas, bem como indicar possíveis soluções para estas.

Apesar de ser uma atividade bastante fluente, intuitiva e de fácil aprendizado, desenvolver aplicativos pode ser uma grande dor de cabeça se não soubermos onde procurar as soluções. Sendo assim, o objetivo principal é fornecer ao aluno as primeiras orientações, mostrar exemplos práticos e caminhos que podem ser seguidos.

Por ser um assunto muito extenso, seria contraproducente tentar abordar todas as áreas do desenvolvimento neste tutorial, o que torna impossível a tarefa de falar sobre as áreas afins. Por este motivo, fica pressuposto que o aluno já domina (ou, ao menos, conhece) conceitos básicos da **programação orientada à objetos** e da linguagem **Java**. Para a última seção, em especial, será necessário algum conhecimento de **banco de dados** e **SQL**.

Sumário

1 Introdução

1.1 História da telefonia

Patenteado pela primeira vez em 1876, o telefone apareceu para revolucionar a forma como nos comunicamos. Apesar da requisição de Dom Pedro II pela instalação da primeira linha telefônica em 1877, só em 1998 os primeiros celulares foram ativados no Brasil; nesses mais de 120 anos, um longo caminho foi trilhado. (1)

A mudança na era da comunicação começou no final do século XIX quando, em 1899, **Guglielmo Marcone** dá mais um passo em direção a evolução da comunicação e transmite, pela primeira vez sem fio, uma mensagem em código morse. E foi justamente a telegrafia sem fios que abriu espaço para uma nova forma de comunicação. (2)

Nas primeiras décadas do século XX, a telefonia sem fio começava a ser desenvolvida, ainda que em passos lento; um exemplo foi a foto, capturada ao final da primeira guerra mundial (em 1917), que retrata soldados alemães utilizando um telefone (Figura 1). Duas décadas depois (em 1938), o exército americano começava a desenvolver os primeiros *Walkie Talkies*, rádios AM portáteis que permitiam comunicação em um raio de até impressionantes 8 quilômetros.

A telefonia seguiu se desenvolvendo até que, no início da segunda guerra mundial (em 1940), a Motorola desenvolve o modelo SCR-300 (Figura 2), um rádio FM portátil largamente usado durante a guerra pelas forças aliadas.(2) A essa época, a telefonia sem fio se restringia a rádios de longo alcance que, em seu ápice de mobilidade, podiam ser equipados em carros. (3)

Figura 1: Imagem retirada de (2).

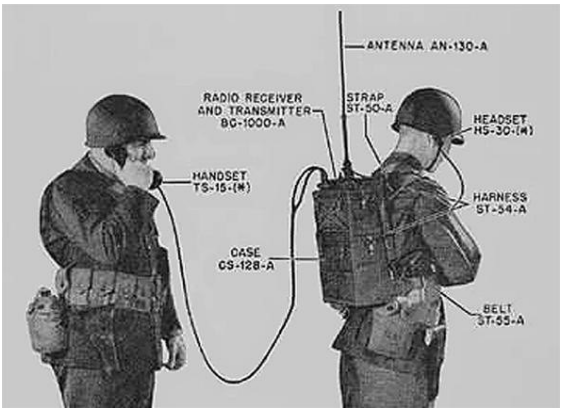

Figura 2: Imagem retirada de (2).

Foi apenas nos Estados Unidos, em 1974, que Martin Cooper apresenta ao mundo o primeiro celular, chamado *Motorola DynaTAC*. Somente após quase uma década, o celular é liberado ao público em 1973; com elevado custo de venda e tarifas de serviço bastante caras, funcionava em rede analógica e já contava com tecnologia para memorizar até 30 números. (4)

Após o sucesso da linha *DynaTAC*, a *Motorola* dá mais um passo na evolução dos telefones: em 1989, cria o primeiro celular com *flip*. Chamado *MicroTAC*, o aparelho, também analógico, tinha o dispositivo de captura de voz dobrado sobre o teclado (como os telefones *flip*), reduzindo significativamente o tamanho e o peso do aparelho. (4)

Foi somente durante a década de 1990 que os celulares começaram a incorporar as tecnologias que conhecemos hoje; como sinal digital e funções além da simples ligação.

Em 1992, o *Simon*, da *IBM*, já anunciava uma novidade ao mundo: os *smartphones*. Apesar de anunciada, a tecnologia ainda não havia sido incorporada pela sociedade.

No final do segundo milênio (em 1998), a *Nokia*, que já vinha tentando lançar algum modelo que superasse os concorrentes, surge com o *Nokia 6160*: um telefone celular pequeno e leve que, por conta de seu preço e facilidade de uso, se tornou o mais vendido da Nokia nos anos 90. (4)

Apesar de ter seus primeiros exemplos no início da década de 1990, foi só no terceiro milênio que os *smartphones* ganharam maior apelo púbico. Capazes de realizar outras tarefas além da ligação, estes aparelhos contavam com tecnologias como tela sensível ao toque (*touchscreen*), PDA (*Personal Digital Assistant*) e fax. E foi assim que a telefonia seguiu seu rumo para o que é hoje em dia: telefones celulares que são capazes de substituir um computador potente na realização de diversas tarefas.

1.2 Sistemas Operacionais e a plataforma *Android*

Um sistema operacional é um conjunto de rotinas e programas que gerenciam recursos, processadores, periféricos e outros; o sistema operacional é responsável pela comunicação entre *hardware* e *software*. Um dispositivo móvel consiste em um computador reduzido, portátil e com funcionalidades inerentes à mobilidade. Um sistema operacional móvel é, como se pode supor pelo nome, um conjunto de rotinas e programas desenvolvidos especificamente para dispositivos móveis.

Apesar de termos ao menos dois grandes sistemas operacionais móveis no mercado e contarmos com tecnologia para desenvolvimento multiplataforma, neste tutorial daremos atenção especial à programação em *Android*, usando o *Android Studio* em específico. Entretanto, antes que demos atenção especial às IDEs e aos códigos, precisamos antes introduzir os primeiros conceitos de arquitetura *Android*.

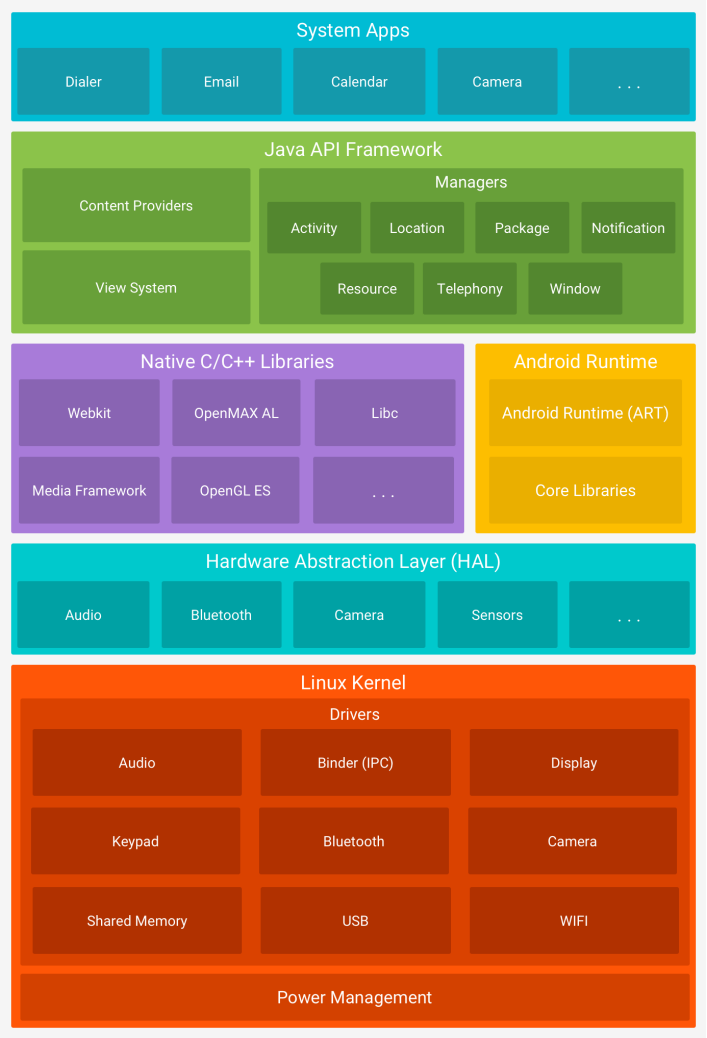

Figura 3: Imagem retirada de (5).

O ***kernel*** (em português, **núcleo**) é o nível de abstração mais baixo, responsável por gerenciar recursos e funcionalidades básicas do sistema. O núcleo faz a ponte das aplicações de *software* com os recursos de *hardware*, sendo ele, na plataforma *Android*, um núcleo *Linux* profundamente modificado. As principais vantagens de usar esse núcleo já bastante utilizado é aproveitar os recursos de segurança e, também, permitir que os fabricantes trabalhem com um núcleo conhecido. (5)

A ***Hardware Abstraction Layer*** (em português, **Camada de Abstração de *Hardware***) fornece interfaces para disponibilizar as funcionalidades do *hardware* em um nível mais alto. Funciona como uma coleção de bibliotecas que viabilizam a comunicação entre as aplicações e os módulos do aparelho, como câmera ou bluetooth. (5)

O ***Android Runtime* (ART)**, para dispositivos com *Android* versão 5.0 (API nível 21), permite que cada aplicativo execute os próprios processos com uma instância individual. Projetado para processar várias máquinas virtuais em dispositivos de baixa memória, executa arquivos em um formato de *bytecode* específico para *Android*, com objetivo de minimizar o consumo de memória. (5)

As chamadas ***Native C/C++ Libraries*** (em português, **Bibliotecas C/C++ Nativas**) são, como o nome sugere, um conjunto de bibliotecas nativas nas linguagens C e C++. Vários componentes do *Android*, como o *ART*, dependem dessas bibliotecas e, apesar da plataforma já oferecer APIs em Java para operar algumas delas, também é possível acessá-las diretamente em seu código. (5)

A ***Java API Framework*** consiste em um conjunto muito rico de recursos do sistema operacional que, programados em Java, permitem ao programador criar aplicações reutilizando componentes e serviços do sistema. Com isso, os programadores têm acesso a todo esse conjunto de *Framework API*s, que são os mesmos utilizados por aplicativos nativos do sistema *Android*. (5)

Por último, na camada mais superficial da estrutura, são as aplicações como são vistas, permitindo ao usuário navegar na internet, ouvir música ou enviar uma mensagem, por exemplo. Vale dizer que as aplicações que o *Android* oferece nativamente não tem nenhuma preferência sobre aquelas que o usuário opte por instalar. Além de disponíveis para o usuário, as aplicações também podem ser acessadas por outros aplicativos, permitindo uma modularização das funções e facilitando o desenvolvimento. (5)

1.3 Ambiente Integrado de Desenvolvimento (*IDE*)

Os Ambientes Integrados de Desenvolvimento (do inglês, *Integrated Development Environment*, ou apenas *IDE*) são *softwares* desenvolvidos para facilitar a tarefa de programar e desenvolver. Reúnem, em apenas uma única aplicação, um conjunto de ferramentas úteis e necessárias para o programador, desde o compilador/interpretador até as funções de colorir código.

Por um lado, uma *IDE* boa fornece muitos poderes ao programador, podendo otimizar o desenvolvimento e maximizar os resultados, reduzindo os erros e facilitando as correções (chamados *bug fix*). Por outro lado, uma *IDE* muito rica pode, facilmente, confundir um usuário iniciante com as várias funcionalidades, o que torna mandatório que conheçamos, ainda que superficialmente, as principais funções e suas utilidades. Tendo isso em mente, apresentaremos ainda nessa seção algumas das funções mais comuns e, em seguida, apresentaremos o ambiente que usaremos no tutorial.

A primeira função importante é o compilador (ou interpretador, a depender da linguagem alvo); é o recurso que permite utilizar o código escrito, viabilizando a execução do mesmo. O processo de compilar consiste em traduzir um código de uma determinada linguagem para outra de baixo-nível (como linguagem de máquina), permitindo que o computador a entenda e execute. Já o processo de interpretação tem um mediador responsável por entender e executar o código, sem a necessidade de traduzir para outra linguagem. É possível, inclusive, executar o processo de forma híbrida, que é o caso da linguagem Java: o código é compilado para uma versão mais simples, permitindo que a JVM (*Java Virtual Machine*, ou Máquina Virtual Java) o interprete.

> Dica: apesar do processo de compilação demorar, no geral, mais do que interpretar diretamente uma linguagem, ele é feito apenas uma vez. Depois de compilado, a versão baixo-nível fica disponível para todas as execuções futuras (enquanto nada for alterado), permitindo que da segunda execução em diante seja mais rápido do que interpretar a cada vez.

Outra função muito importante é o *Debugger* (em português, Depurador), funcionalidade que permite executar o código com vantagens, a fim de facilitar a detecção e correção de defeitos no funcionamento. Apesar de diferentes entre si, a maioria permite executar todo o código (ou só a parte desejada), linha a linha, podendo ver o conteúdo

das variáveis a cada momento e observar as respostas retornadas por cada chamada. Quando bem utilizado, permite detectar não só erros de sintaxe ou falta de permissão, como também perceber problemas na lógica do código.

Existem várias outras funções muito comuns, que são incorporadas por todas as boas *IDE*s. Um exemplo é o ***Syntax Highlighter*** (em português, **marcador de sintaxe**), responsável por colorir as palavras do código de forma diferente a depender de seu tipo (variável, palavra reservada, modificador de escopo e outros). Outras funções são as rotinas de ***Refactor*** (**refatorar**), que consistem em um conjunto de ferramentas para fazer mudanças na estrutura interna de um código (trocar um trecho por outro com mesma função, alterar o nome de uma variável em todos os pontos em que é usada etc.) sem alterar seu comportamento externo.

Para a programação de aplicativos, existem diversas alternativas que trabalham com diferentes linguagens e desenvolvem para diferentes sistemas operacionais. Como neste tutorial pretendemos dar atenção ao *Android*, trabalharemos com o ambiente próprio da plataforma: *Android Studio*.

2 Introdução ao *Android Studio*

2.1 *Download* e instalação

O ambiente que utilizaremos nesse tutorial será o *Android Studio*, uma *IDE* bastante potente e completa, oferecida gratuitamente pela plataforma. Trabalhando tanto com Java quanto com Kotlin, a *IDE* oferece compatibilidade com C++ e NDK, além de dezenas de bibliotecas.

Acesse o *website* por meio da URL *https://developer.android.com/studio/?hl=pt-br*, encontrando a página da Figura 4.

Figura 4: Tela inicial da aba de *download* do *website*.

Caso esteja usando Windows na versão 64 bits, clique em **fazer o download do *Android Studio***; do contrário, entre em **opções de download** e busque a opção que melhor se adeque à sua máquina. Ao escolher sua versão, encontrará a tela da Figura 5.

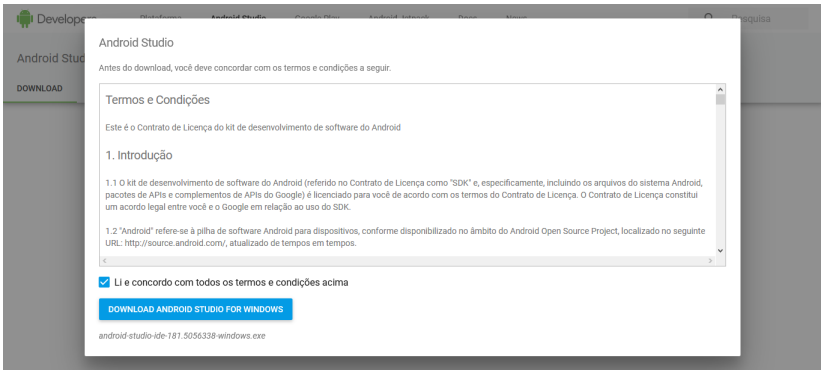

Figura 5: Aceitando os temos e condições.

Leia o termo e marque a caixa que indica **Li e concordo com todos os termos e condições acima**. Depois, pressione o botão em azul que indica **Download**.

Após o término do download, execute o instalador e encontre a tela da primeira imagem da Figura 6. Clique em **Next** nas várias telas até encontrar a tela referente a segunda imagem da mesma figura. Neste ponto, clique no botão **install** e aguarde até ser concluída a instalação da aplicação.

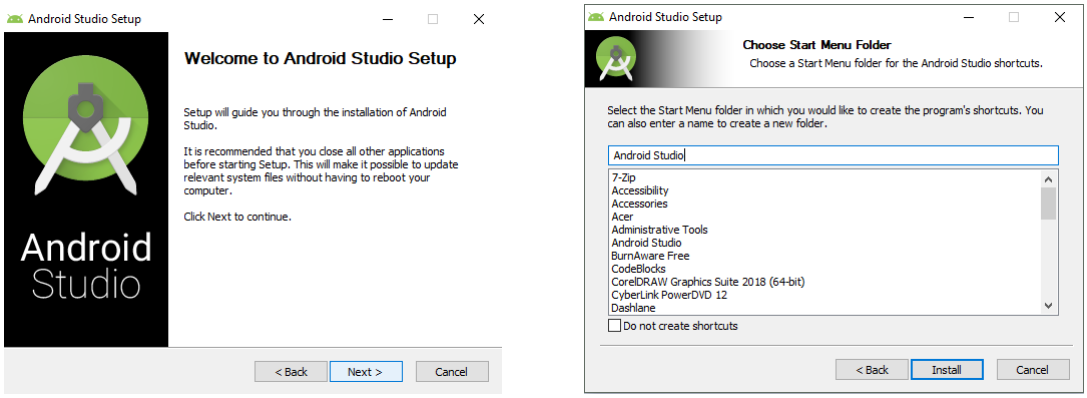

Figura 6: Primeira parte do processo de instalação.

Terminado o processo de instalação, será exibida a tela referente a primeira imagem da Figura 7. Clique no botão **Next** e, ao encontrar tela referente a segunda imagem da mesma figura, pressione a opção **finish**.

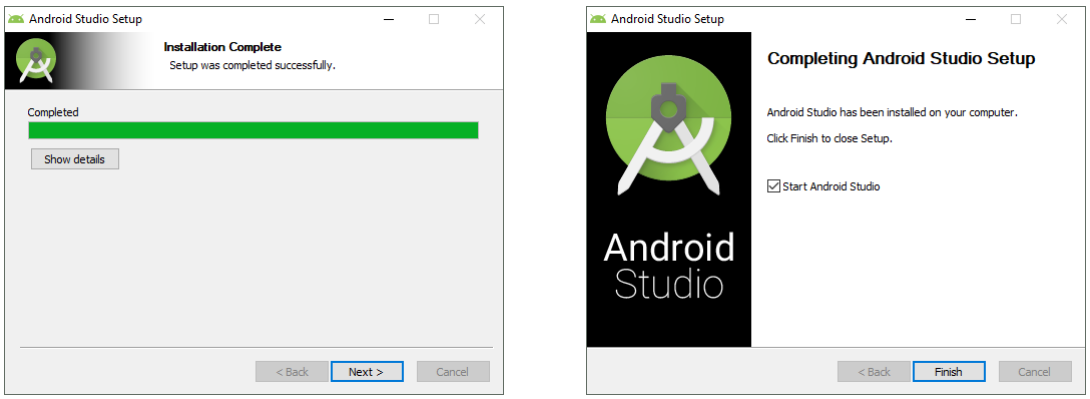

Figura 7: Segunda parte do processo de instalação.

Dica: caso não queira começar a utilizar o ambiente logo após o fim da instalação, desmarque a caixinha que diz **Start *Android Studio***.

Neste ponto, o ambiente já está instalado em sua máquina e já podemos começar a entender um pouco sobre a *IDE* e seus elementos.

2.2 Principais elementos do ambiente

Antes de entendermos sobre a organização do projeto e como trabalhar com ele, vamos conhecer um pouco sobre os elementos do ambiente e, em especial, onde procurar quando quisermos encontrar algo.

Aviso: não se preocupe em encontrar ainda a tela da Figura 13, ela aparecerá ao ser criado o projeto, assunto da próxima seção.

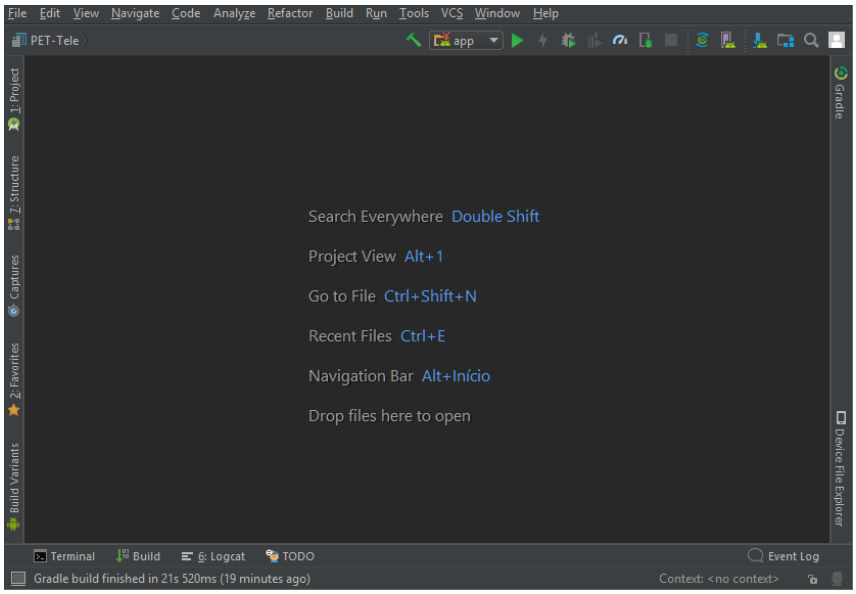

Figura 8: Tela inicial do projeto.

Dica: caso você prefira o ambiente escuro, como no exemplo, basta trocar o tema nas configurações. Para isso, vá em *File* > *Settings* > *Appearance e Behavior* > *Appearance* e, em *UI Options*, mude o *Theme* para **Darcula**.

À direita, vemos uma aba onde diz *Gradle*; esse nome vai se repetir bastante, entendê-lo é importante. O *Gradle* é um sistema de automação de compilação bastante avançado e de código aberto, que reúne conceitos dos sistemas Apache *Ant* e *Maven*. Os sistemas de **automação de compilação** são responsáveis por facilitar uma série de atividades que seriam feitas pelos desenvolvedores, como as verificações, testes, compilação e empacotamento.

O *Gradle* conta com uma série de *scripts* que descrevem configurações do projeto, permitindo até uso de *plugins* para adicionar funcionalidades extras. Uma das características marcantes é, também, seu suporte para **compilações incrementais**; capaz de

identificar quais partes da árvore foram alteradas, ele consegue compilar apenas a parte que falta, não precisando re-executar o projeto inteiro.

O sistema de automação de compilação tem três principais conceitos: *build*, *project* e *task*. Uma *build* é uma versão compilada do projeto, que pode ser tanto um *software* quanto um conjunto de recursos que vão integrar o produto final. Os *projects* compõem a *build* e podem ser tanto itens a serem construídos quanto ações a serem realizadas. As *tasks* são partes do trabalho que uma *build* executa, como a compilação das classes ou a publicação em um repositório.

Na barra inferior à esquerda, temos quatro abas importantes para o desenvolvimento. O desenvolvedor pode, na aba *Terminal*, executar comandos manualmente, como compilação, execução ou, até mesmo, publicação em repositório e resolução de conflitos. Na aba *Build*, o desenvolvedor pode acompanhar o resultado das operações de uma *build*, detalhadamente; inclusive, pode receber indicações do local e do tipo de erro que ocorrer durante a construção.

> Dica: O *Android Studio* oferece grande compatibilidade com sistemas de versionamento, como o *git*, tornando até mesmo a resolução de conflitos bastante tranquila. Num próximo tutorial podemos abordar os sistemas de gerência de versionamento de projetos.

A plataforma *Android* utiliza uma maquina virtual especial (chamada *Dalvik*) que emite as mensagens de erro pela classe *log*; para vê-las, e também qualquer outra mensagem da mesma classe, podemos utilizar o monitor *Logcat* que as exibe em tempo real e, inclusive, mantém um histórico das execução. Esse monitor permite ainda utilizar filtros para buscar a mensagem desejada, podendo ser uma poderosa ferramenta para acompanhar o funcionamento em tempo real do aplicativo e depurá-lo.

Por último, a aba *TODO* permite gerenciar as anotações, facilitando a organização do código e permitindo que se deixe anotada tarefas para serem concluídas depois. Basta deixar um comentário com o prefixo "TODO", o monitor exibe todas as anotações e redireciona o desenvolvedor para o local específico da mensagem.

> Dica: é sempre importante manter bem documentadas os trechos do código e as tarefas a serem realizadas. Um projeto pode parecer bastante compreensível no momento que estamos trabalhando com ele, mas pode ser, depois de alguns meses, muito complicado de se dar suporte.

3 Primeiro projeto

3.1 Criando o primeiro projeto

Iniciado o *Android Studio*, vemos de entrada a tela da Figura 9.

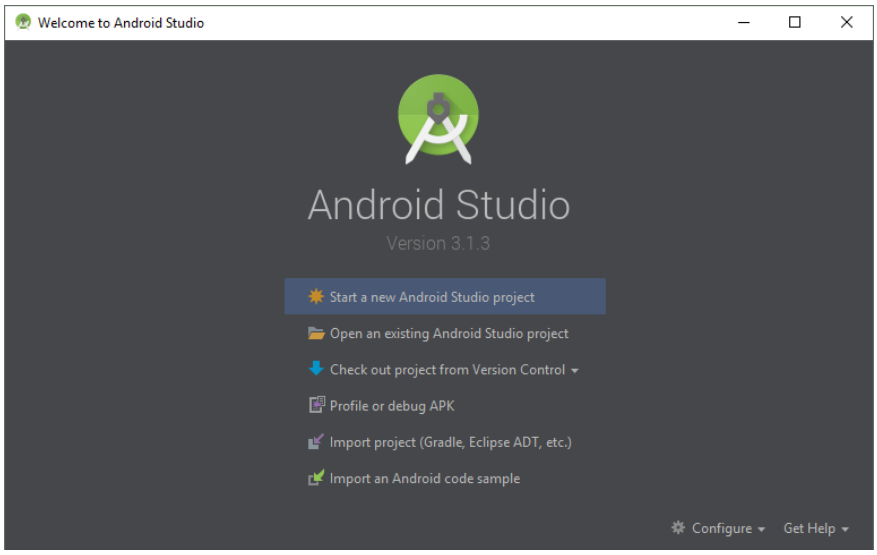

Figura 9: Tela inicial do *Android Studio*.

Nesta tela inicial, temos a opção de criar um novo projeto ou abrir outro já existente, tanto da mesma *IDE* quanto de outros ambientes e sistemas. Para agora, criaremos um novo projeto, clicando na opção ***Start a new Android Studio project***.

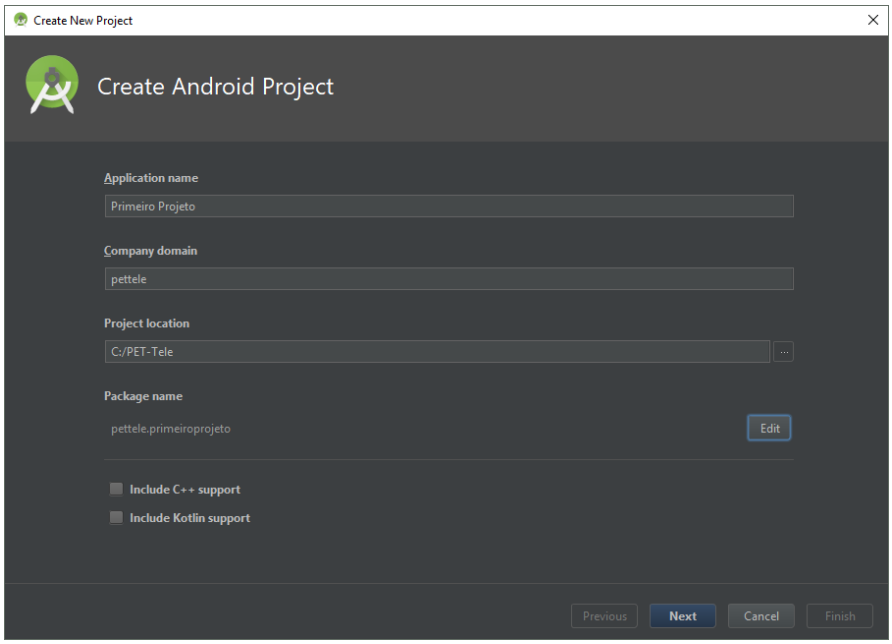

Figura 10: Criando novo projeto.

Na tela seguinte, mostrada na Figura 10, devemos dar um nome (*Application name*), identificar a quem pertence (*Company domain*) e a localização da pasta do projeto (*Project location*). Também podemos incluir suporte para C++ e Kotlin logo nesse passo, mas não será necessário. Após, clique em **Next**.

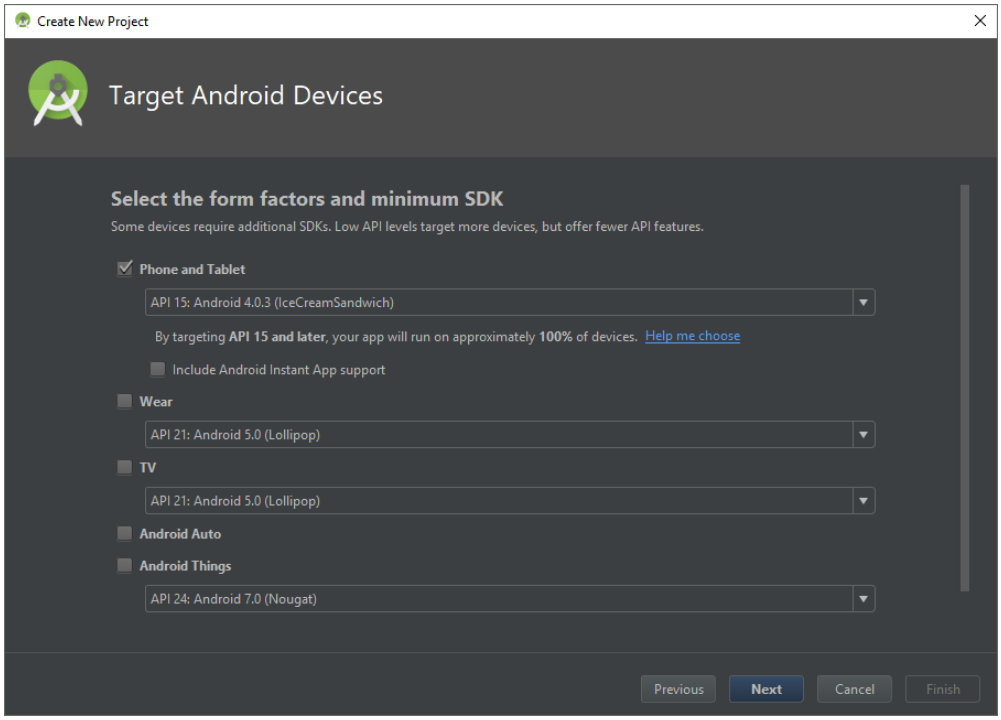

Figura 11: Definindo o suporte oferecido.

A próxima tela, ilustrada na Figura 11, será onde escolheremos a quais dispositivos nosso aplicativo dará suporte, desde sua forma (telefone, tablet, TV etc.) até a versão *Android* mínima necessária (IceCreamSandwich, Lollipop, Oreo etc.).

Apesar das versões mais novas de *Android* fornecerem suporte a tecnologias mais avançadas e permitirem rotinas mais sofisticadas, deve-se observar que, a medida que escolhemos uma versão mais atualizada, diminuímos proporcionalmente a cobertura que podemos oferecer.

> Dica: é possível adicionar, ao código, trechos que só serão considerados quando o aparelho estiver em uma API superior àquela que escolhemos dar suporte no início.

Para este projeto, daremos suporte apenas para telefones e tablets que utilizarem a API 15 na versão **4.0.3 IceCreamSandwich** ou superior. Em seguida, clique em **next**.

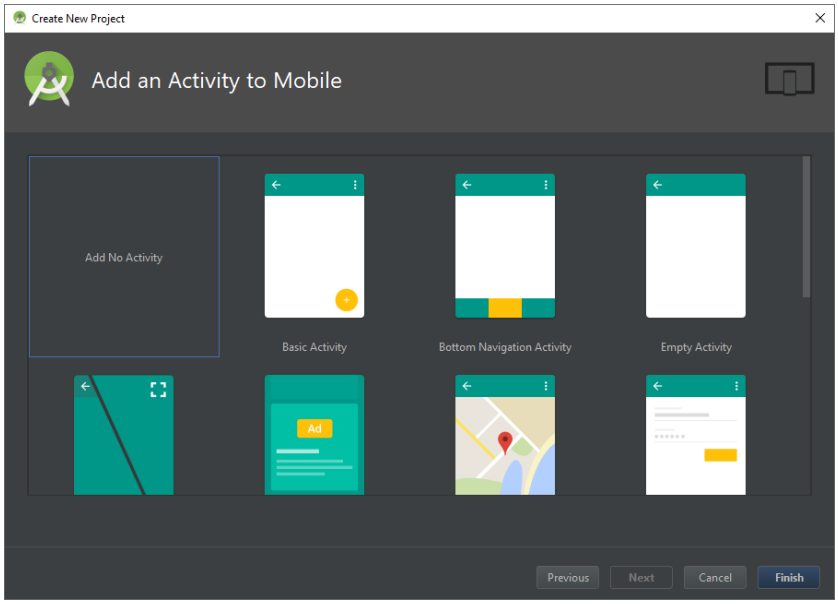

Figura 12: Adicionando *activities*.

A tela seguinte (Figura 12) pergunta se o desenvolvedor quer adicionar alguma *activity* ao projeto, mas como ainda não falamos sobre isso, deixaremos para depois. Por enquanto, selecione a opção **Add No Activity** e clique no botão **Finish**.

Feito isso, o *Android Studio* criará o projeto e as principais pastas. Este processo deve demorar alguns minutos, mas, ao fim, veremos a tela principal do ambiente (Figura 13).

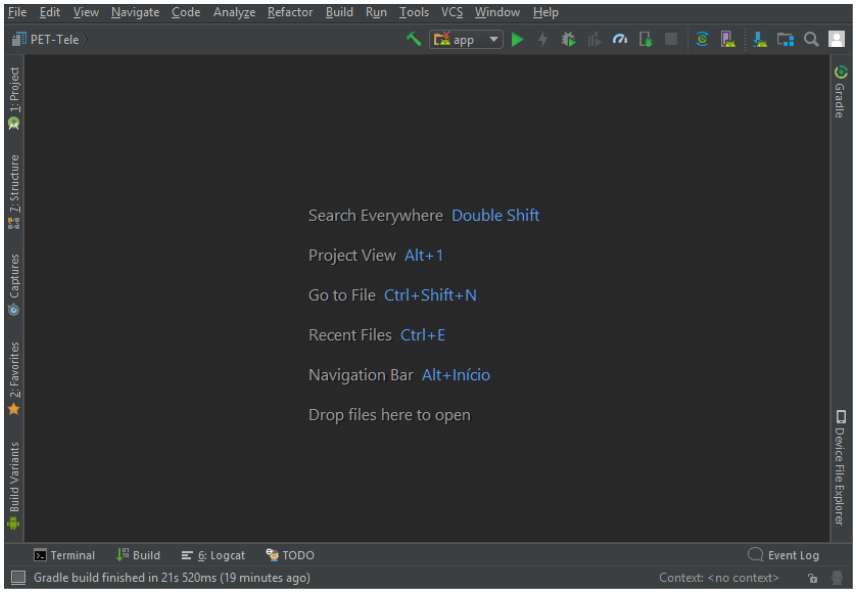

Figura 13: Tela inicial do projeto.

Com o projeto criado, aprenderemos um pouco sobre a estrutura e como utilizá-lo.

3.2 *Activities*

Uma *Activity* é, superficialmente, uma tela do aplicativo. Formalmente, podemos dizer que as *activities* são classes gerenciadoras de interface do usuário (no inglês, *User Interface* ou UI); quando vemos um aplicativo no telefone, estamos olhando para uma de suas *activities*.

3.2.1 Ciclo de vida

O ciclo de vida de uma *activity* é, provavelmente, a parte mais importante de se entender sobre seu funcionamento. Compreender os estados possíveis e os métodos responsáveis pela transição é essencial para programá-las corretamente.

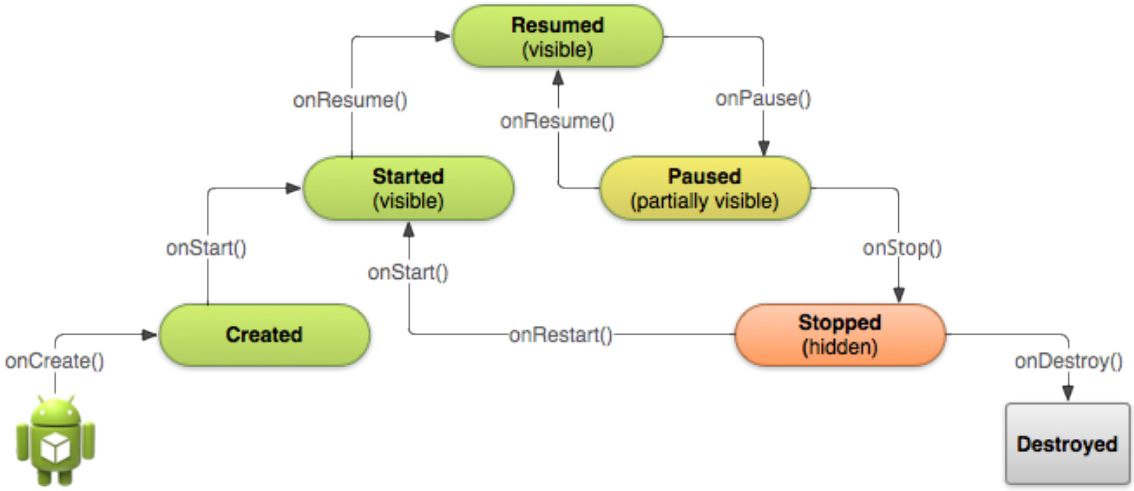

Figura 14: Ciclo de vida de uma *activity*.

Cada estado é chamado por um ou vários métodos e, para que saibamos como a *activity* deve se comportar em cada um, devemos conhecer quando são chamados e para qual estado conduzem:

- **onCreate**: é o primeiro método a ser chamado quando se inicia uma *activity*. Será chamado somente uma vez durante todo o ciclo de vida, por isso é um bom lugar para colocar a construção do conteúdo visual e das instâncias que só serão criadas uma vez.

- **onStart**: é chamado logo após o método *onCreate* e, também, quando a tela volta depois de entrado em segundo plano. É o lugar ideal para se certificar de que os requisitos estejam disponíveis.

- **onResume**: é chamada logo após o método *onStart* e em todos os momentos que a tela volta a ter foco, não necessariamente ao retornar do segundo plano. Bom local para se obter os dados mais atualizados, sendo a última instância visível a ser carregada.

- **onPause**: é chamado quando a tela perde o foco, mas não vai para o segundo plano; quando algo ocupa parcialmente a tela, por exemplo. Pode ser o local para se interromper a execução de tudo que for necessário realizar em foco, como a execução de um vídeo ou propaganda.

- **onStop**: é chamado quando uma tela entra em segundo plano e fica escondida, mas não foi finalizada. É o local para guardar as informações e interromper tudo que não for essencial manter executando, pois o aplicativo estará em segundo plano até ser chamado novamente.

- **onRestart**: é chamado junto ao método *onStart* ao retornar do segundo plano. Pode ser o local certo para se verificar algo que não é necessário logo após a criação, como avaliar o que o usuário fez enquanto o aplicativo estava em segundo plano.

- **onDestroy**: é chamado ao se fechar uma aplicação e é responsável por encerrar todos os processos e destruir a aplicação. Só será chamado uma vez no ciclo e, depois de chamado, não poderá retornar a nenhum outro ponto senão iniciando novamente o aplicativo.

3.2.2 Componentes

Para construir a *activity*, alguns passos devem ser respeitados. O primeiro é definir os atributos visuais, os elementos que comporão o visual da tela; essas características devem ser descritas em um arquivo *XML*. Em seguida, deve-se ter uma classe java que herde a classe ***AppCompatActivity***; essa será responsável por representar a *activitiy*. Essa classe java, através do método **setContentView**, deve ser associada ao arquivo *XML*. Por último, essa classe deve ser declarada no arquivo **AndroidManifest** para que fique disponível ao sistema operacional.

> Aviso: não se preocupe, ainda, com todos esses nomes e informações. Vamos, pouco a pouco, abordar e esclarecê-los.

3.3 Organização das pastas

O primeiro contato com a árvore de pastas pode confundir, afinal são muitos nomes diferentes para se conhecer. Devemos entender ao menos um pouco das pastas, para que as usemos corretamente e mantenhamos um código limpo e bem estruturado.

Na aba vertical à esquerda, clique em **Project** e um campo, como ilustrado na Figura 15, deve se abrir. A primeira coisa a se salientar é que, como indica na barra superior, esta é a organização *Android*; essa organização oculta pastas duplicadas e facilita o entendimento, mas em alguns momentos pode ser interessante usar outras.

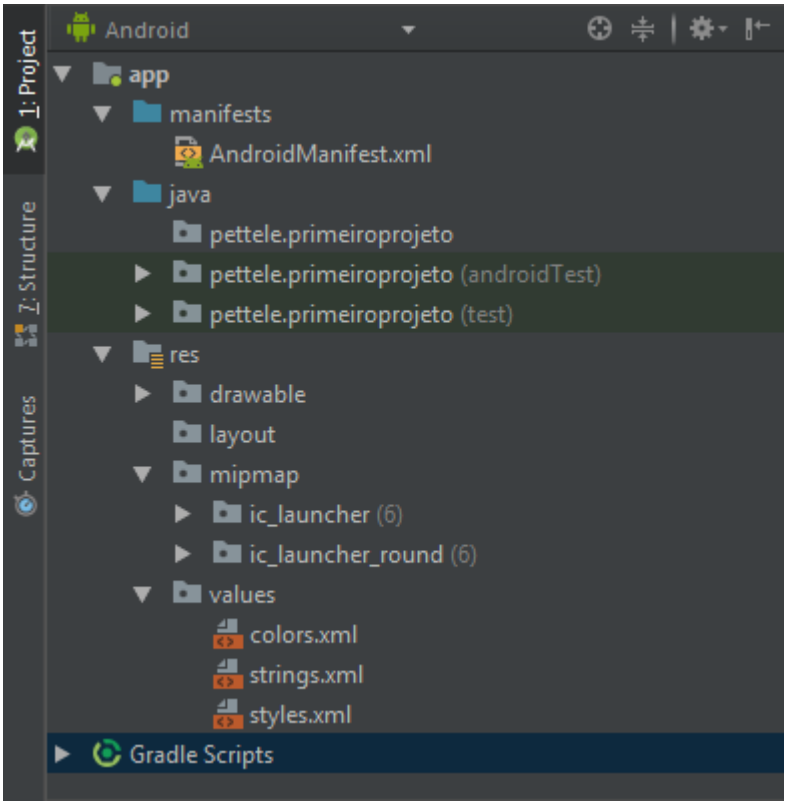

Figura 15: Organização das pastas.

A primeira coisa a se observar é que a árvore se ramifica em **app** e **Gradle Scripts**. É bastante intuitivo observar que a segunda está relacionado com os *scripts* do *Gradle*, automatizador de compilação que já abordamos anteriormente. Ao menos por enquanto, não falaremos desta pasta; daremos enfaze ao ramo do diretório **app**.

O primeiro diretório é o **manifest**, que já conta com um dos principais arquivos: *AndroidManifest*. Esse arquivo, em *XML*, é responsável por declarar para o sistema operacional várias informações, como algumas características e comportamentos das *activities*, as permissões necessárias, a API mínima e outras mais.

Logo a seguir, o diretório **java** é responsável, como é provável intuir, por reunir todos os códigos em java que serão escritos. Percebamos que o diretório é dividido em 3 pastas de mesmo nome, com diferentes anotações ao lado. A primeira, que não conta com nenhum parêntesis, é a principal e é onde deve-se colocar todos as classes java da aplicação. As duas outras pastas são relacionadas a testes, assunto que não abordaremos ainda.

Por último, o diretório **res** (abreviação de **resources** ou recursos, em português) é responsável por guardar todos os arquivos *XML* que definirão os layouts a serem usados, bem como todas as imagens, estilos, cores e demais recursos da aplicação. É subdividido em várias pastas, algumas que já são utilizadas por padrão (no caso, as quatro que aparecem no exemplo) e outras que são utilizadas em situações mais específicas. Por agora, abordaremos as quatro principais e seus usos, sendo elas:

- **drawable**: chama-se tudo aquilo que pode ser desenhado na tela, todo recurso desenhável. Nessa pasta, coloca-se desde arquivos em bitmap (png, jpg etc.) até desenháveis descritos em *XML*.

- **layout**: chama-se um arquivo que descreve a arquitetura de uma interface de utilizador (*UI*) ou parte de uma. São arquivos em *XML* que descrevem os elementos das *activities* ou pedaços que a compõem.

- **mipmap**: usa-se esta pasta para colocar ícones que serão usado pela aplicação, como o ícone que a representará na tela inicial do aparelho. A diferença do *mipmap* para o *drawable* está no uso das diferentes resoluções.

- **values**: essa pasta engloba vários tipos de arquivos que, no geral, guardam padrões para serem usados em outros pontos do código. No caso do arquivo *colors*, ficam definida as cores que serão utilizadas. No arquivo *strings*, ficam, como é de se esperar, as *strings* que serão usadas, como todos os textos. No arquivo *styles*, pode-se definir estilos (grupos de configurações mais gerais), como as cores primárias do tema da aplicação.

  Definição: uma interface de utilizador (*UI*, *User Interface* ou interface de usuário) é tudo aquilo que é visível pelo usuário, responsável pela comunicação entre a aplicação e o utilizador.

Dica: apesar de sempre ser possível definir a cor, texto e até mesmo todo o estilo diretamente nos códigos de *layout*, é uma boa prática que eles fiquem em seus respectivos arquivos e apenas referenciados posteriormente. Organizar dessa forma permite, além de evitar repetições, alterar informações com mais facilidade e precisão, quando necessário.

Sugestão: quando for desenvolver sua aplicação, descreva estilos para cada elemento que deseja manter padrão. Por exemplo, um estilo para as caixas de texto (*TextView*), botões (*Button*) ou caixas editáveis (*EditText*). Dessa forma, fica mais simples de alterar configurações em todas as telas e manter um padrão visual.

Aviso: para além das dicas e sugestões, o *Android Studios* vai apontar um aviso (*Warning*) se algum arquivo estiver fora da pasta que lhe diz respeito. Dessa forma, melhor fica desenvolvimento se nos habituamos a organizar os recursos nas pastas apropriadas.

4 Conhecendo um pouco sobre *layout*

Nessa seção, abordaremos um pouco sobre o processo de desenvolvimento de um *layout*. Para isso, falaremos um pouco sobre os componentes, as ferramentas e os gerenciadores de *layout*. Apesar de existirem, no ambiente, ferramentas visuais para adicionar e organizar os componentes de forma intuitiva (e veremos sobre isso mais adiante), o que fica por trás são códigos em *XML* e conhecê-los é essencial para um bom projeto.

4.1 Introdução ao *XML* e às linguagens de marcação

A sigla *XML* significa e**X**tensible **M**arkup **L**anguage, algo como linguagem extensível de marcação em português. É a linguagem recomendada pela W3C para criação de dados organizados hierarquicamente e é chamada extensível pois permite a definição dos marcadores. (6)

As linguagens de marcações são utilizadas para formatar e exibir informações; aglutinando-as, descrevem a forma como estas devem ser interpretadas e/ou exibidas visualmente. No geral, são orientadas pelo conceito de *tags* ou marcadores, que indicam informações acerca de um dado específico.

Observe o trecho de código a seguir:

```
<receita rendimento="12 cubos" tempo="1 hora">                          1
    <titulo>Gelo</titulo>                                               2
    <ingredientes>                                                      3
        <ingrediente qtd="500" unidade="ml">de agua</ingrediente>       4
        <ingrediente qtd="1" unidade="forma de gelo"/>                  5
    </ingredientes>                                                     6
    <preparo>                                                           7
        <passo>Coloque a forma de gelo em uma mesa</passo>              8
        <passo>Despeje a agua na forma</passo>                          9
        <passo>Coloque a forma no congelador</passo>                    10
    </preparo>                                                          11
</receita>                                                              12
```

Repare que aquilo que chamamos marcadores são os elementos que indicam a abertura (<**texto**>) e fechamento (</**texto**>) de um bloco de informação. Esses marcadores representam um elemento que pode, ou não, aglutinar conteúdo.

Repare que foi aberto um bloco para **ingredientes**; nesse bloco, foram colocados outros elementos **ingrediente** que continham, além de um conteúdo (no caso, o nome do ingrediente), informações sobre si (no caso, quantidade e unidade). Ao encerrar todos os ingredientes, o bloco **ingredientes** foi fechado. O mesmo processo é feito para o bloco de **preparo**, que guarda os **passos** da receita.

Esse código é só um exemplo lúdico, mas a intenção é explicar como funciona a linguagem de marcação. Com um interpretador apropriado, poderia-se ler o código e identificar os elementos, construindo um documento do tipo:

Gelo

Rende 12 cubos em 1 hora

Ingredientes: 500 ml de agua e 1 forma de gelo

Preparo: Coloque a forma de gelo em uma mesa. Despeje a agua na forma. Coloque a forma no congelador.

As linguagens de marcação são importantes para estruturar as informações e, para nós, serão importantes ferramentas para descrever as formas e os elementos que colocaremos em tela, bem como estruturar informações de texto, cor e tamanhos.

> Dica: em linguagens de marcação, é muito comum que se contraia o marcador de fechamento com o de abertura quando não se tem informação entre eles. Por exemplo: <ingrediente qtd="3"></ingrediente>
>
> Como não existe informação entre os marcadores de início e fim, pode-se subtrair último e deixar apenas indicado esse fechamento ao fim do primeiro. Desse modo, ficaria: <ingrediente qtd="3"/>

4.2 Elementos visuais básicos e sua representação em *XML*

Nesta seção, alguns elementos básicos serão apresentados; conheceremos um pouco de seu funcionamento, suas características e como trabalhar com eles em *XML*. Considerando que existem centenas de elementos que podem ser usados, apresentaremos aqui apenas os mais básicos e suas possíveis variações, como as caixas de texto, botões e imagens. Deixaremos para tratar das listas em um outro momento, pois são mais complicadas de se lidar e requerem um nível mais aprofundado de discussão.

4.2.1 Botões

Os botões são componentes muito importantes, pois são responsáveis por grande parte da interatividade entre a *UI* e o usuário. Existem vários tipos de botão e cada um deles tem, além de um desenho gráfico, uma forma de coletar informações e realizar ações. Falaremos apenas de seis deles, a saber: *Button, Checkbox, RadioButton, ToggleButton* e *Switch*.

4.2.1.1 Botões simples (*Button*)

É a classe mais básica de botões; descreve apenas uma caixa visual que, ao ser clicada, chama algum método previamente indicado. Os botões são usados em várias situações onde precisamos de uma confirmação (ou pedido) do usuário para executar uma tarefa.

Os botões são bastante maleáveis e permitem muitas diferentes configurações, desde cores e o texto da caixa até sua forma (quadrada, esférica etc.); são componentes bastante voláteis e interessantes de se usar.

```
<Button                                                    1
    android:id="@+id/button"                               2
    android:layout_width="500px"                           3
    android:layout_height="100px"                          4
    android:onClick="funcao_executar"                      5
    android:text="Texto"                                   6
    android:textSize="22sp"                                7
    android:textStyle="bold" />                            8
```

Observação: no código acima, temos vários elementos que exercem diferentes funções: na linha **2**, temos o identificador do elemento; linhas **3** e **4** são largura e altura, respectivamente; a linha **5** indica qual método na classe relacionada será chamado ao ser clicado; linha **6** indica o texto que estará escrito no botão; e, por último, linhas **7** e **8** descrevem tamanho e estilo da fonte. Existem várias outras informações que podem ser adicionadas, mas são de uso mais específico e não as abordaremos, por enquanto.

O trecho de código acima representa um botão simples, com poucas informações declaradas e o produto será algo como mostrado na Figura 16.

Figura 16: Botão simples.

Para dar mais características, é possível escrever estilos em *XML* para conferir ao botão informações adicionais, como cor, borda e até forma. Depois, basta associar este arquivo ao botão e pronto. Um exemplo pode ser o código a seguir, que dá uma forma retangular com pontas arredondadas, com uma cor azulada e com uma borda grossa e preta.

> Aviso: deve-se avaliar a qual atributo devemos passar a informação. Se estamos falando de estilo, usamos ***theme***, se é uma forma, ou seja, um *drawable*, passamos para ***background***.

```
<shape                                                          1
    xmlns:android="http://schemas.android.com/apk/res/android"  2
    android:shape="rectangle">                                  3
    <corners                                                    4
        android:radius="25dp"/>                                 5
    <solid                                                      6
        android:color="#00BFFF"/>                               7
    <stroke                                                     8
        android:width="3dp"                                     9
        android:color="#000000"/>                               10
</shape>                                                        11
```

> Observação: na linha **3**, definimos a forma do botão que, no exemplo, é retangular; na linha **4**, informações a cerca das quinas, neste caso a linha **5** define o raio dos arredondamentos; na linha **6**, damos características sobre uma cor sólida e, na linha **7**, definimos a cor; e em **8**, descrevemos características sobre a borda, em especial linhas **9** e **10**, que descrevem a espessura e a cor.

O resultado é um botão simples, azulado e retangular; com as quinas arredondadas, inclui também uma borda preta (Figura 17).

Figura 17: Botão simples estilizado.

Também é possível colocar um gradiente no lugar de uma cor sólida, definindo a cor inicial, a cor final e, opcionalmente, uma cor para a transição. Para isso, deve-se trocar a classe **solid** pelo trecho **gradient** a seguir.

```
<gradient                                                    1
    android:startColor="#00008B"                             2
    android:endColor="#6495ED"                               3
    android:centerColor="#4169E1"                            4
    android:angle="45"/>                                     5
```

Observação: na classe **gradient** (que ocupará o lugar de **solid** no código anterior), descreveremos um gradiente que ficará no fundo. Na linha **2**, definimos a cor mais à esquerda; em **3**, aquela mais à direita; e, na linha **4**, declaramos a que ficará no meio, na transição; por último, a linha **5** define o ângulo do gradiente.

Figura 18: Botão simples com gradiente.

4.2.1.2 Caixas marcáveis (*Checkbox* e *RadioButton*)

As *checkbox* são um tipo específico de botão que consiste em uma caixinha, que pode ser marcada ou não, com um texto ao lado. Esse tipo de caixinha é muito usada em formulários para que o usuário possa indicar se concorda ou não com algo.

Como a maioria dos componentes em *XML*, os códigos são muito parecidos entre si e têm vários atributos em comum, diferenciando-se quase que exclusivamente pela *tag*. Para além das informações comuns, as *checkbox* contam, também, com a opção de deixar o botão, por padrão, marcado ou não.

Observação: nas linhas **5** e **6**, define-se as margens superior e inferior. Já na linha **7**, escolhemos se a caixa aparecerá, por padrão, marcada ou não.

```
<CheckBox                                                    1
    android:id="@+id/checkBox"                               2
    android:layout_width="wrap_content"                      3
    android:layout_height="wrap_content"                     4
    android:layout_marginLeft="90dp"                         5
    android:layout_marginTop="50dp"                          6
    android:checked="true"                                   7
    android:text="Texto" />                                  8
```

Figura 19: Checkbox marcada.

Observação: nas linhas **3** e **4**, caso prefira, pode-se usar o termo **wrap_content** para que o tamanho seja só o suficiente para caber todo o conteúdo, ou seja, para que ele englobe o tamanho de seus filhos. Alternativamente, pode-se usar **match_parent**, para que o tamanho se estique até encontrar o elemento diretamente superior na hierarquia (ou pai).

Outro exemplo de caixas marcáveis são os *radioButton*, com a diferença visual principal de ter a caixa circular, e não quadrada.

```
<RadioButton                                                 1
    android:id="@+id/radioButton"                            2
    android:layout_width="wrap_content"                      3
    android:layout_height="wrap_content"                     4
    android:text="Texto"                                     5
    android:textSize="20sp"                                  6
    android:textStyle="bold" />                              7
```

Figura 20: RadioButton desmarcado.

Isoladamente, o *radioButton* tem um uso muito limitado. Diferentemente das *checkbox*, um elemento do tipo *radioButton* **não pode** ser desmarcado depois de marcado,

o que é de pouca utilidade quando usado sozinho. Para agregar mais função, devemos usar o *radioGroup*; funcionando como um *container*, o *radioGroup* agrega vários *radioButton* para trabalharem como um conjunto de forma exclusiva (ou seja, permitindo marcar apenas uma das opções).

```
<RadioGroup                                                    1
        android:id="@+id/group_test"                           2
        android:layout_width="wrap_content"                    3
        android:layout_height="match_parent"                   4
        android:checkedButton="@id/radioButton1" >             5
                                                               6
        <RadioButton                                           7
            android:id="@+id/radioButton1"                     8
            android:layout_width="wrap_content"                9
            android:layout_height="wrap_content"               10
            android:layout_weight="1"                          11
            android:text="RadioButton1" />                     12
                                                               13
        <RadioButton                                           14
            android:id="@+id/radioButton2"                     15
            android:layout_width="wrap_content"                16
            android:layout_height="wrap_content"               17
            android:layout_weight="1"                          18
            android:text="RadioButton2" />                     19
    </RadioGroup>                                              20
```

Sua estrutura é simples, basta incluir, dentro de um elemento *RadioGroup*, um conjunto de outros elementos do tipo *RadioButton*, cada um com seus atributos e um **id**. Desta forma, eles trabalharão conjuntamente e, se o usuário escolher um, todos os outros serão desmarcados.

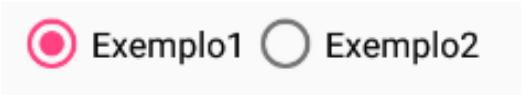

Figura 21: *RadioGroup* com dois *RadioButtons*.

Sugestão: às vezes, é necessário garantir que o usuário escolherá ao menos uma das opções. Para garantir que algum dos elementos estará marcado sem precisar verificar, basta adicionar **android:checkedButton** ao *radioGroup* e identificar o *id* do botão escolhido como padrão, como feito na linha **5**.

4.2.1.3 Alternadores de estado (*ToggleButton* e *Switch*)

Com uma aparência similar a um botão, o *toggle button* é usado para alternar entre dois estados, como **ativado** ou **desativado**. Para indicar se ativo ou não, conta com uma linha colorida logo abaixo do botão e é alternado clicando.

```
<RadioButton                                          1
    android:id="@+id/radioButton"                     2
    android:layout_width="wrap_content"               3
    android:layout_height="wrap_content"              4
    android:text="Texto"                              5
    android:textSize="20sp"                           6
    android:textStyle="bold" />                       7
```

Figura 22: ToggleButton desativado e ativado.

Outra opção para alternar entre estados são os *Switch* que, como o nome indica, tem a aparência semelhante a um interruptor deslizável.

```
<Switch                                               1
    android:id="@+id/switchEx"                        2
    android:layout_width="110dp"                      3
    android:layout_height="wrap_content"              4
    android:text="Texto"                              5
    android:textSize="20sp"                           6
    android:textStyle="bold"/>                        7
```

Figura 23: Switch ativado e desativado.

4.2.2 Caixas de texto editáveis

A forma mais simples de conseguir informações mais específicas do usuário é por meio de caixas de texto editáveis, onde o usuário pode escrever. Essas caixas podem ser editadas de várias formas como para permitir apenas um tipo específico ou um limite máximo de caracteres, por exemplo.

```
<EditText                                                    1
        android:id="@+id/editText3"                          2
        android:layout_width="match_parent"                  3
        android:layout_height="wrap_content"                 4
        android:inputType="number"                           5
        android:hint="teste"                                 6
        android:maxLength="20" />                            7
```

Observação: na linha 5, define-se o tipo de entrada (no exemplo, apenas numérico); na linha 6, podemos dar uma "dica", que é o texto exposto quando a caixa está vazia; por fim, a linhas 7 determina um limite de caracteres.

Figura 24: EditText em foco e digitado.

Ao definir um tipo em *inputType*, a caixa editável reconhecerá apenas o tipo de texto escolhido. Existem várias opções possíveis, como *text*, *number*, *textEmailAdress*, *date* ou *textAutoComplete*. Outra opção de entrada são *textPassword* e *numberPassword*, que escondem o texto digitado (Figura 25).

Figura 25: EditText em foco e digitado.

Dica: para criar uma caixa de senha que permite exibir e esconder o texto, podemos usar as classes ***TextInputEditText*** e ***TextInputLayout***, marcando como verdadeiro o atributo ***passwordToggleEnabled***.

4.2.3 Texto (label)

Um dos elementos mais simples, a *TextView* consiste em um texto a ser exibido, que pode ser personalizado em vários aspectos. É usado quando necessário exibir um conteúdo em tela de forma que o usuário possa apenas ler, mas não alterar (ao menos, não diretamente). Pode ser programado para sumir e reaparecer, alterar o texto, alterar a cor e permite várias ouras alterações dinâmicas.

```
<TextView                                                    1
        android:id="@+id/textView"                           2
        android:layout_width="wrap_content"                  3
        android:layout_height="wrap_content"                 4
        android:text="Texto exemplo"                         5
        android:textSize="20sp"/>                            6
```

Figura 26: *TextView*.

4.2.4 Spinner

O elemento *spinner* funciona como um menu que, ao ser clicado, abre uma lista de itens para que o usuário escolha um deles. Neste caso, é um elemento que vai demandar um conhecimento um pouco mais amplo, sendo necessário utilizarmos os conceitos de *adapter*, *string-array* e escrever um pouco no arquivo Java.

O primeiro passo é criar um *string-array*, para enumerar os elementos a serem exibidos. Para isso, vamos ao arquivo *strings.xml*, em **app** > **res** > **values**, e adicionaremos um elemento **string-array** entre as *tags* **resources** como no exemplo a seguir.

```
<resources>                                                  1
    <string-array name="spinner">                            2
        <item>Um</item>                                      3
        <item>Dois</item>                                    4
        <item>Tres</item>                                    5
    </string-array>                                          6
</resources>                                                 7
```

O próximo passo é incluir no *layout* da aplicação um elemento *Spinner*, como o trecho a seguir.

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

E, por último, devemos ir ao arquivo java respectivo a nossa *activity* e associar o *array* ao *spinner*. Para isso, precisamos primeiro associar o elemento a uma variável e usar um *adapter* que terá a função de servir de ponte entre a estrutura de dado (*string-array*) e a *view* (no caso, o *spinner*).

Dica: Não se preocupe, ao menos por enquanto, em entender todos os passos e elementos; eles serão melhor abordados no desenrolar do tutorial.

Dentro do arquivo Java, no método *onCreate*, colocaremos um trecho como o a seguir, escolhendo alguns estilos de visualização pré-definidos da biblioteca *android*.

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);
ArrayAdapter<CharSequence> adapter =
    ArrayAdapter.createFromResource(this, R.array.spinner,
        android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(adapter);
```

Feitos os passos, o resultado é como aquele mostrado na Figura 27. O elemento exibe a primeira opção por padrão e, ao ser clicado, abre o menu para que o usuário possa escolher o item desejado.

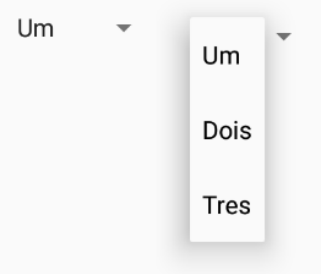

Figura 27: *Spinner* fechado e aberto.

4.2.5 Imagens

Por vezes, desejamos colocar uma imagem em nossa tela, apenas a imagem. Para fazê-lo, são necessários pouquíssimos passos, basicamente dois. O primeiro e mais importante é se certificar de ter um caminho até a imagem. Como caminho, pode-se usar uma URL, URN, URI entre outras formas de identificar. Nesse momento, faremos da forma mais simples: colocaremos a imagem na pasta **drawable** (em **app** > **res**) e, com ela lá, usaremos apenas **@drawable/"nome_do_arquivo"** para identificá-la. Para o exemplo a seguir, a imagem foi salva com o nome **pettele_uff.png**.

A seguir, adicionaremos ao *layout* um trecho como esse a seguir. Como pode-se perceber, em **srcCompat** nós identificamos qual recurso desejamos exibir. O resultado será como da Figura 28.

```
<ImageView                                                    1
        android:id="@+id/imageView"                           2
        android:layout_width="match_parent"                   3
        android:layout_height="wrap_content"                  4
        app:srcCompat="@drawable/pettele_uff" />              5
```

Figura 28: ImageView exibindo a figura *pettele_uff.png*.

Dica: existe uma discussão sobre a diferença de uso entre os métodos **android:src** e **app:srcCompat**, mas acaba envolvendo outros assuntos que não abordaremos nesse tutorial. De forma geral, pode-se dizer que o método **app:srcCompat** permite integrar imagens vetoriais de forma mais sólida.

4.3 Gerenciadores de *layout*

Quando criamos uma tela, nada mais fazemos além de incluir uma série de elementos (*Views*) e grupos de elementos (*ViewGroup*). Esses elementos devem ser alocado em "caixinhas", que podem estar dentro de caixas maiores. Os *layouts* são essas "caixas", são como *containers* que são responsáveis por agregar e organizar seus elementos.

Algumas informações, como altura e largura (*height* e *width*), são obrigatórios para todos os elementos, sejam eles *views* ou *layouts*. Como é de se imaginar, todos os elementos deverão respeitar ao tamanho de seu "pai"(elemento que o engloba), logo, devemos prestar atenção aos tamanhos que atribuiremos.

> Dica: é nesse ponto que o conhecimento sobre as opções **match_parent** e **wrap_content**, já anteriormente apresentadas, se tornam bastante importantes.

É necessário que toda tela tenha um *layout* raiz, aquele que engloba todos os elementos, mas nada impede que se aninhe uma série de outros *layouts*. Dessa forma, podemos criar complexas hierarquias de *layouts* e *views* para construir uma tela da forma que desejamos.

Existem vários diferentes gerenciadores de *layout* e cada um tem sua forma de dispor e organizar os elementos, logo, é necessário que conheçamos um pouco sobre os principais.

4.3.1 *Constraint layout*

Além de ser o gerenciador padrão do *Android Studios*, inclui as principais características do *linear layout* e do *relative layout*, sendo disponível a partir da versão 2.3 (API 9) de *Android*.

É baseado na ideia de *constraints*, que significariam algo como "restrições"ou "coerção", que são as indicações de alinhamento para o objeto, âncoras que indicam a quais elementos o componente está preso. Todo elemento deve ter ao menos uma *constraint* vertical e outra horizontal; o posicionamento padrão é sempre no topo à esquerda na tela.

Dessa forma, quando desejamos colocar dois botões no centro da tela, um logo abaixo do outro, devemos usar um código semelhante ao exemplo a seguir.

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/constraintLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botao 1"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toTopOf="@+id/button2"/>
    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botao 2"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/button1"
        app:layout_constraintBottom_toBottomOf="parent" />
</android.support.constraint.ConstraintLayout>
```

No exemplo, coloca-se um elemento do tipo *constraint layout* por fora, dando características de altura e largura como **match_parent**. Entre as *tags*, aninham-se dois botões simples, como os exemplos anteriores.

Os nomes adotados são **Start** para o lado esquerdo, **End** para lado direito, **Top** para topo e **bottom** para fundo. Dessa forma, cria-se algo como uma âncora, que será a *constraint* responsável por indicar o alinhamento do elemento.

As *constraints* são sempre inicadas por *app:layout_constraint* e, em seguida, define-se como ***lado*_toladoOf=referência**. Dessa forma, as principais funções são:

- Start_toStartOf (esquerda alinhada à esquerda de)
- Start_toEndOf (esquerda alinhada à direita de)
- End_toStartOf (direita alinhada à esquerda de)
- End_toEndOf (direita alinhada à direita de)
- Top_toTopOf (topo alinhado ao topo de)
- Top_toBottomOf (topo alinhado ao fundo de)
- Bottom_toTopOf (fundo alinhado ao topo de)
- Bottom_toBottomOf (fundo alinhado ao fundo de)

  Dica: existem várias outras *constraints* disponíveis, mas as listadas acima já são suficiente pra grande parte das operações.

No caso acima, o pai (*parent*) refere-se ao *constraint layout*, que está esticado até os limites da tela; dessa forma, todas as referências a *parent* tem o efeito de ligar aos limites da tela. Além disso, ambos os botões tiveram sua esquerda alinhada à esquerda da tela e sua direita, à direita. Já no caso do primeiro botão, seu topo foi alinhado ao topo da tela e o fundo, ao topo do segundo botão. No caso do segundo, seu topo está ligado ao fundo do primeiro botão e seu fundo, ao fundo da tela. Dessa forma, o resultado em tela é como aquele ilustrado na Figura 29.

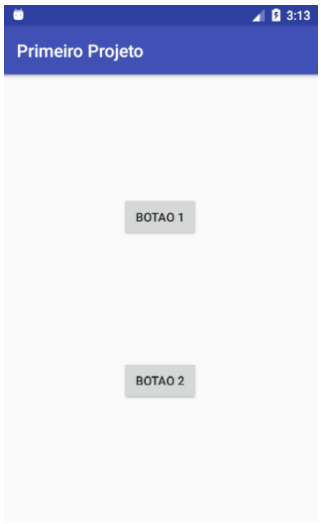

Figura 29: Resultado visual do exemplo com *Constraint layout*.

4.3.2 *Relative layout*

Como pode-se imaginar, o gerenciador *relative layout* organiza os elementos em relação a outros elementos da tela.

```
<RelativeLayout                                                    1
    xmlns:android="http://schemas.android.com/apk/res/android"     2
    android:id="@+id/relativeLayout"                               3
    android:layout_width="match_parent"                            4
    android:layout_height="match_parent">                          5
    <Button                                                        6
        android:id="@+id/button1"                                  7
        android:layout_width="wrap_content"                        8
        android:layout_height="wrap_content"                       9
        android:layout_centerInParent="true"                       10
        android:text="Botao 1" />                                  11
    <Button                                                        12
        android:id="@+id/button2"                                  13
        android:layout_width="wrap_content"                        14
        android:layout_height="wrap_content"                       15
        android:layout_below="@+id/button1"                        16
        android:layout_centerHorizontal="true"                     17
        android:layout_marginTop="40dp"                            18
        android:text="Botao 2" />                                  19
</RelativeLayout>                                                  20
```

Como no caso anterior, o *layout* raiz ocupa todo o espaço da tela; entre suas *tags*, ficam descritos os botões. O primeiro botão recebe apenas um indicador de posição, que o centraliza tanto vertical quanto horizontalmente (linha **10**). No caso do segundo, aninha-se este com o fundo do primeiro (linha **16**) e o centralizamos horizontalmente (linha **17**); para garantir uma distância entre os botões, o segundo ganha também uma margem superior (linha **18**).

> Aviso: os exemplos, tanto este quanto os sequentes, são apenas formas de alinhar dois botões simples e rapidamente. Todo gerenciador *layout* pode fazer qualquer tela, mas alguns tornam determinadas tarefas mais fáceis do que em outros.

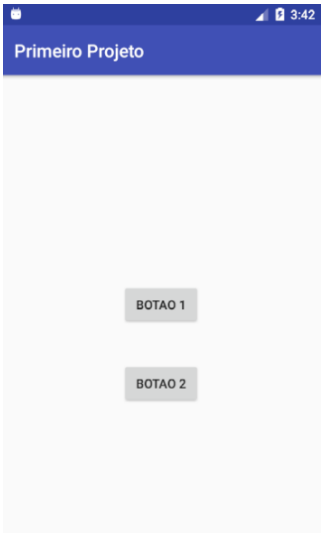

Figura 30: Resultado visual do exemplo com *Relative layout*.

Para definir uma orientação relativa a outro elemento, existem várias formas; pode-se dizer que os principais métodos são:

- ***layout_below***: alinha abaixo da referência
- ***layout_above***: alinha acima da referência
- ***layout_toEndOf***: alinha à direita da referência
- ***layout_toStartOf***: alinha à esquerda da referência

4.3.3 *Linear layout*

O gerenciador *Linear layout* funciona, como o nome sugere, de forma linear, permitindo basicamente duas orientações: vertical e horizontal. Seu comportamento padrão é colocar todos os elementos, seguindo a ordem em que aparecem no código, lado a lado (no caso horizontal) ou um abaixo do outro (no caso vertical).

> Dica: por ter um comportamento bastante simples e intuitivo, é muito utilizado como *subcontainer* para organizar conjuntos de elementos. Um exemplo disso é quando desejamos criar uma fileira de objetos; usar um *linear layout*, neste caso, pode nos poupar várias linhas de *constraints* ou atributos similares.

```
<LinearLayout                                                    1
    xmlns:android="http://schemas.android.com/apk/res/android"   2
    android:id="@+id/linearLayout"                               3
    android:layout_width="match_parent"                          4
    android:layout_height="match_parent"                         5
    android:orientation="vertical">                              6
    <Button                                                      7
        android:id="@+id/button1"                                8
        android:layout_width="wrap_content"                      9
        android:layout_height="wrap_content"                     10
        android:layout_marginTop="210dp"                         11
        android:layout_gravity="center_horizontal"               12
        android:text="Botao 1" />                                13
    <Button                                                      14
        android:id="@+id/button2"                                15
        android:layout_width="wrap_content"                      16
        android:layout_height="wrap_content"                     17
        android:layout_marginTop="40dp"                          18
        android:layout_gravity="center_horizontal"               19
        android:text="Botao 2" />                                20
</LinearLayout>                                                  21
```

Com uso da orientação vertical (linha **6**), centraliza-se horizontalmente os botões (linhas 12 e 19); por fim, adiciona-se uma margem para acertar distâncias.

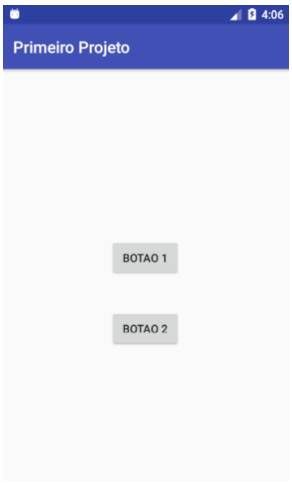

Figura 31: Resultado visual do exemplo com *Linear layout*.

4.3.4 *Grid layout*

A palavra “grid”seria traduzida como “grade”ou “rede”e, como podemos intuir, *grid layout* se trata de um gerenciador com funcionamento semelhante a uma grade. Permitindo que se defina um número de linhas e colunas, os elementos são colocados respeitando a indicação horizontal (coluna) e vertical (linha).

Apesar de ser possível definir uma contagem para o número de células com os métodos **android:rowCount** e **android:columnCount**, o *grid layout* realiza a contagem por meio de células usadas, tornando os métodos dispensáveis. Dessa forma, esse gerenciador acaba tendo um uso bastante restrito, sendo necessário adicionar elementos vazios com tamanhos definidos para simular uma linha ou coluna em branco.

> Dica: uma forma de criar esses espaços em branco é utilizando elementos do tipo **TextView** vazios, com tamanho pré-definido. No exemplo, foram utilizados elementos dessa forma, para criar linhas e colunas vazias.

```
<GridLayout                                                    1
    xmlns:android="http://schemas.android.com/apk/res/android"  2
    android:id="@+id/gridLayout"                                3
    android:layout_width="match_parent"                        4
    android:layout_height="match_parent">                      5
    <TextView                                                  6
        android:layout_height="90dp"                           7
        android:layout_width="142dp"                           8
        android:layout_row="0"                                 9
        android:layout_column="0"/>                            10
    <TextView                                                  11
        android:layout_height="90dp"                           12
        android:layout_width="142dp"                           13
        android:layout_row="1"                                 14
        android:layout_column="2"/>                            15
    <TextView                                                  16
        android:layout_height="90dp"                           17
        android:layout_width="142dp"                           18
        android:layout_row="3"                                 19
```

```
        android:layout_column="2"/>                    20
    <Button                                            21
        android:id="@+id/button1"                      22
        android:layout_width="wrap_content"            23
        android:layout_height="wrap_content"           24
        android:layout_row="2"                         25
        android:layout_column="1"                      26
        android:text="Botao 1" />                      27
    <Button                                            28
        android:id="@+id/button2"                      29
        android:layout_width="wrap_content"            30
        android:layout_height="wrap_content"           31
        android:layout_row="4"                         32
        android:layout_column="1"                      33
        android:text="Botao 2" />                      34
</GridLayout>                                          35
```

Para organizar os elementos, usa-se o método **layout_row** para definir a linha e **layout_column** para coluna, bastando apenas informar o número.

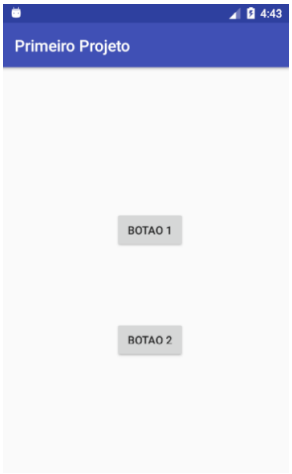

Figura 32: Resultado visual do exemplo com *Grid layout*.

Dica: o *grid layout* é melhor utilizado quando se pretende criar uma tabela de elemento. Apesar de fornecer métodos para alinhamento, não chega a ser intuitivo.

4.3.5 *Absolute layout*

Definitivamente, não é um gerenciador recomendado e, tampouco, prático. O *absolute layout* funciona como um plano cartesiano, organizando os elementos segundo suas coordenadas X e Y. Com coordenadas dadas em dip ("*density independent pixels*"), os elementos podem sofrer distorções bastante preocupantes em seu posicionamento, a depender do formato da tela. O exemplo dos botões, em uma tela específica (Figura 33), poderia ser recriado usando o seguinte exemplo.

```
<AbsoluteLayout                                                    1
    xmlns:android="http://schemas.android.com/apk/res/android"     2
    android:id="@+id/gridLayout"                                   3
    android:layout_width="match_parent"                            4
    android:layout_height="match_parent">                          5
    <Button                                                        6
        android:id="@+id/button1"                                  7
        android:layout_width="wrap_content"                        8
        android:layout_height="wrap_content"                       9
        android:layout_y="180dp"                                   10
        android:layout_x="140dp"                                   11
        android:text="Botao 1" />                                  12
    <Button                                                        13
        android:id="@+id/button2"                                  14
        android:layout_width="wrap_content"                        15
        android:layout_height="wrap_content"                       16
        android:layout_y="300dp"                                   17
        android:layout_x="140dp"                                   18
        android:text="Botao 2" />                                  19
</AbsoluteLayout>                                                  20
```

Dica: quando se usa *absolute layout* em algum projeto, a *IDE* acusa como *deprecated* ("descontinuada", em português); isso se dá por não ser indicada já há muitas versões, devido sua difícil compatibilidade com diferentes aparelhos. Apesar de não usada, fez-se necessário apresentar ao aluno a opção, para mostrar que é possível organizar elementos de forma absoluta.

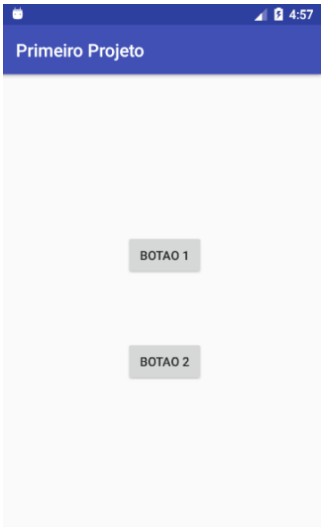

Figura 33: Resultado visual do exemplo com *Absolute layout*.

4.4 Conhecendo o editor de *layout*

Nesta subseção, apresentaremos alguns elementos do editor de *layout* oferecido pelo *Android Studio*.

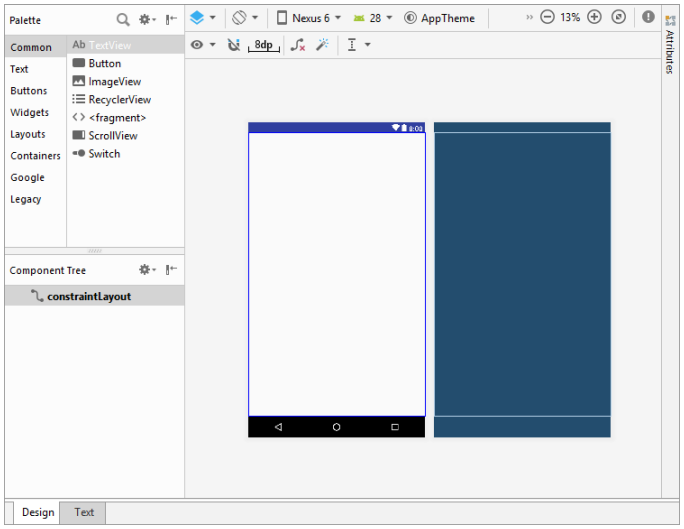

Figura 34: Editor de *layout* em **design**.

Ao criarmos um novo arquivo do tipo **layout resource** (em *XML*), encontraremos a tela do editor (Figura 34). Perceba, na parte inferior, que a aba ativa é **design**; nela, podemos tratar os elementos de forma intuitiva, arrastando-os para a tela.

Na janela **Palette**, encontramos uma lista de elementos dividido em grupos (expostos à esquerda), mas também é possível buscá-los por nome. Logo abaixo, a janela **Component Tree** (árvore de componentes, em português) descreve todos os elementos

em sua respectiva hierarquia; e arrastar os elementos de **Palette** para a posição desejada na árvore é, também, a forma mais segura de incluí-los. Arrastando dois botões, temos uma árvore como a da Figura 35 e, clicando sobre o primeiro botão, abrimos a janela **Attributes** com configurações sobre o botão selecionado (Figura 35).

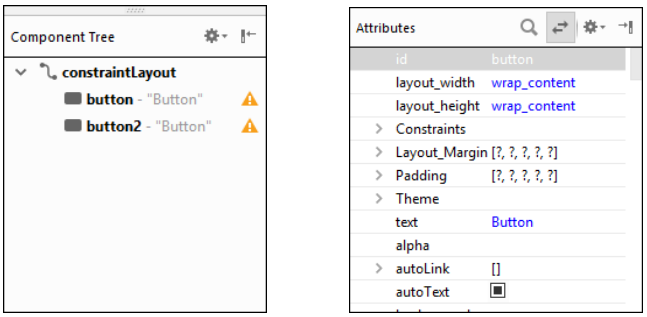

Figura 35: *Component Tree* e *Attributes*.

Com os componentes aparecendo em tela, podemos criar as *constraints* clicando nas bolinhas que aparecem em cada lado do botão e as arrastando até o elemento ao qual desejamos afixar (esquerda), sem precisarmos operar código (Figura 36).

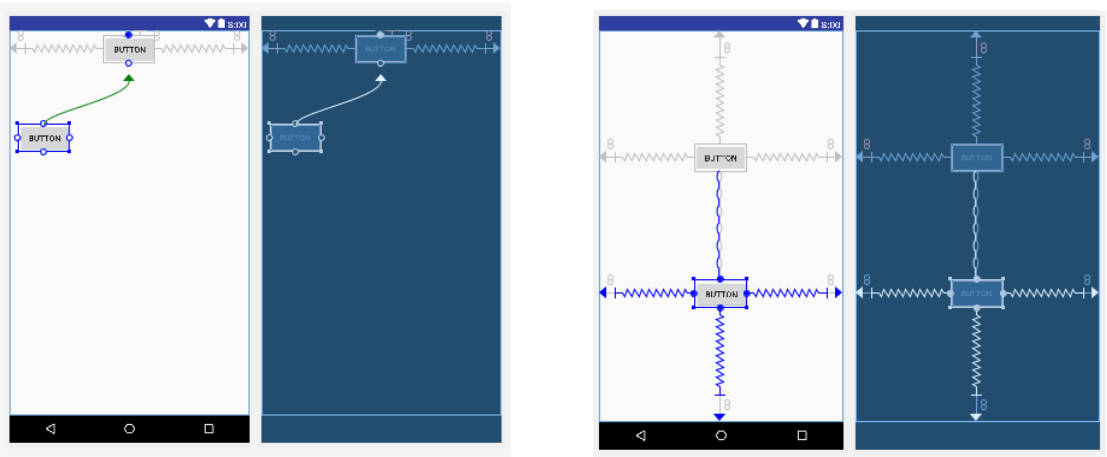

Figura 36: Criando as *constraints*.

Dica: pode ocorrer que os elementos adicionados não estejam aparecendo em tela. Uma forma simples de contornar esta situação é trocar, no arquivo *styles.xml* (**app** > **res** > **values**), a frase

```
parent="Theme.AppCompat.Light.DarkActionBar"
```

por

```
parent="Base.Theme.AppCompat.Light.DarkActionBar"
```

5 Primeiro aplicativo

Agora que já criamos o primeiro projeto e aprendemos um pouco sobre *activity* e *layout*, começaremos o primeiro projeto prático de desenvolvimento; nesta seção, abordaremos passo a passo a criação de um aplicativo. Para todo desenvolvimento de aplicação com tela, é necessário ter, ao menos, uma *activity*; ela será responsável por construir a interface de utilizador (*UI*).

Para adicionar uma *activity*, o modo mais fácil é utilizando a interface visual da própria *IDE*. Basta clicar sobre a pasta **app** com o botão direito e escolher **New** > **Activity** > **Gallery**. Entretanto, as várias opções de telas pré-programadas podem (e vão) confundir qualquer programador pouco experiente, por isso, abordaremos a criação da *activity* passo a passo, sem utilizar uma tela pré-programada.

O primeiro passo é sempre criar um arquivo *XML* para o *layout* e outro Java para a classe da tela. Depois de construir o *layout*, a programação entra em cena na classe criada; essa deve herdar de **AppCompatActivity**, sobrescrever alguns métodos obrigatórios (como o **onCreate**) e incluir uma referência ao *layout*. Com tudo pronto, precisamos apresentar nossa tela ao *Android*, declarando-a no já conhecido **AndroidManifest**.

O aplicativo que desenvolveremos é bastante simples, mas deve incluir vários dos conceitos que já abordamos aqui. A ideia é criar uma tela que gera mensagens, onde o usuário deve escolher um pronome, digitar seu nome e receberá uma mensagem personalizada.

5.1 Desenvolvendo o primeiro *layout*

Para a tela, usaremos uma **EditText** para recolher o nome, **RadioButtons** para que se escolha o pronome e um **Button** para gerar a mensagem, que será exibida com uma **TextView**.

Lembrando sempre das boas práticas, devemos nos certificar de incluir no arquivo **strings.xml** (**app** > **res** > **values**) o seguinte trecho, que contém os vários textos que usaremos em nossa tela.

```
<string name="senhor">Sr.</string>                    1
<string name="senhora">Sra.</string>                  2
<string name="senhorita">Srta.</string>               3
```

```
<string name="outro">Outro</string>
<string name="seuNome">Digite seu nome</string>
<string name="botao">Gerar mensagem</string>
```

Agora, devemos criar um arquivo *XML* que será responsável pelo *layout*. Na pasta **layout** (**app** > **res**), clicamos com o botão direito e seguimos **New** > ***layout resource file*** (Figura 37). Nesta tela, em **File name** devemos dar um nome e para **root element** (elemento raiz) escolheremos um *layout* para englobar todos os elementos.

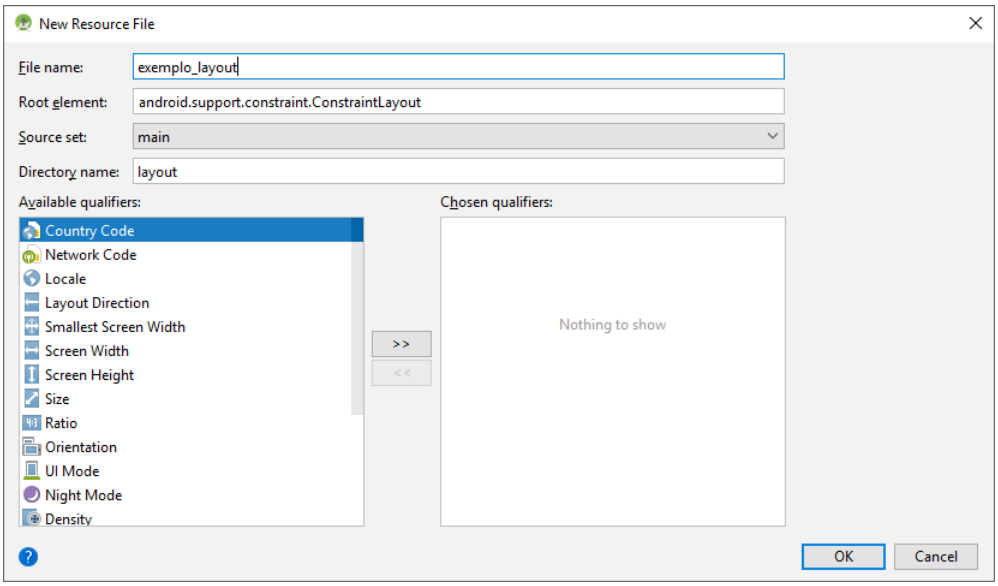

Figura 37: Tela para criar arquivo *layout*.

Com o arquivo criado, podemos começar a desenvolver os elementos; e sabendo que estamos usando um *constraint layout* como raiz, devemos nos atentar em incluir as *constraints*. Seguiremos em ordem de elementos na tela, começando pela caixa editável, seguindo para as caixas marcáveis, depois um botão (que será personalizado) e, por último, o texto.

Para a primeira parte, criaremos apenas uma caixa editável simples que se estende por quase toda a tela horizontalmente. O trecho equivalente pode ser conferido a seguir.

```
<EditText
    android:id="@+id/editText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
```

```
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@+id/linLayout"
    android:hint="@string/seuNome"
    android:layout_marginLeft="20dp"
    android:layout_marginRight="20dp"/>
```

O segundo passo é definir um **RadioGroup** com os diferentes pronomes possíveis; ofereceremos como opção **Sr.**, **Sra.** e **Srta.**, permitindo também que o usuário insira um pronome diferente, se desejar.

```
<LinearLayout
    android:id="@+id/linLayout"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toBottomOf="@+id/editText"
    app:layout_constraintBottom_toTopOf="@+id/button"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent">
    <RadioGroup
        android:id="@+id/radioGroup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:checkedButton="@+id/radioSenhor">
        <RadioButton
            android:id="@+id/radioSenhor"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/senhor"
            android:layout_marginTop="3dp"
            android:textSize="20sp"/>
        <RadioButton
            android:id="@+id/radioSenhora"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/senhora"
```

```
            android:layout_marginTop="3dp"
            android:textSize="20sp" />
        <RadioButton
            android:id="@+id/radioSenhorita"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/senhorita"
            android:layout_marginTop="3dp"
            android:textSize="20sp" />
        <RadioButton
            android:id="@+id/radioOutro"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
    </RadioGroup>
    <EditText
        android:id="@+id/editTextOutro"
        android:layout_width="80dp"
        android:layout_height="wrap_content"
        android:hint="@string/outro"/>
</LinearLayout>
```

Agora, criaremos um botão personalizado que executará a ação. Primeiro, precisamos de um *XML* de estilo. Clicando com o botão direito em **drawable** (**app** > **res**), seguimos **New** > **Drawable resource file**.

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <corners
        android:radius="25dp" />
    <solid
        android:color="#00BFFF"/>
    <stroke
        android:width="3dp"
        android:color="#000000" />
</shape>
```

Com o estilo já feito, voltamos para o arquivo de *layout* principal e criaremos um botão, que incluirá as configurações de estilo como **background**.

```
<Button                                                                    1
    android:id="@+id/button"                                               2
    android:layout_width="200dp"                                           3
    android:layout_height="wrap_content"                                   4
    app:layout_constraintTop_toBottomOf="@id/linLayout"                    5
    app:layout_constraintBottom_toTopOf="@id/textView"                     6
    app:layout_constraintStart_toStartOf="parent"                          7
    app:layout_constraintEnd_toEndOf="parent"                              8
    android:text="@string/botao"                                           9
    android:background="@drawable/botao_estilo"/>                          10
```

Por último, devemos criar uma **textView** que exibirá a mensagem gerada; por enquanto, ela terá apenas um espaço e não incluirá texto.

```
<TextView                                                                  1
    android:id="@+id/textView"                                             2
    android:layout_width="match_parent"                                    3
    android:layout_height="wrap_content"                                   4
    app:layout_constraintTop_toBottomOf="@+id/button"                      5
    app:layout_constraintBottom_toBottomOf="parent"                        6
    app:layout_constraintStart_toStartOf="parent"                          7
    app:layout_constraintEnd_toEndOf="parent"                              8
    android:textSize="20sp"/>                                              9
```

Para que recriemos a tela, basta incluir todas as partes anterior em ordem dentro de um *constraint layout*.

```
<?xml version="1.0" encoding="utf-8"?>                                     1
<android.support.constraint.ConstraintLayout                               2
    xmlns:android="http://schemas.android.com/apk/res/android"             3
    android:layout_width="match_parent"                                    4
    android:layout_height="match_parent"                                   5
    xmlns:app="http://schemas.android.com/apk/res-auto">                   6
                                                                           7
</android.support.constraint.ConstraintLayout>                            8
```

Figura 38: Resultado do exemplo desenvolvido.

5.2 Criando a classe Java

Com o *layout* criado, é a vez de termos uma classe para tratá-lo. Vamos a pasta com o nome do projeto (que criamos em outra seção) seguindo **app** > **java** > **pettele.primeiroprojeto** e, clicando com o direito, seguimos **New** > **Java Class** (Figura 39). É importante reparar que a classe deve herdar de **AppCompatActivity**.

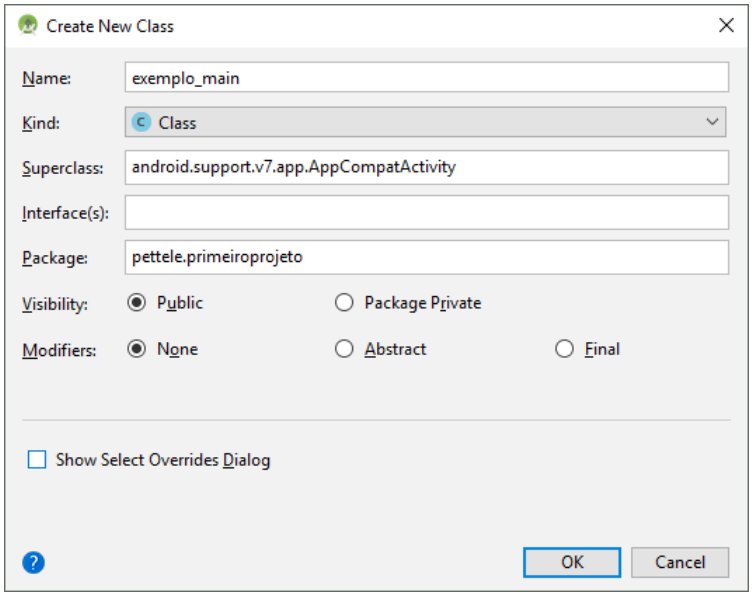

Figura 39: Tela para criar classe Java.

Clicando em **ok**, a classe é gerada do jeito que queremos; agora, precisamos sobrescrever o método **onCreate** e associá-la ao *layout* criado, por meio do método **setContentView**.

```
package pettele.primeiroprojeto;
import ...

public class exemplo_main extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.exemplo_layout);
    }
}
```

Com isso, nós já temos uma classe que representará nossa tela. Agora precisamos, primeiramente, recuperar o controle dos elementos; e faremos isso instanciando objetos com o método **findViewById**.

> Dica: Como o nome do método sugere, a ideia é encontrar as **views** usando suas **ids**, logo, é necessário que todos os elementos estejam identificados. O método retorna uma View (que é a classe de onde todos os elementos descendem), logo, é necessário fazer um *casting* para acessar os métodos específicos.

Assim sendo, devemos instanciar cada um dos elementos como atributos de nossa classe (para facilitar as operações), adicionando o seguinte trecho antes do primeiro (e, atualmente, único) método.

```
protected EditText editText;
protected EditText editTextOutro;
protected RadioGroup radioGroup;
protected Button button;
protected TextView textView;
protected String texto;
protected String pronome;
```

Desta vez dentro do método **onCreate**, devemos instanciá-los com seus elementos respectivos. Com os elementos instanciados, podemos começar a trabalhar.

> Dica: não é necessário instanciar objetos para todos os elementos visuais da tela, apenas para aqueles que pretendemos acessar dentro da classe.

```
editText = (EditText) findViewById(R.id.editText);
radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
button = (Button) findViewById(R.id.button);
textView = (TextView) findViewById(R.id.textView);
editTextOutro = (EditText) findViewById(R.id.editTextOutro);
```

O primeiro passo é adicionar um **listener** ao botão, para que possamos identificar um clique. Para isso, usaremos o método **setOnClickListener** e criaremos um objeto que sobrescreverá o principal método, **onClick**.

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        //aqui colocaremos as atividades a serem realizadas apos
           o clique
    }
});
```

Dica: existe uma forma mais recente e fácil de dar atividade ao botão, não sendo necessário adicionar um **listener**, como fizemos. Para isso, basta criar um método público, sem retorno e que receba uma **View** como parâmetro.

```
    public void metodo_botao(View view) {
    }
```

E, lá no *XML* do botão, adicionar um trecho **onClick** que faz referência ao método criado.

```
    android:onClick="metodo_botao"
```

Desse modo, cada clique chamará o método uma vez.

Agora que já temos um **listener** para identificar os cliques, chegou a hora de criar um responsável, que executará toda a atividade. Para isso, escreveremos mais um método em nossa classe principal, com o nome “metodoBotao”, que não receberá nenhum parâmetro e também não terá retorno. Esse método será chamado dentro do método **onClick**.

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
    ...
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            metodoBotao();
        }
    });
}

public void metodoBotao() {
}
```

O primeiro passo é adicionar um verificador para garantir que algum nome foi escrito na editText; para isso, basta usar o método **getText** sobre a caixa e comparar com uma string vazia. Caso esteja vazia, emitiremos um aviso (**Toast**), que aparecerá, durante pouco tempo, com um mensagem pedindo ao usuário que preencha o campo.

Dica: o método **getText** retorna um objeto do tipo **editable**, logo, é necessário chamar também o método **toString** para poder comparar o conteúdo.

Apresentação: a classe **Toast** permite criar avisos que aparecem no fundo da tela por um tempo curto. O método **makeText** é estático, logo, não requer uma instância; ele recebe como parâmetros um contexto, uma string e uma duração em int. O próprio método já tem duas durações, uma curta e outra mais longa (*LENGTH_SHORT* e *LENGTH_LONG*). Por fim, após criado o **Toast**, deve-se chamar o método **show** para exibí-lo em tela.

```
public void metodoBotao() {
    if (editText.getText().toString().compareTo("") == 0){
        Toast.makeText(this, "Preencha o campo",
            Toast.LENGTH_SHORT).show();
    } else {}
}
```

Figura 40: Resultado de um clique no botão sem preencher a caixa.

Agora que temos certeza que a caixa tem texto e que alguma das opções do pronome está marcada (e garantimos isso escolhendo uma caixa marcada por padrão), devemos descobrir qual o pronome escolhido. Para isso, usaremos o método próprio **getCheckedRadioButtonId**, que retornará um inteiro representando o **id** do botão marcado; e podemos usar a estrutura do **switch** para comparar com as **ids** possíveis. Sabendo o pronome marcado, basta associar nosso atributo **pronome** ao pronome escolhido e, no caso do escolhido ser **outro**, devemos recuperar o texto escrito na caixinha.

```
switch(radioGroup.getCheckedRadioButtonId()){                          1
    case R.id.radioSenhor: pronome = "Sr.";                            2
    case R.id.radioSenhora: pronome = "Sra.";                          3
    case R.id.radioSenhorita: pronome = "Srta";                        4
    case R.id.radioOutro:                                              5
        if(editTextOutro.getText().toString().compareTo("") == 0)      6
           {
            Toast.makeText(this, "Preencha o campo", Toast.            7
               LENGTH_SHORT).show();
        } else {                                                       8
            pronome = editTextOutro.getText().toString();              9
        }                                                              10
}                                                                      11
```

Em posse do pronome e tendo certeza de que a caixa já foi preenchida, basta concatenar as *strings* recolhidas e gerar a mensagem personalizada. Depois, basta associá-la à **textView** e pronto.

```
public void metodoBotao() {
    if (editText.getText().toString().compareTo("") == 0){
        Toast.makeText(this, "Preencha o campo", Toast.
           LENGTH_SHORT).show();
    } else{
        switch(radioGroup.getCheckedRadioButtonId())
        {
            case R.id.radioSenhor:
                pronome = "Sr.";
                break;
            case R.id.radioSenhora:
                pronome = "Sra.";
                break;
            case R.id.radioSenhorita:
                pronome = "Srta";
                break;
            case R.id.radioOutro:
                if(editTextOutro.getText().toString().compareTo("
                   ") == 0){
                    Toast.makeText(this, "Preencha o campo",
                       Toast.LENGTH_SHORT).show();
                } else {
                    pronome = editTextOutro.getText().toString();
                } break;
        }
        texto = pronome+" "+editText.getText().toString()+", seja
            bem vindo!";
        textView.setText(texto);
    }
}
```

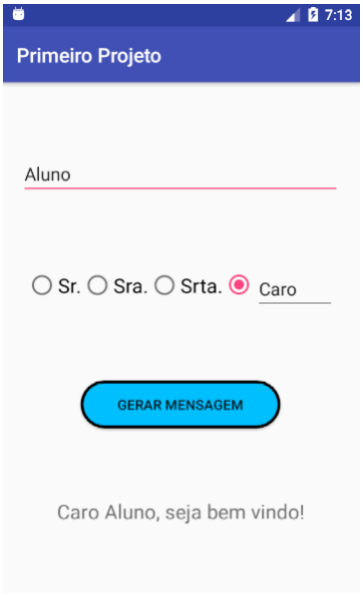

Figura 41: Resultado de uma execução do aplicativo.

5.3 Declarando a *activity* no *androidManifest*

Agora que nossa *activity* já está pronta e funcionando, precisamos declará-la para que o *Android* saiba como reagir ao abrir o aplicativo. Para isso, iremos no arquivo *AndroidManifest.xml* (**app** > **manifest**) e, entre as *tags* **application**, incluiremos um elemento **activity**, com o nome de nossa classe. Dentro das *tags* que acabamos de inserir, devemos colocar uma **intent-filter** e indicar que essa *activity* é a principal e deve ser iniciada junto a aplicação.

```
<manifest>                                                          1
 <application>                                                      2
  <activity android:name=".exemplo_main">                           3
   <intent-filter>                                                  4
    <category android:name="android.intent.category.LAUNCHER" />    5
    <action android:name="android.intent.action.MAIN" />            6
   </intent-filter>                                                 7
  </activity>                                                       8
 </application>                                                     9
</manifest>                                                         10
```

Aviso: no exemplo acima, foram omitidas informações das *tags* **manifest** e **application**, que são geradas por padrão, para simplificar o exemplo.

5.4 Testando

Agora que nossa aplicação já está pronta, falta apenas testar; e como estamos tratando um aplicativo para *smartphone*, o teste deve ser realizado em um aparelho correspondente. Para isso, é possível que usemos nosso próprio aparelho celular ou que criemos um aparelho virtual, utilizando o **Android Virtual Device**.

5.4.1 Executando a aplicação em aparelho virtual

Aviso: antes de tudo, devo alertar que o teste em aparelhos virtuais demanda bastante memória RAM e, ainda sim, costuma demorar um pouco.

Para criarmos um aparelho virtual, devemos ir em **Tools** > **AVD Manager** > **Create Virtual Device**. Na tela seguinte (Figura 42), escolhemos a categoria e o modelo do aparelho que desejamos simular; para o exemplo, foi escolhido um modelo **Nexus 5X**.

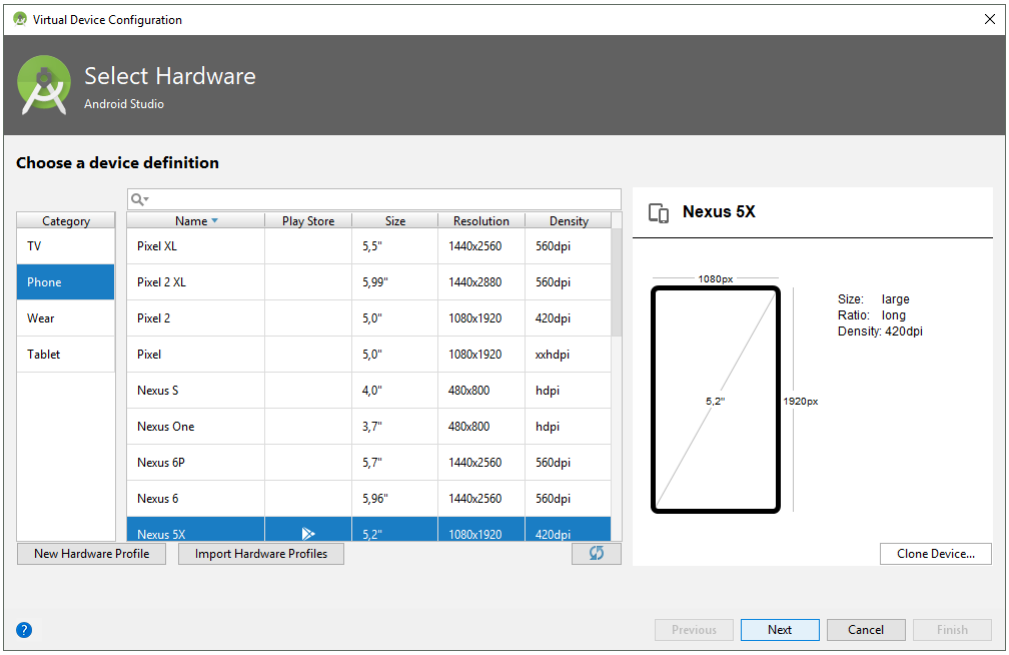

Figura 42: Criando o dispositivo virtual.

O segundo passo é instalar alguma versão *Android* (Figura 43). Se é sua primeira vez neste processo, você terá que baixar alguma versão; para isso, basta clicar em **download** ao lado da versão escolhida e pronto, em pouco tempo a versão estará disponível. Depois basta selecioná-la e clicar em **next**.

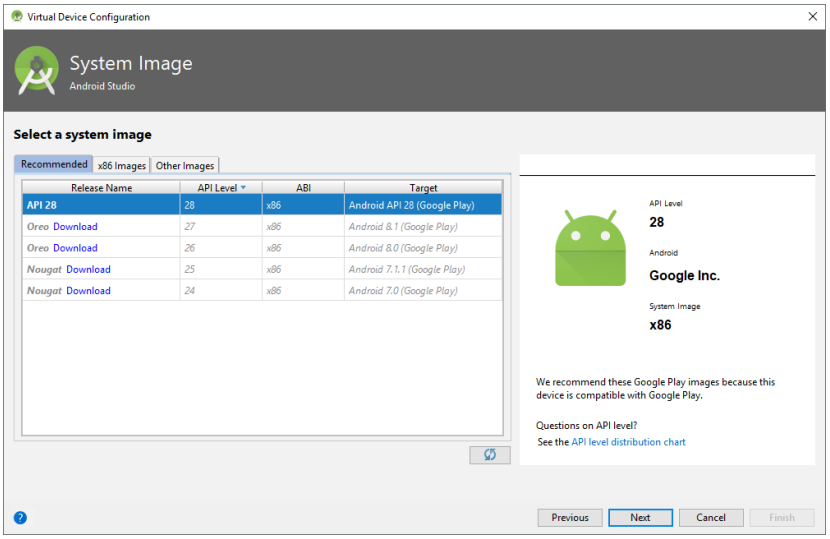

Figura 43: Escolhendo uma versão *Android*.

A última tela te pedirá um nome e uma orientação padrão; clicando em **finish**, seu dispositivo virtual está criado e pronto para ser usado. Para iniciá-lo, basta clicar no triângulo que aparece ao lado do nome (Figura 44).

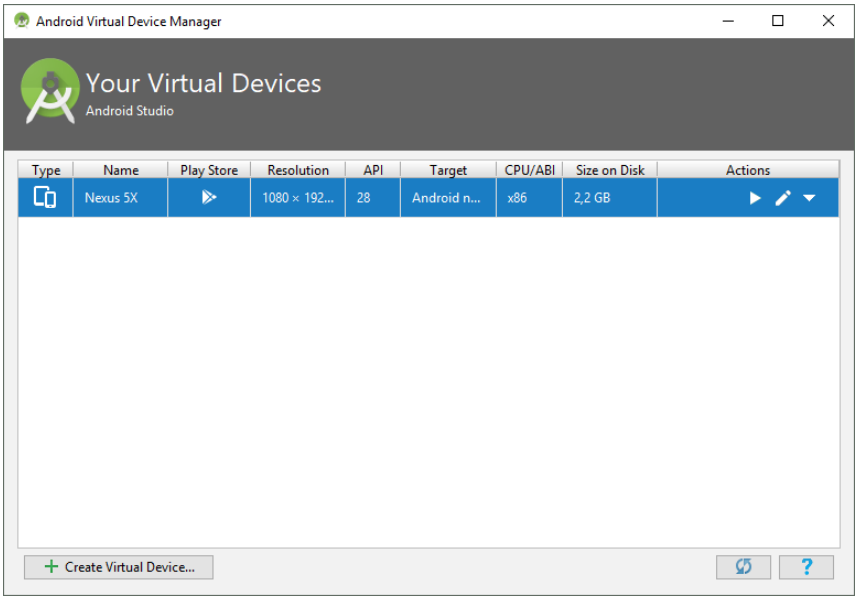

Figura 44: Iniciando o aparelho virtual.

O resultado será um telefone flutuante na tela, que funciona de forma praticamente idêntica a um *smartphone* de verdade, considerando suas limitações físicas. Para executar o aplicativo, basta clicar no triângulo verde no topo direito da *IDE* e selecionar o dispositivo que acabamos de criar. O telefone carregará durante algum tempo e exibirá a aplicação (Figura 45).

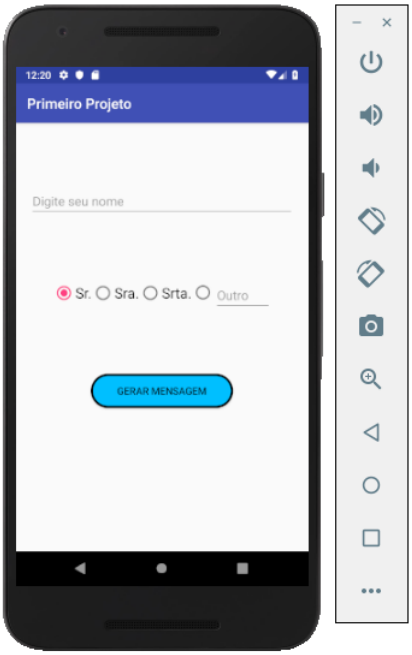

Figura 45: Aparelho virtual executando aplicação.

5.4.2 Executando a aplicação em aparelho físico

Para que executemos a aplicação em nosso próprio *smartphone* (uma solução mais rápida e que demanda menos memória), é necessário, antes, que sejam feitas algumas configurações. Primeiro, conseguir acesso de desenvolvedor no *smartphone*, o que geralmente consiste em seguir **Configuração** > **Sobre o dispositivo** e apertar várias vezes em **Número da versão** até receber uma mensagem de que o acesso está liberado.

Depois, acessamos as configurações de desenvolvedor e precisamos permitir **Instalação via USB** e ativar a **Depuração USB**, **Depuração USB (configuração de segurança)** e **Verificar apps por USB**. Agora que o telefone já permite esse processo, basta ligá-lo ao computador por conexão USB e clicar no triângulo verde no topo direito da *IDE*. Espere até seu aparelho ser reconhecido e aparecer disponível, selecione-o e complete o processo; após alguns momentos, seu aparelho abrirá a aplicação.

> Dica: alguns aparelhos têm certa resistência e não aparecem disponíveis tão facilmente. Nesses casos específicos, é necessário que se realize algumas outras operações (que podem ser encontradas com facilidade nos *websites* de busca).

6 Listas de Views

Um dos principais artifícios para construir boas aplicações são as listas de *views*; elas tornam possível construir *layouts* dinâmicos, que se formam a partir de informações externas.

Suponhamos uma aplicação que gerencia vários produtos de uma loja. Nós não sabemos quantos produtos temos e nem quantos serão no futuro; não há forma de construir um *layout* com infinitos elementos e depois achar e preencher, um a um, na classe principal. Com uma lista de *views*, poderíamos construir um *layout* padrão para cada produto e, com uma lista de produtos, preenchê-la dinamicamente.

6.1 Atualizando as dependências

Para que nós não precisemos construir todos os elementos do zero, o ambiente integrado já nos oferece uma coleção de elementos diversos. Incluindo, *a priori*, uma série de bibliotecas, o desenvolvimento fica mais fácil e organizado.

Alguns elementos não estão por padrão no projeto, mas ainda são essenciais para o desenvolvimento da aplicação. Essas classes necessárias são as **dependências** e, antes que comecemos a trabalhar na nossa lista, devemos atualizar as dependências do projeto.

Para isso, basta ir ao menu no topo da tela e seguir **File** > **Project Structure...** (ctrl + alt + shift + s). Na tela (Figura 46.a), clique em **app** na seção **Modules**. Depois, vá a aba **dependencies** (Figura 46.b).

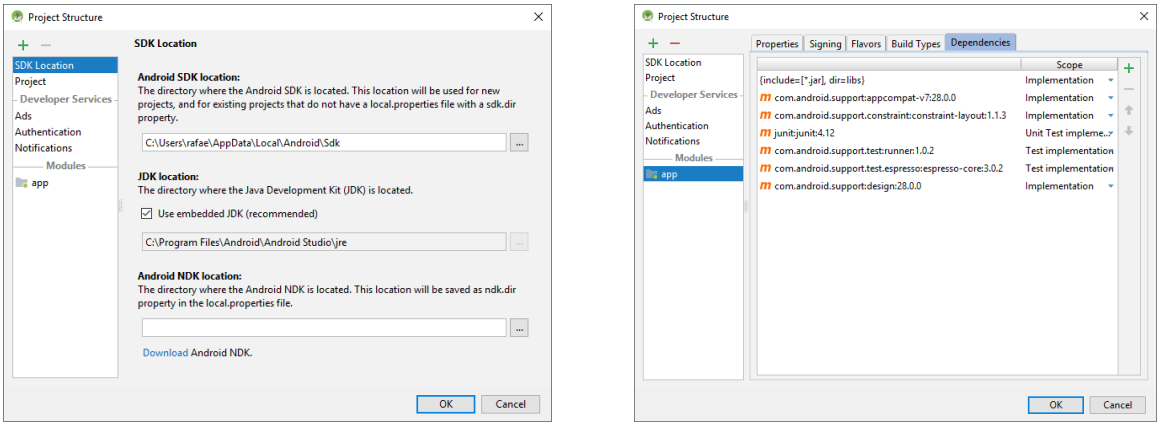

Figura 46: Estrutura do projeto e dependências.

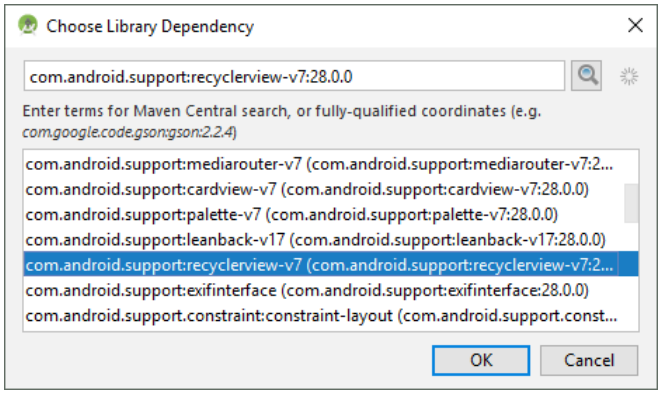

Figura 47: Incluindo a **RecyclerView**.

Clicando no símbolo "+"(topo direito) e escolhendo **Library dependency**, podemos escolher uma dependência para incluir ao projeto (Figura 47). Devemos buscar e incluir **RecyclerView** com **implementation** em *scope*.

Por último, clicamos em **OK** e aguardamos a sincronização do *gradle*. Se não começar automaticamente, irá aparecer no topo uma mensagem informando sobre a necessidade da sincronização, deve-se, então, clicar em **Sync now** e aguardar um pouco.

Agora que já temos as dependências necessárias no projeto, podemos começar a desenvolver os elementos.

6.2 Introdução à classe *View*

A **View** será, para a lista, como um *layout* é para a *activity*; ela descreverá todos os elementos que se repetiram na lista e vai organizar a forma como as informações serão dispostas. Elas serão responsáveis por conter o arquivo *XML* que faremos para guardar as informações.

O primeiro passo é, de fato, construir o *layout*; ele deve conter elementos visuais para exibir todas as informações que desejamos. Para exemplo, exibiremos apenas uma imagem (*ImageView*), um nome (*TextView*) e um curso (*TextView*).

> Dica: a ***RecyclerView*** permite que usemos diferentes *layouts* a depender da posição em questão. Como o exemplo tem todos os elementos iguais, só faremos um *layout*; mas caso o programador deseje, pode definir um arquivo para cada item da lista.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView
        android:id="@+id/image"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginRight="50dp"
        android:layout_marginLeft="30dp"/>
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical" >
        <TextView
            android:id="@+id/textview1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="30sp"
            android:textColor="#bb3333"/>
        <TextView
            android:id="@+id/textview2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"/>
    </LinearLayout>
</LinearLayout>
```

Agora que já temos um *layout resource file* com os elementos que precisaremos, é necessário uma classe que possa organizar as informações a serem exibidas. Esta nova classe, chamada **Aluno**, deve incluir como atributo um nome (*String*), um curso (*String*) e uma imagem (*int*). Além do construtor, que deve receber todos os atributos como parâmetros, implementaremos todos os *getters*.

```
public class Aluno {
    private String nome;
    private String curso;
    private int imagem;

    public Aluno(String nome, String curso, int imagem) {
        this.nome = nome;
        this.curso = curso;
        this.imagem = imagem;
    }
    public String getNome() { return nome; }
    public String getCurso() { return curso; }
    public int getImagem() { return imagem; }
}
```

O próximo passo é criar um *Adapter*, que será responsável por fazer a conexão entre nossa classe **Aluno** e a **View**, associando as informações aos elementos visuais do *layout*.

Nossa classe, chamada **AlunoAdapter**, deve herdar de ***RecyclerView.Adapter*** que, por sua vez, pede uma instância de ***ViewHolder***. Como o projeto pede uma classe adaptada ao *layout* com o qual trabalharemos, devemos criar uma outra classe, de nome ***AlunoViewHolder***, que herdará de *ViewHolder*. A fim de reduzir código e mantê-lo mais coeso, a classe *AlunoViewHolder* será interna a *AlunoAdapter*.

```
public class AlunoAdapter extends
        RecyclerView.Adapter<AlunoAdapter.AlunoViewHolder> {

    class AlunoViewHolder extends RecyclerView.ViewHolder {
    }
}
```

Agora que já temos nossa classe externa (*outer class*) e a classe interna (*inner class*), devemos sobrescrever todos os métodos obrigatórios destas. Também adicionaremos à classe externa, como atributos, um contexto (*Context*) e uma lista (*List<Aluno>*); também implementaremos um construtor que recebe e instância todos seus atributos.

```
public class AlunoAdapter extends                                    1
        RecyclerView.Adapter<AlunoAdapter.AlunoViewHolder> {         2
    private Context contexto;                                        3
    private List<Aluno> alunoList;                                   4
                                                                     5
    public AlunoAdapter                                              6
            (Context contexto, List<Aluno> alunoList) {              7
        this.contexto = contexto;                                    8
        this.alunoList = alunoList;                                  9
    }                                                                10
    @Override                                                        11
    public AlunoViewHolder onCreateViewHolder                        12
            (ViewGroup viewGroup, int tipoDeView) {                  13
    }                                                                14
    @Override                                                        15
    public void onBindViewHolder                                     16
            (AlunoViewHolder alunoViewHolder, int posicao) {         17
    }                                                                18
    @Override                                                        19
    public int getItemCount() {                                      20
    }                                                                21
                                                                     22
    class AlunoViewHolder extends RecyclerView.ViewHolder {          23
        public AlunoViewHolder(View itemView) {                      24
        }                                                            25
    }                                                                26
}                                                                    27
```

Para a classe externa, devemos sobrescrever **onBindViewHolder**, que será responsável por fazer a associação entre a informação e os elementos do *layout*, e o **getItemCount**, que deve retornar o numero de elementos; também devemos sobrescrever o método **onCreateViewHolder**, que fará a associação entre **AlunoViewHolder** e o *layout* que construímos. Para a classe interna, devemos apenas criar um novo construtor que chamará o método **super**.

O primeiro passo é a classe interna; ela deve ter como atributos os elementos visuais com os quais pretendemos trabalhar. Sendo assim, ela deve ter uma imagem (*ImageView*), um nome (*TextView*) e um curso (*TextView*). Em seu construtor, devemos associar seus atributos aos elementos do *layout*, que é recebido como parâmetro (*itemView)*. Para isso, usamos o método ***findViewById*** sobre o *itemView*.

```
class AlunoViewHolder extends RecyclerView.ViewHolder {
    protected ImageView image;
    protected TextView nome;
    protected TextView curso;
    public AlunoViewHolder(View itemView) {
        super(itemView);
        image = itemView.findViewById(R.id.image);
        nome = itemView.findViewById(R.id.textview1);
        curso = itemView.findViewById(R.id.textview2);
    }
}
```

O próximo passo é implementar o método (da classe externa) **onCreateViewHolder**, passando o *layout* ao *ViewHolder* como uma *View*, através de um **LayoutInflater**. Primeiro, instanciamos um *LayoutInflater* passando o contexto da classe como parâmetro. Depois, chamamos o método **inflate**, passando a referência do elemento e de seu pai; nesse caso, passaremos o *layout* da *View* e não indicaremos pai. Por último, retornaremos um elemento **AlunoViewHolder** que recebe essa *view* "inflada"como parâmetro.

> Dica: quando nós não indicamos o pai do elemento a ser inflado, o *Android* irá sobrescrever todas as informações que dependem da informação do pai. Por exemplo, se o *layout* usava "match_parent"como tamanho, a informação será convertida para algo que não precise do pai, como "wrap_content".

```
public AlunoViewHolder onCreateViewHolder
        (ViewGroup viewGroup, int i) {
    LayoutInflater inflater = LayoutInflater.from(contexto);
    View view = inflater.inflate(R.layout.view_layout, null);
    return new AlunoViewHolder(view);
}
```

Em seguida, iremos implementar o método **onBindViewHolder**, que recebe tanto um **AlunoViewHolder** quanto um inteiro como parâmetro; o inteiro indicará qual dos elementos da lista estamos construindo. A partir desse valor recebido, será recolhido um elemento do tipo **Aluno** da posição correspondente na lista e será feita a associação das informações desse objeto com os elementos visuais da **AlunoViewHolder** recebidos, carregando a *view* com informação.

```
public void onBindViewHolder                                          1
        (AlunoViewHolder alunoViewHolder, int posicao) {              2
    Aluno aluno = alunoList.get(posicao);                             3
    alunoViewHolder.image.setImageResource(aluno.getImagem());        4
    alunoViewHolder.nome.setText(aluno.getNome());                    5
    alunoViewHolder.curso.setText(aluno.getCurso());                  6
                                                                      7
}                                                                     8
```

Por último, devemos implementar o método **getItemCount**, que retornará, simplesmente, o tamanho de nossa lista de informações.

```
public int getItemCount() {                                           1
    return alunoList.size();                                          2
}                                                                     3
```

Com todos esses métodos, nós já temos como criar todos as *views* da nossa lista. O próximo passo será tratar a *activity* que incluirá a **RecyclerView** e adicionar as informações.

6.3 Introdução à classe *RecyclerView*

Com uma **ListView**, é possível indicar um *layout* padrão para cada item e preenchê-lo com um *adapter*. Entretanto, este elemento gera tantas *views* quanto forem necessárias e, a depender da quantidade de informação, pode tornar a aplicação muito pesada.

Uma alternativa é a **RecyclerView**. Como o nome sugere, ela recicla as *views* criadas, fazendo com que sejam geradas apenas um numero finito (e pequeno) de elementos visuais que serão preenchidos e reciclados dinamicamente. Dessa forma, a tela é preenchida

e cria-se apenas alguns elementos "externos"que serão substituídos, durante o rolamento da tela, por aqueles que já não estão mais visíveis.

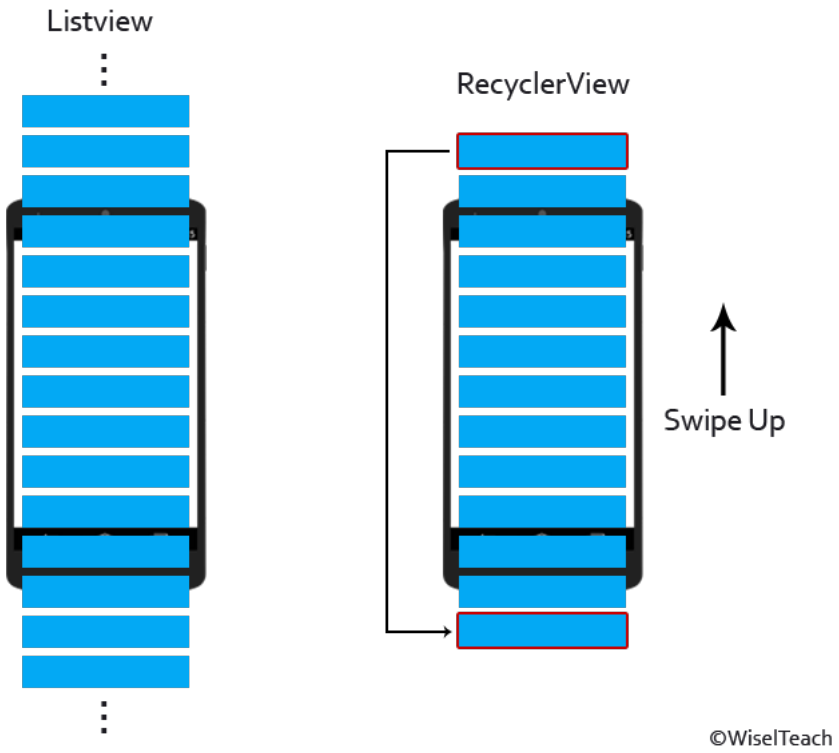

Figura 48: Imagem retirada de (7).

Por ter uma versão mais eficiente e que satisfaz todas as suas funções, a *ListView* está desatualizada (*deprecated*). Por isso, daremos atenção especial à *RecyclerView* e mostraremos exemplos.

O primeiro passo é criar uma *activity* para a tela. Para isso, devemos clicar com o botão direito em **app** e seguir **New** > **Activity** > **Empty Activity**; a ela, daremos o nome de **Lista_aluno**. Desta forma, será gerada uma classe java e um arqivo *XML* automaticamente.

No arquivo *XML*, devemos acrescentar apenas um elemento **RecyclerView** que se estenderá em todos os sentidos. É importante que esse elemento tenha um **id**, para que o encontremos depois.

```
<android.support.v7.widget.RecyclerView                    1
    android:id="@+id/recyclerview"                          2
    android:layout_width="match_parent"                     3
    android:layout_height="match_parent"/>                  4
```

Já na classe principal, o primeiro passo é adicionar, como atributo, uma **RecyclerView**, um **AlunoAdapter** e uma lista de alunos (**List<Aluno>**).

```
public class Lista_aluno extends AppCompatActivity {
    private RecyclerView recyclerView;
    private AlunoAdapter adapter;
    private List<Aluno> alunoList;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista_aluno);
    }
}
```

Dentro do método **onCreate**, devemos buscar e instanciar a nossa *recyclerview*. Além disso, como o tamanho dos filhos não irá alterar em nada o tamanho da *recyclerview*, indicaremos seu tamanho como fixo. As listas de *views* também demandam um gerenciador de *layout*, que será responsável por distribuir os elementos gerados na lista.

```
recyclerView = (RecyclerView) findViewById(R.id.recyclerview);
recyclerView.setHasFixedSize(true);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

Em seguida, a lista será criada e preenchida com casos de teste. O importante aqui é ter a lista ocupada, independente da forma; para o exemplo, as informações serão inseridas de forma manual.

```
alunoList = new ArrayList<>();
alunoList.add(new Aluno("Joao Silva","Artes",
                        R.drawable.pettele_uff));
alunoList.add(new Aluno("Marta Cristina","Ciencia da computacao",
                        R.drawable.pettele_uff));
alunoList.add(new Aluno("Rodolfo Silva","Artes ",
                        R.drawable.pettele_uff));
alunoList.add(new Aluno("Josana Riveiro","Egenharia de producao",
                        R.drawable.pettele_uff));
alunoList.add(new Aluno("Rivalda Dantas","Ciencias Sociais",
                        R.drawable.pettele_uff));
```

Por último, basta instanciar um novo **AlunoAdapter** passando, como parâmetros, o contexto e a lista de elementos. Em seguida, passaremos esse *adapter* para a *RecyclerView*.

```
adapter = new AlunoAdapter(this, alunoList);
recyclerView.setAdapter(adapter);
```

Figura 49: Resultado final da lista.

7 Caixas de diálogo (*AlertDialog*)

Quando queremos que o usuário tenha a atenção direcionada a algo, podemos usar ferramentas de diálogo. Existem várias formas de passar uma mensagem ou requisitar respostas; uma forma simples é utilizando a classe ***AlertDialog***, a qual daremos ênfase nesta seção.

Um elemento *AlertDialog* pode incluir botões, textos e outros elementos visuais que já conhecemos; pode ser temporária ou pode aguardar uma resposta do usuário. Sendo bastante versátil, oferece várias formas de construir a interface. Com o *Builder*, podemos utilizar tanto os métodos próprios para construir a interface (que incluem até três botões, título, mensagem e outros elementos), quanto inserir um *layout* personalizado. Nesta seção, veremos exemplos tanto com os métodos do *Builder* quanto inflando um *layout* próprio.

7.1 Utilizando os métodos do *Builder*

A proposta é construir uma caixa de diálogo que apresente um título, uma mensagem e três botões: negativo, neutro e positivo. O usuário deve escolher algum destes botões, fechando a caixa de diálogo e apresentando um *Toast* com a mensagem correspondente.

O primeiro passo é declarar um elemento *AlertDialog* e um *AlertDialog.Builder*, que será instanciado. O Builder deve receber um título (**setTitle**), uma mensagem (**setMessage**) e três botões de opinião (**setNegativeButton**, **setNeutralButton** e **setPositiveButton**); cada um desses botões deve receber uma ação (**onClickListener**) para gerar um *Toast*. Por fim, o método *create* retorna uma instância de *AlertDialog*, para qual chamamos o método **show**.

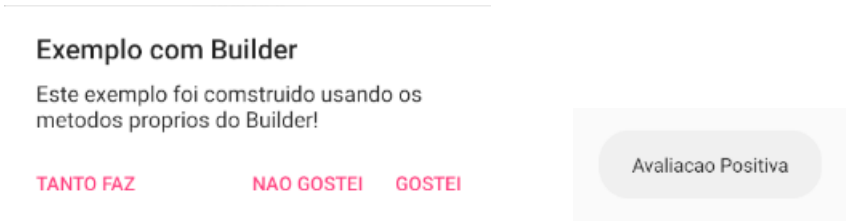

Figura 50: Imagem de *AlertDialog* e mensagem gerada com clique em Gostei.

```
AlertDialog alertDialog;
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Exemplo com Builder");
builder.setMessage("Este exemplo foi construido usando os metodos
    proprios do Builder!");
builder.setNegativeButton("Nao gostei",
    new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        Toast.makeText(this, "Avaliacao Negativa",
            Toast.LENGTH_SHORT).show();
    }
});
builder.setNeutralButton("Tanto faz",
    new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        Toast.makeText(this, "Avaliacao Neutra",
            Toast.LENGTH_SHORT).show();
    }
});
builder.setPositiveButton("Gostei",
    new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        Toast.makeText(this, "Avaliacao Positiva",
            Toast.LENGTH_SHORT).show();
    }
});
alertDialog = builder.create();
alertDialog.show();
```

7.2 Utilizando um *layout* personalizado

Outra forma de construir uma caixa de diálogo é criando um *layout* próprio e depois carregando-o em um *Builder*. Nesse caso, o primeiro passo é construir um *layout*.

```
<LinearLayout                                                    1
    xmlns:android="http://schemas.android.com/apk/res/android"   2
    android:layout_width="match_parent"                          3
    android:layout_height="match_parent"                         4
    android:orientation="vertical">                              5
    <TextView                                                    6
        android:layout_width="wrap_content"                      7
        android:layout_height="wrap_content"                     8
        android:layout_gravity="center_horizontal"               9
        android:text="Exemplo com layout"                        10
        android:textSize="30dp"/>                                11
    <EditText                                                    12
        android:id="@+id/alertDialog_edit"                       13
        android:layout_width="wrap_content"                      14
        android:layout_height="match_parent"                     15
        android:layout_gravity="center_horizontal"               16
        android:layout_marginTop="20dp"                          17
        android:hint="Digite uma mensagem"/>                     18
    <Button                                                      19
        android:id="@+id/alertDialog_botao"                      20
        android:layout_width="wrap_content"                      21
        android:layout_height="wrap_content"                     22
        android:layout_gravity="center_horizontal"               23
        android:layout_marginTop="20dp"                          24
        android:text="Criar Toast"/>                             25
</LinearLayout>                                                  26
```

O segundo passo é, também, declarar um elemento *AlertDialog* e um *Builder* (*AlertDialog.Builder*). Em seguida, precisamos instanciar um *LayoutInflater*, que será responsável por gerar uma *View* com nosso *layout*.

Para que trabalhemos com os elementos que incluímos , é necessário que os recuperemos; como estes são parte da *view* construída, o método **findViewById** deve ser chamado sobre ela. Em seguida, passamos a *view* para o *Builder* e instanciamos um *alertDialog*. Antes de mostrar, daremos atividade aos botões (**setOnClickListener**).

> Dica: devemos, sempre, ter em mente o contexto em que estamos. Por isto, para mencionar a classe principal dentro da interna, devemos usar o nome (no caso, *exemplo_dialog*). Para que a caixa feche, chamamos o método **dismiss**.

```
final AlertDialog alertDialog;                                          1
LayoutInflater inflater = getLayoutInflater();                          2
View view = inflater.inflate(R.layout.layout_alertdialog, null);        3
final EditText edit = (EditText)                                        4
    view.findViewById(R.id.alertDialog_edit);                           5
Button botao = (Button)                                                 6
    view.findViewById(R.id.alertDialog_botao);                          7
AlertDialog.Builder builder = new AlertDialog.Builder(this);            8
builder.setView(view);                                                  9
alertDialog = builder.create();                                         10
botao.setOnClickListener(new View.OnClickListener(){                    11
    @Override                                                           12
    public void onClick(View view) {                                    13
        Toast.makeText(exemplo_dialog.this, edit.getText().             14
           toString(), Toast.LENGTH_SHORT).show();
        alertDialog.dismiss();                                          15
    }                                                                   16
});                                                                     17
alertDialog.show();                                                     18
```

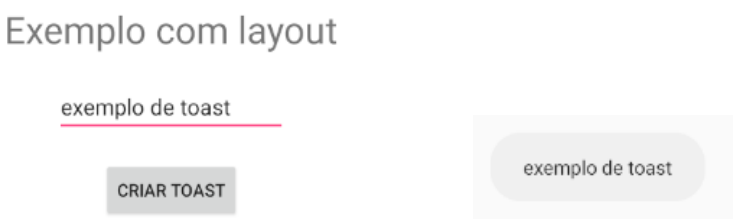

Figura 51: Imagem de *AlertDialog* e mensagem passada como exemplo.

8 Introdução à classe *Intent*

Durante o desenvolvimento de uma aplicação, é razoável que desejemos nos comunicar ou fazer requisições, tanto para nossa própria aplicação quanto para outras. Uma classe essencial para se trabalhar é a ***Intent***; além de viabilizar a conexão entre diferentes componentes, permite iniciar serviços e outras funções.

Sendo muito versáteis, podemos usar *Intents* em muitas situações: quando desejamos passar informações para outros componentes, quando evocamos alguma outra aplicação para realizar uma tarefa ou, simplesmente, quando desejamos ir de uma tela a outra. Nessa seção, falaremos um pouco sobre trafegar entre *activities*, iniciar serviços e chamar outras aplicações.

8.1 Trafegando entre *activities*

Com aplicações cada vez mais robustas e complexas, dificilmente encontramos aplicativos com apenas uma tela. Mas não basta termos várias *activities*, é necessário que exista uma troca entre elas (afinal, o usuário só verá uma por vez). Para realizar esse tráfego, podemos usar objetos da classe *Intent*.

> Aviso: como vimos no ciclo de vida, a *activity* continua existindo até que seja chamado o método *onDestroy*. Não se destrói a tela anterior chamando outra nova; logo, ao fechá-la, reaparecerá a anterior. Para resolver isso, basta chamar o método **finish** após a chamada da *Intent*.

Para exemplificar o funcionamento, faremos uma aplicação com apenas um botão; ao ser clicado, o botão chama uma nova tela, que terá apenas um texto. Para isso, criaremos duas *activities*, uma com apenas um botão e outra com apenas um texto, ambos centralizados.

```
<Button                                                    1
    android:layout_width="wrap_content"                    2
    android:layout_height="wrap_content"                   3
    android:onClick="IniciarNovaTela"                      4
    android:text="Proxima tela"                            5
    android:textSize="20sp" />                             6
```

```
<TextView                                                    1
    android:id="@+id/texto"                                  2
    android:layout_width="wrap_content"                      3
    android:layout_height="wrap_content"                     4
    android:text="Ola, sou a nova tela!"                     5
    android:textSize="35sp" />                               6
```

Agora, no método **IniciarNovaTela**, devemos criar uma *Intent* passando o contexto atual e o nome da classe da *activity* que desejamos iniciar. Por último, chamamos o método **startActivity** dentro do contexto da *activity*, passando nossa *Intent* como parâmetro.

```
public void IniciarNovaTela(View view) {                    1
    Intent it = new Intent(this, trocando_activities_2.class);  2
    startActivity(it);                                       3
}                                                            4
```

O resultado é simples, mas exemplifica bem como transitar de uma tela para outra. Ao iniciar, exibe um botão centralizado; quando clicado, inicia uma nova tela com um texto.

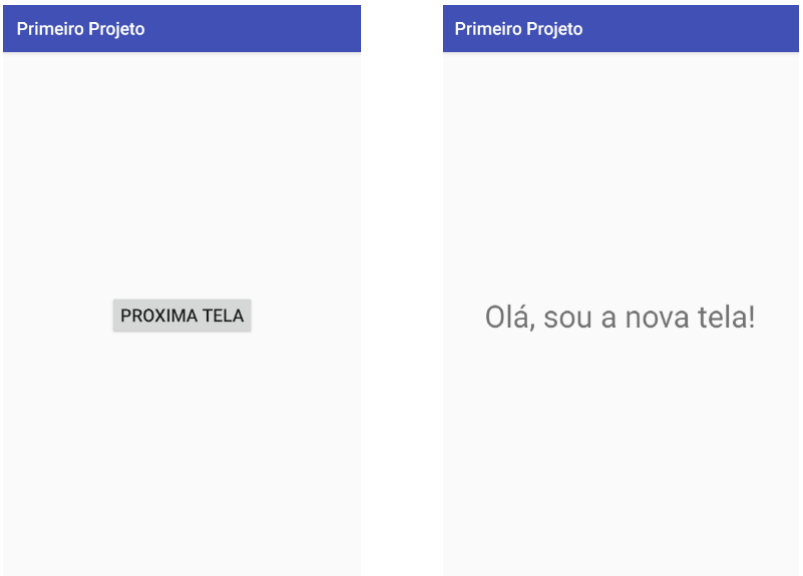

Figura 52: Exemplo antes e depois do clique no botão.

8.2 Transferindo informação entre *activities*

Além de trocar de tela, grande parte das vezes precisamos manter alguma informação, ainda que brevemente. Pense, por exemplo, em uma aplicação com uma lista de produtos; ao clicarmos em um deles, só é preciso guardar o código e enviar à próxima *activity*, que o receberá e carregará a tela com informações.

Como se trata de um armazenamento por tempo muito curto e não organizado, não é necessário usar nenhum mecanismo sofisticado de persistência de dados. Para tal, basta incluir uma informação na *Intent* e chamar o método **startActivity** normalmente.

Para exemplificar, usaremos as mesmas telas anteriores, incluindo apenas uma caixa editável (*EditText*). O usuário deve digitar alguma informação na caixa e, ao ser chamada a nova *activity*, o texto digitado é passado e deve ser carregado na tela.

```
<EditText                                                    1
    android:id="@+id/editText"                               2
    android:layout_width="match_parent"                      3
    android:layout_height="wrap_content"                     4
    android:layout_marginLeft="40dp"                         5
    android:layout_marginRight="40dp" />                     6
<Button                                                      7
    android:layout_width="wrap_content"                      8
    android:layout_height="wrap_content"                     9
    android:onClick="IniciarNovaTela"                        10
    android:text="Proxima tela"                              11
    android:textSize="20sp" />                               12
```

O primeiro passo é recolher a informação da caixa editável; depois, com a *Intent* já instanciada, esse dado deve ser adicionado. Para isso, usaremos o método **putExtra**, que recebe uma chave e o valor.

```
public void IniciarNovaTela(View view) {                    1
    EditText edit = (EditText) findViewById(R.id.editText);  2
    Intent it = new Intent(this, trocando_activities_2.class);  3
    it.putExtra("TEXTO", edit.getText().toString());         4
    startActivity(it);                                       5
}                                                            6
```

Já na segunda *activity*, no próprio método **onCreate**, devemos conseguir a informação da *Intent*. Depois, chamamos o método **getStringExtra**, que recebe uma chave e retorna o valor associado a ela. Por fim, passamos essa informação para o elemento de texto da tela, com o método **setText**.

```
TextView texto = (TextView) findViewById(R.id.texto_activity);
String info = getIntent().getStringExtra("TEXTO");
texto.setText(info);
```

Figura 53: Exemplo com o texto a ser passado e resultado após o clique.

Outro caso é quando uma nova *activity* é evocada e ainda se aguarda uma resposta, uma informação que deve retornar para a tela original. Nestes casos, deve-se usar o método ***startActivityForResult***, que recebe, além da *Intent*, um *RequestCode*. A classe chamada, antes de ser fechada, deve criar uma *intent*, adicionar informação (*putExtra*) e chamar o método ***setResult***, passando um código de resultado (*ResultCode*) e a *intent*. A classe original deve sobrescrever o método **onActivityResult**, que receberá como parâmetros o *ResultCode*, o *RequestCode* e os dados adicionados.

> Dica: o *requestCode* serve para ajudar a identificar de onde veio a resposta. Uma aplicação que chame, a depender do caso, diferentes *Intents*, deve passar um código para cada e poderá identificar a origem da informação. Já o *resultCode* serve para retornar um resultado, informando se a requisição foi atendida ou se teve problemas.

8.3 Chamando outros componentes para executar tarefas

Uma grande característica da programação *Android* é a modularidade do desenvolvimento; quando queremos executar uma atividade, nós não precisamos desenvolver todos os componentes para realizar a ação. Se queremos uma foto, não é preciso desenvolver uma aplicação que lide com a câmera do aparelho; podemos apenas fazer uma requisição para que outro aplicativo, capacitado para realizar esta tarefa, a execute.

Às *Intents* que carregam, declaradamente, a classe que estão chamando, nós damos o nome de **explícitas**. Já as *Intents* **implícitas**, como o nome sugere, não estão associadas a um componente específico, mas sim a uma ação (ou informação) que se deseja realizar.

Ao fazer a requisição, o *Android* deve buscar todas as aplicações que são capazes de realizá-la. Caso encontre apenas um aplicativo, este será executado diretamente; do contrário, abre uma caixa de diálogo para que o usuário escolha por qual caminho deseja seguir.

Podemos buscar por diversas ações, desde compartilhar um texto até tirar fotos e reproduzir vídeos. Saber utilizar estes recursos, além de evitar que recriemos tecnologias que já estão disponíveis, permite dar opções para o usuário.

Para exemplo, utilizaremos a primeira tela do exemplo anterior; entretanto, no lugar de iniciar uma nova *activity* com o texto digitado, será requisitado uma tarefa de compartilhamento, para que o usuário possa escolher enviar o texto digitado por algum dos mensageiros disponíveis.

O primeiro passo é instanciar uma nova *intent*, sem parâmetro algum. Depois, devemos dar a ela uma ação (**setAction**), um conteúdo (**putExtra**) e um tipo (**setType**). Por último, basta iniciá-la como nos demais exemplos; o próprio sistema operacional gerenciará a janela de diálogo e o trâmite da informação.

```
public void IniciarNovaTela(View view) {                                    1
    EditText edit = (EditText) findViewById(R.id.editText);                  2
    Intent it = new Intent();                                                3
    it.setAction(Intent.ACTION_SEND);                                        4
    it.putExtra(Intent.EXTRA_TEXT, edit.getText().toString());               5
    it.setType("text/plain");                                                6
    startActivity(it);                                                       7
}                                                                            8
```

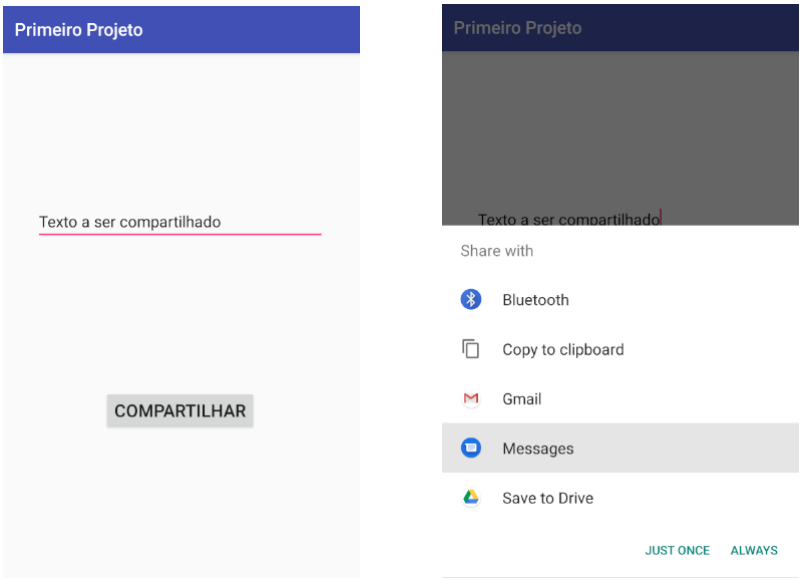

Figura 54: Exemplo antes do clique no botão e depois, com a caixa de diálogo aberta.

8.4 Filtros de *Intent* (*Intent Filters*)

Nós já vimos que podemos usar *intents* para chamar componentes que estejam habilitados a tratar uma determinada informação; a pergunta que fica é: como o sistema consegue identificá-lo? E a resposta está nos filtros!

No arquivo *AndroidManifest.xml* (**app** > **manifests**), dentro da *tag* **application**, existe uma lista das *activities* do projeto. Para cada uma, é possível adicionar um *intent filter*, onde podemos declarar quais tipos de *intents* aquele componente (neste caso, *activity*) está preparado para receber.

Para definir corretamente, precisamos tomar cuidado com as **ações**, **categorias** e **dados**.

- **Action**: define a ação que o componente pode realizar, como enviar (ACTION_SEND) ou exibir (ACTION_VIEW).
- **Category**: permite adicionar uma informação específica sobre o componente que processará a *intent*, relacionada ao local onde foi iniciada.
- **Data**: descreve o tipo de dado que o componente pode receber. Para indicar, pode-se usar *MIME*, prefixo de URI etc.

  Dica: toda *activity* que não tiver um *intent filter* declarado só pode ser chamado em uma *intent* explícita.

Um exemplo de *intent filter* é aquele que usamos para indicar, no *AndroidManifest.xml*, qual das *activities* irá gerar o ícone na tela inicial do usuário e será a primeira a ser exibida.

```
<intent filter>                                                    1
    <category android:name="android.intent.category.LAUNCHER" />  2
    <action android:name="android.intent.action.MAIN" />           3
</intent-filter>                                                   4
```

Nesse caso, a ação *MAIN* indica que esta *activity* será a porta de entrada da aplicação. Ela será a primeira a ser criada quando a aplicação for iniciada. Por sua vez, a categoria *LAUNCHER* indica que o ícone deve estar visível na lista de aplicativos do usuário. Para garantir que esteja visível e que seja reproduzido da forma como esperamos, devemos sempre lembrar de adicionar essas duas informações em nosso projeto.

9 Salvando preferências

Em algumas situações, é relevante que tenhamos uma informação persistente, ou seja, que armazenemos algum dado para além da execução da aplicação. Quando fazemos *login* em algum aplicativo, nós podemos salvar essas informações e, mesmo depois de fechar a aplicação e retornar dias depois, os dados continuam lá, da mesma forma.

> Aviso: quando falamos sobre *login*, estamos dizendo sobre o usuário. Existe uma forte recomendação para que não sejam guardados informações sobre senhas de usuário e dados importantes como preferências.

Devemos deixar claro que, caso seja do interesse do programador guardar grandes quantidades de informações que se relacionam, a melhor opção é desenvolver um projeto mais complexo, como aquele que será apresentado no Capítulo 10. Entretanto, quando o dado é simples, o *Android* oferece os recursos de ***SharedPreferences***.

> Observação: apesar de permitirem guardar uma quantidade considerável de dados e conjuntos em diferentes arquivos, as *SharedPreferences* não são recomendadas para projetos complexos pois não tem métodos de busca e processamento otimizados, como no caso do SQLite.

> Dica: as *SharedPreferences* funcionam muito bem para guardar detalhes de configurações que o usuário fez, como sons (música de fundo, som de toque etc.), cor de elementos (alto contraste, cor do fundo etc.) ou preferência de exibição (ordem dos elementos ou tamanho dos itens, por exemplo).

9.1 Apresentando o exemplo

Uma apresentação simples é dizer que a *SharedPreferences* é uma interface que permite acessar e modificar vários tipos de dados, em um sistema **chave-valor**. Para apresentar esse recurso, a ideia é uma aplicação com apenas um campo (*EditText*) e um botão (*Button*), onde o usuário pode escrever um dado, salvar e fechar a aplicação. A informação da caixa será salva como preferência e, quando o usuário voltar à aplicação, será exibida como "dica".

```
<EditText                                                    1
    android:id="@+id/texto"                                  2
    android:layout_width="match_parent"                      3
    android:layout_height="wrap_content"                     4
    android:layout_marginLeft="20dp"                         5
    android:layout_marginRight="20dp"                        6
    app:layout_constraintTop_toTopOf="parent"                7
    app:layout_constraintStart_toStartOf="parent"            8
    app:layout_constraintEnd_toEndOf="parent"                9
    app:layout_constraintBottom_toTopOf="@+id/botoes"/>      10
<LinearLayout                                                11
    android:id="@+id/botoes"                                 12
    android:layout_width="wrap_content"                      13
    android:layout_height="wrap_content"                     14
    app:layout_constraintTop_toBottomOf="@+id/texto"         15
    app:layout_constraintStart_toStartOf="parent"            16
    app:layout_constraintEnd_toEndOf="parent"                17
    app:layout_constraintBottom_toBottomOf="parent"          18
    android:orientation="horizontal"                         19
    android:gravity="center_horizontal">                     20
    <Button                                                  21
        android:id="@+id/botaoSalvar"                        22
        android:layout_width="wrap_content"                  23
        android:layout_height="wrap_content"                 24
        android:text="Salvar preferencia"/>                  25
    <Button                                                  26
        android:id="@+id/botaoApagar"                        27
        android:layout_width="wrap_content"                  28
        android:layout_height="wrap_content"                 29
        android:text="Apagar preferencia"/>                  30
</LinearLayout>                                              31
```

9.2 Criando a classe principal

Quando pretendemos salvar com certa regularidade uma preferência em especial, é uma boa prática que utilizemos uma classe dedicada a essa tarefa. No exemplo, criaremos uma classe chamada **preferencias** que será responsável por gerir todas as operações de escrita e leitura.

Como atributo, guardaremos um nome de arquivo (*String*), uma chave (*String*), um contexto (*Context*) e também um elemento *SharedPreferences* e outro *SharedPreferences.Editor*. Também implementaremos um método **construtor**, que deve receber um contexto para que possamos instanciar nossos atributos.

> Dica: existem vários modos de se abrir um arquivo; o modo privado é um deles, que faz com que as preferências só possam ser acessadas pela mesma aplicação.

```
public class Preferencias {                                          1
    private SharedPreferences sharedPreferences;                     2
    private SharedPreferences.Editor editor;                         3
    private static final String NOME = "exemplo.preferencias";       4
    private static final String CHAVE = "chave_para_salvar";         5
                                                                     6
    public Preferencias(Context contexto) {                          7
        this.contexto = contexto;                                    8
        this.sharedPreferences = contexto.getSharedPreferences       9
            (NOME, contexto.MODE_PRIVATE);                           10
        this.editor = this.sharedPreferences.edit();                 11
    }                                                                12
}                                                                    13
```

Para fazer a inscrição de uma informação, devemos chamar o editor e passar a informação no formato **chave-valor**; depois de passadas todas as informações, deve-se chamar o método **commit** ou **apply** para salvá-las.

Dica: existem diferenças entre os métodos **apply** e **commit**, embora ambos salvem as informações. O método **commit** é uma solução um pouco mais antiga e age de forma síncrona, usando a *thread* principal; também tem um retorno *boolean*, que indica se ocorreu algum problema. Já o método **apply** está presente em versões posteriores de API e realiza a operação de forma assíncrona, deixando a *thread* principal livre; também não tem retorno.

```
public void salvar(String texto){
    editor.putString(CHAVE, texto);
    editor.apply();
}
```

Para recordar as informações, podemos usar o método **getString** de *SharedPreferences*, passando a chave e um valor padrão (que será retornado caso a chave não seja encontrado).

```
public String recordar(){
    return sharedPreferences.getString
        (CHAVE, "Sem preferencia salva");
}
```

Caso desejemos, podemos apagar as preferencias salvas; para isso, basta chamar o método **remove** do *editor* e passar a chave que se deseja remover. Importante lembrar de chamar o método *apply*, para salvar o estado.

```
public void deletar(){
    editor.remove(CHAVE);
    editor.apply();
}
```

Em nossa classe principal, basta chamar os métodos quando necessário. No caso do exemplo, foi necessário instanciar um objeto **Preferencias**, passando o contexto, e instanciar objetos relativos aos elementos visuais.

```
final EditText texto = (EditText) findViewById(R.id.texto);
final Preferencias preferencias = new Preferencias(this);
Button botaoSalvar = (Button) findViewById(R.id.botaoSalvar);
Button botaoApagar = (Button) findViewById(R.id.botaoApagar);
```

Depois, nós recuperamos a informação salva como preferência e a colocamos como dica da caixa editável.

```
texto.setHint(preferencias.recordar());
```

Por último, bastou adicionar um *listener* ao clique dos botões e atribuir os métodos correspondentes.

```
botaoSalvar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        preferencias.salvar(texto.getText().toString());
    }
});
botaoApagar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        preferencias.deletar();
    }
});
```

O resultado da aplicação é simples. O usuário, não tendo informação salva, verá uma mensagem padrão (Figura 55.a); caso tenha, verá a informação como dica (Figura 55.b).

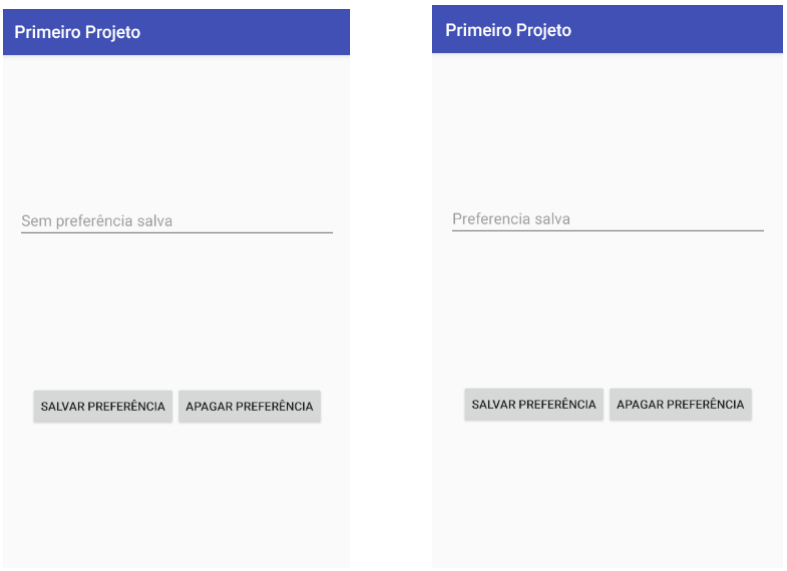

Figura 55: Resultado da aplicação antes e após salvar informação.

10 Banco de Dados

Cada aplicação é como uma "ilha"e, no geral, só pode processar as informações que ela mesmo cria. Entretanto, às vezes, é necessário construir comunicações mais complexas em bancos de dados. Em algumas situações, é necessário que armazenemos uma quantidade de dado de forma persistente e organizada, em volume e complexidade maior que o sistema de preferência, visto na seção 9, nos permite. Para isso, o *Android* oferece os recursos do **SQLite**.

Nessa seção, aprenderemos um pouco sobre essa ferramenta e veremos um exemplo aplicando todas as operações básicas do zero, desde a criação da base até a construção de uma aplicação exemplo e a execução das operações.

10.1 Introdução ao SQLite

Chamamos **SQLite** um potente sistema gerenciador de bancos de dados relacionais. Desenvolvido por uma extensa comunidade internacional, que inclui a Google como contribuidora, o sistema conta com uma estrutura robusta e confiável, permitindo criar complexos bancos por meio de comandos em SQL.

Estando integrado (e contido) ao aparelho e estrutura do sistema operacional, o sistema gerenciador não demanda o uso de servidor para funcionar. Com isso, é bastante prático e rápido, mantendo o conceito já conhecido de *Cursor*.

> Aviso: devo alertar ao aluno que, nessa seção, fica pressuposto algum conhecimento sobre linguagem SQL.

> Observação: um **cursor**, em banco de dados, se refere a uma estrutura de controle que permite percorrer sobre os registros (linhas) do banco. Assemelhando-se com a ideia de ponteiros, os **cursores** são como um iterador, permitindo manipular um conjunto de resultados, processando as linhas individual e sequencialmente.

Nesta subseção, abordaremos um pouco sobre como criar um banco e trabalhar com ele, realizando as quatro operações básicas com dados: criação, consulta, atualização e remoção, popularmente chamadas **CRUD** (*Create, Read, Update and Delete*).

Durante essa seção, usaremos uma aplicação como exemplo.

10.2 Apresentando o exemplo

Para exemplo, usaremos uma aplicação simples; ela terá quatro campos editáveis (nome, sobrenome, idade e matrícula), quatro botões (Adicionar, Buscar, Atualizar e Remover) e um texto (que exibirá a resposta da busca).

```
<?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=".exemplo_bd_interno">
    <EditText
        android:id="@+id/nome"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:layout_marginTop="40dp"
        android:hint="Nome"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>
    <EditText
        android:id="@+id/sobrenome"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:layout_marginTop="40dp"
        android:hint="Sobrenome"
        app:layout_constraintTop_toBottomOf="@id/nome"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>
```

```
    <EditText                                                                    31
        android:id="@+id/idade"                                                  32
        android:layout_width="match_parent"                                      33
        android:layout_height="wrap_content"                                     34
        android:layout_marginLeft="20dp"                                         35
        android:layout_marginRight="20dp"                                        36
        android:layout_marginTop="40dp"                                          37
        android:hint="Idade"                                                     38
        android:inputType="number"                                               39
        app:layout_constraintTop_toBottomOf="@id/sobrenome"                      40
        app:layout_constraintStart_toStartOf="parent"                            41
        app:layout_constraintEnd_toEndOf="parent"/>                              42
    <EditText                                                                    43
        android:id="@+id/matricula"                                              44
        android:layout_width="match_parent"                                      45
        android:layout_height="wrap_content"                                     46
        android:layout_marginLeft="20dp"                                         47
        android:layout_marginRight="20dp"                                        48
        android:layout_marginTop="40dp"                                          49
        android:hint="Matricula"                                                 50
        app:layout_constraintTop_toBottomOf="@id/idade"                          51
        app:layout_constraintStart_toStartOf="parent"                            52
        app:layout_constraintEnd_toEndOf="parent"/>                              53
    <LinearLayout                                                                54
        android:layout_width="match_parent"                                      55
        android:layout_height="wrap_content"                                     56
        android:orientation="horizontal"                                         57
        android:layout_marginTop="40dp"                                          58
        app:layout_constraintTop_toBottomOf="@id/matricula"                      59
        app:layout_constraintStart_toStartOf="parent"                            60
        app:layout_constraintEnd_toEndOf="parent"                                61
        app:layout_constraintBottom_toBottomOf="parent"                          62
        android:gravity="center_horizontal">                                     63
        <Button                                                                  64
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Adicionar"
            android:onClick="Adicionar"/>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Buscar"
            android:onClick="Buscar"/>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Atualizar"
            android:onClick="Atualizar"/>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Remover"
            android:onClick="Remover"/>
    </LinearLayout>
    <TextView
        android:id="@+id/texto"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:layout_marginTop="40dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/linBotao"
        android:textSize="25dp" />

</android.support.constraint.ConstraintLayout>
```

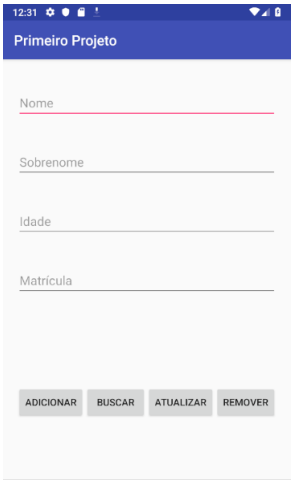

Figura 56: Tela da aplicação exemplo.

Na classe principal, os atributos serão **EditText**, instanciados com as caixas editáveis no método **onCreate**.

```
private EditText nome;
private EditText sobrenome;
private EditText idade;
private EditText matricula;
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_exemplo_bd_interno);
    nome = (EditText) findViewById(R.id.nome);
    sobrenome = (EditText) findViewById(R.id.sobrenome);
    idade = (EditText) findViewById(R.id.idade);
    matricula = (EditText) findViewById(R.id.matricula);
}
```

Agora que já conhecemos a aplicação com a qual vamos exemplificar as operações, vamos apresentar os elementos necessários.

10.3 Criando as classes principais

Uma boa prática da programação com banco de dados é delegar, a uma única classe, o controle de conexão ao banco de dados. Essa classe será responsável por criar o banco (na primeira execução) e estabelecer conexão com o mesmo.

A classe que criaremos deve herdar de **SQLiteOpenHelper** e, obrigatoriamente, sobrescrever os métodos **onCreate** e **onUpgrade**. Quando a aplicação rodar pela primeira vez, o método *onCreate* será chamado e terá a função de criar as tabelas necessárias, tendo em vista que o construtor criará o banco de dados. Caso a aplicação seja atualizada, o bando de dados pode precisar de mudanças e essa atualização será tarefa do método *onUpdate*.

```
public class ConexaoBD extends SQLiteOpenHelper {
    @Override
    public void onCreate(SQLiteDatabase bd) {
    }

    @Override
    public void onUpgrade(SQLiteDatabase bd, int i, int i1) {
    }
}
```

Para que criemos nosso banco de dados, precisamos definir um nome e versão. Com isso, devemos adicionar aos atributos de nossa classe alguns elementos.

```
private static final String NOME_BD = "exemplo.db";
private static final int VERSAO_BD = 1;
```

Para instanciar, devemos criar um **construtor** passando para o método **super** o contexto, o nome do banco, uma classe alternativa para o cursor (que colocaremos nulo, para mantermos a classe padrão) e uma versão.

```
public ConexaoBD (Context contexto){
    super(contexto, NOME_BD, null, VERSAO_BD);
}
```

Agora vamos começar a trabalhar um pouco com os comandos SQL. O objetivo é fazer uma *query* que crie uma tabela com os elementos nome, sobrenome, idade e matrícula, mantendo a **matricula** como chave primária. Com essa *query*, vamos ao método **onCreate** e incluiremos o comando para que seja criada a tabela. Assim, nós definimos qual o banco de dados e chamamos, sobre sua instância, o método que recebe um comando SQL.

```
@Override
public void onCreate(SQLiteDatabase bd) {
    bd.execSQL("create table registro(nome TEXT, sobrenome TEXT,
       idade TEXT, matricula TEXT PRIMARY KEY)");
}
```

Agora que já temos os métodos **construtor** e **onCreate**, criaremos uma classe para organizar os elementos que salvaremos no banco. Essa classe deve ter, como atributos, todos os elementos que salvaremos. Adicionaremos a ela os **getters** e **setters** de cada atributo.

```
public class Registro{

    private String nome;
    private String sobrenome;
    private String idade;
    private String matricula;

    public Registro() {
    }
}
```

Tudo pronto, agora já temos uma classe que cuidará do nosso banco de dados e outra para representar os elementos, precisamos de uma classe para tratar o acesso ao banco. As classes DAO (*Data Access Object*) são responsáveis por definir "contratos"para o acesso ao banco, padronizar o processo. Criaremos uma classe **RegistroDAO**, que será responsável por todas as operações no banco.

Além de ter como atributos uma instância da **ConexaoBD**, a classe contará com um objeto do tipo **SQLiteDatabase**, onde colocaremos instâncias para escrita e leitura no banco.

```
public class RegistroDAO {
    private ConexaoBD conexao;

    public RegistroDAO(Context contexto) {
        conexao = new ConexaoBD(contexto);
    }
```

```
                                                                    7
    public long inserir(Registro registro){                         8
    }                                                               9
    public Registro buscar(String nome){                            10
    }                                                               11
    public int atualiza(String matricula){                          12
    }                                                               13
    public int deleta(String matricula){                            14
    }                                                               15
}                                                                   16
```

10.3.1 Inserção (create)

Depois de criado o banco de dados, o primeiro passo é inserir informações e popular o banco. No exemplo, uma tela captura as informações do usuário, aloca em uma instância de **Registro** e chama o método **Inserir** da classe **RegistroDAO**.

```
public void Adicionar(View view) {                                  1
    registro.setNome(nome.getText().toString());                    2
    registro.setSobrenome(sobrenome.getText().toString());          3
    registro.setIdade(idade.getText().toString());                  4
    registro.setMatricula(matricula.getText().toString());          5
    long id = dao.inserir(registro);                                6
    if (id == -1) {                                                 7
        Toast.makeText(this, "Matricula ja cadastrada", Toast.      8
           LENGTH_SHORT).show();
    } else {                                                        9
        Toast.makeText(this, "Cadastrado com sucesso", Toast.       10
           LENGTH_SHORT).show();
    }                                                               11
}                                                                   12
```

No método **inserir**, os valores serão alocados em uma instância de **ContentValues**, que recebe estruturas de chave e valor. Depois, declarando um objeto do tipo **SQLiteDatabase** (bd), daremos uma instância com o método **getWritableDatabase**. Por último, usamos o método **insert** passando o nome da tabela, o nome das colunas que

receberão valor *null* e a estrutura com os valores.

```
public long inserir(Registro registro){
    SQLiteDatabase bd = conexao.getWritableDatabase();
    ContentValues valores = new ContentValues();
    valores.put("nome", registro.getNome());
    valores.put("sobrenome", registro.getSobrenome());
    valores.put("idade", registro.getIdade());
    valores.put("matricula", registro.getMatricula());
    return bd.insert("registro", null, valores);
}
```

Com o retorno do tipo *long*, o método **insert** retorna, no caso de uma inserção bem sucedida, o índice da linha onde a informação foi alocada e, no caso de erro, o valor **-1**. Como todos os campos devolvem uma *String* (ainda que vazia) e os elementos do banco são do tipo *Text*, o único erro é a tentativa de inserir um novo valor com uma mesma chave (no caso, a matrícula). Com essa informação, podemos verificar se foi cadastrado corretamente ou se a chave está duplicada.

10.3.2 Busca (Read)

Todo banco de dados existe para que seja consumido, afinal, não tem sentido armazenar informações que não serão usadas. Como temos a **matricula** como chave, implementaremos a busca por este termo.

O primeiro passo é criar o método **buscar**, que será chamado quando o botão equivalente for clicado. A ideia é receber o retorno do método **buscar** da classe Registro-DAO e processá-lo: se for nulo, deixar o texto em branco e, estando preenchido, expor as informações no campo de texto.

```
public void Buscar(View view) {
    Registro registro;
    registro = dao.buscar(matricula.getText().toString());
    if (registro.getMatricula() == null) {
        texto.setText("");
        Toast.makeText(this, "Matricula nao encontrada", Toast.
           LENGTH_SHORT).show();
    }
```

```
    else {
        String string = "Nome: " + registro.getNome() + " " +
            registro.getSobrenome() +
            "\nIdade: " + registro.getIdade() +
            "\nMatricula: " + registro.getMatricula();
        texto.setText(string);
    }
}
```

Agora, devemos implementar o método "buscar"da classe **RegistroDAO**; esse será responsável por todo o processo de acesso ao bando de dados.

Para fazer a busca, usaremos o método **query**, que pede o nome da tabela, a lista das colunas que serão buscadas (projeção), as restrições da busca (seleção), os elementos aos quais deve-se comparar a seleção (argumentos da seleção) e ainda o agrupamento, condição de agrupamento e método de ordenação.

```
public Registro buscar(String matricula) {
    Registro registro = new Registro();
    String[] projecao = {"nome","sobrenome","idade","matricula"};
    String selecao = "matricula = ?";
    String[] selecaoArg = {matricula};
    SQLiteDatabase bd = conexao.getReadableDatabase();
    Cursor cursor = bd.query("registro", projecao, selecao,
       selecaoArg, null, null, null);
    if (cursor.getCount() > 0) {
        cursor.moveToFirst();
        registro.setNome(cursor.getString(0));
        registro.setSobrenome(cursor.getString(1));
        registro.setIdade(cursor.getString(2));
        registro.setMatricula(cursor.getString(3));
    }
    return registro;
}
```

Observação: apesar de ser possível executar a busca por SQL, o método trata detalhes de segurança (como *SQL Injection*) e também organiza melhor.

No caso do exemplo, a projeção será uma lista com os nomes de todas as colunas, a seleção será uma comparação pela coluna **matricula** e o argumento será a matrícula recebida do usuário. O agrupamento, seus argumentos e a ordenação serão nulos.

Diferentemente do caso da inserção, ao invés de recebermos uma instância *SQLiteDatabase* pelo método *getWritableDatabase*, usaremos o método **getReadableDatabase**. Depois, chamaremos o método **query**, que devolve um **cursor** com os elementos resultantes da busca.

Com a lista, o primeiro passo é verificar se ela não está vazia; não estando, vamos ao topo e pegamos as informações da primeira linha (através do método **getString**, que recebe um inteiro relativo ao índice da coluna desejada e retorna o valor correspondente, que deve ser uma *string*), alocamos em um elemento do tipo **Registro** e o retornamos. Como só podemos cadastrar um elemento com uma determinada matrícula, a lista sempre será de um elemento só.

10.3.3 Remoção (Delete)

Às vezes, além de inserir e ler, faz-se necessário deletar uma ou mais linhas do banco. Para isso, a solução é bastante similar à busca. O primeiro passo é recolher a informação do campo **matrícula** e verificar se está preenchida. Estando com algum valor, chamamos o método **deletar** da classe RegistroDAO passando a matrícula como parâmetro.

```
public void Remover(View view) {                                    1
    String mat = matricula.getText().toString();                    2
    int resp = dao.deletar(mat);                                    3
    if(resp == 1){                                                  4
        Toast.makeText(this, "Deletado com sucesso", Toast.         5
           LENGTH_SHORT).show();
    } else {                                                        6
        Toast.makeText(this, "Matricula nao encontrada", Toast.     7
           LENGTH_SHORT).show();
    }                                                               8
}                                                                   9
```

O passo seguinte é construir o método **deletar** na classe RegistroDAO que será

responsável por processar a exclusão no banco de dados.

```
public int deletar(String matricula){
    SQLiteDatabase bd = conexao.getWritableDatabase();
    String selecao = "matricula = ?";
    String[] selecaoArg = {matricula};
    return bd.delete("registro", selecao, selecaoArg);
}
```

A primeira parte é sempre instanciar um acesso à base, depois usamos o método **delete**, que pede como parâmetros o nome da tabela, as restrições desejadas (seleção) e os valores aos quais se deseja comparar (argumentos da seleção). O método retorna o número de linhas afetadas; no caso do exemplo, deve retornar 1 se proceder corretamente e 0, caso haja algum erro.

10.3.4 Atualização (Update)

É bem possível que, para se atualizar um dado, nós apenas realizemos uma deleção e, em seguida, uma inserção com os valores atualizados. Entretanto, utilizar o método de atualizar nos permite alterar apenas algumas informações e otimizar o processo.

Primeiro, recebemos todos os valores dos campos e os alocamos numa instância de Registro. Depois, passamos esse objeto para o método **atualiza** da classe RegistroDAO.

```
public void Atualizar(View view) {
    Registro registro = new Registro();
    registro.setNome(nome.getText().toString());
    registro.setSobrenome(sobrenome.getText().toString());
    registro.setIdade(idade.getText().toString());
    registro.setMatricula(matricula.getText().toString());
    int resp = dao.atualizar(registro);
    if(resp == 1){
        Toast.makeText(this, "Atualizado com sucesso", Toast.
           LENGTH_SHORT).show();
    }else{
        Toast.makeText(this, "Matricula nao encontrada", Toast.
           LENGTH_SHORT).show();
    }
```

```
}
```

O método **atualizar** da classe RegistroDAO, por sua vez, recebe uma instância de Registro e verifica cada um dos atributos; se não forem nulos, são colocados em um objeto do tipo **ContentValue** com sua respectiva coluna como chave. Em seguida, devemos criar uma instância de *SQLiteDatabase* e chamar o método **update**, que recebe como parâmetros o nome da tabela, um *ContentValue* com os valores, as condições de restrição (seleção) e os argumentos dessa condição (argumentos da seleção).

```
public int atualizar(Registro registro){
    SQLiteDatabase bd = conexao.getWritableDatabase();
    ContentValues valores = new ContentValues();
    if(!registro.getNome().isEmpty()){
        valores.put("nome", registro.getNome());
    }
    if(!registro.getSobrenome().isEmpty()){
        valores.put("sobrenome", registro.getSobrenome());
    }
    if(!registro.getIdade().isEmpty()){
        valores.put("idade", registro.getIdade());
    }
    String selecao = "matricula = ?";
    String[] selecaoArgs = {registro.getMatricula()};
    return bd.update("registro", valores, selecao, selecaoArgs);
}
```

O método *update*, bem como o *delete*, retorna o número de colunas afetadas; no caso do exemplo, onde a seleção só deve encontrar uma única matrícula, o valor retornado será 1 se a operação tiver sido bem sucedida e 0, caso contrário.

10.3.5 Outras operações (comandos SQL genéricos)

Apesar da classe *SQLiteDatabase* já fornecer métodos bastante eficientes e organizados para todas as principais operações, é possível, também, realizar qualquer comando *SQL* de forma direta.

Para isso, basta construir uma instância *SQLiteDatabase* e chamar o método **execSQL** passando a string com o comando desejado.

```
public void comando(String sql){
    SQLiteDatabase bd = conexao.getWritableDatabase();
    bd.execSQL(sql);
}
```

10.4 Resultado do exemplo

O resultado foi uma aplicação bastante robusta, que permite trabalhar com o banco de dados criado de forma fácil e intuitiva.

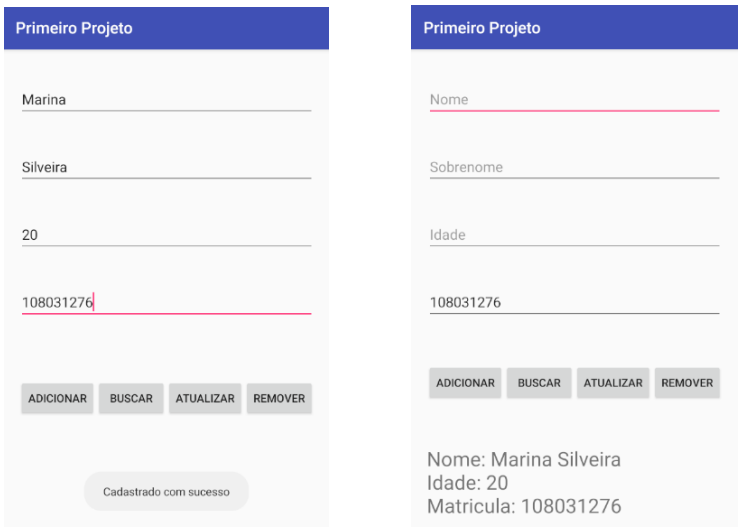

Figura 57: Inserindo e buscando novo registro.

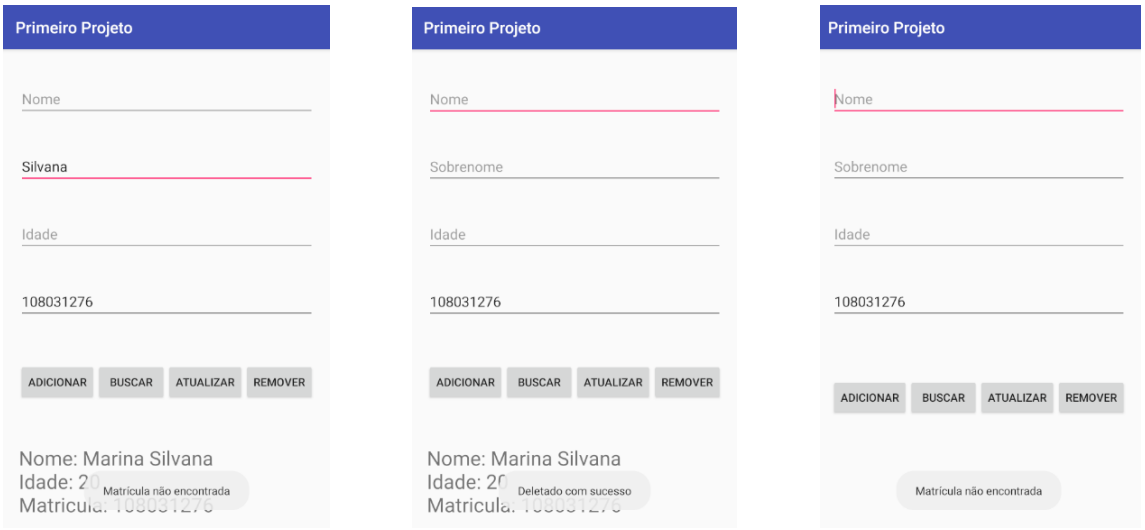

Figura 58: Atualizando, deletando e buscando novamente o registro.

11 Projeto final

Como projeto final, o objetivo foi construir uma aplicação mais complexa, que envolvesse todas as seções que apresentamos ao longo do tutorial. O projeto foi desenvolvido durante uma aula piloto e foi feito em duas partes: a primeira realizada em sala e a segunda, em casa.

A aplicação seria uma lista de músicas, exibindo informações sobre título, álbum e artistas. Na tela inicial, fica exposto, além da lista das músicas, um botão para adicionar novos títulos e uma caixa para escolher por onde ordenar a lista. A tela principal também inclui o recurso de atualizar a lista ao ser puxado para baixo.

Clicando no botão de adicionar, abre-se uma tela com campos para preencher título, álbum e artista; ao fim, um botão para adicionar. Caso o título já tenha sido adicionado, emite-se um aviso para o usuário.

Na tela principal, o clique em um dos itens na lista abre uma outra tela, que exibe as informações e contém dois botões: atualizar e deletar. O clique em deletar abre uma caixa de diálogo, para confirmar a ação; em atualizar, abre outra tela.

A tela de atualizar é bastante similar à adicionar, tem os campos editáveis e dois botões: cancelar e atualizar. O clique em cancelar fecha a página, mas mantém a informação do jeito que está; em atualizar, insere a nova informação no banco e retorna à pagina de informações.

O projeto está disponível na plataforma Github, e pode ser acessado gratuitamente por meio da seguinte URL: `https://github.com/PET-Tele/ListaMusica`. O projeto conta, também, com uma pasta de imagens, que inclui capturas das várias telas da aplicação.

Referências

[1] TELECOMUNICAÇÕES, M. das. *História das Telecomunicações*. Acesso em 03/01/2019. Disponível em: <http://museudastelecomunicacoes.org.br/historia-das-telecomunicacoes/>.

[2] HAMMERSCHMIDT, R. *A incrível (e surpreendentemente antiga) história dos telefones celulares*. Acesso em 04/01/2019. Disponível em: <https://www.tecmundo.com.br/celular/75617-incrivel-surpreendentemente-longa-historia-telefones-celulares.htm>.

[3] NOTíCIAS, B. *Custava R$ 31 mil: como era primeiro celular do mundo*. Acesso em 25/02/2019. Disponível em: <https://www.techtudo.com.br/artigos/noticia/2012/06/historia-dos-telefones-celulares.html>.

[4] RENATO, F. *A história dos telefones celulares*. Disponível em: <https://noticias.bol.uol.com.br/ultimas-noticias/tecnologia/2017/06/30/custava-r-13-mil-como-era-primeiro-celular-do-mundo.htm>.

[5] DEVELOPER, A. *Arquitetura da plataforma*. Acesso em 12/01/2019. Disponível em: <https://developer.android.com/guide/platform/?hl=pt-br>.

[6] TECMUNDO. *O que é XML?* Acesso em 17/01/2019. Disponível em: <https://www.tecmundo.com.br/programacao/1762-o-que-e-xml-.htm>.

[7] MINDORKS. *DiffUtils : Improving performance of RecyclerView*. Acesso em 18/02/2019. Disponível em: <https://medium.com/mindorks/diffutils-improving-performance-of-recyclerview-102b254a9e4a>.