

Kotlin com Android

Crie aplicativos de maneira fácil e divertida



Casa do
Código

KASSIANO RESENDE

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Vivian Matsui

Carlos Felício

[2020]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-75-5

EPUB: 978-85-94188-76-2

MOBI: 978-85-94188-77-9

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Gostaria de agradecer à minha família que sempre esteve comigo em todos os momentos, aos amigos próximos que me apoiaram desde o começo da ideia de escrita deste livro. Agradeço aos alunos aos quais já tive oportunidade de dar aulas pela inspiração e troca de conhecimento.

Deixo aqui também meu muito obrigado à Vivian Matsui pela oportunidade e por acreditar no projeto deste livro, agradeço à Bianca Hubert pelo suporte e atenção prestada durante todo o processo de escrita e estendo os agradecimentos a toda a equipe da Casa do Código.

Agradeço a você, leitor desta publicação, pelo interesse no assunto e vontade de aprender. O nosso futuro depende de pessoas engajadas e com vontade de fazer a diferença, e o conhecimento é o caminho para isso.

PÚBLICO ALVO, PRÉ-REQUISITOS E CÓDIGO-FONTE

Este livro é indicado a todos os desenvolvedores que querem começar a utilizar a linguagem Kotlin para criar aplicativos Android. A abordagem utilizada abrange desde o leitor iniciante no mundo de desenvolvimento até os mais experientes que queiram conhecer a linguagem Kotlin.

É um pré-requisito para o completo entendimento do conteúdo aqui abordado conhecer lógica de programação e suas estruturas básicas em qualquer linguagem. Não é um pré-requisito saber conceitos de Orientação a Objetos nem a linguagem Java, mas se você já possuir esse conhecimento será de grande valia.

Ao longo deste livro, você desenvolverá alguns pequenos projetos e verá muitos exemplos de códigos. Você pode encontrar o código-fonte de todos os projetos deste livro no seguinte repositório do GitHub: <https://github.com/kassiano/livrokotlin>

SOBRE O AUTOR

Iniciei minha carreira em tecnologia em meados de 2007, quando fiz o curso técnico de Redes de Computadores no SENAI. Foi lá que tive os primeiros contatos com programação e conheci a linguagem Visual Basic, juntamente com a plataforma .NET. Como meu curso técnico era de 2 anos e só tive programação em um semestre, tive que começar a estudar programação sozinho. Passava horas e horas em fóruns da Microsoft pesquisando, respondendo dúvidas e tirando dúvidas também! Com toda certeza, estar engajado com a comunidade naquele momento foi crucial para o meu desenvolvimento pessoal como desenvolvedor.

Ainda como estudante do SENAI, eu participei do projeto *Student to Business* da Microsoft, no qual aprendi a linguagem C# e conceitos de Orientação a Objetos. Isso foi revolucionário para mim.

Meses depois, entrei na FATEC para o curso de Tecnologia e Desenvolvimento de Jogos Digitais. Foram 3 anos e meio de muito trabalho, aprendizado e resiliência e, apesar de eu nunca ter trabalhado de fato com desenvolvimento de jogos, programar um jogo envolve desafios complexos de programação que me ajudaram muito a resolver problemas do dia a dia de qualquer sistema.

Em paralelo à faculdade, eu trabalhava com desenvolvimento ASP.NET com C# para uma startup chamada Donuts4U e lá fiquei por 6 anos trabalhando com muitos sistemas, às vezes alocado em clientes, às vezes desenvolvendo produtos internos, mas sempre

com muito aprendizado envolvido. Nesse período, integrei uma equipe altamente qualificada e com visão moderna das coisas e trabalhar junto a essas pessoas contribuiu muito para meu desenvolvimento, com toda a certeza.

Hoje sou professor do SENAI no curso técnico de Desenvolvimento de Sistemas. Ministro aulas de programação, mas principalmente na disciplina de Desenvolvimento mobile com Android e Java. Sigo me dedicando ao ensino desde 2015, realmente é algo que faço com paixão.

Tenho experiência em diversas linguagens de programação, C, C#, Java, PHP, JavaScript, Swift, Python, Kotlin. A questão é que a linguagem de programação é simplesmente nossa ferramenta de trabalho, não nossa razão de viver. Ontem, a linguagem para Android era Java e eu adoro Java; hoje, a linguagem que faz sentido para Android é Kotlin e eu também adoro Kotlin! Quando sair uma linguagem nova, uma plataforma diferente eu estarei lá aprendendo também. Hoje meu foco de estudos em tecnologia está voltado a *Deep Learning*.

Sou pós-graduado em Psicopedagogia também, meu lado de humanas. Mas ser professor é ser de humanas, às vezes não importam os *ifs* e *elses*, os bytes ou códigos, o ensino é transcendente ao meramente técnico. O ensino toca a alma e, citando uma frase de um dos autores de que eu mais gosto na Psicologia, Carl Jung: "*Conheça todas as teorias, domine todas as técnicas, mas ao tocar uma alma humana, seja apenas outra alma humana.*"

PREFÁCIO

Para mim, um bom livro tem que ter uma boa narrativa. Não importa que seja um livro técnico, ele precisa envolver o leitor no processo de aprendizagem com fluência, didática e boas pitadas de embasamento conceitual revelador.

É o que eu identifico neste livro. Apesar de não ter participado do processo de construção do livro, ao lê-lo tive a sensação de que ele tinha sido feito para mim. Não me senti "sozinho" tendo acesso a um corpo de conhecimento, me senti amparado de perto por alguém que queria me mostrar uma novidade, trazendo argumentos e exemplos muito conectados com a minha realidade de desenvolvimento de software.

Faço parênteses para me apresentar: sou Daniel Makiyama e trabalho na área de software há 16 anos. Já aproveito para dizer que, hoje estando no cargo de gestor de desenvolvimento de software e professor de pós-graduação, continuo sendo um desenvolvedor de software, pois continuo aprendendo e desenvolvendo sistemas, e isso nunca deve parar. Aliás, faço um apelo aqui: gestores, nunca parem de desenvolver!

Trabalho atualmente em um time de Arquitetura Global de uma empresa adquirida pela Capgemini, consultoria de TI internacional, responsável pelos sistemas que rodam nos restaurantes do McDonald's no mundo, e na FIAP como professor do MBA de Arquitetura e Desenvolvimento na Plataforma .NET.

Um certo tempo antes de me juntar à empresa na qual trabalho atualmente, decidi em 2009, junto com outros sócios e funding

próprio, criar uma startup chamada Donuts4U. Foi neste momento de grandes mudanças na minha vida que tive a felicidade de conhecer nosso autor. Após uma frustrante e extensiva busca por profissionais formados e experientes em desenvolvimento C# .NET para a startup, tive a sorte de permitir uma chance ao currículo de um jovem de 17 anos que estava fora dos requisitos da vaga. O currículo apresentava um profissional da área de redes de computadores que estranhamente parecia nutrir um grande interesse em programação.

Tamanha foi a minha surpresa quando, durante a entrevista, percebi que ali estava o profissional com maior propriedade em programação que eu havia entrevistado até então. Versado nos conceitos, com exemplos concretos e principalmente muita disposição para aprender. Vivemos uma enxurrada de vivências, desafios intensos e aprendizado contínuo. Construímos uma grande amizade.

Este livro é extremamente prático. Os capítulos nos compelem a abrir o Android Studio e experimentar. É vibrante acompanhar as pequenas dicas focadas em produtividade e organização. Este livro atende de forma precisa ao ímpeto que temos de experimentar rapidamente na linguagem, evitando setups em um primeiro momento e indo direto ao código, aí sim intercalando pequenos setups com códigos bem direcionados, alavancando o que a JetBrains tem a oferecer para a iniciação dos interessados.

É interessante ver como a preocupação com performance, qualidade e produtividade está "embedada" na linguagem e ela já nos direciona neste caminho, o que é muito valioso se pensarmos que se trata de um ambiente mobile. Assim como as importantes

regras de validação do compilador Java para Android, as abstrações propostas pelo Kotlin evitam o emprego de práticas nocivas, inocentes ou ineficazes no código. Neste sentido, o Kotlin apresenta funcionalidades como templates de string, *operador safe call*, *single-expression functions* e classes de dados, todas inspiradas em movimentos convergentes em diversas comunidade de software. Sem contar os métodos fluentes que tornam o código mais semântico e facilitam a sua leitura. Ter menos código para ler significa muito: facilita o entendimento e simplifica o trabalho de manutenção futura.

O livro é fluído como uma aula de curso profissionalizante, direto ao ponto e injetando conceitos à medida que são realmente necessários para a continuidade da atividade prática, de forma até divertida, com a responsabilidade e respeito de trazer um conhecimento acessível para muitos, independente de suas experiências prévias com outras linguagens.

Este é um livro que você abre em um kindle ao lado do seu computador ou coloca em um segundo monitor e vai lendo, codificando, testando e embarcando nesta viagem cheia de interjeições de dúvidas seguidas de epifanias, tudo isso no seu tempo, com calma e com as etapas necessárias para um entendimento mais completo, que é como realmente todo processo de aprendizagem deveria ser. A linguagem do livro é jovial e envolvente, e para mim o conhecimento sempre fica mais fácil desta forma mais informal e prática. As seções de resumo são a cereja do bolo, ajudando-nos a retomar os conceitos e fechar o entendimento sobre o tema abordado no capítulo.

No livro, o autor apresenta muitas dicas de uso da IDE do

Android Studio, como configurar o emulador e outros passos importantes para tornar suave o processo de desenvolvimento e deploy de suas Apps. Isso tudo foi muito esclarecedor para mim, que não tenho muita experiência com a plataforma Android. É incrível como esta plataforma tem um vocabulário e um funcionamento bem próprio (e bem organizado) que vai muito além do ambiente de desenvolvimento clássico em Java. É muito interessante quando uma linguagem nos força a utilizarmos boas práticas através de uma boa estrutura. Isso é um dos motivos do sucesso do Android.

Kassiano constrói os layouts e simula um processo real de desenvolvimento, com as tomadas de decisão quanto a como organizar o código - aqueles questionamentos importantes que levam a boas decisões de como e por que fazer daquela forma. Inicialmente, o autor foca em uma App muito comum e muito utilizada em tutoriais de diversas outras linguagens, que é o aplicativo de listas. Isso facilita que o leitor foque mais no aprendizado da linguagem, pois uma aplicação de listas já é de domínio da maioria das pessoas, além de poder enxergar onde a linguagem Kotlin está, frente a outras linguagens. Por fim, Kassiano fecha com chave de ouro apresentando o processo de desenvolvimento de uma calculadora de bitcoins que se tornou realmente um aplicativo da Play Store e que foi concebido inicialmente para o livro. Isso mostra a dedicação que o autor teve durante a construção do livro.

Fica claro que as ferramentas usadas no livro são de alta produtividade pois este é um dos grandes objetivos da criação desta linguagem. A utilização da ferramenta de ORM no caso de banco de dados é um ótimo exemplo da preocupação deste livro

em apresentar ao leitor uma forma rápida de desenvolver seus aplicativos Android. Sem contar a utilização de conceitos bem atuais, como a criação do banco de dados e tabelas através de código, o que permite a migração estrutural caso a estrutura das tabelas mude ao longo do tempo. Dessa forma, todo o conhecimento sobre o seu aplicativo fica autocontido na solução de código. Isso é muito vantajoso para a manutenção e evolução do seu sistema.

Ao término deste livro, você terá visto na prática todos os passos necessários para construir e publicar na Play Store aplicações para smartphones desenvolvidas na linguagem Kotlin, no ambiente de desenvolvimento Android Studio. O escopo das aplicações apresentadas no livro vai além das funcionalidades da linguagem, com a utilização de banco de dados, acesso a APIs REST e funcionalidades dos smartphones, como notificações, acesso à galeria de imagens, câmera e localização através de GPS.

Bom, não me estendo mais pois vejo que a cada novo parágrafo que escrevo, mais demoro para liberá-lo para esta experiência prazerosa. Sendo assim, ajeite-se na sua cadeira, ligue o notebook ou computador (caso ainda não estiver ligado), coloque este livro ao lado, ou projetado em outro monitor ou kindle e vá montar sua próxima aplicação usando Kotlin agora mesmo! Afinal, o que você está esperando =D?

Daniel Makiyama

Gerente de desenvolvimento de software da Capgemini

Sumário

1 A linguagem Kotlin	1
1.1 Um breve histórico	1
1.2 Interoperabilidade, como assim?	2
1.3 Uma linguagem concisa	3
1.4 Livre-se das referências nulas	6
1.5 Resumindo	8
2 Programando em Kotlin	10
2.1 Conhecendo o Try Kotlin	11
2.2 Inserindo comentários no código	16
2.3 Definindo variáveis val e var	18
2.4 Tipos de dados	23
2.5 Estruturas de decisão	30
2.6 Estruturas de repetição	33
2.7 Funções	34
2.8 Orientação a Objetos	37
2.9 Resumindo	42
3 Configurando o ambiente de desenvolvimento	43

3.1 Android Studio	44
3.2 Configurando o SDK	49
3.3 Primeiro projeto: Hello World	50
3.4 Criando um emulador Android	64
3.5 Executando o projeto	70
3.6 Habilitando o Auto Import	72
3.7 Resumindo	74
4 Anatomia da plataforma Android	75
4.1 Activities	75
4.2 Acessando recursos – Classe R	87
4.3 Views – Componentes visuais	88
4.4 Resumindo	98
5 Primeiro projeto – Cálculo da aposentadoria	99
5.1 Fase de planejamento	99
5.2 Criando o projeto	101
5.3 Criando uma Activity	105
5.4 Criando o layout da Activity	108
5.5 Programando a lógica	121
5.6 AndroidManifest	129
5.7 Resumindo	134
6 Lista de compras	136
6.1 Planejamento	136
6.2 Criando o projeto	137
6.3 Criando o layout	141
6.4 Programando o aplicativo	143

6.5 Pequenas melhorias	156
6.6 Removendo um item da lista	158
6.7 Resumindo	163
7 Lista de compras 2.0	165
7.1 Planejamento	166
7.2 Criando uma nova Activity de cadastro	169
7.3 Personalizando a ListView	184
7.4 Criando a classe de dados	189
7.5 Customizando o adaptador	192
7.6 Cadastrando itens	205
7.7 Exibindo itens cadastrados	211
7.8 Somando os itens da lista	214
7.9 Acessando a galeria de imagens	217
7.10 Resumindo	223
8 Persistência de dados com SQLite	225
8.1 A biblioteca Anko SQLite	225
8.2 Criando o banco de dados	231
8.3 Inserindo dados no banco de dados	243
8.4 Selecionando registros do banco de dados	254
8.5 Removendo um registro	260
8.6 Atualizando um registro	262
8.7 Resumindo	263
9 Calculadora de Bitcoin	265
9.1 Planejamento	266
9.2 Componentizando o layout	268

Sumário	
9.3 API de dados - Mercado Bitcoin	276
9.4 Buscando os dados da API	280
9.5 Efetuando o cálculo de conversão	293
9.6 Resumindo	298
10 Notificações, permissões, localização e publicação	299
10.1 Notificações	299
10.2 Solicitação de permissões	311
10.3 Localização por GPS	317
10.4 Publicação na Play Store	325
11 Conclusão	342

Versão: 24.6.20

CAPÍTULO 1

A LINGUAGEM KOTLIN

Podemos definir Kotlin como uma linguagem de programação pragmática que combina os paradigmas de Orientação a Objetos e Programação Funcional com foco em interoperabilidade, segurança, clareza e suporte a ferramentas - essa é a definição que seus desenvolvedores deram a ela. Já a equipe do Android a define como uma linguagem expressiva, concisa e poderosa.

A linguagem foi criada pela JetBrains, a mesma empresa criadora do Android Studio, e veio como uma alternativa ao Java para desenvolvimento Android. Temos de ter clareza que a intenção do Kotlin não é substituir o Java, mas ser uma linguagem moderna de alta produtividade.

1.1 UM BREVE HISTÓRICO

Na verdade, o Kotlin não nasceu ontem, o primeiro anúncio da linguagem veio lá em 2011 durante o JVM Language Summit em que a JetBrains revelou que estava trabalhando havia quase um ano no projeto Kotlin. A ideia era criar uma nova linguagem estaticamente tipada para a JVM. O projeto Kotlin juntava anos de experiência na criação de ferramentas para diversas outras linguagens e tinha como objetivo criar uma linguagem que fosse

produtiva e, ao mesmo tempo, de simples aprendizado.

Algum tempo depois, já em 2016, a versão 1.0 foi lançada, considerada a primeira versão estável da linguagem. Naquele momento você já conseguiria utilizar a linguagem no Android Studio através da instalação de um plugin. Tenho que deixar claro que Kotlin não é uma linguagem exclusiva do Android, de acordo com seus desenvolvedores, ela é para uso geral. Ela é uma linguagem que roda na JVM, ou seja, onde funciona Java, o Kotlin também funciona!

Em 2017, durante o evento Google I/O, foi feito o anúncio de que Kotlin é oficialmente a linguagem para desenvolvimento Android. Isso já era esperado, porque Kotlin vinha recebendo muitos feedbacks positivos da comunidade de desenvolvedores, então era muito mais lógico fazer do Kotlin uma linguagem oficial para desenvolvimento Android do que uma outra linguagem, como GoLang por exemplo. De acordo com a equipe do Android, a escolha do Kotlin se deve ao fato de ela ser uma ótima linguagem e tornar a criação de aplicativos mais produtiva e divertida. Outro ponto chave é que todo ecossistema Android é compatível com Kotlin, então tudo o que já foi desenvolvido, todas as bibliotecas do Android funcionam sem a necessidade de serem reescritas.

A seguir, veremos mais detalhes sobre os principais pontos da linguagem: interoperabilidade, concisão e proteção contra nulos.

1.2 INTEROPERABILIDADE, COMO ASSIM?

Chamamos de interoperabilidade a capacidade de uma linguagem se comunicar com outra. Quando dizemos que o Kotlin

tem total interoperabilidade com Java, isso significa que podemos utilizar códigos que foram escritos em Java no nosso código em Kotlin e isso é fantástico! Fantástico porque podemos aproveitar qualquer biblioteca ou classe escrita em Java nos nossos aplicativos feitos em Kotlin, e mais ainda, se durante o desenvolvimento você entende que determinada funcionalidade ficaria melhor se fosse escrita em Java, você poderá simplesmente criar a classe em Java e integrar de forma totalmente transparente em seu código Kotlin.

Vamos a um exemplo prático, vamos supor que você criou uma classe em Java chamada `Calculadora` e, dentro dela, você criou um método chamado `somar`, que faz a soma de dois números inteiros. O código seria o seguinte:

```
public class Calculadora{  
    public int somar(int a, int b){  
        return a+b;  
    }  
}
```

E agora você gostaria de utilizar essa mesma classe no seu código Kotlin:

```
val calc = Calculadora()  
val resultado = calc.somar(2,2)  
println(resultado)
```

E você simplesmente pode fazer isso, isso é interoperabilidade, isso é muito legal!

1.3 UMA LINGUAGEM CONCISA

Quando falamos que Kotlin é uma linguagem concisa, basicamente estamos falando que precisamos escrever menos código para fazer a mesma coisa que uma outra linguagem faria

com mais código. A linguagem que utilizaremos como parâmetro será o Java.

Vamos a mais um exemplo prático, o código Android a seguir define que o click de um botão que tenha um identificador `bt_login` chamará um método de nome `efetuarLogin`:

```
Button bt_login = (Button) findViewById(R.id.bt_login);
bt_login.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        efetuarLogin();
    }
});
```

Na plataforma Android, é muito comum definir IDs de elementos com letras minúsculas separadas por *underline* (_). Essa convenção de nomenclatura se estende a qualquer elemento de resource, nome de arquivos, configuração de cores etc. Utilizar essa mesma convenção no código pode ser particularmente útil pois você sabe facilmente quais variáveis se referem a componentes de UI.

Em Kotlin, poderíamos escrever o mesmo código da seguinte maneira:

```
bt_login.setOnClickListener{
    efetuarLogin()
}
```

Incrível, não é? Vamos ver mais um exemplo, vamos supor que você precise criar uma classe de modelo que vai representar um

cliente da sua aplicação. Para simplificar, esse cliente terá somente um e-mail e um nome de usuário. Em Java, sua classe ficaria assim:

```
public class Cliente {  
  
    private String email;  
    private String nomeUsuario;  
  
    public Cliente(String email, String nomeUsuario) {  
        this.email = email;  
        this.nomeUsuario = nomeUsuario;  
    }  
  
    public String getNomeUsuario() {  
        return nomeUsuario;  
    }  
  
    public void setNomeUsuario(String nomeUsuario) {  
        this.nomeUsuario = nomeUsuario;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
}
```

Já em Kotlin, poderíamos escrever a mesma classe assim:

```
data class Cliente(var email:String, var nomeUsuario: String)
```

É isso mesmo, você escreveria a mesma classe com somente uma linha de código!! Durante todo o livro veremos muito mais exemplos de códigos concisos, e se você ainda não estiver convencido de que Kotlin é uma linguagem concisa, certamente se convencerá.

1.4 LIVRE-SE DAS REFERÊNCIAS NULAS

Um erro de referência nula ou o famoso **NullPointerException**, é um tipo de erro muito comum no mundo de desenvolvimento de software. Quando temos um erro de referência nula no nosso programa significa que estamos tentando acessar um objeto que ainda não possui um espaço alocado na memória RAM da máquina. Complicado? Na verdade, não. Veja o seguinte exemplo em Java:

```
String nomeUsuario = null;
```

O código declara uma variável `nomeUsuario` porém atribui a ela um valor nulo. Isso significa que ainda não existe um espaço na memória do computador para essa variável, e se tentarmos acessá-la teríamos um erro de referência nula, `NullPointerException`:

```
String nomeUsuario = null;  
  
// Essa linha de código gera um erro de NullPointerException  
Log.d("TAG", nomeUsuario);
```

O erro ocorre pois o método `LOG.d` tenta acessar uma variável nula. Se você já programa em alguma outra linguagem é possível que você já tenha visto esse tipo de erro muitas vezes. A boa notícia é que, programando em Kotlin, você raramente verá esse tipo de erro! E isso porque o Kotlin possui uma característica muito interessante, que é a proteção contra nulos (*null safety*). Isso diminui drasticamente os erros do tipo `NullPointerException` e ela resolve isso de uma forma muito prática.

Em Kotlin, por padrão nenhuma variável ou objeto pode ter um valor nulo, logo, dificilmente você encontrará um erro de

`NullPointerException`. Vamos ver um exemplo:

```
var nomeUsuario :String = null
```

Esse código simplesmente não compila porque a variável `nomeUsuario` não pode ter um valor igual a `null`. O seguinte erro será gerado:



```
var nomeUsuario :String = null
```

Null can not be a value of a non-null type String

Figura 1.1: Erro valor nulo

Este erro diz "*Null can not be a value of a non-null type String*", que em português podemos traduzir como "Valores nulos não podem ser definidos ao tipo não nulo String". Na prática, quer dizer que essa variável não pode ter um valor nulo. Para resolver, devemos inicializar a variável com um valor diferente de nulo:

```
var nomeUsuario :String = ""
```

Agora sim o código compila, pois inicializamos a variável com uma string vazia, o que definitivamente não é nulo.

Mas Kotlin não é uma ditadura. Se o desenvolvedor desejar, ele pode deixar explícito que a variável pode receber um valor nulo. Para isso, basta adicionar o operador `?` após o tipo da variável, assim:

```
var nomeUsuario :String? = null
```

Dessa forma, a variável pode receber um valor nulo se assim o desenvolvedor desejar. No entanto, para utilizar essa variável o

compilador nos obriga a fazer uma validação antes:

```
var nomeUsuario :String? = null  
  
if (nomeUsuario !=null){  
  
    println( nomeUsuario.length )  
}
```

Dentro do `if`, o compilador assegura que aquela variável terá um valor e não ocasionará um erro de referência nula. Podemos ainda utilizar operador `? ,` chamado de `safe call`, que simplesmente ignora a chamada se a variável for nula:

```
var nomeUsuario :String? = null  
println( nomeUsuario?.length )
```

Ainda existe outra forma de escrever esse código, mas não é uma maneira recomendada - e segundo a documentação oficial, essa seria a forma dos apaixonados por `NullPointerException` de escrever códigos. O Kotlin abre a possibilidade de forçarmos o acesso à referência de alguma variável através do operador `!! .`

```
var nomeUsuario :String? = null  
println( nomeUsuario!!.length )
```

Este código gera um erro do tipo `KotlinNullPointerException` pois tentamos acessar uma variável nula e assim voltamos ao problema inicial. Por esse motivo, não é recomendada a utilização deste operador. O recomendado é sempre fazer a validação das variáveis ou utilizar o operador `safe call ? .`

1.5 RESUMINDO

Neste capítulo, conhecemos um pouco da história da linguagem, sua evolução e suas principais características. No

próximo capítulo, vamos botar a mão na massa e escrever códigos em Kotlin. Vamos aprender como criar variáveis, constantes, como trabalhar com estruturas de decisão e repetição, como criar funções, classes e objetos. Tudo vai nos dar base para criar aplicativos fantásticos! Ao final, montaremos alguns projetos bem legais que vão explorar recursos como GPS, câmera, banco de dados e muito mais! Vamos lá!

CAPÍTULO 2

PROGRAMANDO EM KOTLIN

Chegou a hora de botar a mão na massa e desenvolver alguns códigos em Kotlin! Neste capítulo, vamos aprender como manipular variáveis, como fazer estruturas condicionais, trabalhar com coleções de dados e estruturas de repetição. Porém, existem outros tantos recursos da linguagem que veremos depois, durante a execução dos projetos práticos. Nossa objetivo agora é conhecer a estrutura básica da linguagem.

Todos os exemplos que vamos utilizar podem ser testados no site <https://try.kotlinlang.org>. Esta é uma plataforma online em que é possível executar códigos em Kotlin. Lá você também encontrará vários exemplos da linguagem.

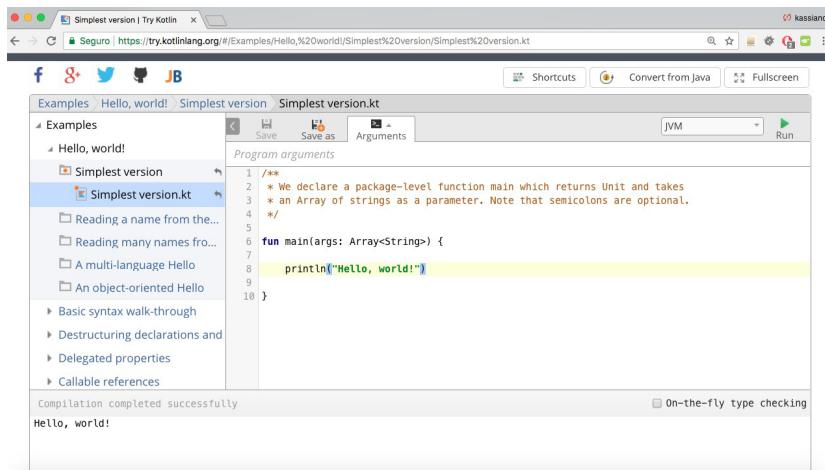
Usaremos a interface do Try Kotlin para aprender as bases da linguagem e rodar nossos códigos, no entanto você precisará estar conectado à internet para utilizá-la pois é uma plataforma on-line. Alternativamente, você poderá instalar o compilador de linha de comando e rodar todos os códigos sem necessidade de conexão com a internet. Nesse caso, existe um pequeno tutorial na página oficial do Kotlin:

<https://kotlinlang.org/docs/tutorials/command-line.html>

Mas neste capítulo vamos nos basear na plataforma Try Kotlin.
Vamos lá!

2.1 CONHECENDO O TRY KOTLIN

Ao acessar o link <https://try.kotlinlang.org> você verá a tela inicial do Try Kotlin:



The screenshot shows the Try Kotlin web application. At the top, there's a header with the title "Simplest version | Try Kotlin". Below the header, the URL "Seguro : https://try.kotlinlang.org/#/Examples/Hello,%20world!/Simplest%20version/Simplest%20version.kt" is visible. The main area is a code editor with the following code:

```
1 /**
2 * We declare a package-level function main which returns Unit and takes
3 * an Array of strings as a parameter. Note that semicolons are optional.
4 */
5
6 fun main(args: Array<String>) {
7
8     println("Hello, world!")
9
10 }
```

To the left of the code editor, there's a sidebar with a tree view of examples under "Hello, world!":

- Simplest version
- Simplest version.kt
- Reading a name from the...
- Reading many names fro...
- A multi-language Hello
- An object-oriented Hello
- Basic syntax walk-through
- Destructuring declarations and
- Delegated properties
- Callable references

At the bottom of the code editor, it says "Compilation completed successfully" and "Hello, world!". There are also buttons for "Shortcuts", "Convert from Java", and "Fullscreen".

Figura 2.1: Try Kotlin

Vamos observar essa tela com mais detalhes. Do lado direito e
tomando a maior parte da tela, temos um editor de códigos:



Figura 2.2: Editor Try Kotlin

1. Editor de códigos em si. Aqui você escreverá seus códigos em Kotlin.
2. Botões para salvar seus códigos. Você só poderá salvar algum código se estiver logado na plataforma.
3. Caixa de seleção para escolher se seu código será executado em JVM, JavaScript, JavaScript(Canvas) ou JUnit.
4. Botão para executar o código escrito no editor.

Do lado esquerdo, temos uma estrutura de pastas com diversos exemplos de código. Você pode clicar em qualquer arquivo, que seu código será mostrado no editor.

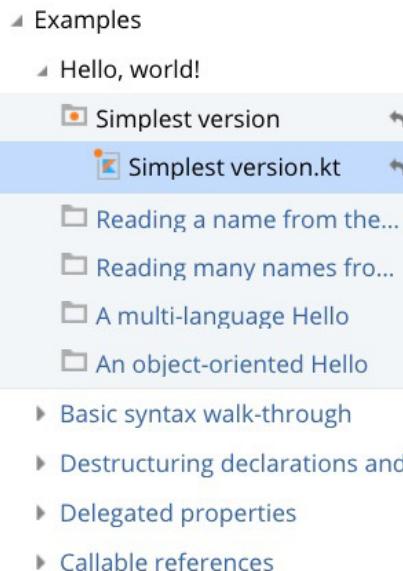


Figura 2.3: Exemplos de código Try Kotlin

Na parte inferior da tela, vemos o console:

```
Compilation completed successfully
Hello, world!
```

Figura 2.4: Console Try Kotlin

É aqui que serão exibidos erros ou mensagens programadas por você.

Ao entrar na plataforma, o editor de códigos já vem selecionado com o arquivo `Simplest Version.kt`, que contém o código do famoso "Hello World" feito na versão mais simples.

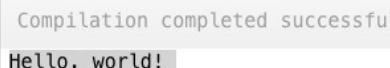
Vamos entender um pouco melhor esse código.

Repare que esse código começa com a seguinte linha:

```
fun main(args: Array<String>)
```

Essa é a função `main` do Kotlin, ela recebe um `Array` de `String` chamado `args`, que são os argumentos que podem ser passados para o programa. A função `main` é o ponto de partida de todo programa feito em Kotlin. É claro que podemos criar outros arquivos, classes, funções e métodos, mas o ponto de partida de um programa em Kotlin sempre será a função `main`, por isso o comando para exibir a mensagem de "Hello, world!" : `println("Hello, world!")` está dentro da função `main`.

Vamos executar esse programa e ver seu resultado. Para isso, clique no botão `Run` localizado na parte superior direita da tela e observe o resultado exibido no console:



```
Compilation completed successfully
Hello, world!
```

Figura 2.5: Hello World em Kotlin

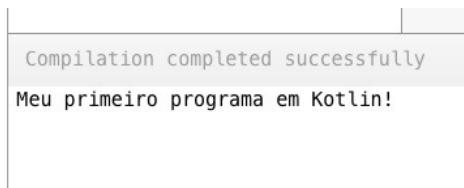
Vamos modificar um pouco esse código. Mude a linha:

```
println("Hello, world!")
```

para:

```
println("Meu primeiro programa em Kotlin!")
```

Execute o programa novamente e veja o resultado exibido no console:



```
Compilation completed successfully
Meu primeiro programa em Kotlin!
```

Figura 2.6: Meu primeiro programa em Kotlin

A esse ponto você já deve ter percebido que a função `println` recebe um texto como parâmetro e o exibe no console! Existe também a função `print` que faz a mesma coisa, com uma única diferença: o `println`, além de exibir a mensagem no console, também pula uma linha no final e assim a próxima mensagem será exibida na linha abaixo; já o `print` não pula uma linha ao final da mensagem, sendo assim, a próxima mensagem será exibida na frente da anterior. Vamos fazer um teste, modifique seu código para ficar assim:

```
fun main(args: Array<String>) {
    print("Meu primeiro ")
    print("programa em Kotlin!")
}
```

Execute e veja o resultado no console. Mudou alguma coisa da execução anterior utilizando o comando `println`? Não mudou nada, porque o comando `print` não pulou uma linha no final e apesar de termos 2 `print`s nesse programa ele exibiu no console como uma única linha. Troque agora a função `print` por `println` e execute o programa novamente.

```
Compilation completed success  
Meu primeiro  
programa em Kotlin!
```

Figura 2.7: Resultado `println`

Perceba que agora ele quebrou o texto em duas linhas, pois após o primeiro `println` ele pulou uma linha e o próximo foi exibido na linha posterior!

Lembre-se: a função `main` é o ponto de partida de todo programa em Kotlin, por isso todos exemplos de código desse capítulo devem estar dentro da função `main`.

2.2 INSERINDO COMENTÁRIOS NO CÓDIGO

Comentários são textos inseridos no código, mas que serão ignorados pelo compilador e não vão interferir no funcionamento do programa. São úteis para explicar alguma lógica desenvolvida ou simplesmente o que faz, de fato, algum comando. Para quem está aprendendo, é muito importante comentar o código para depois você mesmo poder entender o que está acontecendo.

Podemos comentar um programa em Kotlin de duas maneiras. A primeira é utilizando duas barras e, em seguida, o comentário:

```
//Este é um comentário em Kotlin!
```

Esta forma de comentário serve para comentar uma única

linha. A segunda maneira de se comentar um código é usando os delimitadores: `/*` para marcar o início do comentário e `*/` para marcar o fim do comentário. Essa maneira é particularmente útil quando o comentário possui mais de uma linha, veja um exemplo:

```
/*
Este é um comentário em Kotlin,
Com mais de uma linha!
*/
```

Vamos aproveitar e inserir alguns comentários no "Hello World" que fizemos:

```
/*
A função main é o ponto de partida de qualquer programa em Kotlin,
Todos os códigos devem estar dentro dela, ou ser chamados a partir dela
*/
fun main(args: Array<String>) {
    // o comando print exibe uma mensagem no console, mas não pula a linha.
    print("Meu primeiro ")

    // o comando println exibe a mensagem no console e pula uma linha no final.
    println("programa em Kotlin!")
}
```

Dica: no contexto de aprendizagem, é uma ótima ideia comentar todos os seus códigos, todos os comandos novos que você aprender. No contexto de uma aplicação real, utilize comentários com cautela, comente o que for necessário para não ficar poluindo o código com comentários que não são pertinentes à lógica daquele programa.

2.3 DEFININDO VARIÁVEIS VAL E VAR

Em programação, é muito comum o uso de variáveis. Utilizamos variáveis para guardar algum valor que usaremos posteriormente no código. Por exemplo, vamos supor que eu gostaria de criar um programa que efetue a soma de dois números inteiros. Logo de cara eu identifico que preciso desses dois números inteiros, então preciso de pelo menos duas variáveis para guardar cada um. Darei um nome a elas, vamos supor `x` e `y`, sendo que `x` contém um valor e `y` contém outro valor, então, meu programa fará a operação de `x + y`. Naturalmente, eu gostaria de saber qual o resultado dessa operação, logo eu precisaria de mais uma variável para guardar o resultado da operação. Poderia ser uma variável `r`, então a operação ficaria assim: `r = x + y`.

Podemos definir uma variável como um espaço reservado na memória do computador em que eu posso guardar valores e resgatá-los através de um nome.

Para criar variáveis em Kotlin, utilizamos a palavra-chave `var`

seguida da definição de seu tipo, assim:

```
var idade:Int = 22
```

Esse código indica que a variável `idade` é do tipo `Int` e tem um valor inicial de `22`. Em Kotlin, é obrigatório que a variável tenha um valor inicial, porém não é obrigatória a indicação de seu tipo. Por exemplo, o mesmo código poderia ser escrito da seguinte maneira:

```
var idade = 22
```

Dessa forma, a variável `idade` assume o tipo `Int` porque o valor que foi atribuído a ela é `Int`. O Kotlin faz isso com um recurso chamado *inferred type* ou inferência de tipo, em português. Isso não é uma novidade no mundo da programação, você encontra facilmente outras linguagens com o mesmo recurso, como o C# ou Scala.

Devemos tomar certos cuidados em relação à definição correta dos valores para que o Kotlin consiga inferir o tipo correto da variável. Imagine uma situação em que você deve guardar a altura de uma pessoa. Como a altura normalmente não é um número exato, poderíamos utilizar uma variável do tipo `Double` para guardar uma altura, assim é possível guardar valores quebrados, `1.60`, `1.75`, `1.80` etc. Mas imagine que, na definição da variável, você defina o valor inicial de uma pessoa com 2 metros de altura, assim: `altura = 2`. Dessa forma, o compilador assume que a variável `altura` é do tipo `Int` porque `2` é um número inteiro!

Para não cair nesse tipo de erro, você pode sempre definir o tipo da variável ou quando for definir seu valor, passar um valor `Double`, assim: `idade = 2.0`. Com isso, o Kotlin entende que a

variável `idade` é do tipo `Double` e infere o tipo correto.

Os tipos básicos que o Kotlin aceita são:

Tipo	Descrição
<code>Double</code>	Tipo numérico para valores de ponto flutuante, com <code>Double</code> é possível guardar valores com precisão bem grande pois ele ocupa 64 bits
<code>Float</code>	Tipo numérico para valores de ponto flutuante, com a diferença de que tem uma precisão menor que o <code>Double</code> pois ele ocupa 32 bits
<code>Long</code>	Usado para valores inteiros, a diferença dele para o <code>Int</code> é que, como ele ocupa 64 bits, você consegue representar números muito maiores
<code>Int</code>	Talvez um dos tipos numéricos mais comuns, é para guardar números inteiros com uma precisão de 32 bits
<code>Short</code>	Também guarda valores inteiros, porém com capacidade máxima de metade de um <code>Int</code> , pois ocupa somente 16 bits
<code>Byte</code>	Também guarda números inteiros, porém seu número máximo é 127 pois ele ocupa somente 8 bits
<code>String</code>	É um tipo para guardar valores em texto, uma frase, um nome ou qualquer valor que seja um texto
<code>Char</code>	É um tipo para guardar um único caractere, diferente da <code>String</code> , em que você pode guardar textos
<code>Boolean</code>	É usado para valores booleanos, ou seja, valores que são verdadeiro (<code>True</code>) ou falso (<code>False</code>)

Mais adiante darei exemplos de uso dos tipos mais comuns.

Variáveis imutáveis

Utilizamos a palavra-chave `val` quando queremos definir uma variável imutável, ou seja, uma variável que não terá seu valor alterado posteriormente. Há diversos casos em que o uso de uma variável imutável é mais adequado do que de uma variável comum. A principal diferença é que a variável, como seu próprio nome diz, não possui um valor fixo, podendo variar de acordo com a

necessidade; já uma variável imutável tem seu valor fixo, e uma vez definido seu valor este não será mais alterado.

Imagine que vamos construir um App que calcula a área de um círculo. Para esse cálculo, você utilizará o valor de Pi, que é um valor fixo. Pi vale 3,1415926, aproximadamente, e nunca terá seu valor alterado, então, dentro do programa é mais adequado declararmos Pi como uma variável imutável em vez de uma variável comum.

Voltando ao exemplo anterior, se eu declarasse a variável `idade` como `val` e posteriormente tentasse mudar seu valor, o compilador reclamaria e o código não iria compilar:

```
val idade = 22
idade = 26
/*Esse código não compila pois estou tentando alterar o valor de
uma variável imutável */
```

Gerando o seguinte erro:

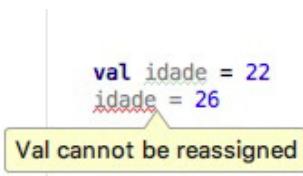


Figura 2.8: Erro variável imutável

Esse erro "*Val cannot be reassigned*" pode ser traduzido como "Val não pode ter seu valor redefinido", ou seja, se utilizamos um `val` não podemos mais alterar seu valor depois que ele foi definido.

No entanto, se ela for declarada como `var`, ela pode ter seu

valor alterado normalmente:

```
var idade = 22  
idade = 26  
println(idade)
```

Mas então, quando usar `var` e quando usar `val`? Bem, por via de regra, defina suas variáveis como `val` e, caso haja necessidade, mude para `var`. Isso melhorará o resultado final dos seus códigos em questão de segurança e clareza. Segurança, porque você garante que variáveis imutáveis não terão seu valor trocado por qualquer motivo; e clareza, porque ficam claras as intenções de design utilizadas na construção do código.

Vamos fazer um pequeno exercício e criar um pequeno programa que calcula a área de um retângulo, um cálculo bem simples de ser feito. Para isso, abra o Try Kotlin e apague todo o código existente, pois vamos criar um código novo.

Para o cálculo de área de um retângulo, basta multiplicar sua base pela altura, então no nosso programa usaremos primeiro uma variável para guardar o valor da `base`, depois uma variável para guardar o valor da `altura` e, por fim, uma variável para guardar o `resultado` do cálculo:

```
fun main(args: Array<String>) {  
  
    val base = 3  
    val altura = 7  
  
    val resultado = base * altura  
    println(resultado)  
}
```

Execute esse código no site Try Kotlin e veja o resultado. Perceba que nesse exemplo todas as variáveis foram definidas

como `val` pois não houve a necessidade de que nenhuma das variáveis mudasse seu valor.

2.4 TIPOS DE DADOS

Em Kotlin, tudo é um objeto, diferentemente de Java, em que existia uma diferenciação de tipo de dados primitivos e objetos. Na prática, isso quer dizer que todas as variáveis e tipos que usamos em Kotlin possuem propriedades e métodos, isto é, todo objeto possui características e comportamentos específicos.

Podemos fazer um comparativo com o tipo `int` do Java e o tipo `Int` do Kotlin. Em essência, ambos servem para a mesma coisa: guardar números inteiros, porém o funcionamento é diferente. O tipo `int` do Java é um tipo primitivo, isso quer dizer que é uma variável que somente guarda um valor e não tem nenhum comportamento atrelado a ela. Já o tipo `Int` do Kotlin é um objeto e, sendo um objeto, além de guardar um valor inteiro ele possui métodos que podem nos auxiliar no desenvolvimento.

O `Int` do Kotlin se assemelha ao tipo `Integer` também do Java, que é um objeto que contém um *wrapper* para o tipo primitivo `int`, então, por dentro, tanto o `Int` do Kotlin quanto o `Integer` do Java contêm um tipo primitivo `int` dentro deles.

Um exemplo são os métodos de conversão de tipos. Imagine que você tenha uma variável do tipo `Int` e precise convertê-la para `Double`, ou `Float`, ou até mesmo para `String`. Para isso, basta utilizar os métodos `toDouble()`, `toFloat()` e `toString()`. Veja:

```
val x: Int = 10;

var y: Double = x.toDouble() //Retorna um objeto Double a partir do valor de x

var z: Float = x.toFloat() //Retorna um objeto Float a partir do valor de x

var a: String = x.toString() //Retorna um objeto String a partir do valor de x
```

Dados numéricos

Os tipos numéricos suportados em Kotlin são muito parecidos com os suportados pelo Java. Existem os tipos `Double` , `Float` , `Long` , `Int` , `Short` e `Byte` , todos eles muito parecidos com os tipos suportados em Java, mas com a principal diferença de que todos são objetos.

A seguinte tabela mostra o tipo e a quantidade de bits que cada tipo ocupa na memória:

Tipo	Bit
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Uma coisa interessante da linguagem é a utilização do *underline* para facilitar a legibilidade do código na definição de variáveis numéricas. Por exemplo, se eu definir uma variável com valor de 1 milhão, ela ficaria assim:

```
val umMilhao = 1000000
```

O que não tem nada de errado, mas fica um pouco difícil de ler qual número é aquele sem contar a quantidade de zeros. Para facilitar a leitura desse código, poderíamos utilizar o caractere `_` para fazer uma separação de milhar:

```
val umMilhao = 1_000_000
```

Strings e caracteres

Uma String é um tipo de dado utilizado para guardar conteúdos do tipo texto. Esse conteúdo deve ser definido utilizando aspas duplas, assim:

```
val texto = "Boa tarde, seja bem-vindo ao sistema"
```

Uma das coisas bem legais que a linguagem possui são os templates. Templates de Strings são trechos de código inseridos diretamente em uma String e que têm seu resultado concatenado junto à String.

Uma vantagem dos templates é que se evita concatenação de Strings, e a operação de concatenação é custosa ao computador, pois ele deve realocar o texto em novo espaço de memória. E se essa operação for repetida diversas vezes durante o programa, com toda a certeza o desempenho geral do App será comprometido.

Com esse recurso, é possível utilizar o valor de uma variável dentro da String através do caractere `$`. Veja um exemplo:

```
fun main(args: Array<String>) {  
  
    val nomeUsuario = "Kassiano"  
    val saudacao = "Bem-vindo, $nomeUsuario"
```

```
    println(saudacao)
}
```

Execute esse código no Try Kotlin e veja o resultado.

Você pode definir também um texto com várias linhas, basta utilizar o delimitador de 3 aspas duplas """ :

```
val text = """
    Exemplo de texto
    com mais de uma
    linha
"""
```

Repare que inicio com 3 aspas duplas e finalizo com mais 3 aspas duplas, desta maneira eu consigo escrever textos com várias linhas.

Booleanos

O tipo booleano é definido como Boolean , e só pode receber valores true (verdadeiro) ou false (falso). Com tipos booleanos, podemos executar as seguintes operações:

Operação	Operador
E	&&
OU	|
NÃO	!

Uma coisa interessante é que o tipo booleano possui funções para essas operações. São elas and , or e not , veja:

```
fun main(args: Array<String>) {
    val b1 = true
    val b2 = false
```

```
    val c1 = b1.and(b2) //Retorno será false
    val c2 = b1.or(b2) //Retorno será true
    val c3 = b1.not() //Retorno será false

    println("$c1 $c2 $c3")
}
```

Execute esse código no Try Kotlin e veja que será impresso no console o seguinte texto: `false true false`.

Listas e Arrays

Quando precisamos armazenar mais de um valor em uma variável, podemos utilizar um `Array`. Um `Array` é como se agrupássemos várias variáveis em uma única variável. Imagine o cenário em que você precise armazenar a nota de 10 alunos. Nada impede de você criar 10 variáveis e armazenar cada nota em uma variável, mas e se fossem 100 alunos, e se fossem 1000 alunos, seria viável criar mil variáveis? Nesse caso, o uso de um `Array` é necessário. Você pode definir um `Array` de mil posições e em cada posição armazenar uma nota.

Um `Array` sempre terá um tamanho fixo e eu consigo definir e resgatar valores através de seu índice utilizando colchetes (`[]`).

Veja um exemplo, vou definir um `Array` de inteiros de 4 posições:

```
val arrayInt :Array<Int> = arrayOf(1, 2, 3, 4)
```

E para acessar ou definir um valor do `Array`, basta utilizar os colchetes passando sua posição:

```
val arrayInt :Array<Int> = arrayOf(1, 2, 3, 4)
```

```
//passando o índice 2 para acessar o valor da posição 2 do array  
val x = arrayInt[2]  
  
println(x)
```

Nesse exemplo, foi definido um `Array` de inteiros com 4 valores (1, 2, 3, 4). Em seguida, foi criada uma variável `x` que recebe o valor que está na posição 2 deste `Array`. Alguns podem pensar que o valor da segunda posição é justamente o número 2 , mas todo `Array` tem seu início na posição 0, então o valor da posição 2 é 3 , nesse caso.

No entanto, nem sempre sabemos antecipadamente a quantidade de valores que precisamos armazenar. E se eu precisasse adicionar mais um valor nesse `Array` ? Sendo um `Array` , isto não é possível, e para esses casos usamos listas.

As estruturas de listas são parecidas com as estruturas de `Array` , com algumas diferenças. Listas essencialmente são estruturas cujo tamanho não preciso saber antecipadamente e posso adicionar novos valores conforme a necessidade. No entanto, em Kotlin há uma diferenciação para listas mutáveis e listas não mutáveis, isso quer dizer que existe um tipo específico de listas que aceita a adição de novos valores, as mutáveis, e um tipo de lista que não aceita novos valores, as imutáveis.

Para definir uma lista mutável, podemos usar a seguinte sintaxe:

```
val lista = mutableListOf(1,2,3,4)
```

Eu ainda consigo utilizar os colchetes para acessar os valores da lista assim como em `Arrays` . A novidade agora é que eu posso adicionar valores à lista utilizando a função `add` , veja:

```
val lista = mutableListOf(1,2,3,4)
lista.add(5) //adicionando um novo valor a lista
```

Além dessa característica, listas possuem algumas funções muito úteis no dia a dia. Veremos 3 delas, a função `first`, `last` e `filter`. A função `first` retorna sempre o primeiro elemento da lista:

```
val lista = mutableListOf(1,2,3,4)
val item = lista.first() //a variável item ficará com valor 1
```

A função `last` retorna o último item da lista:

```
val lista = mutableListOf(1,2,3,4)
val item = lista.last() //a variável item ficará com valor 4
```

A função `filter` aplica um filtro específico na lista, ela é bem bacana e economiza bastantes linhas de código. Vamos supor que nesta mesma lista nós quiséssemos aplicar um filtro e obter somente números pares. Isso é possível com o seguinte código:

```
val lista = mutableListOf(1,2,3,4)
val numerosPares = lista.filter { it % 2 == 0 }
```

Este código cria uma nova lista chamada `numerosPares` somente com os números pares! A função `filter` realiza uma interação na lista e aplica o filtro de acordo com o código que escrevemos dentro das chaves. Nesse caso, o código verificava se cada elemento da lista dividido por 2 é igual a 0.

Para definir uma lista imutável, podemos usar a seguinte sintaxe:

```
val lista = listOf(1,2,3,4)
```

Todos os métodos utilizados na lista mutável funcionam em listas imutáveis, com exceção do método `add`, pois em uma lista

imutável não é possível adicionar novos valores.

Nos capítulos seguintes, na execução dos projetos veremos outras funções úteis de listas e também vamos ver outros tipos de coleções como mapas.

2.5 ESTRUTURAS DE DECISÃO

Utilizamos uma estrutura de decisão em nosso programa quando queremos que determinado pedaço de código seja executado somente quando uma condição for satisfeita. Imagine um código de login, que deve receber um usuário e uma senha e verificar se esses dados estão corretos. Somente com usuário e senha corretos o sistema deve ser liberado. Para isso, utilizamos uma estrutura de decisão, nesse caso, uma expressão `if` cairia bem.

A expressão `if` é uma estrutura de decisão, ou seja, conseguimos mudar o fluxo do programa de acordo com o resultado de um `if`. Vamos voltar ao exemplo do login e supor que se a senha for igual a `123` o programa exibe uma mensagem de "Acesso concedido", caso contrário, o programa exibe uma mensagem de "Senha incorreta". O código seria assim:

```
fun main(args: Array<String>) {  
  
    val senha = "123"  
  
    if( senha == "123"){  
        println("Acesso concedido")  
    }else{  
        println("Senha incorreta")  
    }  
}
```

Um `if` não precisa ter necessariamente um `else`, ele pode ter somente a condição `if`. Veja o seguinte exemplo em que a estrutura verifica se um valor é maior que outro:

```
val a = 10
val b = 5

if( a > b) {
    println("$a é maior que $b")
}
```

O código executa o `print` caso a variável `a` seja maior que a variável `b`, caso contrário, não faz nada. Podemos ainda utilizar um `else` para indicar o que fazer caso a condição seja falsa, veja:

```
val a = 10
val b = 5

if( a > b) {
    println("$a é maior que $b")
} else{
    println("$a é menor que $b")
}
```

Outra forma interessante de se fazer um `if` em Kotlin é criar toda a estrutura em uma única linha. Em alguns casos isso pode fazer sentido e economizar algumas linhas de código. Vamos supor que no exemplo anterior eu queira saber qual variável é maior, `a` ou `b`. Nesse caso, eu poderia utilizar a mesma estrutura anterior e simplesmente armazenar o resultado em outra variável, mas eu também poderia resumir o código da seguinte maneira:

```
val a = 10
val b = 5

val maior = if( a > b) a else b
```

Nesse caso, o `if` foi usado como uma expressão e não como um simples controle de fluxo, mas veja como o código ficou claro, ele pode ser lido da seguinte forma: "Se `a` for maior que `b`, então `maior` recebe `a`, caso contrário, `maior` recebe `b`"

Outra estrutura legal do Kotlin é a estrutura `when`. Usamos o `when` quando precisamos fazer várias verificações em seguida. É claro que isso poderia ser feito com vários `if`s um atrás do outro, mas a estrutura `when` é própria para esses casos. Em um comparativo com Java, o `when` seria o substituto do `switch`, veja:

```
when (x) {  
    1 -> print("x == 1")  
    2 -> print("x == 2")  
    else -> {  
        print("x possui outro valor")  
    }  
}
```

Essa é a forma mais simples e clara de se usar o `when`. Perceba que com essa estrutura temos 2 verificações e ainda um `else` caso nenhuma das duas seja a verdadeira.

Ainda temos mais algumas opções interessantes no uso do `when`. Se eu quisesse, por exemplo, testar se a variável `x` tem valor 1 ou 2, poderia fazer assim:

```
when (x) {  
    0, 1 -> print("x == 0 ou x == 1")  
    else -> print("x tem outro valor")  
}
```

Poderia também verificar se `x` está dentro de um intervalo de valores. Por exemplo, gostaria de verificar se `x` está entre 1 e 10:

```
when (x) {  
    in 1..10 -> print("x está no intervalo")  
    else -> print("x está fora do intervalo")  
}
```

2.6 ESTRUTURAS DE REPETIÇÃO

Uma estrutura de repetição é utilizada quando queremos repetir determinado trecho de código. Imagine sua lista de contatos do WhatsApp, ela provavelmente está armazenada em um banco de dados e, quando você abre o aplicativo, a lista é montada na tela para você ter acesso a seus contatos. Este é um bom exemplo de uma estrutura de repetição em ação. Imagine que, para montar um contato na tela, o programador montou um bloco de código, mas esse código precisa ser executado para todos os contatos da lista. Então esse código é colocado dentro de uma estrutura de repetição que itera por todos os contatos da lista e monta cada um na tela!

Veremos duas estruturas de repetição em Kotlin. A linguagem possui mais, mas estas são as mais utilizadas. São elas o `for` e o `while`. Vamos ver primeiro o `for`.

O `for` é amplamente utilizado para iterar listas, sendo adequado quando sabemos o número de interações que o programa precisa fazer. Veja um exemplo:

```
val lista = listOf(1, 2, 3, 4)  
  
for(i in lista) {  
    println("Item: $i")  
}
```

Nesse exemplo, o `for` itera uma lista com valores inteiros, e esse `for` é executado em todos os itens da lista, desde o primeiro

até o último. A cada interação, ele modifica a variável `i` com o valor daquela posição da lista.

Em alguns casos, além do valor, precisamos do índice em que aquele valor está na lista. Para esses casos poderíamos usar o seguinte código:

```
val lista = listOf(1,2,3,4)

for((indice ,valor) in lista.withIndex()){
    println("índice: $indice  valor: $valor")
}
```

Outra estrutura de repetição disponível na linguagem é o `while`, cujo funcionamento é diferente do `for`, pois repete um trecho de código enquanto uma condição for verdadeira. Por exemplo:

```
var x = 0
while (x < 10) {
    println(x.toString())
    x++
}
```

O código repete o `print` enquanto a variável `x` tiver um valor menor que 10.

2.7 FUNÇÕES

Podemos definir uma função como um conjunto de comandos agrupados em um bloco, que recebe um nome e, através deste nome, pode ser chamado em outras partes do código. Na prática utilizamos funções para separar melhor nossa lógica.

Imagine um aplicativo igual ao Facebook. Pense na ação de *like*

do Facebook. Certamente existem algumas linhas de código que fazem esta ação, no entanto, a ação de *like* está espalhada em diversas partes do Facebook. Imagine ter que repetir as mesmas linhas de código em todos os lugares onde é possível dar um *like*. Isso dificultaria muito a programação e principalmente a manutenção desse código. Por isso, certamente existe uma função para isso no sistema do Facebook. Em todos os lugares em que se utiliza essa ação é só chamar a função. O melhor disso é que, se acontecer alguma mudança na lógica da função, todos os lugares que a utilizam já estariam atualizados pois só houve mudanças na função.

Para definir funções em Kotlin utilizamos a palavra `fun`, seguida do nome da função, seus argumentos e seu retorno. Vamos a um exemplo prático, vou definir uma função com nome `somar` que recebe 2 argumentos inteiros e retorna a soma:

```
fun somar(n1: Int , n2: Int): Int{  
    return n1 + n2  
}
```

E para chamar a função:

```
val resultado = somar(5 , 7)
```

Esse exemplo é de uma função que possui um retorno, por isso eu defini qual é este retorno e utilizei a palavra `return` para devolver o resultado.

Nem todas as funções possuem um retorno. No exemplo, a função `soma` recebe dois números e retorna a sua soma, então faz sentido nesse caso retornar um valor. No entanto, há casos em que não é necessário retornar nenhum valor. Para esses casos

definimos que o retorno da função é `Unit` e não precisamos utilizar a palavra `return`.

```
fun imprimir(texto: String): Unit{  
    println(texto)  
}
```

Perceba que essa função tem uma anatomia diferente da anterior: ela não possui a palavra `return`, pois ela não retorna nada, apenas exibe um texto no console. Utilizei a palavra `Unit` para indicar que essa função não possui retorno. No entanto, quando uma função é do tipo `Unit`, eu posso omitir a palavra `Unit` e o Kotlin vai entender sozinho que essa função não possui retorno. Então essa mesma função poderia ser reescrita da seguinte forma:

```
fun imprimir(texto: String){  
    println(texto)  
}
```

O código completo ficaria assim:

```
fun main(args: Array<String>){  
  
    val n1 = 5  
    val n2 = 7  
  
    val resultado = somar(n1, n2)  
  
    imprimir("A soma de $n1 + $n2 = $resultado")  
}  
  
fun somar(n1: Int, n2: Int): Int{  
    return n1 + n2  
}  
  
fun imprimir(texto: String){  
    println(texto)  
}
```

Execute esse código no Try Kotlin e veja o resultado no console.

Um recurso bem interessante da linguagem são as **Single-Expression functions** (Funções de expressão única). Esse recurso simplifica a definição de uma função quando ela possui apenas uma linha, não sendo necessário o uso das chaves ({ }). As mesmas funções somar e imprimir poderiam ser escritas da seguinte maneira:

```
fun somar(n1: Int , n2: Int) = n1 + n2  
fun imprimir(texto: String) = println(texto)
```

Perceba que, no caso da função somar , eu não precisei utilizar a palavra return e nem definir o tipo de retorno da função, essas informações o próprio compilador descobre sozinho porque ambas são *Single-Expression functions*.

Modifique o código anterior no Try Kotlin e veja que, apesar de a sintaxe estar menor, o resultado continua o mesmo.

2.8 ORIENTAÇÃO A OBJETOS

O Kotlin é uma linguagem que trabalha com o paradigma de Orientação a Objetos (OO) além do Paradigma Funcional. No contexto de aplicativos, é fundamental entender o básico de Orientação a Objetos e como trabalhar orientado a objetos com Kotlin.

Podemos definir como Programação Orientada a Objetos um modelo baseado em objetos. Um objeto é uma abstração de algo que pode ser do mundo real ou não. Assim como no mundo real,

pode ter características e pode executar ações. Vamos imaginar um objeto do mundo real, um carro. Ele tem diversas características como modelo, cor, rodas, motor, bancos etc. Um carro também pode executar algumas ações como acelerar, frear, virar à direita, virar à esquerda etc.

Em programação, chamamos essas características de **propriedades** e suas ações de **métodos**. Então um objeto pode ter propriedades, que são características que o definem, e métodos, que são ações que ele pode executar.

Para se criar um objeto em Kotlin devemos criar uma **classe**. Uma classe é onde colocaremos a programação do objeto. Programaremos suas propriedades e seus métodos e, através da classe, podemos criar instâncias deste objeto. Pense na classe como um molde para criação do objeto, de modo que na programação eu posso ter uma classe e, desta classe, criar N objetos.

Para se criar uma classe em Kotlin utilizamos a palavra `class` seguida pelo nome da classe, veja um exemplo:

```
class Carro{  
}
```

Toda a programação da classe ficará entre os parênteses (`{ }`) que indicam seu início e fim. Para definir suas propriedades, podemos criar variáveis em seu escopo:

```
class Carro{  
    var cor: String = ""  
    var modelo:String = ""  
}
```

Desta forma, temos a classe `carro` com 2 propriedades, `cor`

e modelo , ambas do tipo String . Para definir métodos em uma classe, basta criar funções nela, veja:

```
class Carro {  
  
    var cor: String = ""  
    var modelo:String = ""  
  
    fun acelerar(){  
        println("Acelerando")  
    }  
  
    fun frear(){  
        println("freando")  
    }  
}
```

Desta forma, implementamos 2 métodos na classe, o acelerar e o frear . Perceba que a Programação Orientada a Objetos simplesmente é uma abstração, nesse caso, uma abstração de um carro. A grande vantagem dessa forma de programar é que, agora, sempre que eu quiser utilizar este objeto, basta criar uma instância dele: val c = Carro() , então através da variável c eu tenho acesso às propriedades e métodos:

```
val c = Carro()  
c.cor = "Azul"  
c.modelo = "Nissan 350z"  
c.acelerar()
```

Herança

Uma das características mais marcantes da Orientação a Objetos é a capacidade de se fazer uma herança entre classes. Isso quer dizer que conseguimos criar subclasses de alguma outra classe. Por exemplo, imagine que precisaremos criar um modelo de carro específico com propriedades e métodos novos, porém

mantendo as propriedades e métodos de um carro comum.

Temos duas opções nesse caso, a primeira seria criar essa classe nova e repetir todo o código da classe `carro`. Essa opção não me parece muito legal, uma vez que uma das vantagens da Orientação a Objetos é a **reutilização** de código. A melhor opção seria utilizar um recurso de linguagens orientadas a objetos chamado **herança**.

O conceito de herança em OO é bem parecido com o conceito de herança no mundo real. Assim como uma pessoa pode herdar características genéticas de seus pais, em programação, uma classe pode herdar características e métodos de outra classe!

Para esse caso, seria a solução perfeita. Poderíamos então criar uma nova classe `CarroEspecial` e fazer uma **herança** da classe `Carro` e somente implementar as propriedades e métodos novos.

Para se fazer a herança de uma classe, devemos colocar : na frente do nome da classe e, em seguida, instanciar a classe pai:

```
class CarroEspecial : Carro(){  
    fun fazerDrift(){  
        //implementação  
    }  
}
```

Desta forma, a classe `CarroEspecial` herda automaticamente todas as propriedades e métodos da classe `Carro`, e implementa um método novo. Mas para isso funcionar, a classe `Carro` deve permitir a explicitamente que se faça herança dela.

Para isso, devemos utilizar a palavra `open` na definição da classe, assim: `open class Carro`. Veja como ficaria o código completo:

```
open class Carro {  
  
    var cor: String = ""  
    var modelo:String = ""  
  
    fun acelerar(){  
        println("Acelerando")  
    }  
  
    fun frear(){  
        println("freando")  
    }  
  
}  
  
class CarroEspecial : Carro(){  
  
    fun fazerDrift(){  
        //implementação  
    }  
}
```

Classe de dados (Data class)

Durante o desenvolvimento de software, é muito comum utilizar classes de dados. São classes que geralmente não possuem nenhum método, somente propriedades, e nos ajudam a transitar e organizar os dados em uma aplicação.

Imagine a modelagem de um usuário de um aplicativo. Vamos supor que este usuário terá um nome, um e-mail e também uma senha. Podemos modelar uma classe para isso e, como será uma classe somente para guardar essas informações, ela pode ser uma **data class**.

A grande vantagem das classes de dados em Kotlin é a sua simplificação de implementação. Podemos criar essa classe somente com uma linha de código! Veja:

```
data class Usuario(var nome:String, var email:String, var senha:String)
```

Isto é incrível! As data class nos ajudam muito a agilizar e simplificar o desenvolvimento de um aplicativo.

Existem muitas outras coisas para se falar de Orientação a Objetos, é um assunto bem amplo e há diversos livros na Casa do Código que tratam sobre isso. Aqui só vimos o básico para começar a desenvolver aplicativos.

2.9 RESUMINDO

Neste capítulo, passamos por todas as estruturas importantes da linguagem e aprendemos como fazer a maioria das coisas básicas. Vimos como trabalhar com variáveis e seus diferentes tipos, como trabalhar com estruturas de decisão e repetição, como trabalhar com listas e arrays. No próximo capítulo, vamos configurar nosso ambiente de desenvolvimento para começar a criar aplicativos para Android. Veremos como fazer a instalação do Android Studio, configuração do SDK e a criação de um emulador para testarmos nossos aplicativos. Vamos nessa!!

CAPÍTULO 3

CONFIGURANDO O AMBIENTE DE DESENVOLVIMENTO

Para começar a criar nossos projetos, precisamos configurar nosso ambiente de desenvolvimento. Para isso, vamos instalar o Android Studio, a IDE oficial para desenvolvimento Android. Junto ao Android Studio, está o Android SDK, o emulador e tudo mais que for necessário para o desenvolvimento.

O QUE É O ANDROID SDK?

Podemos dizer que o Android SDK é nosso kit de desenvolvimento. A sigla SDK significa *Software Development Kit*. O SDK do Android vem com todas as ferramentas, APIs e bibliotecas necessárias para se trabalhar com a plataforma Android. Junto ao SDK, também estão os emuladores que você utilizará para testar seus aplicativos. Não se preocupe em instalá-lo, pois ele vem junto com o Android Studio.

3.1 ANDROID STUDIO

O Android Studio será nossa IDE de desenvolvimento. Vamos usá-lo porque basicamente ele é a IDE oficial do Android, é muito bom e possui diversos recursos que agilizam o desenvolvimento. Seu editor de código é muito bom, e você não precisa de nenhuma outra ferramenta adicional, isto é, consegue fazer tudo somente com o Android Studio!

Vamos lá. Para baixar o Android Studio, você deve entrar em <https://developer.android.com/studio/index.html>. Lá você verá uma tela parecida com esta:



Figura 3.1: Home Android Studio

O site é todo em português, o que facilita bastante a busca por informações se você não domina tão bem o inglês. Repare que existe um botão grande escrito DOWNLOAD ANDROID STUDIO ; é aí que você deve clicar para baixar o programa.

O site automaticamente vai identificar seu sistema operacional

e encaminhar para o download correto. Você pode desenvolver para Android utilizando um computador com Windows, Mac ou até mesmo Linux!

Para Windows, os requisitos do sistema são:

- Microsoft® Windows® 7/8/10 (32 ou 64 bits);
- Mínimo de 3 GB de RAM, 8 GB de RAM recomendados, mais 1 GB para o Android Emulator;
- Mínimo de 2 GB de espaço livre em disco, 4 GB recomendados (500 MB para o IDE + 1,5 GB para o Android SDK e as imagens do sistema do emulador);
- Resolução de tela mínima de 1.280 x 800;
- Para o emulador acelerado: sistema operacional de 64 bits, processador Intel® compatível com Intel® VT-x, Intel® EM64T (Intel® 64) e a funcionalidade Execute Disable (XD) Bit.

Já para Mac:

- Mac® OS X® 10.10 (Yosemite) ou posterior, até a versão 10.12 (macOS Sierra);
- Mínimo de 3 GB de RAM, 8 GB de RAM recomendados, mais 1 GB para o Android Emulator;
- Mínimo de 2 GB de espaço livre em disco;
- 4 GB recomendados (500 MB para o IDE + 1,5 GB para o Android SDK e as imagens do sistema do emulador);
- Resolução de tela mínima de 1.280 x 800.

Por último, para Linux, precisaremos de:

- Área de trabalho GNOME ou KDE;

- Testado no Ubuntu® 12.04, Precise Pangolin;
- Distribuição de 64 bits, capazes de executar aplicativos de 32 bits;
- Biblioteca C do GNU (glibc) 2.19 ou posterior;
- Mínimo de 3 GB de RAM, 8 GB de RAM recomendados, mais 1 GB para o Android Emulator;
- Mínimo de 2 GB de espaço livre em disco;
- 4 GB recomendados (500 MB para o IDE + 1,5 GB para o Android SDK e as imagens do sistema do emulador);
- Resolução de tela mínima de 1.280 x 800;
- Para o emulador acelerado: processador Intel® compatível com Intel® VT-x, Intel® EM64T (Intel® 64) e a funcionalidade Execute Disable (XD) Bit, ou processador AMD compatível com AMD Virtualization™ (AMD-V™).

Estes são os requisitos mínimos que você precisa para rodar o Android Studio. Entretanto, quero compartilhar também as minhas recomendações pessoais baseadas na experiência que tive ao utilizar o Android Studio.

Primeiramente, 4 GB de memória RAM é pouco! É possível rodar com 4 GB, mas provavelmente você ficará frustrado com a performance geral, então, recomendo 8 GB para rodar bem. Se você somente dispõe de um computador com 4 GB de memória, recomendo não utilizar o emulador do Android Studio, e sim testar os aplicativos no próprio aparelho.

Uma das coisas que consome uma boa fatia de memória é o emulador. Isso porque é um sistema operacional Android completo rodando em seu computador. A boa notícia é que você pode facilmente rodar seus Apps diretamente em um aparelho

Android!

Em questões de processamento, um Intel core i5, ou um equivalente, dá conta do recado. Particularmente, utilizo o Android Studio em dois computadores diferentes, o meu pessoal e o do trabalho. O pessoal é um Macbook Pro com 10 GB de RAM, core i5 e SSD de 240 GB, e o do trabalho é um Windows 8, core i7 com 16 GB de RAM e HD de 1 TB, e ambos rodam muito bem o Android Studio.

Agora com o Android Studio baixado, vamos partir para sua instalação. Não há muito segredo, é basicamente next-next-finish . Durante a instalação, o Android Studio vai configurar o SDK e provavelmente baixar algumas atualizações. Isso pode demorar mais alguns minutos. Ao término, você verá a seguinte tela:

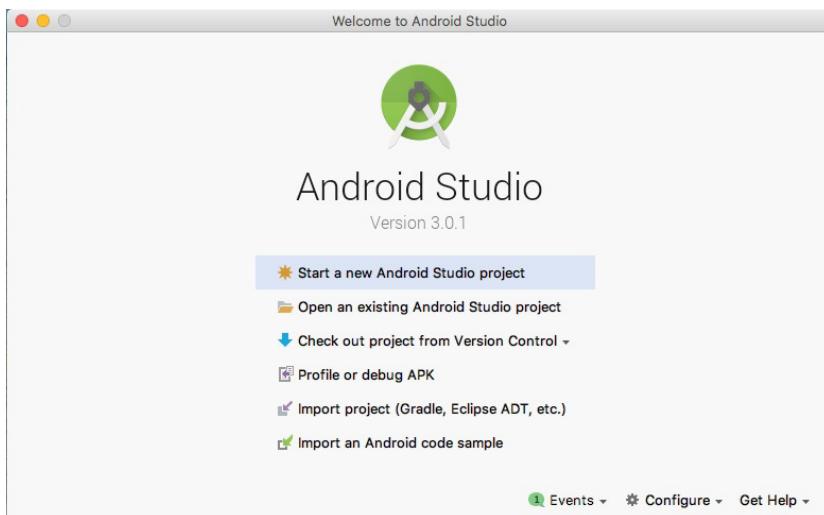


Figura 3.2: Tela inicial

Esta é a tela inicial do Android Studio. A partir dela, você poderá criar um projeto novo, abrir um existente, fazer um *check out* direto de algum controlador de versão, debugar um APK, importar um projeto Gradle ou importar um exemplo de código. Existe um pequeno botão **configure** com as seguintes opções:

Opção	Descrição
SDK Manager	Gerenciador do SDK, no qual conseguimos atualizar e instalar novas plataformas e ferramentas.
Preferences	Configurações gerais da IDE, como tamanho de fonte, tema da IDE, controle de versões, formas de compilação etc.
Plugins	Gerenciador de plugins, no qual podemos instalar novos ou atualizar existentes.
Import Settings	Aqui, você pode importar configurações de outra instalação do Android Studio.
Export Settings	É possível exportar as configurações do Android Studio.
Setting Repository	Esta opção habilita a entrada de uma URL para a configuração de um repositório externo.
Check for update	Verifica se há atualizações.
Project Defaults	Esta opção abre links para configurações padrão da IDE.

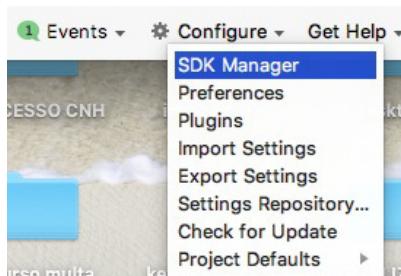


Figura 3.3: Configuração

3.2 CONFIGURANDO O SDK

Um pré-requisito para que tudo funcione corretamente é a configuração correta do SDK. O Android Studio já instala a última versão disponível durante a instalação do programa, então, provavelmente já está tudo configurado corretamente para começarmos a programar.

No entanto, é importante checar se a instalação está correta. Também é importante conhecer onde se atualiza o SDK. Abra a opção **SDK Manager** pelo link **Configure**, visto anteriormente. Isto abrirá a seguinte janela:

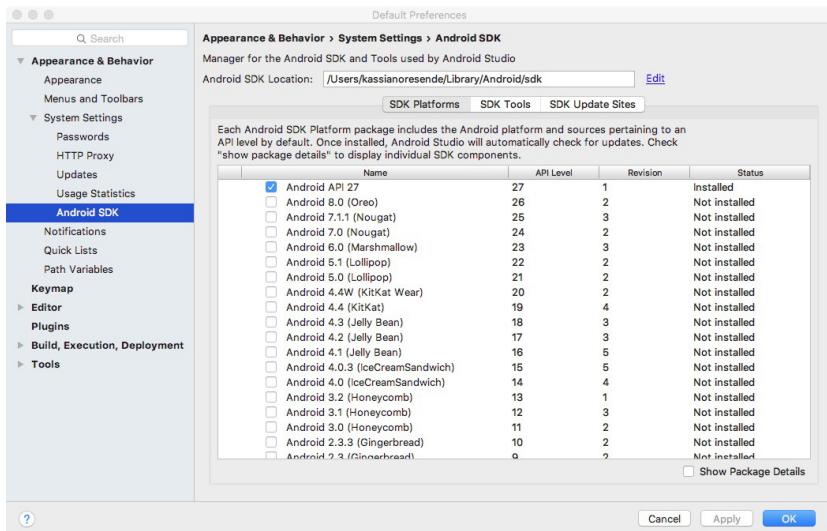


Figura 3.4: SDK Manager

Esta é dividida por três abas: **SDK Platforms**, **SDK Tools** e **SDK Update Sites**. Em **SDK Platform**, podemos escolher as plataformas Android com qual trabalharemos. Por padrão, já vem

instalada a plataforma mais recente, neste caso, a API 27 – correspondente ao Android Oreo 8.1.0.

Para cada atualização do sistema operacional, é lançada uma nova versão da API também, sendo que cada versão tem uma enumeração. Assim, o Android 8.1.0 é a API 27; a 26, o Android 8.0; a 25, o Android Nougat 7.1.1 e assim por diante. Você pode conferir todas as versões de API diretamente em: <https://developer.android.com/guide/topics/manifest/uses-sdk-element.html?hl=pt-br>.

Caso você queira mudar a pasta de instalação do SDK, basta clicar no link `Edit` ao lado do campo `Android SDK Location` e escolher uma nova.

3.3 PRIMEIRO PROJETO: HELLO WORLD

Agora que temos nosso ambiente todo configurado, é hora de criar nosso primeiro projeto! Vamos fazer um clássico Hello World. Nele, não utilizaremos praticamente nenhum código em Kotlin. A intenção é que você conheça um pouco da IDE e também configure um emulador para testar os futuros aplicativos.

Então, vamos lá. Na tela inicial, clique em `Start a new Android Studio Project`. A primeira tela na qual você precisa preencher algumas informações sobre o projeto será:

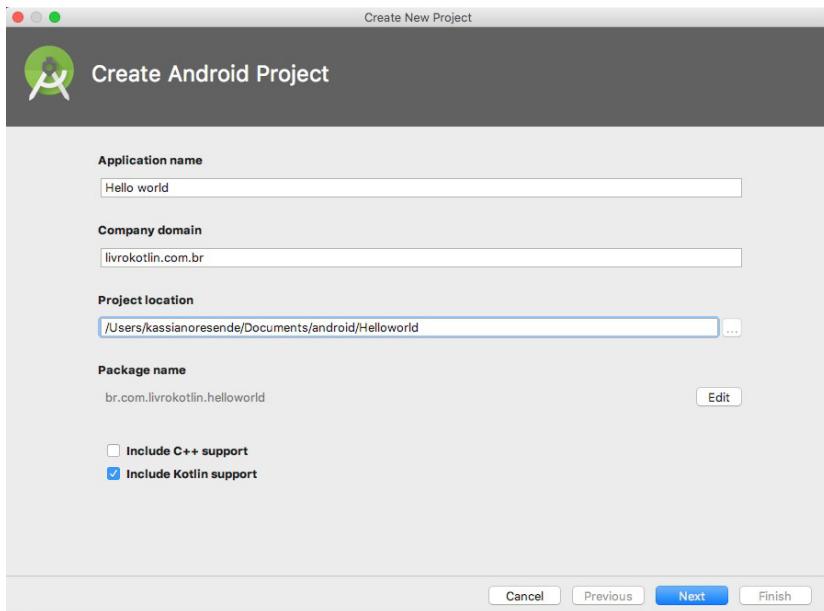


Figura 3.5: Configurações do projeto

Em `Application Name`, deixei como `Hello world`, já que este é o nome do projeto. Em `Company Domain`, coloquei `livrokotlin.com.br`, mas, em um projeto real, você deve colocar o domínio da empresa, pois isso será usado posteriormente pela IDE para a criação do nome de pacote da aplicação.

Em `Project Location`, você deve indicar a pasta para salvar o projeto. No meu caso, configurei uma pasta chamada `android` dentro de `Documentos`, deixando todos os projetos lá.

Por fim, deixe marcada a opção `Include Kotlin support` e clique em `Next`. A próxima tela é a de `Target Android Devices`:

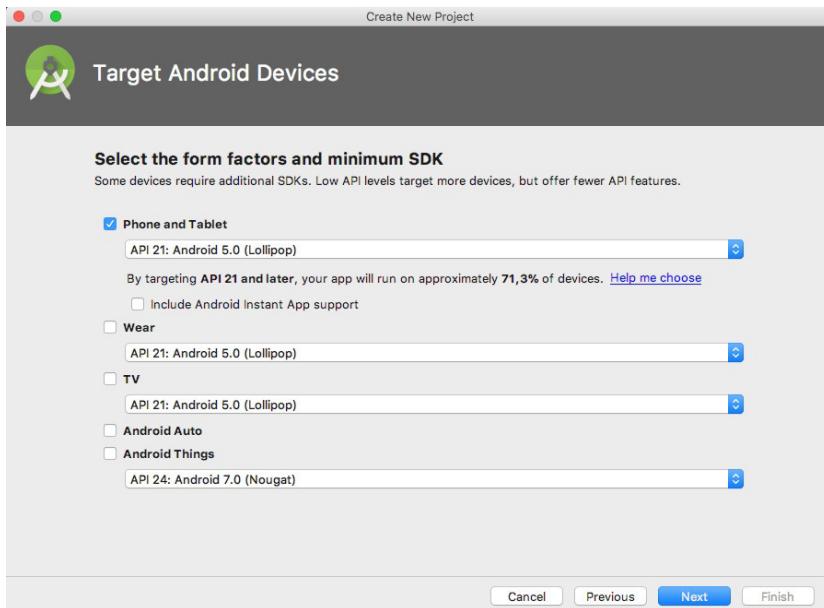


Figura 3.6: Target devices

Nesta tela, você deve configurar quais dispositivos seu App suportará. No nosso caso, deixarei selecionado somente a opção Phone and Tablet e a API 21 : Android 5.0 . Isso significa que nosso App funcionará em telefones e tablets que tenham Android a partir da versão 5.0!

Não há uma regra para essa escolha, mas a versão mais nova do Android é atualmente a 8.1. Então, se eu escolher a versão 8.1, meu aplicativo funcionará em pouquíssimos aparelhos, porque as empresas (Motorola, LG, Samsung etc.) costumam demorar um certo tempo para lançar as atualizações de sistema. Portanto, é uma boa ideia deixar um suporte a algumas versões anteriores, garantindo que nosso App vai funcionar em uma gama extensa de aparelhos.

Não se preocupe que, se depois você precisar mudar essa informação e passar a suportar uma versão mais antiga, é totalmente possível. Então, clique em `Next`. Na próxima tela, podemos escolher um modelo de Activity:

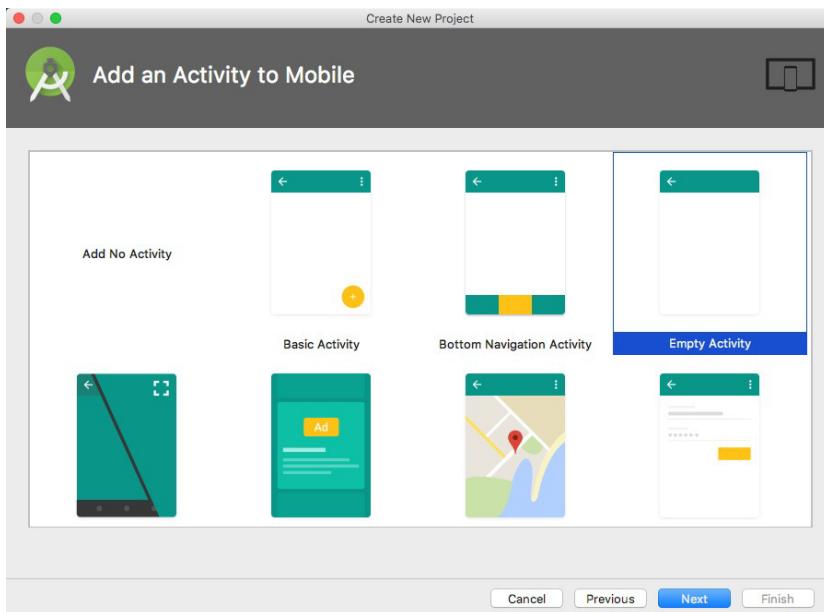


Figura 3.7: Add Activity

Aqui podemos escolher algum modelo de tela inicial. O Android Studio possui vários que são muito úteis no dia a dia, como modelos com mapa, telas de login, barra de navegação etc.

Em Android, uma Activity significa basicamente uma tela; é o ponto central no qual o usuário interage com o App. Dentro de uma Activity, podemos colocar botões, imagens, menus etc., assim, qualquer objeto de interação com o usuário estará de alguma forma dentro da Activity.

Para nosso projeto, escolherei o modelo `Empty Activity`, porque essa opção criará toda a estrutura básica de uma tela e não precisaremos nos preocupar com alguns detalhes de implementação. Depois, selecione `Next`.

Na próxima tela, você deve configurar o nome dessa Activity:

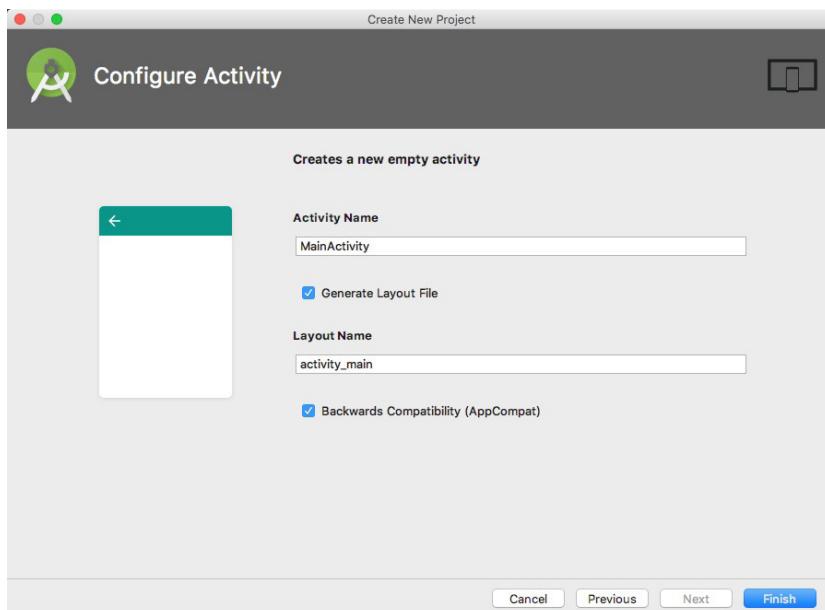


Figura 3.8: Activity name

Deixarei o nome padrão, `MainActivity`, e, em Layout, ficaremos com `activity_main`. Aqui são dois nomes diferentes pois essa tela utilizará um arquivo XML para o layout e um arquivo em Kotlin para o código.

Por convenção, os arquivos de layout têm nomenclatura toda minúscula e, quando é um nome composto, este é separado por `_` (*underline*), também por convenção os arquivos relacionados a

Activities têm o prefixo "activity".

Já os arquivos de código, Kotlin ou Java, a convenção é ter um sufixo "Activity" para arquivos relacionados a uma Activity e todas as palavras são iniciadas com letras maiúsculas. É convenção também que, quando é um nome composto, este não é separado por nenhum caractere.

Clique em Next . Como é o primeiro projeto, é possível que demore um pouco até abrir a tela, isso é normal pois o Android Studio está baixando alguns pacotes de compilação. Após o termino, você verá a seguinte tela:

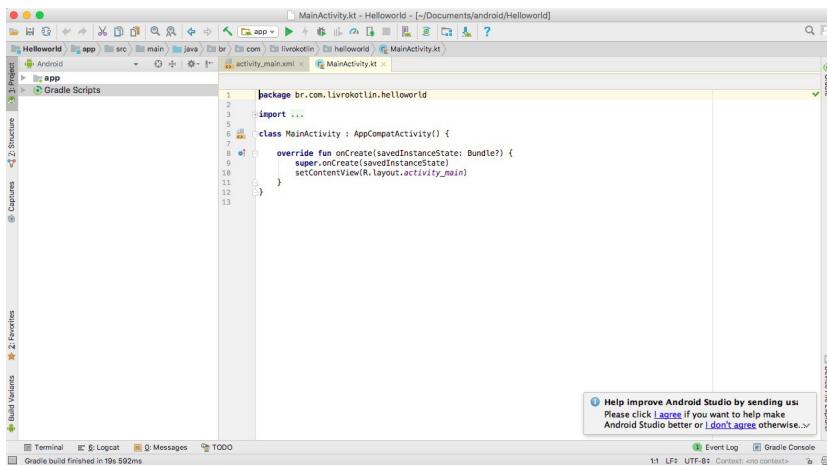


Figura 3.9: IDE do Android Studio

Está é a IDE do Android Studio! Este será nosso ambiente de desenvolvimento e você precisará se familiarizar com ele.

A tela está dividida basicamente em duas partes, do lado direito está aberto o código em Kotlin e, do lado esquerdo, a navegação de arquivos. Essa forma de separação é bem característica da

JetBrains, que é a empresa criadora do Android Studio. Se você já utilizou alguma outra plataforma deles como PhpStorm, PyCharm ou IntelliJ você se familiarizará facilmente pois todas utilizam o mesmo padrão.

Não se preocupe com o código por enquanto, vou explicá-lo detalhadamente mais adiante. Vamos focar por enquanto no lado esquerdo da tela. Repare que existe uma pasta chamada `app` e uma chamada `Gradle Scripts`. Vamos começar pela `Gradle Scripts`, clique duas vezes nela para mostrar os arquivos que ela contém:

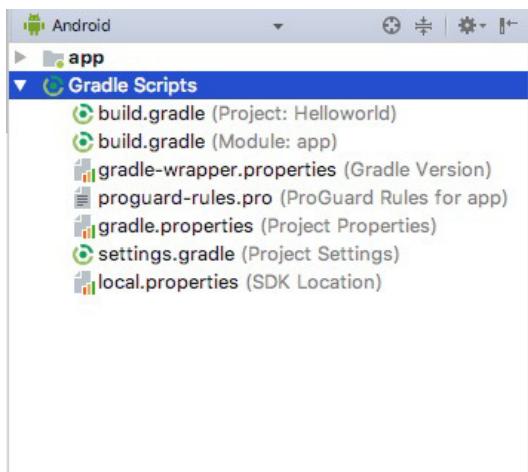


Figura 3.10: Gradle Scripts

O Gradle é uma ferramenta que automatiza a compilação de um software. O Android Studio utiliza o Gradle para compilação e também como gerenciador de dependências. No projeto, encontramos o arquivo `build.gradle` padrão para todos os projetos, e o arquivo `app/build.gradle`, com as configurações

referentes ao módulo.

Com o Android Studio na visualização Android, você diferenciará um arquivo do outro pela indicação que terá ao lado do nome de cada um entre parênteses. O arquivo referente ao projeto terá a indicação (Project :) e o arquivo referente ao módulo terá a indicação (Module :). Caso o Android Studio esteja na visualização Project você encontrará o build.gradle do projeto na pasta raiz e o build.gradle do módulo na pasta app .

Você pode conferir em qual tipo de visualização seu Android Studio está na parte superior da tela como mostra na imagem:

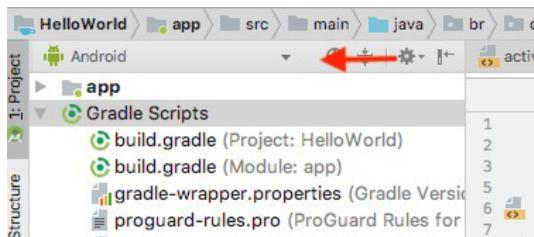


Figura 3.11: Visualização Android

Neste caso, estamos na visualização Android que na minha opinião é a forma mais adequada de se trabalhar com um projeto Android, mas isso pode ser alterado e afetará diretamente como a estrutura de pastas e arquivos é exibida. Você pode mudar o modo de visualização clicando no DropDown :

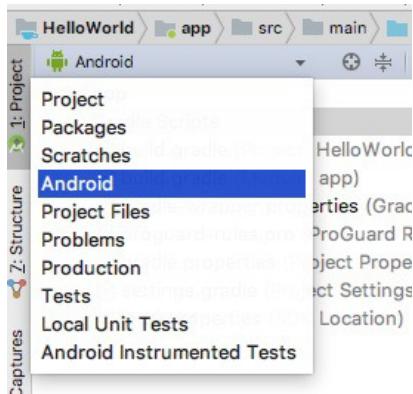


Figura 3.12: Alterar modo de visualização

Dentre todas as opções de visualização que você pode utilizar, a mais comum é `Android`, porém não é difícil você ver pessoas trabalhando na visualização `Project`. Neste caso os arquivos `build.gradle` do modulo e `build.gradle` do projeto estarão respectivamente na pasta `app` e na pasta raiz.

Veja como fica a estrutura de pastas em modo de visualização `Project`:

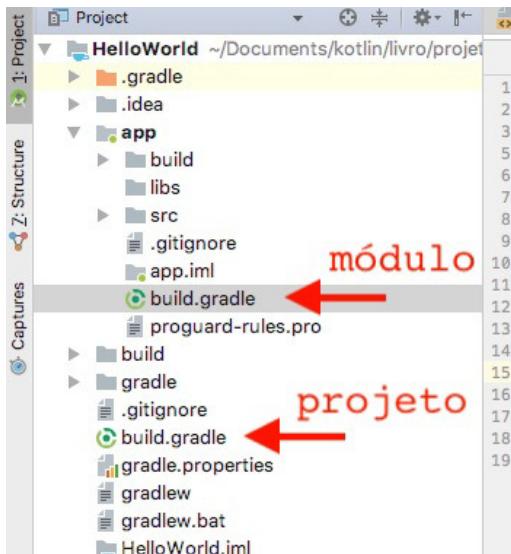


Figura 3.13: Alterar modo de visualização

O arquivo `build.gradle` do projeto contém configurações relacionadas ao projeto todo, como as configurações de repositórios de dependências, versão do plugin do Gradle e versão do Kotlin. Na prática, dificilmente mexemos nessas configurações. Segue o arquivo comentado:

```
buildscript {  
    //versão do Kotlin  
    ext.kotlin_version = '1.2.10'  
  
    repositories {  
        //repositórios de dependências  
        google()  
        jcenter()  
    }  
    dependencies {  
  
        //Versão do plugin do gradle  
        classpath 'com.android.tools.build:gradle:3.0.1'
```

```
        //Configuração do Kotlin no projeto
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

Já o arquivo `build.gradle` do módulo contém configurações referentes ao aplicativo em si, como versão mínima do sistema, versão de compilação e também as bibliotecas necessárias. A seguir, o arquivo `build.gradle` do módulo comentado:

```
//Plugin do Android
apply plugin: 'com.android.application'

//Plugin da linguagem Kotlin
apply plugin: 'kotlin-android'

//Adiciona funcionalidades estendidas ao Kotlin
apply plugin: 'kotlin-android-extensions'

android {
    //versão de compilação do app
    compileSdkVersion 26
    defaultConfig {

        //applicationId é o nome do pacote do projeto, este deve
        ser único
        applicationId "br.com.livrokotlin.helloworld"

        //versão mínima suportada
        minSdkVersion 21

        //versão alvo do aplicativo
        targetSdkVersion 26

        //número da versão do código
        versionCode 1

        //nomenclatura da versão do código
        versionName "1.0"
```

```
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {

            //Configuração para minificação do código
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

//dependências do projeto
dependencies {

    //inclui a pasta libs na compilação
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    //Biblioteca do Kotlin
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"

    //Bibliotecas de compatibilidade do google
    implementation 'com.android.support:appcompat-v7:26.1.0'

    //Bibliotecas do constraint layout
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'

    //Bibliotecas de testes
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
        androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

Não vamos modificar nada neste arquivo também.

Vamos olhar agora a estrutura de arquivos em que vamos trabalhar. Voltando ao menu do lado esquerdo, abra a pasta app

para exibir a seguinte estrutura:

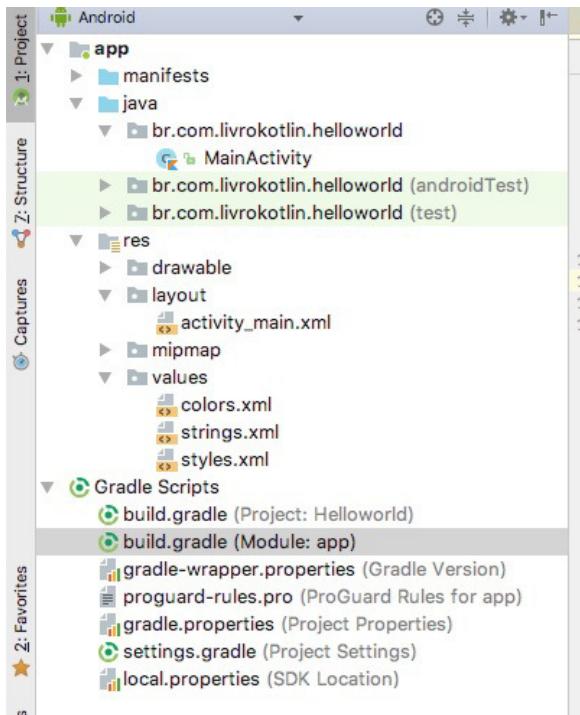


Figura 3.14: Arquivos Android

Observe que existe uma pasta `java`, e dentro dela uma pasta `br.com.livrokotlin.helloworld` e, por fim, um arquivo chamado `MainActivity`. É nessa pasta que criaremos todos os arquivos em Kotlin - é aqui que fica todo o código do nosso aplicativo! Observe também que logo a seguir existem duas pastas de testes. Não vamos utilizá-las por enquanto, mas o Android Studio já cria para nós uma estrutura para trabalhar com testes.

Um pouco mais abaixo existe uma pasta chamada `res`, que é uma abreviação de "resources" (recursos). É lá que ficam todos os

arquivos de layout, imagens que usaremos nos projetos, assim como arquivos de configurações. Ou seja, tudo o que não é código o Android entende como um recurso e este deve estar dentro da pasta `res`.

A seguir uma explicação de cada pasta dentro do `res`:

- `drawable` : é onde colocaremos todas as imagens que utilizaremos no nosso aplicativo.
- `layout` : é onde colocaremos todos os arquivos `xml` relacionados ao layout do App. Repare que já existe um arquivo chamado `activity_main.xml`, ele foi criado pelo Android Studio quando selecionamos um modelo de "Empty Activity". Este arquivo contém o código em `xml` responsável pela criação da tela do nosso app.
- `mipmap` : nesta pasta ficam os ícones do App.
- `values` : nesta pasta ficam os arquivos de `colors`, `strings` e `styles`.
 - `colors.xml` : neste arquivo podemos configurar cores para usar no nosso projeto. Imagine que você defina que a cor dos botões do seu App será padronizada, com o tom de azul `#4169E1`, então em vez de configurar esse hexadecimal em cada botão, você pode criar uma entrada no arquivo de cores e definir essa cor:
`<color name="azul_btn">#4169E1</color>`. Na hora de definir a cor no botão é só referenciar o nome da cor da seguinte forma: `@colors/azul_btn`.
 - `strings.xml` : neste arquivo podemos configurar textos que serão usados no nosso App. Imagine que você tenha uma mensagem padrão para quando

acontece algum erro e essa mensagem pode ser exibida em vários lugares. Em vez de fixar esse texto em todos os lugares, você pode definir uma entrada no arquivo de strings:

```
<string  
name="mensagem_erro">Ocorreu  
um  
erro</string>
```

e depois simplesmente referenciá-la pelo nome: @strings/mensagem_erro .

- styles.xml : neste arquivo você pode configurar temas para o App. Um tema pode agrupar várias configurações dentro dele como configurações de cor e dimensões. Assim como no arquivo de colors e strings, você pode definir um tema no arquivo de styles e depois referenciá-lo pelo nome.

3.4 CRIANDO UM EMULADOR ANDROID

Agora que você já teve uma visão macro do Android Studio, é hora de criar um emulador para rodarmos nosso aplicativo. O emulador virtualiza o sistema Android e dessa forma conseguimos testar o aplicativo sem a necessidade de ter um aparelho. Outra vantagem é que podemos criar vários emuladores virtualizando modelos de celulares diferentes, e também com sistemas de diferentes versões. Assim conseguimos testar como nosso App se comporta em diferentes versões do Android.

Para criar um emulador, entre no menu Tools > Android > AVD Manager ; isso abrirá a seguinte tela:



Figura 3.15: AVD Manager

Se tivesse algum emulador já criado, ele seria listado nesta tela, mas como estamos criando o primeiro, aparece somente um botão **Create Virtual Device**. Clique nele e a seguinte tela aparecerá:

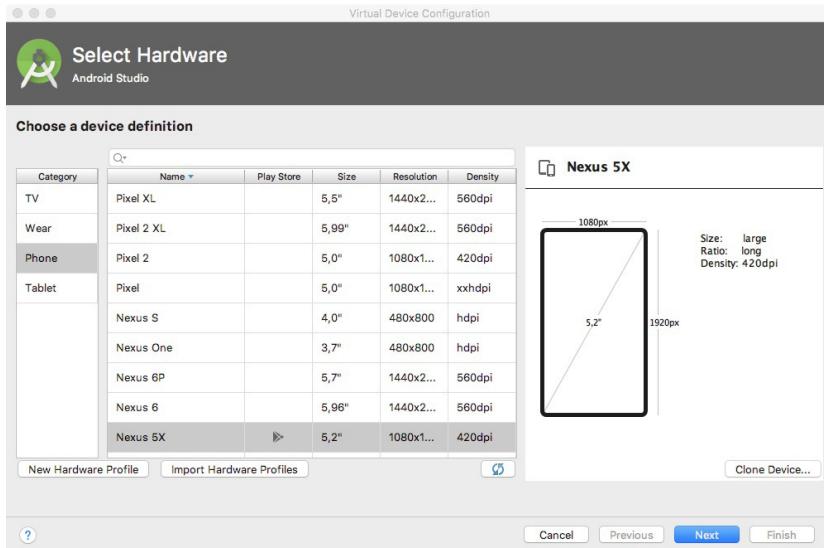


Figura 3.16: Selecionar dispositivo

Nessa tela, você pode escolher o modelo do dispositivo virtual que vamos criar. Escolheremos o modelo Nexus 5X porque ele faz parte da família Nexus, que é própria do Google, possui uma tela de alta resolução (1080 x 1920). Essas características o tornam um bom modelo para testar diferentes aplicativos. Então escolha o modelo Nexus 5X na categoria Phone e clique em Next .

Na próxima tela você deve escolher o sistema operacional do emulador:

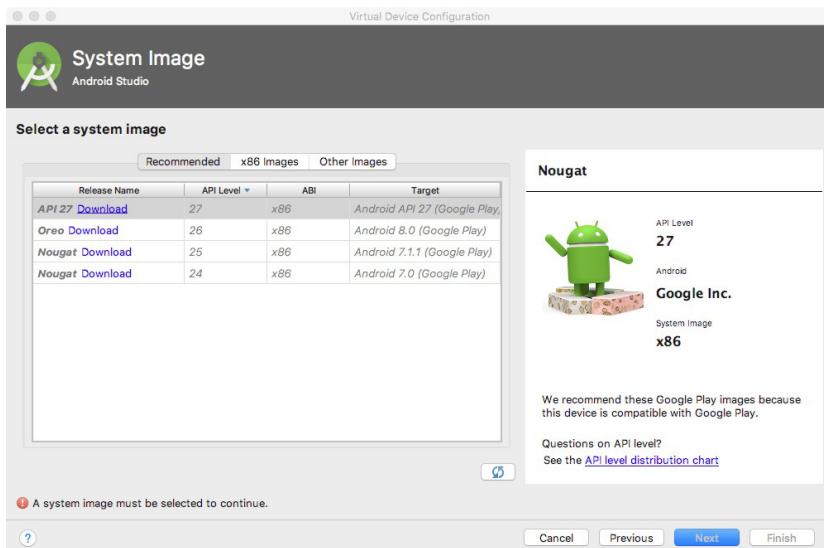


Figura 3.17: Selecionar dispositivo

Dentro das opções recomendadas, escolha a primeira opção, API 27 , referente ao Android Oreo 8.1, a versão mais recente do Android. Porém, antes de prosseguir precisamos fazer o download da imagem do sistema. Repare que, ao lado do nome da versão, existe um link Download , click nele para fazer o download da imagem que será usada no nosso emulador. Assim que terminar o download, escolha a versão Oreo e clique em Next .

Na próxima tela, você poderá mudar o nome de visualização do emulador, trocar o dispositivo ou sistema operacional, assim como escolher a posição do dispositivo entre retrato e paisagem. Não faremos nenhuma alteração nessa tela pois não é necessário.

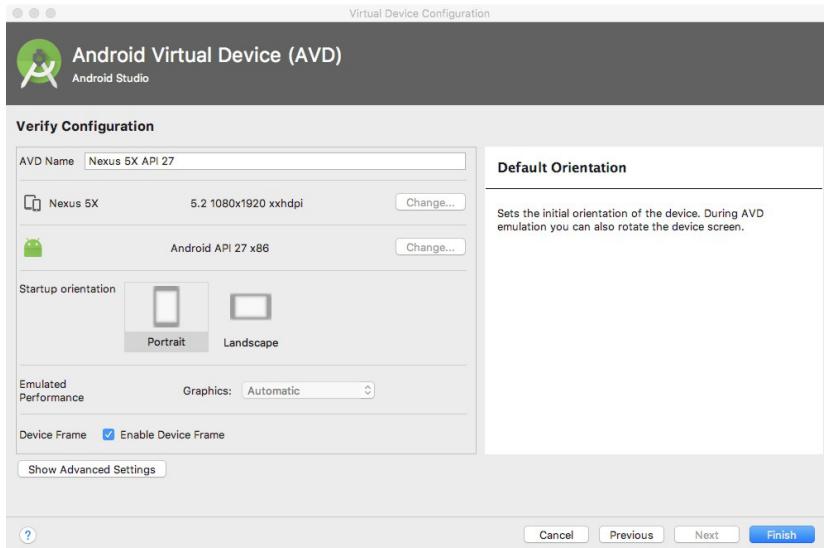


Figura 3.18: Finalizar

Ao clicar em **Finish**, o sistema criará o emulador com as configurações que escolhemos, o que pode demorar alguns segundos. Assim que terminar, você verá a tela com a lista de emuladores:



Figura 3.19: Lista de emuladores

Vamos dar o play no emulador para iniciá-lo. Para isso, clique no botão Play em verde na coluna Actions .



Figura 3.20: Emulador pronto

Se tudo ocorreu bem você verá o emulador com o sistema Android em funcionamento.

3.5 EXECUTANDO O PROJETO

Agora podemos executar nosso projeto para ver como fica no emulador, para isso localize o botão de play verde na parte superior da IDE:

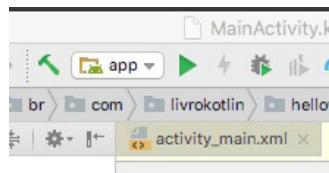


Figura 3.21: Botão Play

Ao "dar o play" no projeto será exibida uma tela para você selecionar onde gostaria que o aplicativo fosse executado:

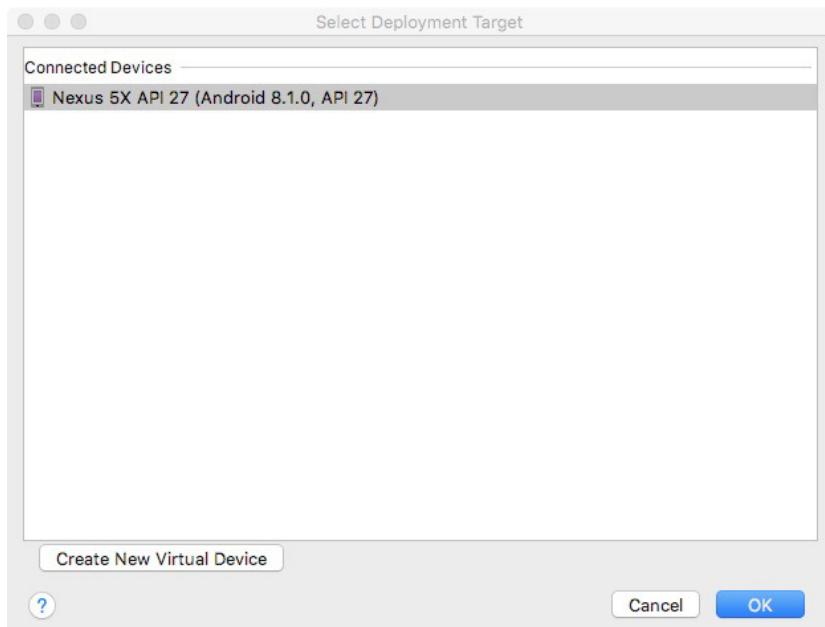


Figura 3.22: Selecionar emulador

Selecione o emulador que acabamos de criar e clique em **OK**.

Aguarde um pouco para que o emulador inicialize e a compilação e instalação da APK seja realizada. Ao término, você

verá a seguinte tela:



Figura 3.23: Hello World

Pronto! Nossa "Hello World" está em funcionamento!

3.6 HABILITANDO O AUTO IMPORT

Uma configuração que eu gosto de usar é habilitar a opção **Auto Import** (importação automática) do Android Studio. Essa

opção automaticamente verifica qual biblioteca você está utilizando no seu código e já faz a importação de maneira automática.

Para configurar o Auto Import , abra o menu Preferences > Editor > General > Auto Import .

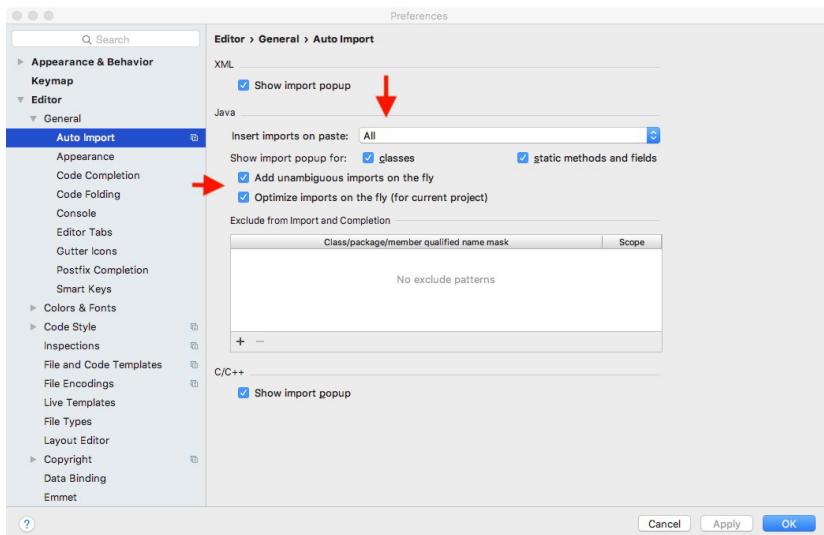


Figura 3.24: Auto Import

Nesta tela, selecione a opção All (Todos) em Insert imports on paste (Inserir importação ao colar). Essa configuração inserirá o import automaticamente caso você esteja colando um pedaço de código de outro lugar quer necessite de import.

Marque também as opções:

Add unambiguous imports on the fly (Adicionar importações não ambíguas em tempo real). Essa configuração

adiciona automaticamente os imports, quando necessário, em tempo real da escrita do código.

`Optimize imports on the fly` (Otimizar a importação em tempo real). Essa configuração remove os imports que não estão sendo usados na classe.

Essa configuração ajuda muito o processo de desenvolvimento, assim o programador quase não precisa se preocupar com a importação das bibliotecas. No entanto, às vezes a IDE não consegue localizar a biblioteca correta para importação, ou faz a importação de uma biblioteca equivocada que tenha nomes ambíguos e nesses casos o programador deve estar atento às bibliotecas que são importadas no projeto.

3.7 RESUMINDO

Neste capítulo, você aprendeu a instalar o Android Studio, a configurar o SDK e a criar um novo projeto. Você teve uma visão geral da IDE passando pelos principais pontos. Você aprendeu também a criar um emulador e a rodar o projeto! Vimos bastante coisa e tudo servirá de base para os próximos capítulos, em que vamos criar projetos reais utilizando a linguagem Kotlin!

CAPÍTULO 4

ANATOMIA DA PLATAFORMA ANDROID

Agora que você já conheceu um pouco do nosso ambiente de desenvolvimento, gostaria de mostrar para você um pouco de como é a anatomia do Android. Vamos ver questões fundamentais da plataforma e também aprender sobre o vocabulário utilizado nela.

A minha experiência como professor mostra que o maior erro dos alunos quando estão aprendendo a desenvolver aplicativos pra Android é não conhecer bem a plataforma antes de sair codificando. Esse é um erro cometido até por programadores experientes. Tenha em mente que programar pra Android é diferente de programar para Web, é diferente de programar para Desktop e é diferente de programar pra iPhone - a experiência de programar pra Android é única.

4.1 ACTIVITIES

Um dos componentes mais comuns no desenvolvimento Android se chama **Activity** (ou simplesmente **atividade**, em português). Podemos entender uma Activity como uma tela do nosso aplicativo, onde colocamos componentes visuais de

interação com o usuário – como botões, caixas de texto, caixas de seleção etc.

Normalmente, um aplicativo possui várias Activities, mas não há problema se ele possuir uma única, isso tudo vai depender da sua necessidade. Em um App, devemos eleger uma Activity como **principal**, geralmente chamada de `MainActivity` (Atividade principal). É ela que é apresentada ao usuário quando ele inicia o aplicativo.

Cada Activity pode iniciar outras Activities para executar tarefas diferentes. Imagine um aplicativo de mensagens, cuja Activity principal seja uma lista dos contatos que você tem na agenda. Ao clicar em cima de algum contato, essa Activity principal deve iniciar uma outra Activity, a da conversa. Quando isso acontece, a Activity que foi iniciada é empilhada sobre a principal.

Se por acaso fosse iniciada uma terceira Activity na Activity da conversa, esta ficaria empilhada sobre a Activity da conversa. Perceba que assim o Android vai criando uma **pilha de navegação** e a partir do momento em que você vai clicando nos botões de voltar , as Activities vão sendo desempilhadas e você vai voltando uma a uma.

A seguir uma imagem que ilustra todo esse fluxo de funcionamento:

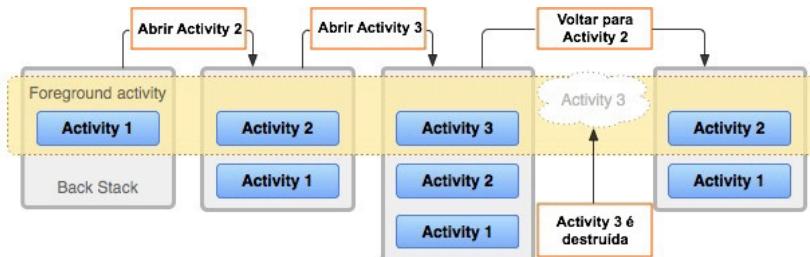


Figura 4.1: Pilha de navegação Activity

Um erro muito comum entre os desenvolvedores na hora de criar a ação de voltar é iniciar novas Activities em vez de simplesmente tirar a Activity atual da pilha. Imagine o mesmo exemplo do aplicativo de mensagens, em que o usuário clica em um contato e entra na conversa. Neste momento, sabemos que o sistema tem a Activity principal na pilha de navegação.

Mas suponha que na ação de "voltar" da tela de conversa, em vez de finalizar a Activity, o programador tirou da pilha de navegação, e usou um código que inicia a Activity principal novamente. Será que funcionaria? O que aconteceria?

Bom, funcionaria, o App abriria novamente a Activity principal e, para o usuário, o efeito seria o mesmo. O grande problema dessa forma, você já deve imaginar, é que agora teríamos 2 Activities na pilha de navegação! Isso é um problema porque a memória RAM do aparelho ficaria se enchendo de "lixo" – "lixo" porque a pilha de navegação só iria crescer.

Em um cenário como o exemplo que eu dei, talvez não seja tanto problema, pois são somente duas telas. Mas imagine um aplicativo com muitas telas e todas sendo feitas dessa forma, ai teríamos um problemão.

Criando uma Activity

Criar uma Activity é muito simples. Basta criar uma classe e herdar da classe `Activity` do Android. Lembra do conceito de herança utilizada em Programação Orientada a Objetos? É isso que usamos aqui.

Então, se eu criar uma classe qualquer e herdar da classe `Activity`, automaticamente essa minha classe passará a ter todos as características de uma Activity! Vamos ver um exemplo de código:

```
import android.app.Activity  
  
class MainActivity : Activity(){  
}
```

A declaração `import android.app.Activity` serve para nos dar acesso à classe base `Activity`. Fazendo uma analogia, quando começamos nosso código do zero, é como comprar um carro básico, sendo assim, se quisermos ar-condicionado ou uma roda diferente, devemos adicionar esses acessórios. A declaração `import` seria análoga à adição de acessórios ao nosso código. Você vai perceber que as declarações `import` serão comuns e pode haver várias em um mesmo código.

Simples, né? Desta forma já temos uma Activity, porém faltam algumas coisas para ela funcionar de fato. Uma delas é a implementação dos métodos de *callback* (chamada de retorno) do

sistema. Oi?

Apesar do nome complicado, o conceito é bem simples também! Os métodos de callback do sistema nada mais são que métodos da Activity que são acionados pelo sistema operacional quando acontece alguma alteração de estado. Vamos a um exemplo prático.

Imagine que você pega seu celular Android e abra sua rede social preferida. Neste momento, já sabemos que o Android vai procurar pela Activity principal deste aplicativo e vai abri-la, mas ele precisa avisar essa Activity de que ela está sendo aberta, algo como: "Ei, o usuário está abrindo o App, é melhor você fazer o que tiver que fazer aí para estar tudo pronto".

Esse aviso que o sistema manda é a chamada ao método de callback. No caso de inicialização de tela, o sistema chama o método `onCreate` (em criação). Este método é o que devemos implementar na Activity para ela conseguir receber esse retorno do sistema. Veja como ficaria o código:

```
import android.app.Activity  
import android.os.Bundle  
  
class MainActivity : Activity(){  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
    }  
}
```

O método `onCreate` é o único obrigatório, porque, sem ele, a tela não se constrói! Perceba que ele tem toda uma anatomia que devemos seguir. Primeiro, ele começa com a palavra `override`,

isso indica que esse é um método da classe mãe (`Activity`) e que estamos sobrescrevendo-o para dar um novo comportamento, isto é, estamos sobrescrevendo-o para colocar o código de criação da nossa tela.

Rpare também que este método recebe a variável `savedInstanceState: Bundle?` por parâmetro, e como quem aciona este método é o próprio sistema operacional, quem é responsável por passar essa variável também é o sistema operacional! A grande questão é: "o que é essa variável?".

A variável `savedInstanceState` guarda informações do estado da `Activity`, guarda quais componentes estão na tela e informações que o usuário tenha preenchido. Ele guarda essas informações porque se o Android precisar tirar a sua `Activity` da memória enquanto outro aplicativo mais prioritário requisita mais memória, quando a sua `Activity` for reconstruída o Android consegue voltar ao estado que esta estava, isso tudo sem necessidade de nenhum código adicional!

Por fim, toda essa lógica está programada internamente no código padrão da `Activity`, então é fundamental a chamada `super.onCreate(savedInstanceState)` pois ela executa essa lógica!

Existem diversos outros métodos de retorno que podemos implementar, mas o único obrigatório é o `onCreate` (em criação).

A seguir listarei todos os métodos que podem ser implementados e qual o momento em que o sistema os chama:

Método	Tradução	Descrição
--------	----------	-----------

<code>onStart</code>	Em inicialização	O método <code>onStart</code> é executado logo após o <code>onCreate</code> , a grande diferença é que durante o <code>onCreate</code> o aplicativo não está visível ao usuário, mas no <code>onStart</code> sim. O método <code>onStart</code> é executado rapidamente assim como o <code>onCreate</code> e o aplicativo não fica parado nesse estado. Logo após a execução do <code>onStart</code> , o aplicativo passa para o estado de "resumido", chamando o método <code>onResume</code>
<code>onRestart</code>	Em reinicialização	O método <code>onRestart</code> é acionado quando uma Activity está no estado "parada" e está sendo iniciada novamente
<code>onResume</code>	Em retomada	O método <code>onResume</code> vem logo após o <code>onStart</code> e deixa o aplicativo pronto para uso. É nesse estado que o usuário consegue interagir com o aplicativo.
<code>onPause</code>	Em pausa	O método <code>onPause</code> é acionado pelo sistema como um primeiro indicador de que o usuário está saindo do aplicativo, isso não quer dizer necessariamente que o usuário está fechando-o, ele pode estar simplesmente abrindo outro aplicativo e deixando este em segundo plano.
<code>onStop</code>	Em parada	O método <code>onStop</code> é acionado na sequência do <code>onPause</code> , e indica que o aplicativo já está totalmente invisível ao usuário.
<code>onDestroy</code>	Em destruição	O método <code>onDestroy</code> é acionado pelo sistema momentos antes de a Activity ser "destruída", ou seja, totalmente finalizada. Isso pode acontecer quando o usuário encerra totalmente o aplicativo ou quando o aplicativo está no estado "parado" e o sistema precisa de memória. Ele pode eleger sua Activity para destruir e ganhar um pouco de espaço.

A seguir, veja uma imagem que ilustra o ciclo de vida de uma Activity:

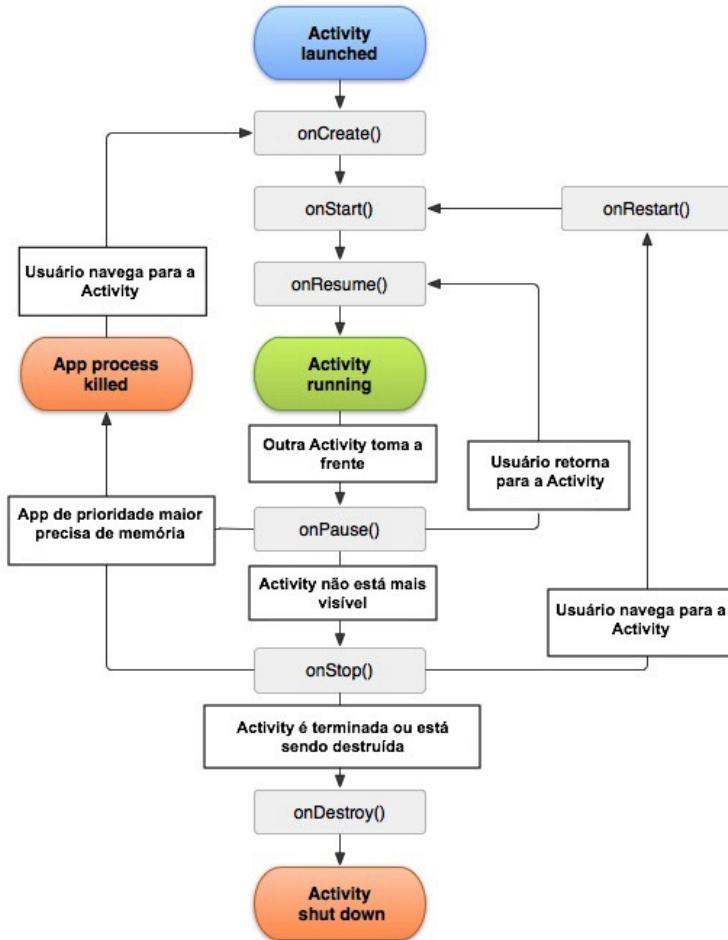


Figura 4.2: Ciclo de vida de uma Activity

Definindo o conteúdo de uma Activity

Definir o conteúdo de uma Activity significa definir o conteúdo que aparecerá na tela, para isso o Android possui o método `setContentView`, cuja tradução seria "Definir conteúdo

de visualização", exatamente o que ele faz!

No entanto, para esse método funcionar, devemos passar o conteúdo que vamos montar na tela para ele. Existem essencialmente 2 formas de passar esse conteúdo. A primeira é criando todo o layout da tela com código em Kotlin e passá-lo no método `setContentView`, esta opção é mais trabalhosa porque o programador tem que montar a tela toda via código e só conseguiria ver o resultado ao compilar e rodar a aplicação.

Outra desvantagem é que o código relacionado ao layout fica misturado ao código relacionado à lógica do aplicativo e, quando há um problema, fica mais difícil para o desenvolvedor saber se o problema é no código do layout ou na lógica do aplicativo. Em alguns casos, é mais vantajoso criar os layouts via código, quando é necessária a criação dinâmica de componentes. Por exemplo, em jogos, a tela é constantemente atualizada, a todo o momento tem objetos sendo criados e sendo retirados da tela. Nesse caso, é mais vantajoso fazer todo esse gerenciamento via código. No entanto, quando se tem um aplicativo estático, já não é mais vantagem a criação do layout via código porque seu código é muito mais complexo e de difícil manutenção.

A seguir está um exemplo de criação de um layout muito simples via código:

```
import android.app.Activity  
import android.os.Bundle  
import android.widget.TextView  
  
class MainActivity : Activity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)
```

```
//Criando um objeto de texto  
val texto = TextView(this)  
texto.text = "Hello Kotlin"  
  
//definindo o conteúdo da tela  
setContentView( texto )  
  
}  
}
```

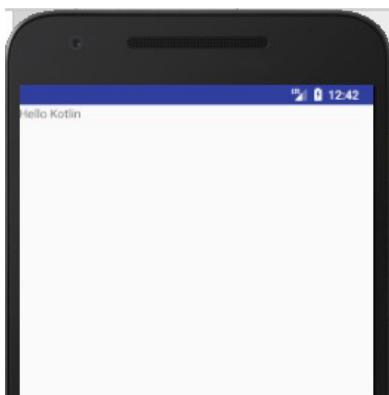


Figura 4.3: Hello Kotlin

O layout criado simplesmente cria um texto na tela escrito "Hello Kotlin". Para isso, eu primeiro criei o objeto de texto com o código: `val texto = TextView(this)` , em seguida, defini o texto que aparecerá na tela: `texto.text = "Hello Kotlin"` e depois utilizei o método `setContentView` , passando como parâmetro o objeto que eu criei acima: `setContentView(texto)` .

A outra forma de criação de layout, e a mais comum, é utilizar um arquivo `xml` e indicá-lo no método `setContentView` . Utilizando essa abordagem, precisamos ter um arquivo `xml` , que

terá a definição do layout do App, e ficará dentro da pasta `res/layout` do nosso projeto. Para esse exemplo utilizarei um arquivo `xml` com o nome `activity_main.xml` e com o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="Hello Kotlin"
/>
```

Esse código é responsável pela criação de um objeto `TextView` na tela, com a propriedade `text` preenchida com o texto "Hello Kotlin". Assim como no exemplo anterior, a diferença é que anteriormente faríamos isso no código em Kotlin.

Desta forma, o código da `Activity` ficará assim:

```
import android.app.Activity
import android.os.Bundle

class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        //definindo o conteúdo da tela
        setContentView( R.layout.activity_main )
    }
}
```

Um detalhe que deve ser observado é a forma como arquivo está sendo passado ao método `setContentView`, que é por meio do código `R.layout.activity_main`. Essa linha pode ser quebrada em 3 partes, mas é muito simples de entender. Vamos analisá-la de trás para frente. Essa linha de código pressupõe que

exista um arquivo `xml` chamado `activity_main`, que está em uma pasta especial do Android, chamada `layout`, e através da variável `R` conseguimos acessar essa pasta; por isso: `R.layout.activity_main`.

Perceba que dessa forma o código em Kotlin diminui bastante, isso porque estamos usando um exemplo com um único componente na tela. Imagine montar a tela de um App mais complexo, toda no código da Activity! Separar o layout da tela em um arquivo `xml` possui diversas vantagens, uma delas é que as responsabilidades ficam separadas e aí fica muito mais fácil para o programador encontrar um problema tanto no código quanto no layout. Outra vantagem de separar o layout em um arquivo é que este arquivo pode ser reutilizado em alguma outra Activity que utilize o mesmo layout.

Um porém de se utilizar esse método é que simplesmente olhando o código da Activity não conseguimos saber qual o layout da tela, temos que olhar diretamente no arquivo `xml`. A estrutura de pastas do projeto tem uma pasta específica para se colocar esses arquivos, a `layout`. Mas não se preocupe, que logo detalharemos a estrutura de pastas do Android.

À primeira vista, parece que esse código é mais complicado que o anterior, mas não é! Primeiro porque a estrutura de dados em `xml` é muito fácil de entender e mesmo se você não tem experiência com arquivos `xml`, você perceberá que a curva de aprendizado é muito rápida.

O segundo ponto é que a IDE nos ajuda muito na criação do código, basta você digitar `<Tex` que a IDE já vai nos sugerir que estamos criando um objeto `TextView` (visualização de texto) e vai

autocompletar o código. Assim funciona para qualquer outro objeto que você for criar no `xml`, é muito fácil!

Outra vantagem é que dessa forma podemos utilizar a janela de Preview (pré-visualização) do Android Studio e assim conseguimos ver a tela sendo produzida em tempo real:

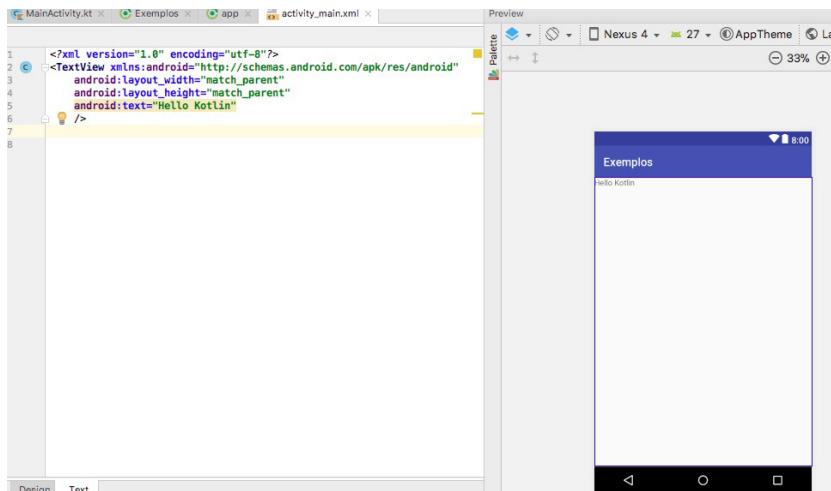


Figura 4.4: Pré-visualização do layout

4.2 ACESSANDO RECURSOS – CLASSE R

Durante o processo de desenvolvimento de um aplicativo, é comum o programador precisar acessar algum recurso através do código. Este recurso pode ser uma imagem, um arquivo `xml` de layout, uma configuração de cor ou texto ou até mesmo o id de um objeto específico. Para facilitar o acesso aos recursos do projeto, o Android possui uma classe chamada `R`. Simples assim mesmo, uma única letra em maiúsculo.

O nome `R` é uma abreviação de "Resources" (Recursos), o que

faz todo sentido. A ideia de chamar a classe somente de `R` é facilitar a vida do desenvolvedor. Como é uma classe que pode ser usada em muitos lugares do código de um aplicativo, fica muito mais fácil para o programador escrever simplesmente `R` do que o nome completo, `Resources`.

Na seção anterior, utilizamos a classe `R` para acessar o arquivo de layout, olhe o código novamente: `setContentView(R.layout.activity_main)`. Preste atenção no uso da classe `R`; esse código acessa o arquivo `activity_main`, que está dentro da pasta `layout`, por isso, o comando `R.layout.activity_main`! Faz todo o sentido!

Agora repare em mais um detalhe: o arquivo `activity_main` é um arquivo em `xml`, mas quando o acessamos através da classe `R`, não é necessário colocar a extensão. Desta forma, podemos acessar qualquer outro recurso do projeto.

Vamos supor que exista uma imagem chamada `background.png` dentro da pasta `drawable`; podemos acessá-la com `R.drawable.background`. Agora vamos supor que exista uma cor chamada azul definida no arquivo `colors`; podemos acessá-la com `R.color.azul`, e essa mesma lógica segue para qualquer recurso dentro da pasta.

4.3 VIEWS – COMPONENTES VISUAIS

As *Views* (visualizações) são todos os componentes visuais que podem ser usados na criação de um aplicativo. A classe `View` é a classe base de qualquer outro componente que podemos usar em uma Activity. Botões, caixas de texto, caixas de seleção, objetos de

imagens etc., todos são derivados da classe `View`.

A imagem a seguir mostra toda a hierarquia da `View`:

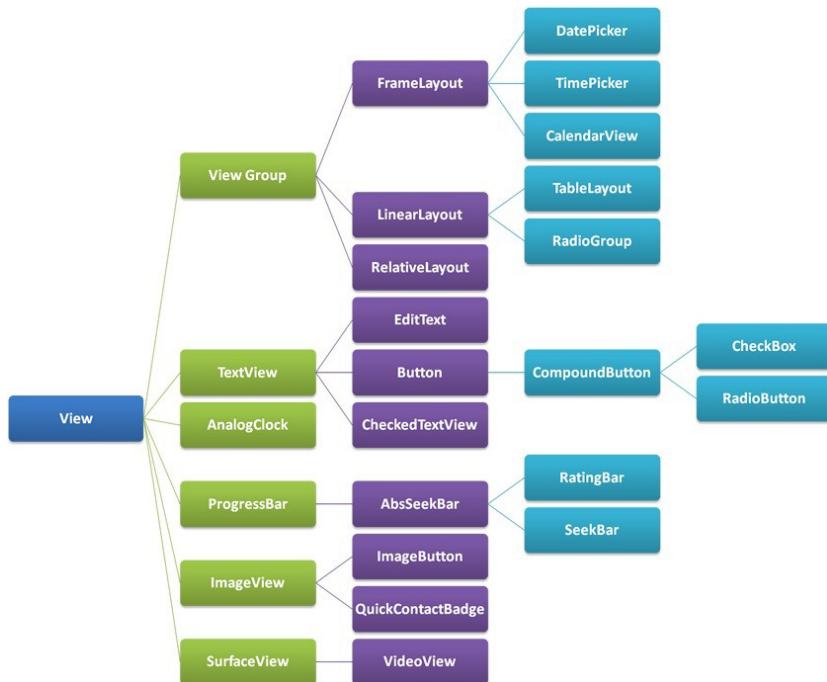


Figura 4.5: Hierarquia da `View`

A seguir, veremos os componentes visuais mais comuns usados em aplicativos.

TextView (visualização de texto)

O `TextView` é um componente de visualização de texto, usado quando queremos mostrar alguma informação escrita para o usuário. Ele pode ser definido pela tag `<TextView />`, e podemos usar sua propriedade `text` para definir o texto exibido na tela.

Veja um exemplo completo:

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Text exemplo"  
/>
```

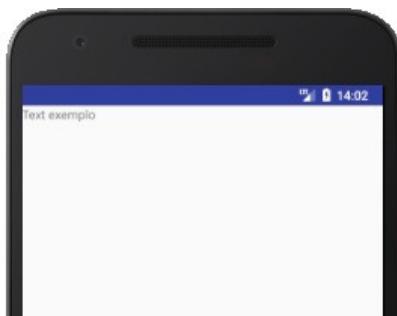


Figura 4.6: TextView

EditText (edição de texto)

O `EditText` é o componente que usamos quando queremos que o usuário informe algo ao aplicativo, conhecido também como caixa de texto. O `EditText` pode ser definido pela tag `<EditText />`, e ele também possui a propriedade `text`, em que podemos definir algum texto.

Outra propriedade de uso muito comum nesse componente é a propriedade `hint` (sugestão). Ela serve para indicar ao usuário qual o conteúdo que deve ser digitado naquele `EditText`, por exemplo, podemos usar o `hint` para indicar que aquele campo deve ser preenchido com seu nome de usuário:

`android:hint="Nome de usuário"`. Veja um exemplo completo:

```
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:hint="Nome de usuário"  
/>
```

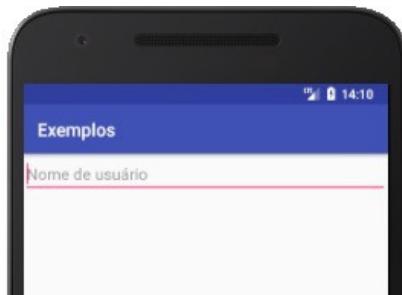


Figura 4.7: EditText

Button (botão)

Outro componente muito comum é o `Button`, que cria um botão na tela e pode disparar alguma ação. Imagine uma tela de cadastro, em que o usuário preenche todas as informações e, ao final, é comum haver um botão `cadastrar`, que dispara uma ação no código, que faz o cadastro do usuário.

Um botão pode ser definido pela tag `<Button />` e também possui a propriedade `text` para definir o texto do botão. Veja a declaração completa:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="cadastrar"  
/>
```

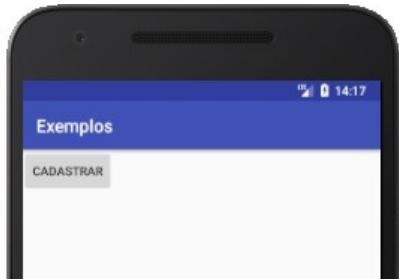


Figura 4.8: Button

LinearLayout (layout linear)

O `LinearLayout` é um pouco diferente das outras Views porque ele faz parte da família do `ViewGroup` (grupo de visualização), o que quer dizer que sua função é agrupar outros componentes dentro dele. Podemos definir um layout linear através da tag `<LinearLayout />`.

O `LinearLayout` possui uma propriedade chamada `orientation` (orientação) que define como os componentes serão exibidos dentro do layout. Essa definição pode acontecer de duas formas: com orientação horizontal ou vertical. Em um layout vertical, todos os componentes dentro dele ficarão um embaixo do outro; já em um layout horizontal, todos os componentes ficarão um ao lado do outro.

Veja um exemplo de um `LinearLayout` com orientação vertical com dois botões dentro:

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:orientation="vertical">
```

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Botão 1"  
/>  
  
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Botão 2"  
/>  
  
</LinearLayout>
```

Veja o resultado visual:

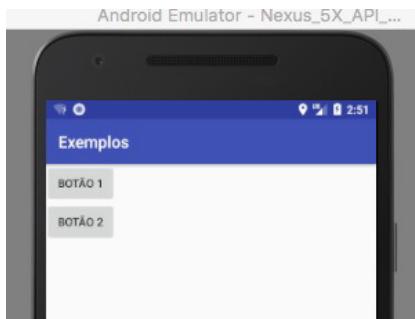


Figura 4.9: Layout vertical

Veja agora como ficaria simplesmente trocando a propriedade orientation do LinearLayout para horizontal :



Figura 4.10: Layout horizontal

Existem diversos outros tipos de layouts além dos lineares, eu particularmente gosto deles porque são simples de entender e utilizar, e flexíveis para montar qualquer tipo de layout mais complexo.

Definindo a altura e a largura de uma View

Em todas as Views, é obrigatório que se definam duas propriedades, `layout_width` (largura) e `layout_height` (altura). Sendo obrigatórias, caso você não as defina, o código não vai compilar, e também não há um valor padrão para elas, ou seja, você sempre terá que as definir.

Mas na prática você verá que, sempre que abrir uma tag, a IDE já vai pedir para você definir esses valores, sendo quase impossível esquecer de fazer isso. Podemos definir essas propriedades de 3 maneiras.

A primeira é definindo uma largura ou altura fixa em dp (pixel independente de densidade), Veja um exemplo em que eu defino a largura de um botão fixa com 130dp:

```
<Button  
    android:layout_width="130dp"
```

```
    android:layout_height="wrap_content"
    android:text="Botão 1"
/>
```



Figura 4.11: Largura fixa

Em casos em que precisamos definir um valor fixo, utilizamos a medida `dp`. A sigla `dp` vem de "Density-independent Pixels", que, em português ficaria: pixel independente de densidade. A utilização do `dp` garante que os componentes sempre terão o mesmo tamanho, mesmo em aparelhos com resolução diferente. Dispositivos diferentes podem ter resolução de tela diferentes, desta forma, o mesmo número de pixels pode corresponder a diferentes tamanhos físicos.

A segunda opção é usar a constante `wrap_content`. Esta opção, quando utilizada na altura ou largura de um componente indica que esta largura ou altura será variável de acordo com o conteúdo dela. No exemplo a seguir, o `Botão 2` está definido com uma largura `wrap_content`, veja:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Botão 2 - Texto Maior Teste"
/>
```

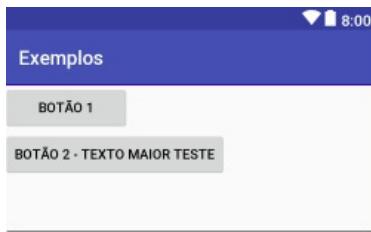


Figura 4.12: Largura wrap_content

A terceira opção é usar a constante `match_parent`. Ela indica que o componente tomará todo o espaço disponível em relação ao layout em que ele está inserido. Veja como ficaria se eu mudasse a largura do Botão 2 para `match_parent`:

```
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Botão 2 - Texto Maior Teste"  
/>
```



Figura 4.13: Largura match_parent

findViewById - Localizando uma visualização pelo id

Quando precisamos acessar algum elemento que foi definido no arquivo de layout pelo código em Kotlin, precisamos localizar aquele elemento pelo seu id, que deve ser definido pelo programador. Imagine uma tela de login de um aplicativo, em que

o usuário digita seu e-mail e sua senha, e o aplicativo verifica se é um usuário válido. Para isso, através do código em Kotlin devemos acessar os campos de e-mail e senha que foram definidos no layout. Para isso, usamos o método `findViewById`, que está disponível na Activity!

O método `findViewById` localiza no layout uma View a partir de um id passado! Vamos fazer um pequeno exemplo. Vamos definir um botão no layout e, em seguida, localizá-lo através do método `findViewById`. Não podemos esquecer de declarar o id do botão através da propriedade `android:id`. Veja a implementação no arquivo de layout:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="login"  
    android:id="@+id/btn_login"  
/>
```

Agora, fora do arquivo de layout, já no arquivo da Activity em Kotlin, vamos acessar esse botão através do método `findViewById`:

```
import android.widget.Button  
...  
  
findViewById<Button>(R.id.btn_login)
```

Perceba que acessamos o id do botão através da classe `R` também! Outro detalhe é que o método `findViewById` pode encontrar qualquer objeto da tela, então devemos fazer a tipagem indicando que o objeto retornado é um botão `findViewById<Button>`. Se fosse um `EditText` bastaria trocar o tipo.

No entanto, utilizar o método desta maneira não nos traria muitos benefícios, já que não estamos guardando uma referência do objeto para poder utilizar. Se não guardarmos uma referência do objeto, logo na próxima linha já não teria mais acesso nenhum ao elemento. O que devemos fazer nesse caso é guardar o resultado do método `findViewById` em uma variável:

```
val btn_login = findViewById<Button>(R.id.btn_login)
```

Desta forma, temos uma variável `btn_login` guardando uma referência do botão que foi definido na tela!

4.4 RESUMINDO

Neste capítulo, vimos alguns pontos básicos do Android, é claro que existe muito mais em um projeto Android, mas eu acredito que esse conteúdo é o básico que você precisa saber para começar a desenvolver aplicativos, independente de ser em Kotlin ou em Java. Nos próximos capítulos, começaremos a trabalhar com alguns projetos, que darão muito mais ênfase na prática com a linguagem Kotlin. Vamos nessa!

CAPÍTULO 5

PRIMEIRO PROJETO – CÁLCULO DA APOSENTADORIA

Agora que você já está com seu computador configurado, vamos iniciar nosso primeiro projeto em Kotlin! Como é o primeiro projeto será simples, porém você aprenderá muita coisa importante que dará base para projetos mais complexos que estão por vir.

Em nosso primeiro projeto, vamos fazer um App que calcula quanto tempo falta para uma pessoa se aposentar! Seria interessante para uma pessoa saber quanto tempo falta ainda para sua aposentadoria, então que tal criarmos nosso próprio App que calcula isso?

5.1 FASE DE PLANEJAMENTO

Antes de botar a mão na massa e sair codificando, seria bom planejar nosso aplicativo. Como ele será visualmente? De que informações ele precisa para funcionar? Qual saída o App terá? Como é feito esse cálculo? São todas questões que precisam ser respondidas antes de partir para o código. Lembre-se, um bom

planejamento é a chave do sucesso do seu projeto.

Vamos começar definindo como será feito o cálculo, porque a partir daí teremos uma ideia de quais informações o usuário precisará informar ao App. A lógica que usaremos no nosso App será simples, considerará somente aposentadoria por idade de acordo com as regras vigentes que utilizam como base idade mínima de 65 anos para homens e 60 anos para mulheres. É claro que há uma série de outros fatores que poderíamos considerar, mas vamos iniciar com um cálculo simples e, ao final do capítulo, você terá total condições de incrementar a lógica para considerar outros fatores.

Para saber quanto tempo falta para uma pessoa se aposentar de acordo com a nossa lógica, devemos primeiro saber se a pessoa é do sexo masculino ou do sexo feminino e aí, caso a pessoa seja do sexo masculino utilizaremos a fórmula: $65 - \text{idade}$. Caso a pessoa seja do sexo feminino, a fórmula é: $60 - \text{idade}$.

Agora que já sabemos como esse cálculo é feito, vamos definir quais informações o usuário precisará informar ao App. Olhando para nosso cálculo, podemos observar que o usuário precisará informar ao App qual seu sexo e qual sua idade, então essas serão as entradas: sexo e idade.

Também precisamos pensar na tela e em como o usuário vai interagir com o App e como o App responderá ao usuário! Como já temos definido quais entradas o App terá, vamos pensar em como o usuário entrará com essas informações. O usuário precisará informar primeiro qual seu sexo, que pode ser masculino ou feminino.

Para isso poderíamos utilizar uma caixa de seleção em que o usuário seleciona a opção correspondente. Em seguida, o usuário precisará informar qual a sua idade, para isso podemos utilizar uma caixa de entrada de texto em que ele digita sua idade de forma numérica. Depois dos valores inseridos no App, o usuário precisará executar uma ação para indicar que ao App que o cálculo já pode ser feito, e para isso podemos incluir um botão "Calcular" em que o usuário clicará para efetuar o cálculo.

Por fim, precisamos pensar em como o resultado será exibido ao usuário. Acho que seria interessante se embaixo do botão aparecesse um texto escrito "Faltam x anos para você se aposentar", sendo x o resultado do cálculo.

Por enquanto, nossa tela terá os seguintes componentes: 1 caixa de seleção, 1 caixa de entrada de texto, um botão e um campo de texto fixo para mostrar o resultado. Perceba que é importante fazer esse planejamento independente da linguagem ou plataforma de desenvolvimento em que você esteja trabalhando. Sempre planeje antes de começar um projeto.

5.2 CRIANDO O PROJETO

Vamos iniciar criando nosso projeto, abra o Android Studio e clique em `Start a new Android Project`. Na tela seguinte, preencha o Application name como `CalculoAposentadoria`, Company domain como `livrokotlin.com.br`, e escolha a pasta em que salvará o projeto. Não se esqueça de deixar a opção `Include Kotlin support` (Incluir suporte ao Kotlin) marcada. Esta opção cria o projeto já configurado para trabalhar com a linguagem Kotlin. Clique em

Next .

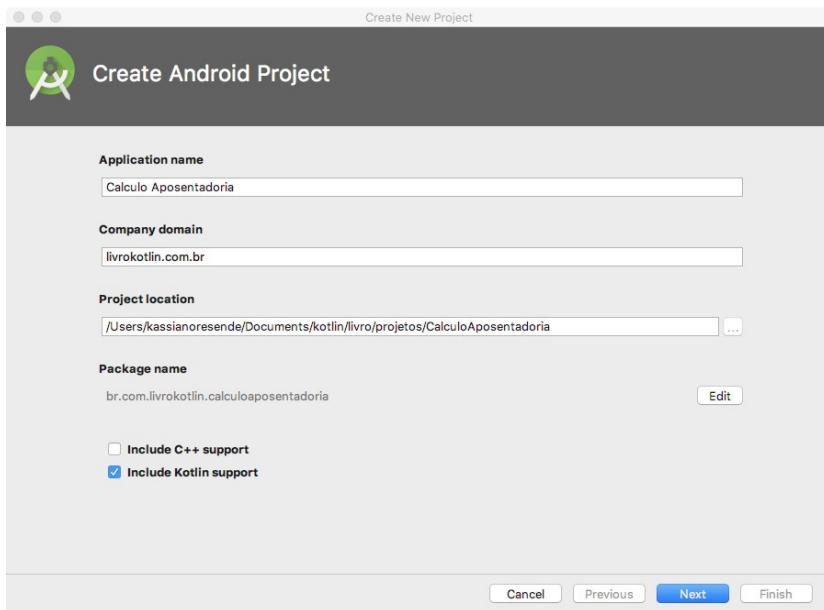


Figura 5.1: Configurações iniciais

Se você já estiver com um projeto aberto no Android Studio, você consegue criar um novo clicando em File > New > New Project .

Na próxima tela, vamos selecionar a versão mínima do Android que nosso App suportará, sugiro que seja a partir da versão 5.0 porque teríamos uma gama muito grande de aparelhos em que nosso aplicativo funcionará. Selecione a API 21: Android 5.0 e clique em Next .

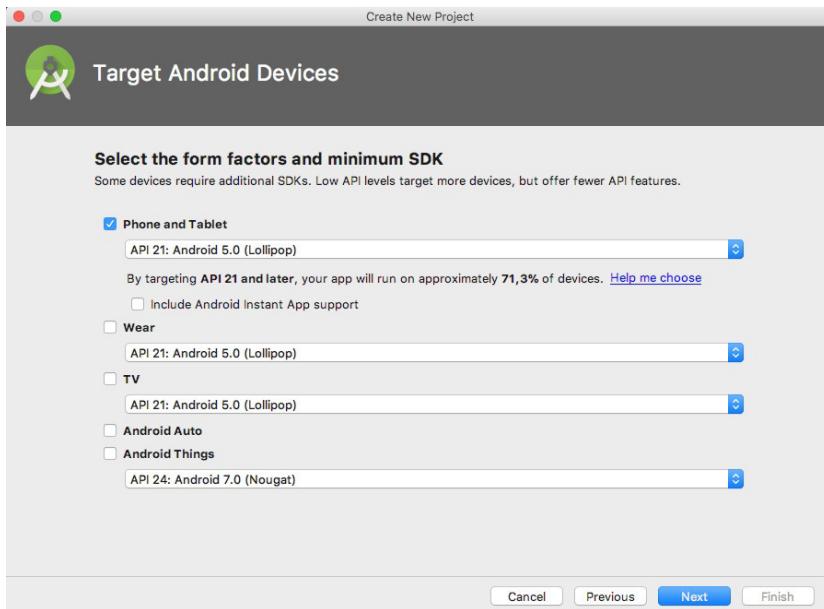


Figura 5.2: Seleção de API mínima

Por fim, selecione a opção `Add` no `Activity` (Não adicionar atividade) e clique em `Finish`.

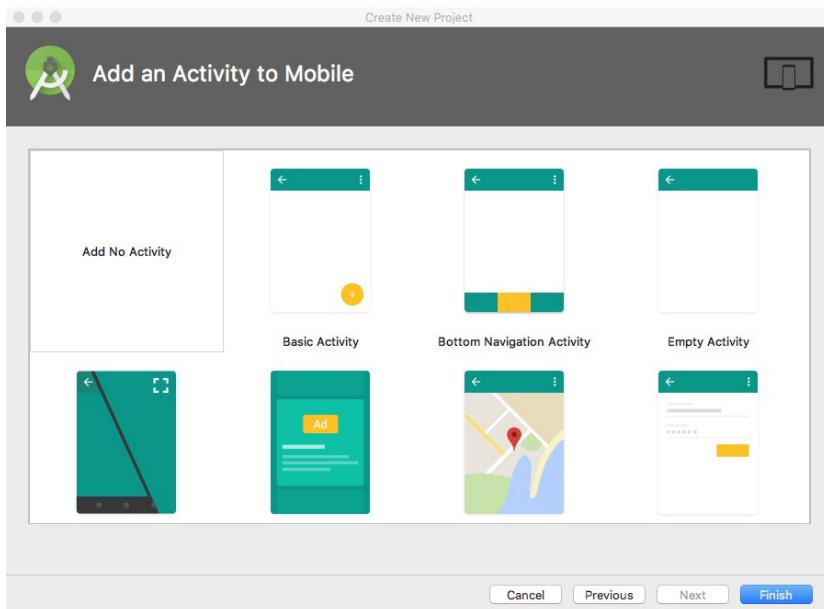


Figura 5.3: Seleção de Activity

Essa opção Add no Activity não criará nenhuma Activity para nós, já que faremos todo o processo do início e você vai aprender passo a passo como criar uma Activity no Android!

Observe que agora o Android Studio abriu o projeto totalmente vazio e, com as abas fechadas, isso porque ele não criou nenhum arquivo de Activity.

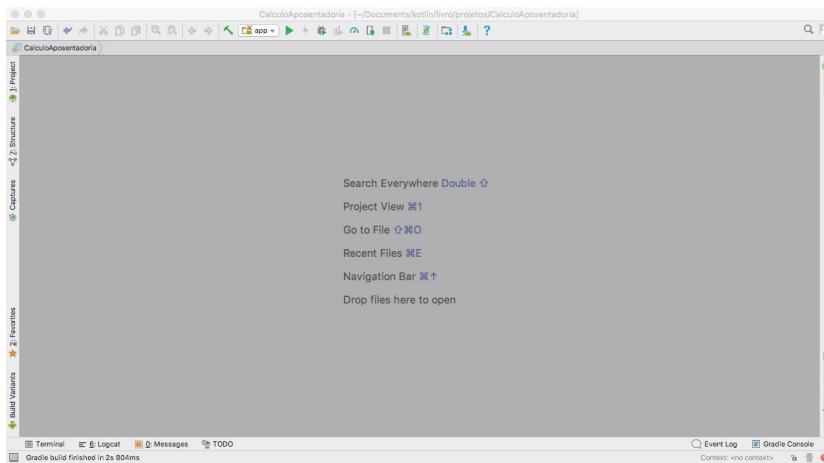


Figura 5.4: Android Studio IDE

5.3 CRIANDO UMA ACTIVITY

Precisamos criar a Activity principal do nosso projeto. Esta será a tela em que funcionará toda a lógica do aplicativo e também, como atividade principal, será a tela que o Android vai inicializar quando o usuário abrir o aplicativo.

Para criar nossa Activity o processo é muito simples, basta criar um arquivo Kotlin dentro da pasta `br.com.livrokotlin.calculoaposentadoria`, que está dentro de `app/java`, e neste arquivo criar uma classe que herde a classe `Activity`.

Vamos começar criando o arquivo Kotlin para nossa Activity, para isso siga os passos a seguir:

Abra a aba de `Project` do lado esquerdo e localize a pasta `br.com.livrokotlin.calculoaposentadoria`, que está dentro

de app/java .

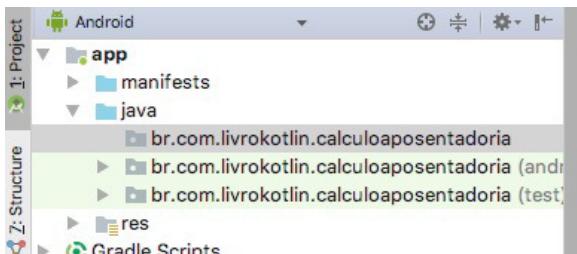


Figura 5.5: Estrutura de pastas

Como vimos no capítulo anterior, é nela que colocamos nossos arquivos de código então é aqui que vamos criar o arquivo para a Activity do nosso App. Para isso, clique com o botão direito sobre a pasta `br.com.livrokotlin.calculoaposentadoria` e escolha a opção `new (novo)` e, em seguida, `Kotlin File/Class` (Arquivo/Classe Kotlin)

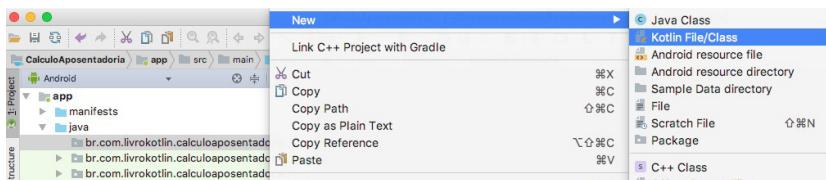


Figura 5.6: Criação de novo arquivo em Kotlin

Nomeie o arquivo como `MainActivity` , já que essa será nossa tela principal. Seguir esse padrão é uma boa prática de programação pois fica muito fácil de localizar os arquivos relacionados a alguma tela.

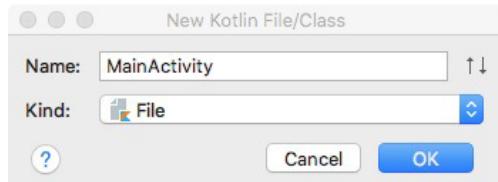


Figura 5.7: Nomear arquivo MainActivity

Isso criará um arquivo vazio dentro da pasta, no qual vamos programar nossa Activity, para isso vamos criar uma classe com nome `MainActivity` :

```
package br.com.livrokotlin.calculoaposentadoria

class MainActivity {
```

```
}
```

Agora precisamos fazer a herança da classe `Activity` para o Android entender que nossa classe é relacionada a uma tela do Android:

```
package br.com.livrokotlin.calculoaposentadoria

import android.app.Activity

class MainActivity : Activity(){
```

```
}
```

Por fim, vamos criar o método `onCreate`. Este é o método que o sistema operacional Android vai chamar para criar a tela.

```
package br.com.livrokotlin.calculoaposentadoria

import android.app.Activity
import android.os.Bundle
```

```
class MainActivity : Activity(){

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

    }

}
```

Dica: para implementação do método `onCreate`, comece a digitá-lo e a própria IDE nos sugere a implementação do método!

Excelente, já temos o esqueleto da nossa Activity pronto! Mas ele ainda não tem conteúdo nenhum, então, é hora de pensar no layout da nossa Activity.

Vamos fazer nosso layout em `xml` porque isso vai deixar nosso código mais organizado, uma vez que os arquivos de layout estão separados da lógica da Activity.

5.4 CRIANDO O LAYOUT DA ACTIVITY

Vamos iniciar pela criação do arquivo `xml`, para isso devemos criar um arquivo dentro da pasta `layout` localizada dentro da pasta `res`, esta é a pasta destinada aos arquivos de layout do projeto:

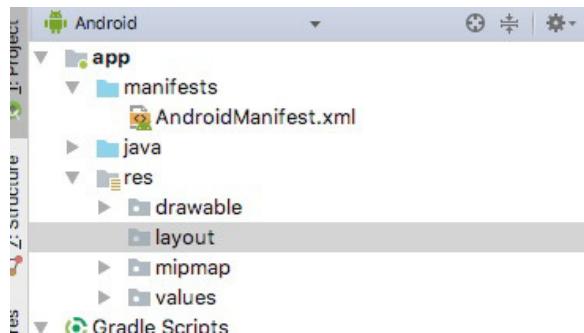


Figura 5.8: Pasta layout

Observação: caso não exista a pasta layout dentro da pasta res , você deve criá-la, para isso basta clicar com o botão direito em res , em seguida, New > Directory e nomear como layout .

Clique com o botão direito na pasta layout e escolha a opção New (novo) e então Layout resource file (Arquivo de recurso de layout):

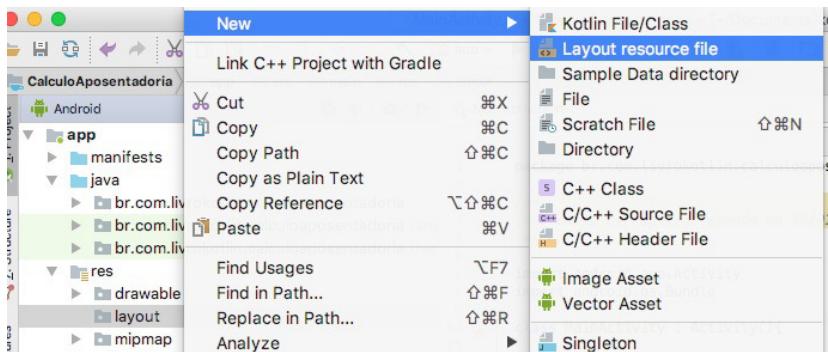


Figura 5.9: Novo layout resource file

Abrirá a seguinte janela:

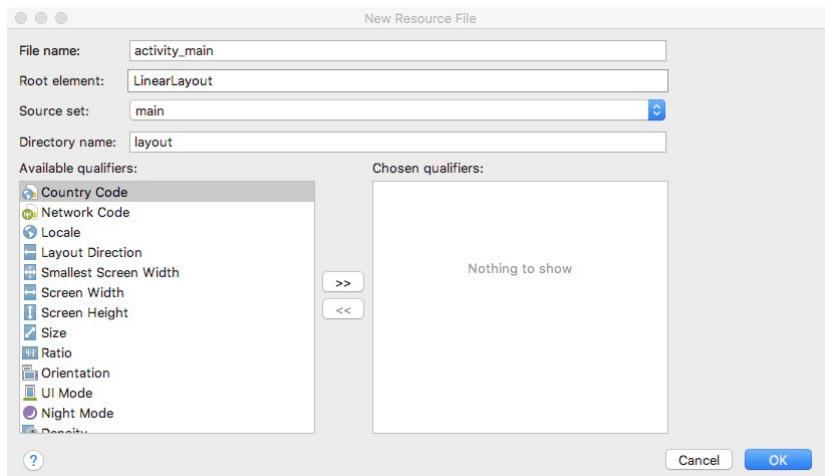


Figura 5.10: layout resource file

Preencha o campo `File name` (Nome do arquivo) com `activity_main` e o campo `Root Element` (Elemento raiz) como `LinearLayout` e clique em `OK`. A seguir, uma explicação do significado de cada campo:

- **File name:** este é o nome do arquivo de layout que será criado.
- **Root Element:** este campo indica o elemento raiz, ou seja, o primeiro elemento deste arquivo.
- **source set:** é onde o arquivo será salvo, deixe a opção `main` selecionada para que o arquivo seja criado no diretório principal do aplicativo.
- **Directory name:** é o nome da pasta em que o arquivo será salvo, preencha como `layout`.

- **Available qualifiers:** são configurações que podemos usar para especificar algumas condições especiais para alguns arquivos, por exemplo, poderíamos configurar um tipo de layout específico para quando o celular estiver em modo paisagem, podemos configurar um arquivo específico para determinado idioma que o aparelho estiver configurado. Para nosso caso, não escolheremos nada desta opção.

Ao clicar em **OK**, um novo arquivo será criado na pasta **layout** e a IDE já o abrirá para nós no modo design:

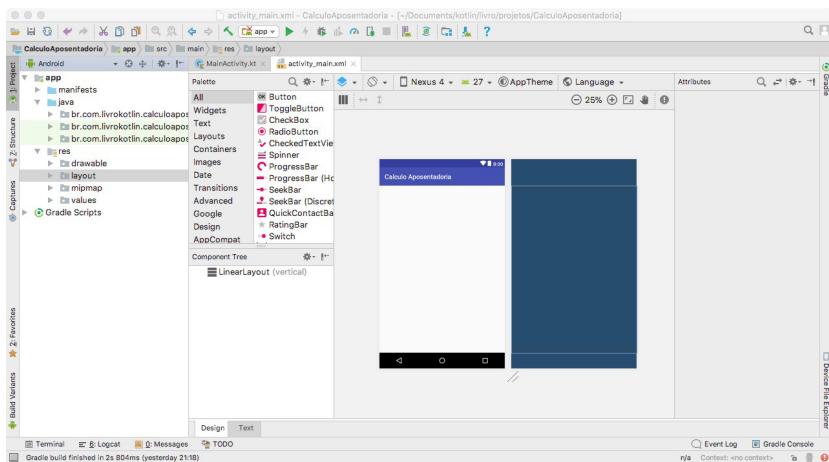


Figura 5.11: Modo design

Através do modo design, é possível montar a tela arrastando os componentes que estão disponíveis na paleta central. Nem sempre a melhor forma de se montar o layout é pelo modo design, isso porque você precisa entender antes como funciona o posicionamento dos elementos na tela em diferentes tipos de **ViewGroup** (grupo de visualização).

Vamos programar nossa tela pelo modo texto, o que nos permite trabalhar diretamente no código-fonte da tela. Parece mais trabalhoso, inicialmente, porém na prática você vai perceber que a IDE é muito produtiva e a hierarquia do XML é muito fácil e sempre podemos deixar a aba de preview (pré-visualização) aberta para ver como a tela está ficando.

Localize na parte inferior da tela o botão Text e clique nele para mudar do modo design para o modo de edição de texto.

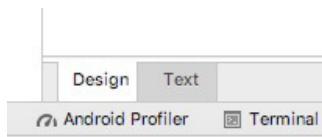


Figura 5.12: Botão modo texto

Desta maneira, se abrirá o código-fonte desse arquivo:

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

</LinearLayout>
```

Esse código foi criado automaticamente pela IDE, porque na criação do arquivo escolhemos como Root Element (elemento raiz) um LinearLayout (layout linear).

Agora vamos relembrar a fase de planejamento desse projeto e em como pensamos nossa tela. A tela do nosso aplicativo deve ter 1 caixa de seleção, 1 campo de entrada de texto, um botão e um texto fixo para mostrar o resultado. A seguir, vamos ver como criar cada

componente desses individualmente e depois vamos juntar tudo para criar nossa tela.

O primeiro componente que precisamos criar é uma caixa de seleção em que o usuário vai selecionar seu sexo. Existe um componente no Android chamado `Spinner` (controle giratório), que, ao ser clicado, mostra um menu suspenso com as opções que o usuário pode escolher. Veja um exemplo de `Spinner`:

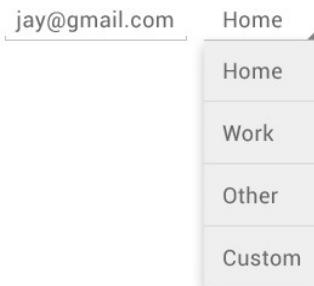


Figura 5.13: Exemplo componente Spinner

Outras plataformas utilizam o nome de `DropDown` ou `ComboBox` para representar componentes semelhantes ao `Spinner`.

Para criar um `Spinner` no layout utilizamos a tag `<Spinner>`. Veja sua implementação completa:

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

Este código cria um `Spinner` na tela, porém do jeito que está não haveria nenhum item disponível para selecionar. Para inserir itens a este controle devemos fazer através do código em Kotlin. Veremos isso mais adiante.

O próximo componente de que precisamos no nosso layout é uma caixa de entrada de texto para que o usuário informe sua idade. Para isso, podemos usar o componente `EditText` (edição de texto), que permite que o usuário digite alguma informação através de uma caixa de texto na tela. Veja como fica sua implementação no layout:

```
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/txt_idade"  
/>
```

Agora nosso layout precisa de um botão! Este botão será usado para acionar o cálculo no aplicativo depois que o usuário terminar de preencher as informações. Vamos utilizar o componente `Button` para criá-lo:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id	btn_calcular"  
    android:text="Calcular"  
/>
```

Por fim, precisamos criar um componente de texto para mostrar o resultado do cálculo para o usuário. Para isso, vamos utilizar o `TextView` (visualização de texto), que nos permite exibir um texto na tela:

```
<TextView  
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:id="@+id/txt_resultado"
  />
```

Agora que sabemos como criar cada componente, vamos juntar todos os códigos e montar o layout completo dentro do arquivo `activity_main`. Lembre-se de que este arquivo foi criado dentro da pasta `layout` para servir como o arquivo de layout da nossa Activity.

A ordem do arquivo ficará assim: primeiro, teremos um `LinearLayout` com orientação vertical para que cada componente fique um embaixo do outro; em seguida, criaremos um `Spinner` que permitirá a seleção do sexo do usuário; na sequência, criaremos uma caixa de texto para o usuário informar sua idade; depois criaremos um botão que será responsável por acionar o cálculo do aplicativo; e por fim, criaremos um componente de texto para mostrar o resultado do cálculo para o usuário.

A ordem de criação dos componentes fará diferença no resultado final, pois, como estamos utilizando um layout linear com orientação vertical, os componentes ficarão um embaixo do outro seguindo a ordem de criação de cada um.

Faça a implementação completa deste código conforme o código seguinte:

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

    <Spinner>
```

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

<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/txt_idade"
    />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Calcular"
    android:id="@+id/btn_calcular"
    />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/txt_resultado"
    />

</LinearLayout>
```

Para ter uma pré-visualização de como está ficando a tela, abra a aba `Preview` localizada do lado direito da IDE:

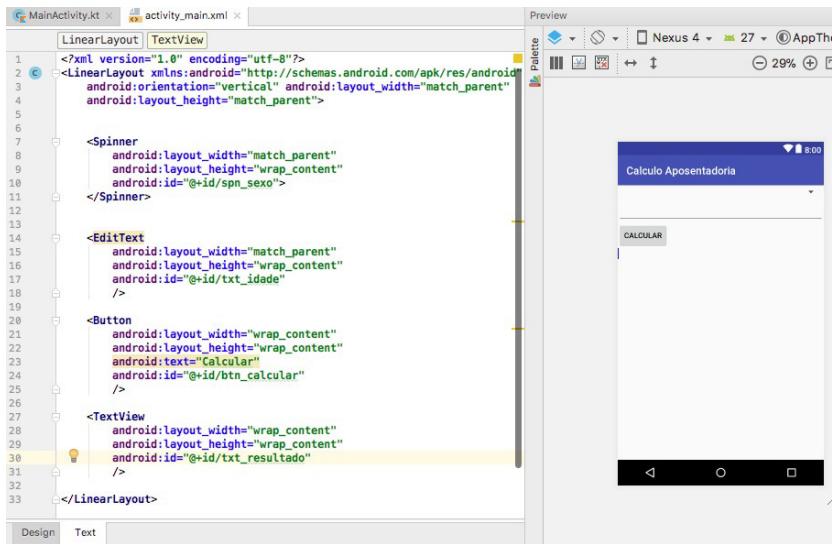


Figura 5.14: Preview

Agora temos o layout do nosso aplicativo! Mas antes de partir para o código em Kotlin, gostaria de fazer pequenos ajustes nesse layout para melhorar um pouco. O primeiro ajuste que farei é definir um padding (preenchimento) no `LinearLayout` para que os componentes não fiquem tão grudados nos cantos. Essa propriedade nos permite definir um número em dp (densidade de pixels) que dá um espaço entre o componente e seu conteúdo.

Localize a tag do `LinearLayout` e inclua a propriedade `android:padding`, ficará assim:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">
```

Outro ajuste que gostaria de fazer é incluir alguns `TextViews` com textos que indiquem ao usuário o que ele deve fazer. Logo de início, colocarei um texto sobre o que se trata o App, que será centralizado na tela. Para isso, usarei a propriedade `android:gravity="center_horizontal"`. O texto também terá um tamanho de fonte `16sp` para dar um destaque e, para isso, usarei a propriedade `android:textSize="16sp"`.

Para tamanho de fontes usamos `sp` em vez de `dp`. `sp` significa "*Scale-independent Pixels*" (pixels independentes de escala). Na prática, a medida `sp` funciona de forma semelhante ao `dp`, com a diferença de que ela também sofrerá alterações de tamanho de acordo com as configurações de fonte do aparelho celular!

Por fim, definirei uma margem inferior de `10dp` para dar um espaço para o próximo componente. Veja:

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Calcule quanto tempo falta para sua aposent  
adora !"  
    android:gravity="center_horizontal"  
    android:textSize="16sp"  
    android:layout_marginBottom="10dp"  
/>
```

Também colocarei um texto indicativo em cima do `Spinner` e do `EditText` para o usuário saber o que são esses campos. O código final fica assim:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Calcule quanto tempo falta para sua aposentadoria !"
        android:gravity="center_horizontal"
        android:textSize="16sp"
        android:layout_marginBottom="10dp"
    />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Escolha o sexo:"
        android:gravity="center"
    />
    <Spinner
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/spn_sexo">
    </Spinner>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Digite sua idade:"
        android:gravity="center"
    />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txt_idade"
        android:gravity="center"
    />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Calcular"
        android:gravity="center"
    />

```

```
        android:id="@+id/btn_calcular"
    />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/txt_resultado"
    />

</LinearLayout>
```

O resultado final deve ser o seguinte:

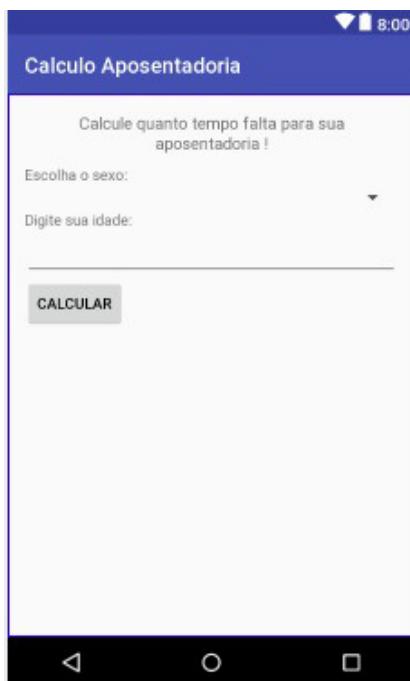


Figura 5.15: Preview

Ótimo! Já deu uma boa melhorada no layout! Agora podemos partir para o código e programar a lógica do nosso aplicativo.

5.5 PROGRAMANDO A LÓGICA

Abra o arquivo `MainActivity` que foi criado no começo deste capítulo, vamos desenvolver nossa lógica nele. Para começar, precisaremos definir que a `Activity` usará o arquivo `activity_main` como layout. Para isso, usaremos o método `setContentView` e indicaremos o arquivo que criamos na pasta `layout`. Isso será feito no método `onCreate`, pois é o método de criação de tela:

```
import android.app.Activity
import android.os.Bundle

class MainActivity : Activity(){

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        //definindo o arquivo de layout
        setContentView(R.layout.activity_main)

    }
}
```

O próximo passo é definir as variáveis referentes aos componentes da tela. Como todos os componentes visuais foram definidos no `xml`, precisamos acessá-los de alguma forma aqui no código em Kotlin. Para isso, o Android nos disponibiliza o método `findViewById`, capaz de instanciar um objeto procurando pelo id definido no `xml`.

Quando montamos nosso layout em `xml`, definimos um id para cada componente: o controle giratório ficou com `spn_sexo` como id, a caixa de texto ficou como `txt_idade`, o botão ficou

como `btn_calcular` e o componente de texto ficou como `txt_resultado`. Então, para cada componente destes teremos também uma variável aqui no código em Kotlin que o referencia.

Vamos fazer primeiro o `findViewById` do `Spinner`, vendo todos seus detalhes, e depois faremos para o resto dos componentes. Para isso o comando é o seguinte:

```
val spn_sexo = findViewById<Spinner>(R.id.spn_sexo)
```

Vamos detalhar essa linha de código em algumas etapas. Primeiro, repare que estamos definindo o tipo do componente no comando `findViewById`. No caso, estamos definindo que o componente é um `Spinner`, por isso o comando fica `findViewById<Spinner>`. Em seguida, passamos o `id` do componente como parâmetro: `R.id.spn_sexo` e, por fim, pegando todo esse retorno e colocando na variável `spn_sexo`, assim temos o comando completo: `val spn_sexo = findViewById<Spinner>(R.id.spn_sexo)`

Faremos todo esse código ainda no método `onCreate` pois toda essa lógica precisa ser executada assim que o aplicativo for aberto. Nesse código, vamos instanciar objetos do tipo `Spinner`, `Button`, `EditText` e `TextView`, por isso precisaremos dos seguintes `imports` também:

```
import android.widget.Button
import android.widget.EditText
import android.widget.Spinner
import android.widget.TextView
```

Mais adiante, usaremos um objeto do tipo `ArrayAdapter` também, que será explicado na hora certa, mas já vamos fazer seu `import`:

```
android.widget.ArrayAdapter
```

Agora veja como fica a implementação deste código na Activity:

```
import android.app.Activity
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.Spinner
import android.widget.TextView
import android.widget.ArrayAdapter

class MainActivity : Activity(){

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        //definindo o arquivo de layout
        setContentView(R.layout.activity_main)

        //acessando o spinner
        val spn_sexo = findViewById<Spinner>(R.id.spn_sexo)

        //acessando a caixa de idade
        val txt_idade = findViewById<EditText>(R.id.txt_idade)

        //acessando o botão de calcular
        val btn_calcular = findViewById<Button>(R.id.btn_calcular
    )

        //acessando o texto de resultado
        val txt_resultado = findViewById<TextView>(R.id.txt_resul
tado)
    }
}
```

Agora que já temos todas as variáveis, precisamos definir os valores que o usuário poderá selecionar no Spinner . Este campo terá duas opções de seleção, masculino ou feminino . Para fazer isso, vamos utilizar o objeto ArrayAdapter , que será responsável

por passar os itens que queremos preencher para o Spinner e também por definir o layout do item. O ArrayAdapter é muito importante no Android, porque o utilizamos também para listas, então não se preocupe muito agora com os detalhes deste trecho de código, vamos estudar o ArrayAdapter profundamente nos capítulos seguintes.

Para adicionar os itens masculino e feminino ao Spinner , usaremos o seguinte código:

```
spn_sexo.adapter = ArrayAdapter<String>(this, android.R.layout.si  
mple_spinner_dropdown_item,  
        listOf("masculino", "feminino"))
```

Vamos agora definir o evento de clique do botão. Gostaríamos que o cálculo seja executado somente quando o usuário clicar no botão, para isso devemos definir esta ação. Se você já possui algum conhecimento em Java, isso não será nenhuma novidade, mas se não possui, não tem problema. Para definir a ação de clique do botão, precisamos definir um listener (ouvinte) para esse botão!

Cada componente pode possuir diversos tipos de métodos de listeners (ouvintes), esses métodos funcionam como verdadeiros ouvintes de algum evento. No nosso caso, não queremos que o cálculo seja executado direto, queremos que ele seja executado somente quando o usuário clica no botão, então no botão existe um método ouvinte que se chama `setOnClickListener` . Este é o ouvinte do evento de clique, ou seja, o código deste método somente será executado quando ele detectar que o botão recebeu um clique, por isso os chamados de métodos ouvintes.

Vamos então implementar o método `setOnClickListener`

do botão da seguinte maneira:

```
btn_calcular.setOnClickListener {  
    //aqui vai o código que será executado quando houver um click  
    do botão  
  
}
```

Agora podemos partir para o cálculo em si, o fluxo de ações do usuário será o seguinte:

1. Escolher o sexo;
2. Digitar a idade;
3. Clicar no botão `Calcular` ;
4. Verificar o resultado exibido em tela.

Internamente, o aplicativo deve seguir o seguinte fluxo quando o usuário clicar no botão `Calcular` :

1. Capturar o sexo da pessoa de acordo com o escolhido na caixa de seleção;
2. Capturar o valor digitado na caixa de idade;
3. Verificar se o sexo escolhido é igual a `masculino`
 - Caso sim, efetuar o cálculo: $65 - \text{idade}$
 - Caso contrário, efetuar o cálculo: $60 - \text{idade}$
4. Exibir o resultado de acordo com o cálculo.

Vamos ao código! A primeira coisa que precisamos fazer é capturar o sexo que a pessoa escolheu no controle giratório da tela. Para isso, o componente `Spinner` que utilizamos possui uma propriedade chamada `selectedItem` , que nos retorna justamente o item selecionado!

Podemos então utilizar a nossa variável `spn_sexo` e acessar

sua propriedade `selectedItem` para podermos capturar o item selecionado. No entanto, somente capturar o item selecionado não é suficiente, precisamos ainda guardá-lo em uma variável, porque vamos utilizá-lo um pouco mais adiante no código. Veja:

```
val sexo = spn_sexo.selectedItem
```

Nesse código, é criada uma variável com nome `sexo`, que guardará o valor que o usuário selecionou na tela. Mas tem mais um detalhe com que precisamos nos preocupar nessa linha de código: a propriedade `selectedItem` do `Spinner` internamente implementa o tipo `Object` e isso dá uma liberdade para o programador criar `Spinner` de vários tipos diferentes.

O nosso controle giratório é um `Spinner` de `String` porque o conteúdo que colocamos dentro dele na inicialização foi `String`, então devemos tomar o cuidado de retornar o tipo correto também. Para isso, utilizamos a palavra-chave `as` do Kotlin e em seguida indicamos o tipo do objeto. Veja:

```
val sexo = spn_sexo.selectedItem as String
```

Desta forma, garantimos que o valor da propriedade `selectedItem` nos retorne em `String` e consequentemente a variável `sexo` será do tipo `String`.

Agora precisamos capturar a idade digitada pelo usuário. Para isso, podemos acessar a propriedade `text` do componente `EditText` que utilizamos no layout. Essa propriedade guarda o texto que foi digitado na pelo usuário.

Vamos utilizar a nossa variável `txt_idade`, que faz referência à caixa de texto exibida na tela, e acessar sua propriedade `text`, mas precisamos também guardar esse valor em uma outra variável

para podermos usar no cálculo. Veja:

```
val idade = txt_idade.text
```

Neste ponto, precisamos de atenção a uma condição: quando o cálculo for executado, o valor da idade deve ser um valor numérico e inteiro; se não, o programa não conseguirá calcular corretamente. No entanto, a propriedade `text` de um `EditText` nos retorna um objeto do tipo `Editable`, que nada mais é do que uma interface que a classe `String` implementa.

Na prática, isso quer dizer que o retorno da propriedade `text` não é numérico mas sim em texto, o que faz sentido pois o `EditText` pode ser utilizado para entrada de qualquer tipo de texto. Pensando nisso, teremos que converter esse valor de texto para inteiro.

A linguagem Kotlin através da classe `String` possui alguns métodos que fazem esse tipo de conversão. Para converter uma `String` em um número inteiro, basta utilizarmos o método `toInt()`, que a conversão será feita!

Só devemos nos atentar para o fato de que o retorno da propriedade `text` do `EditText` não é diretamente uma `String`, mas sim um `Editable`. Por esse motivo, antes de utilizar o método `toInt()`, devemos utilizar o método `toString()`. O método `toString` é disponível a qualquer objeto do Kotlin e sua função é retornar uma representação em texto do objeto em questão.

O código ficará assim:

```
val idade = txt_idade.text.toString().toInt()
```

O próximo passo é fazer a verificação do sexo informado pelo usuário, isso é importante pois toda a lógica dependerá dessa verificação. Faremos um cálculo específico caso o sexo seja masculino e um cálculo diferente caso o sexo seja feminino . Para essa verificação utilizaremos o comando `if` do Kotlin.

Vamos fazer um `if` para verificar se a variável `sexo` é igual a `masculino` ; caso sim, faremos o cálculo referente ao sexo masculino, caso contrário faremos o cálculo referente ao sexo feminino. Perceba que, com essa lógica, caso o sexo do usuário não seja masculino só pode ser feminino, pois só deixamos essas duas opções, sendo assim não há necessidade de um segundo `if` , bastando usar o comando `else` . Veja a implementação:

```
if(sexo == "masculino"){
    //efetuar cálculo para homem
}else{
    //efetuar cálculo para mulher
}
```

O próximo passo é efetuar o cálculo em si, mas precisamos ter em mente que vamos utilizar o resultado desse cálculo depois para atualizar a tela, então precisaremos criar uma variável para guardar esse resultado. Vamos chamá-la de `resultado` .

```
var resultado = 0
```

Agora sim, vamos programar o cálculo e guardar o resultado na variável `resultado` :

```
var resultado = 0
if(sexo == "masculino"){
    resultado = 65 - idade
}else{
    resultado = 60 - idade
}
```

O último passo é atualizar a tela com o resultado calculado! Para isso, vamos simplesmente atualizar a propriedade `text` da variável `txt_resultado`. Lembre-se de que essa variável faz referência ao elemento `TextView` definido no layout.

```
txt_resultado.text = "Faltam $resultado anos para você se aposentar."
```

E agora o código completo do botão:

```
btn_calcular.setOnClickListener {  
  
    //capturando o sexo selecionado  
    val sexo = spn_sexo.selectedItem as String  
  
    //capturando a idade digitada  
    val idade = txt_idade.text.toString().toInt()  
  
    //variável para guardar o resultado do cálculo  
    var resultado = 0  
  
    //verificando o sexo da pessoa  
    if(sexo == "masculino"){  
        resultado = 65 - idade  
    }else{  
        resultado = 60 - idade  
    }  
  
    //Atualizando a tela de acordo com o resultado do cálculo  
    txt_resultado.text = "Faltam $resultado anos para você se aposentar."  
}
```

Agora precisamos testar nosso aplicativo, mas ainda falta um último detalhe para nossa Activity funcionar: nós devemos registrá-la no arquivo `AndroidManifest`.

5.6 ANDROIDMANIFEST

Todo aplicativo precisa ter um arquivo chamado `AndroidManifest.xml`, que pode ser encontrado dentro da pasta `app/src/main/` se você estiver utilizando a visualização no modo `Project`. Se você estiver no modo de visualização `Android`, esse arquivo se encontrará em uma pasta chamada `manifests`. Vale ressaltar que essa pasta `manifests` só existe no contexto de visualização no modo `Android`, ela não existe fisicamente na estrutura de pastas do seu projeto.

Em um projeto criado no `Android Studio`, a própria IDE já criou o arquivo de `manifest` para nós. Ele apresenta as informações essenciais do aplicativo para o sistema operacional `Android`, e é nele que definimos as permissões que o aplicativo precisa ter, qual o nome do pacote, quais `Activities` o App terá, entre outras coisas.

Então localize o arquivo e abra-o. Você encontrará as seguintes configurações:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.livrokotlin.calculoaposentadoria">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" />
</manifest>
```

A primeira configuração que devemos fazer é registrar nossa `Activity`, para isso basta adicionar a tag `<activity />` e definir o nome da classe na propriedade `name`. Ficará assim:

```
<activity android:name=".MainActivity"></activity>
```

Outra configuração que devemos fazer é definir os filtros de intenção através da tag `<intent-filter></intent-filter>`. Os filtros de intenção são configurações que se comunicam diretamente com o sistema operacional Android e é através delas que o sistema sabe do que se trata uma Activity.

Por exemplo, quando você vai compartilhar algum conteúdo através de qualquer aplicativo Android, abre-se uma tela para você escolher com qual aplicativo deseja compartilhar tal conteúdo. O Android faz isso lendo os filtros de intenção dos aplicativos que você tem instalado e assim ele sabe exatamente quais Activities e quais aplicativos aceitam aquele tipo de compartilhamento. A mesma coisa acontece quando você vai abrir uma imagem e tem mais de um aplicativo de imagens no celular. O Android lê os filtros de intenção dos aplicativos e pede para você selecionar em qual aplicativo você deseja abrir a imagem. Isso só funciona porque nos filtros de intenção desses aplicativos está configurado que são aplicativos que tratam imagens!

As configurações que devemos fazer aqui são duas: definir a `action` (ação) e a `category` (categoria). A configuração de `action` diz ao sistema a ação que aquela Activity aceita. Esse valor deve ser uma String literal passada através da propriedade `name`. Existem diversos tipos de ações que já fazem parte do sistema Android, ou você poderá passar um valor próprio.

No nosso caso, usaremos a ação `MAIN` (principal), que informará ao sistema que essa é a Activity principal do nosso aplicativo. Essa configuração será feita da seguinte maneira:

```
<action android:name="android.intent.action.MAIN"></action>
```

Você pode consultar a documentação oficial para ver mais detalhes sobre *actions*:
<https://developer.android.com/guide/topics/manifest/action-element.html>

Outra configuração que devemos fazer é informar a categoria da Activity, isto é, se ele é um navegador de internet, uma calculadora, um aplicativo de e-mail etc. Assim como a ação, em categoria devemos informar uma String literal passada através da propriedade `name`. Existem diversos tipos de categorias no sistema, mas você pode passar um valor próprio.

Aqui vamos definir a categoria `LAUNCHER`, que vai informar ao sistema que essa é a Activity de lançamento do aplicativo. A configuração será feita da seguinte maneira:

```
<category android:name="android.intent.category.LAUNCHER"/>
```

Essas configurações em conjunto vão dizer ao sistema operacional que essa Activity é a principal e é ela que deve ser inicializada quando o usuário clicar no ícone do App.

A configuração completa da tag da Activity é a seguinte:

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"></action>
    >
        <category android:name="android.intent.category.LAUNCHER"
    />
    </intent-filter>
</activity>
```

Essa configuração fica dentro da tag de application (aplicação). O código completo do AndroidManifest ficará assim:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    package="br.com.livrokotlin.etanolougasolina"

        <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
            android:roundIcon="@mipmap/ic_launcher_round"
            android:supportsRtl="true"
            android:theme="@style/AppTheme">

            <activity android:name=".MainActivity">
                <intent-filter>
                    <action android:name="android.intent.action.MAIN">
                </action>
                    <category android:name="android.intent.category.LAUNCHER"/>
                </intent-filter>
            </activity>

        </application>
</manifest>
```

Lembre-se de que as configurações de activity ficam dentro da tag de application

Agora sim podemos rodar nosso aplicativo! Aperte o botão de Play do Android Studio, escolha um emulador ou algum dispositivo Android conectado ao computador e aperte OK . Se tiver alguma dúvida nessa parte, ou nenhum emulador aparecer

disponível, consulte o capítulo 3. *Configurando o ambiente de desenvolvimento*.



Figura 5.16: Preview

Faça algumas simulações para testar a nossa lógica e verifique se os resultados estão corretos.

5.7 RESUMINDO

Eba, nosso primeiro projeto está em funcionamento! É claro que poderíamos fazer diversas melhorias nele, tanto de design

quanto de usabilidade, e até mesmo de código. Nos próximos capítulos aprenderemos algumas técnicas mais avançadas e aí você poderá voltar a esse projeto e fazer as modificações por si próprio.

Aprendemos muita coisa neste capítulo! Foi um projeto simples, porém trabalhoso: aprendemos a criar uma Activity do zero, também aprendemos a criar arquivos `xml` de layout, a manipular botões, caixas de texto e caixas de seleção, e definimos uma ação para um botão - tudo isso em um aplicativo real e funcional.

Nos próximos capítulos, veremos outros componentes e funcionalidades do Android, e como a linguagem Kotlin junto à IDE pode nos ajudar simplificando as coisas. Você verá que muito do que fizemos neste projeto com um certo trabalho pode ser feito com muito menos esforço graças ao Kotlin e também à IDE, em alguns casos. Nos vemos no próximo capítulo ;)

CAPÍTULO 6

LISTA DE COMPRAS

Neste projeto vamos implementar uma lista de compras! Você provavelmente já passou pela situação de ir ao supermercado para fazer compras e teve que levar uma listinha para se lembrar de quais produtos você precisava comprar. A ideia desse projeto é criar um aplicativo para criar uma lista de compras que você poderia usar em um supermercado ou em qualquer outro lugar.

Para implementação desse projeto, vamos trabalhar com um dos principais objetos no Android, que são as listas. Se você reparar nos aplicativos que você utiliza no dia a dia, você perceberá que a grande maioria utiliza listas em algum momento, por exemplo: os aplicativos de mensagem usam uma lista para mostrar seus contatos, uma rede social usa uma lista para mostrar as últimas postagens, o aplicativo de e-mail usa uma lista para mostrar seus e-mails. Se você pensar bem, quase todos os aplicativos que você utiliza no dia a dia utilizam listas para mostrar informações ao usuário, então é superimportante saber trabalhar com essa estrutura no Android.

6.1 PLANEJAMENTO

Vamos começar a pensar no funcionamento do nosso App.

Começaremos pelo layout da tela: que informações o usuário precisará informar para o aplicativo? E que informações o aplicativo deve mostrar na tela?

Como estamos pensando em uma lista de compras, seria interessante o usuário poder inserir o nome do produto que deseja adicionar à lista. Uma coisa já sabemos, precisaremos de uma caixa de texto no nosso aplicativo para que o usuário possa inserir o nome do produto. Se esse produto será inserido em uma lista posteriormente, então precisaremos também de um botão para o usuário clicar e inserir o item digitado na lista! Por fim, o aplicativo deve mostrar uma lista com todos os produtos que o usuário inseriu, então devemos ter na tela um componente de lista!

6.2 CRIANDO O PROJETO

Agora que já temos em mente como nosso aplicativo deve funcionar, vamos começar nosso projeto. Abra o Android Studio e crie um projeto com as seguintes características:

- **Nome do aplicativo:** Lista de compras
- **Domínio:** livrokotlin.com.br
- **API Mínima:** 21 - Android 5.0
- **Modelo de Activity:** Add No Activity

Não se esqueça de marcar a opção `Include Kotlin support`. Se houver dúvidas na criação do projeto, revisite a seção "Primeiro projeto: Hello World" do capítulo 3.

Após a criação do projeto, precisamos criar a Activity principal do nosso App. Diferente do projeto do capítulo passado em que criamos a Activity do zero (criando seu arquivo `xml`, depois o arquivo `kotlin` e, por fim, declarando-a no arquivo `AndroidManifest`), desta vez utilizaremos o menu de criação de Activity da IDE. Desta forma, a criação da tela como um todo é agilizada uma vez que a IDE cria os arquivos necessários e já registra a Activity no manifesto do aplicativo!

Em comparação com a forma anterior, há diversas vantagens de se criar todas as Activities dessa maneira, primeiro porque o processo fica agilizado e o programador ganha tempo, e segundo porque nos dá a segurança de que o programador não vai se esquecer de nenhum passo. No entanto, é importante conhecer bem esse processo porque se houver algum problema o programador saberá onde procurar.

Vamos então criar nossa Activity. Para isso, clique no menu `File > New > Activity > Empty Activity`.

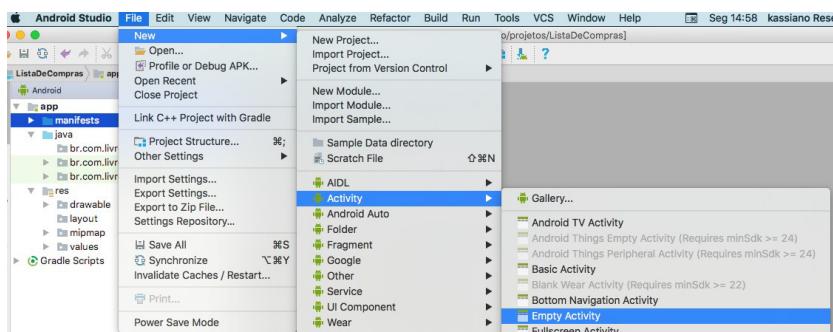


Figura 6.1: Criar Activity

Na próxima tela, mantenha o nome `MainActivity` e

`activity_main` que a IDE sugere, marque a opção `Launcher Activity` (atividade de inicialização), que indica que queremos tornar essa tela como principal no nosso aplicativo. Sendo assim, a IDE vai definir as configurações necessárias no arquivo de manifesto do aplicativo.

Deixe marcada a opção `Generate Layout File` (gerar arquivo de layout). Essa opção fará com que a IDE já crie para nós um arquivo XML de layout.

Deixe marcada também a opção `Backwards Compatibility (AppCompat)` (compatibilidade retroativa), que criará uma `Activity` do tipo `AppCompatActivity`, e não uma `Activity`. Um pouco mais adiante falarei sobre as diferenças entre a classe `Activity` e a `AppCompatActivity`.

Em `Package name` (nome de pacote) mantenha com `br.com.livrokotlin.listadecompras`. Aqui você poderia escolher um outro pacote para esse arquivo se seu projeto tiver mais de um pacote.

Em `Source Language` (linguagem do código-fonte) escolha `Kotlin`. Aqui ainda temos a opção de criar arquivos em Java e, como o Kotlin e o Java têm total interoperabilidade, conseguiríamos trabalhar com as duas linguagens no mesmo projeto.

E por fim, clique em `Finish` (finalizar).

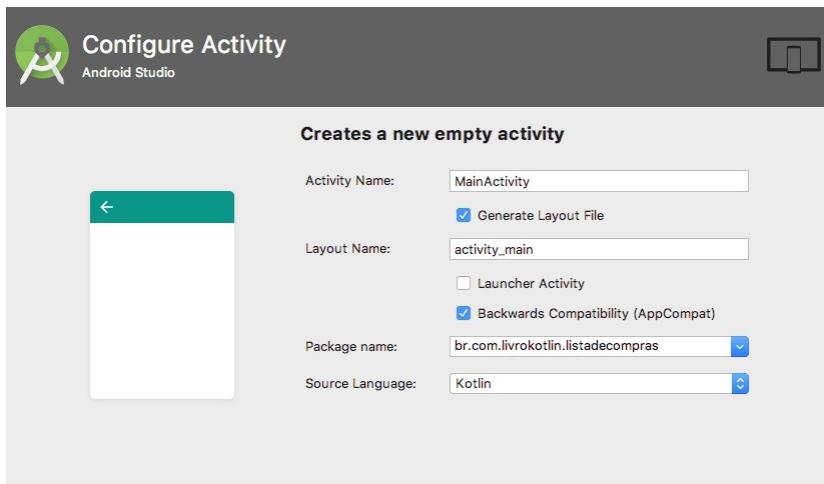


Figura 6.2: Configuração Activity

Vamos manter os nomes `MainActivity` e `activity_main` porque essa será nossa tela principal, então o nome juntando `Main` (principal) e `Activity` é bom, pois facilita a localização do arquivo posteriormente, logo a sugestão da IDE é bem-vinda.

A opção `Empty Activity` (atividade vazia) cria todo o esqueleto de uma tela vazia para trabalharmos: o arquivo Kotlin com o nome `MainActivity`, o arquivo de layout com o nome `activity_main` e também já registra a Activity no manifesto do Android!

Com isso, já temos os arquivos necessários para começar a trabalhar!

6.3 CRIANDO O LAYOUT

Agora vamos trabalhar no layout do nosso aplicativo. Para criá-lo, vamos usar os seguintes componentes visuais:

- 1 `EditText` (caixa de edição de texto) para o usuário digitar o nome do produto;
- 1 `Button` (botão) para chamar a ação de inserir o produto na lista;
- 1 `ListView` (visualização em lista) para mostrar todos os produtos cadastrados.

Desses componentes, o único que não abordamos ainda é a `ListView`. Vamos então entender um pouco mais dela para podermos usá-la. Uma `ListView` é o componente visual do Android que exibe uma lista no aplicativo. Essa lista pode exibir um texto simples ou pode ser totalmente personalizada pelo programador, isto é, podemos exibir imagens, textos e qualquer informação em uma lista.

A definição da `ListView` no arquivo é muito simples, basta adicionar a tag `<ListView />`, definir a altura, largura e o id. Veja um exemplo:

```
<ListView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/list_view"/>
```

A definição dos itens dessa lista e também a definição de como ficará o layout de cada item será feita no código em Kotlin, isso porque precisaremos de um objeto para gerenciar os itens da lista, chamado de `Adapter` (adaptador), veremos como criar um adaptador para a lista um pouco mais adiante.

Agora que temos uma noção de todos os componentes visuais que usaremos, vamos começar a montar o layout. Abra o arquivo `activity_main` localizado na pasta `res > layout`.

Neste arquivo, vamos criar um layout linear com orientação vertical para que os componentes fiquem um embaixo do outro. Também já incluirei a propriedade `padding` com valor de `16dp` para que os componentes não fiquem grudados nas laterais da tela. O código ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.livrokotlin.listadecompras.MainActivity"

    android:orientation="vertical"
    android:padding="16dp">

</LinearLayout>
```

Em seguida, vamos montar a tela dentro desse layout linear. Criaremos um componente de `EditText`, em seguida, um `Button` e, por último, uma `ListView`. O código ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.livrokotlin.listadecompras.MainActivity"

    android:orientation="vertical"
    android:padding="16dp">
```

```
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/txt_produto"  
    android:hint="Nome do produto"  
/>  
  
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="inserir"  
    android:id="@+id	btn_inserir"  
/>  
  
<ListView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/list_view_produtos"></ListView>  
  
</LinearLayout>
```

Podemos executar o aplicativo para ver como está ficando.

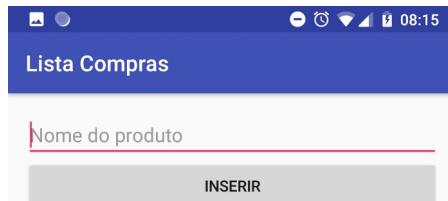


Figura 6.3: Execução do aplicativo

6.4 PROGRAMANDO O APLICATIVO

Agora vamos partir para a programação do aplicativo! Abra o

arquivo `MainActivity` localizado na pasta `java`. Você encontrará uma parte do código da Activity já pronta, que é o mínimo que uma Activity precisa para funcionar, então a IDE já nos traz pronta uma classe com a herança implementada: `class MainActivity : AppCompatActivity() { ... }`. Também nos traz o método `onCreate` implementado dentro da classe: `override fun onCreate(savedInstanceState: Bundle?) { ... }`, e dentro do método a chamada para o conteúdo da tela: `setContentView(R.layout.activity_main)`. Veja o código completo:

```
package br.com.livrokotlin.listadecompras

import android.os.Bundle
import android.support.v7.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

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

Repare que a classe `MainActivity` faz uma herança da classe `AppCompatActivity` e não da classe `Activity` como estávamos usando no capítulo anterior. A classe `AppCompatActivity` é uma classe derivada da classe `Activity`, ou seja, ela ainda possui todas as características de uma Activity, mas com algumas diferenças. A `AppCompatActivity` é uma classe de suporte que nos permite utilizar o componente `ActionBar` (barra de ações) mesmo em versões mais antigas do Android. É uma boa ideia utilizar o `AppCompatActivity` porque garantimos que nosso App se comportará da mesma maneira em uma quantidade maior de aparelhos e versões de sistema operacional.

Esse código então basicamente cria a Activity: `MainActivity` : `AppCompatActivity()` , implementa o método de criação da tela: `override fun onCreate(savedInstanceState: Bundle?)` e define o layout utilizado pela Activity: `setContentView(R.layout.activity_main)` .

Nossa tarefa agora é programar a lógica do nosso aplicativo. Não será uma lógica complexa, muito pelo contrário, nós simplesmente queremos que, após o usuário preencher o nome do produto e clicar no botão `inserir` , que este produto seja inserido na lista! Para isso, precisamos entender como as listas funcionam e como podemos inserir novos itens nelas!

A ideia geral de uma lista é bem simples, ela é um componente que exibirá dados em forma de lista. No entanto, as listas são dinâmicas, podendo exibir qualquer estrutura de dados pré-programada pelo desenvolvedor. No nosso caso, queremos que a lista seja simplesmente uma exibição de um texto, o nome do produto que foi previamente inserido pelo usuário. Mas é possível exibir imagens, botões e qualquer componente dentro de uma lista.

Criando um adaptador

Para a estrutura de `ListView` funcionar, precisamos de um objeto que fará a ligação entre o código e a lista definida na tela, que é o `Adapter` (adaptador). O `Adapter` é responsável por gerenciar a criação da lista, a exibição dos itens e também por definir o layout que será utilizado nos itens da lista. Sua função é abstrair toda a criação da lista, sendo assim, o objeto `ListView` não precisa saber como seus itens internos estão sendo montados e nem precisa saber do layout que está sendo implementado, ela só

precisa ter um adaptador, que fará todo o resto.

Esse modelo de funcionamento é muito interessante pois é por conta dos adaptadores que conseguimos personalizar qualquer lista, então se precisarmos de uma lista diferente não mexeremos no objeto de `ListView`, mas sim implementaremos um adaptador diferente. Pense em um adaptador do Android como um adaptador na vida real. Imagine que você tenha um computador com uma tomada de 3 pinos, mas você precisa ligá-lo em uma tomada de 2, como você resolveria isso? Você mudaria a tomada do seu computador, mudaria a rede elétrica ou utilizaria um adaptador de tomadas? Provavelmente, você utilizaria um adaptador. E se a rede elétrica fosse de 3 pinos, mas no padrão americano? Bastaria utilizar um adaptador diferente!

Para nossa lista precisaremos de um adaptador também. O Android tem uma classe chamada `ArrayAdapter`, que implementa um adaptador genérico, isto é, um adaptador multiuso. Para criar um `ArrayAdapter` precisaremos de algumas informações: primeiro, precisaremos definir qual informação esse adaptador vai representar, isto é, com qual tipo de dado esse adaptador trabalhará. Faremos isso inferindo o tipo do adaptador. Parece algo complicado, mas na prática é muito simples, veja o seguinte exemplo: `ArrayAdapter<String>`. Nesse código, estamos dizendo que esse adaptador representa o tipo `String`, por isso passamos `String` entre o sinal `< e >`. Desta forma, o adaptador sabe que internamente os dados com que ele vai trabalhar serão do tipo `String`. Seguindo essa lógica, podemos criar um adaptador de qualquer tipo. Por exemplo, se eu criar uma classe nova chamada `Pessoa` eu posso criar um adaptador que trabalhe com esse tipo: `ArrayAdapter<Pessoa>`.

No capítulo 5 já utilizamos o `ArrayAdapter` para criar o `Spinner` com as opções masculino e feminino .

Para a nossa lista, vamos utilizar um adaptador de `String` porque a informação que ele vai representar é um texto simples, o nome do produto.

Para criação do objeto, precisamos ainda passar alguns parâmetros pelo construtor. O primeiro parâmetro obrigatório é o contexto, que nesse caso é a própria tela, a própria Activity, então passaremos `this`, que se refere à própria classe. Ainda precisamos passar qual o layout que vamos utilizar para a montagem de cada item da lista. Aqui podemos utilizar algum modelo que o Android já possui ou criar nosso próprio modelo baseado em um arquivo `xml`.

Por enquanto, vamos utilizar um modelo pronto do Android, mas um pouco mais adiante vamos personalizar nossa lista e criar um layout personalizado! Para acessar esse modelo de layout já disponível no Android, podemos utilizar o código: `android.R.layout.simple_list_item_1`. Este é um modelo simples para exibição de um único texto, exatamente o que precisamos no momento!

O Android possui diversos layouts já definidos que podemos utilizar. Para uma lista completa consulte a documentação oficial no endereço:
<https://developer.android.com/reference/android/R.layout.html>

A declaração completa do adaptador é a seguinte:

```
val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1)
```

Ainda precisamos saber como inserir e remover itens deste adaptador. Para isso, a classe `ArrayAdapter` implementa dois métodos: `add` e `remove`. O primeiro é usado para adicionar itens na lista e o segundo para remover, sendo que ambos recebem o parâmetro do item que você deseja remover ou adicionar. Veja um exemplo:

```
val item = "Feijão"  
adapter.add(item)
```

E para remover um item existente:

```
adapter.remove(item)
```

Existem alguns casos em que você vai precisar criar um adaptador já com itens inseridos e trazer uma lista na tela já com informações. Imagine, por exemplo, que você tenha uma lista de contatos, que estão guardados em um banco de dados, e você deseja exibi-los em uma `ListView`. Para isso, você lerá o banco de dados para buscar os contatos e criará um adaptador já com esses contatos preenchidos, assim quando o usuário abrir a tela já

aparecerá uma lista exibindo informações. Para esses casos, podemos passar um terceiro parâmetro na criação do adaptador, que deve ser uma lista com as informações. Veja o exemplo a seguir da criação de um Adapter sendo criado com uma lista de nomes de pessoas:

```
//lista com contatos  
var contatos = mutableListOf("Mariana", "João", "Francisco", "Vitória")  
  
//criação do adaptador passando a lista como o terceiro parâmetro  
val produtosAdapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, contatos )
```

Agora que já criamos nosso adaptador, falta ligá-lo na ListView. O processo é muito simples. A ListView possui um atributo chamado adapter, que é justamente para definição do adaptador da lista. Pensando no exemplo de que eu falei acima, é como se o atributo adapter da ListView fosse o plug da tomada em que ligaremos o adaptador.

Veja como é feita a ligação do adaptador na lista:

```
list_view_produtos.adapter = produtosAdapter
```

Agora, o código completo do método onCreate ficará assim:

```
package br.com.livrokotlin.listadecompras  
  
import android.os.Bundle  
import android.support.v7.app.AppCompatActivity  
import android.widget.ArrayAdapter  
  
import kotlinx.android.synthetic.main.activity_main.*  
  
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)
```

```
//Implementação do adaptador
val produtosAdapter = ArrayAdapter<String>(this, android.
R.layout.simple_list_item_1)

//definindo o adaptador na lista
list_view_produtos.adapter = produtosAdapter
}

}
```

ELIMINANDO O `FINDVIEWBYID`

Perceba que aqui eu não utilizei o método `findViewById` para acessar a `list_view_produtos`. Essa mágica pode ser feita graças ao plugin `kotlin-android-extensions`, que já vem configurado no projeto quando estamos trabalhando com o Kotlin. Você pode conferir se ele está implementado no arquivo `build.gradle` do módulo, que deve conter a seguinte linha logo abaixo da configuração do plugin do Kotlin:

```
apply plugin: 'kotlin-android-extensions'
```

Confira também o arquivo `build.gradle` do projeto. Na seção de `dependencies` deste arquivo deve ter a seguinte linha:

```
classpath "org.jetbrains.kotlin:kotlin-android-extensions:$kotlin_version"
```

Esse plugin faz um mapeamento dos componentes declarados no `xml` e seus respectivos Ids e nos dá acesso a eles como se fossem propriedades da Activity. E você deve ter percebido também um novo `import`:

```
kotlinx.android.synthetic.main.activity_main.*
```

É através desse `import` que todas as propriedades serão mapeadas. Geralmente, a IDE fará esse `import` automaticamente. Com isso, não nos preocuparemos mais com métodos `findViewById` nas Activities.

Adicionando items ao adaptador

Agora que temos o adaptador criado e já devidamente definido na lista, precisamos agora tratar o clique do botão para adicionar novos itens na lista. De fato, não adicionamos os itens diretamente na `ListView`, como é o adaptador que gerencia os itens dentro de uma `ListView` vamos adicioná-los no adaptador e ele será responsável por notificar a lista de que existem novos itens a serem exibidos.

No nosso caso, queremos que os itens sejam adicionados somente quando o usuário clica no botão e o item que será adicionado deverá ser o texto que o usuário digitou na caixa `txt_produto`.

Vamos primeiro tratar o clique do botão, e para isso vamos implementar um `listener` (ouvinte) no botão. Já usamos este método no projeto do capítulo anterior, mas vamos relembrar seu funcionamento:

Os `listeners` (ou métodos ouvintes) são utilizados quando queremos tratar alguma ação de algum componente. O exemplo mais clássico é do próprio botão, a ação mais comum em um botão é o clique, então conseguimos monitorá-lo por meio de um `listener` de click do botão!

Para conhecer mais sobre os eventos de entrada do Android, consulte a documentação oficial através do link:
<https://developer.android.com/guide/topics/ui/ui-events.html?hl=pt-br>

Como queremos que o item seja adicionado somente quando o usuário clica no botão, devemos então monitorar este evento, e para isso podemos utilizar o método `setOnClickListener` do botão. Este método é acionado toda vez que ocorre uma ação de clique no botão. Veja como essa implementação pode ser feita:

```
btn_inserir.setOnClickListener {  
}
```

Dentro das chaves {} é que colocaremos o código que será executado a cada vez que esse botão for clicado. Precisamos primeiro pegar o nome do produto que o usuário digitou na caixa de texto e precisaremos de uma variável para guardar esse texto. Podemos fazer assim:

```
val produto = txt_produto.text.toString()
```

Em seguida, usaremos o método `add` para inserir a variável `produto` no adaptador:

```
adapter.add(produto)
```

Inserimos a variável `produto` no adaptador porque ela guarda o nome do produto inserido pelo usuário!

Veja como ficará o código completo do botão:

```
btn_inserir.setOnClickListener {  
    val produto = txt_produto.text.toString()  
    adapter.add(produto)  
}
```

Agora temos todas as partes do aplicativo programada, fizemos o adaptador da lista, tratamos o clique do botão e adicionamos novos itens na lista! Vamos ver como ficará o código completo dessa Activity:

```
package br.com.livrokotlin.listadecompras  
  
import android.os.Bundle  
import android.support.v7.app.AppCompatActivity  
import android.widget.ArrayAdapter  
import kotlinx.android.synthetic.main.activity_main.*  
  
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        //Implementação do adaptador  
        val produtosAdapter = ArrayAdapter<String>(this, android.  
R.layout.simple_list_item_1)  
  
        //definindo o adaptador na lista  
        list_view_produtos.adapter = produtosAdapter  
  
        //definição do ouvinte do botão  
        btn_inserir.setOnClickListener {  
            //pegando o valor digitado pelo usuário  
            val produto = txt_produto.text.toString()  
  
            //enviado o item para a lista  
            produtosAdapter.add(produto)  
        }  
    }  
}
```

```
        }  
    }  
}
```

Chegou a hora de testar nosso App e ver como ele ficou! Execute o aplicativo e veja o resultado final:

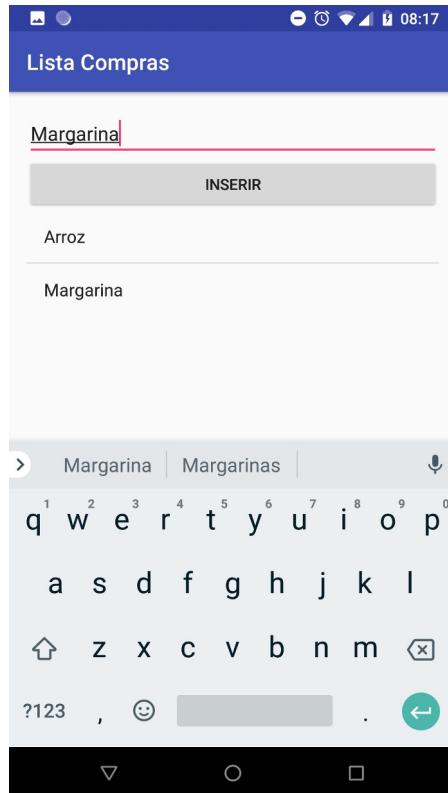


Figura 6.4: Aplicativo em execução

Faça alguns testes e veja se a lista é atualizada assim que um novo item é inserido!

6.5 PEQUENAS MELHORIAS

A base do nosso aplicativo já está funcionando, mas existem alguns pequenos ajustes que podemos fazer para tornar a experiência do usuário ainda melhor. A primeira melhoria que podemos fazer é verificar se a caixa não está vazia antes de inserir na lista. Pense na possibilidade de o usuário não digitar nada na caixinha e mesmo assim clicar no botão, nesse caso seria inserido um item vazio na lista.

Para fazer essa verificação, podemos usar o método `isNotEmpty` (não está vazio), que é um método booleano, ou seja, seu retorno será ou `true` (verdadeiro) ou `false` (falso). Caso o método nos retorne um valor verdadeiro, significa que a caixa não está vazia, ou seja, o usuário digitou alguma coisa então podemos adicionar o item na lista; caso contrário, não adicionaremos nada na lista e o avisaremos de que deve preencher um valor.

Vamos implementar a verificação no código do botão:

```
//verificando se o usuário digitou algum valor
if (produto.isNotEmpty()) {
    //enviado o item para a lista
    produtosAdapter.add(produto)
}
```

Essa verificação evita que seja inserido um item vazio na lista, mas ainda não avisa ao usuário de que ele deve digitar um valor. Para isso, incluiremos o comando `else` ao `if` para tratar essa situação. Precisamos também avisar que ele precisa preencher um valor naquele campo, para isso vamos utilizar a propriedade `error` da `EditText`. Ela exibe uma mensagem de erro na caixa de texto, e será personalizada pelo valor passado a ela em formato de texto. No nosso caso, avisaremos ao usuário de que este campo

precisa ser preenchido com algum valor, então poderíamos fazer o seguinte código: `txt_produto.error = "Preencha um valor"`. Desta forma, o usuário saberá qual ação ele deve tomar. Veja como fica a implementação no código:

```
if (produto.isNotEmpty()) {  
    //enviado o item para a lista  
    produtosAdapter.add(produto)  
}  
else{  
    txt_produto.error = "Preencha um valor"  
}
```

O resultado final ficará assim:

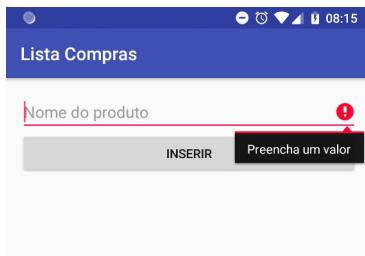


Figura 6.5: Exibição de mensagem de erro

Outra melhoria que poderíamos fazer é limpar o campo após a inserção do produto. Do jeito que está, quando inserimos um novo item na lista, o campo com o nome do produto continua com o nome lá e, se o usuário quiser inserir outro produto, ele tem que apagar o valor anterior e inserir um novo valor. Esse comportamento pode ser evitado se, quando o item fosse inserido na lista, nós apagarmos o texto da caixinha através do código.

Para isso, podemos usar o método `clear` (limpar), cuja função é simplesmente limpar um conteúdo de texto. Poderíamos também simplesmente definir uma `String` vazia a caixa de texto,

no entanto, o método `clear` é um pouco mais elegante no código. Podemos usá-lo da seguinte maneira:
`txt_produto.text.clear()` e pronto, ele vai limpar todo o conteúdo da caixa de texto deixando-a pronta para receber um novo valor. Veja o código completo:

```
if (produto.isEmpty()) {  
    //enviado o item para a lista  
    produtosAdapter.add(produto)  
  
    //limpando a caixa de texto  
    txt_produto.text.clear()  
  
} else{  
    txt_produto.error = "Preencha um valor"  
}
```

6.6 REMOVENDO UM ITEM DA LISTA

Nosso App já está bem legal, o usuário consegue inserir vários itens na lista, já colocamos algumas restrições e melhorias para a experiência do usuário, mas ainda não pensamos em uma questão: e se o usuário quiser remover um item da lista? É possível?

Do jeito que o aplicativo está, atualmente não é possível remover um item da lista, somente adicionar novos itens! O que precisamos para implementar mais essa funcionalidade?

Precisamos de duas coisas: primeiro, temos que saber como se remove um item da lista, qual comando faz isso? Segundo, devemos pensar em que ação o usuário fará para remover um item da lista.

Remover um item da lista não é nada complicado, basta usarmos o método `remove` do adaptador passando como

parâmetro o objeto que desejamos remover da lista. Lembre-se, não manipulamos os itens da lista diretamente, manipulamos o adaptador e ele refletirá as alterações na lista!

O próximo ponto que devemos pensar é na ação que o usuário executará para remover o item da lista, que pode ser a de `longclick` (clique longo). Essa ação é executada quando o usuário clica em um item da lista e segura por alguns segundos. Podemos definir que, quando o usuário clicar em um item da lista e segurar por alguns segundos, esse item será removido.

Para utilizar essa ação devemos implementar mais um método ouvinte, dessa vez na `ListView`. O método ouvinte que utilizaremos é o `setOnItemClickListener`, acionado quando um item da lista recebe um clique longo.

Na implementação desse método, é obrigatório que sejam declaradas 4 variáveis: `adapterView`, `view`, `position` e `id`. Elas já vão estar preenchidas toda vez que um clique longo ocorrer. Através delas, o programador saberá em qual posição da lista exatamente ocorreu o clique (`position`), qual o adaptador da lista (`adapterView`), qual o id da linha (`id`) e qual a visualização do item clicado (`view`). Veja a seguir a função de cada uma dessas variáveis:

- `adapterView` : a lista na qual o item foi clicado;
- `view` : a `view` do item que foi clicado;
- `position` : a posição desse item na lista;
- `id` : Um número que corresponde ao ID da linha que item se encontra

E, por fim, o método deve retornar um booleano indicando se

o clique longo foi realizado ou não. Sua implementação no código será assim:

```
list_view_produtos.setOnItemLongClickListener{ adapterView: AdapterView<*>, view: View, position: Int, id: Long ->

    //retorno indicando que o click foi realizado com sucesso
    true
}
```

A notação `->` faz parte da sintaxe do Kotlin e é usada em alguns contextos diferentes. Esta sintaxe é similar à de expressões lambda do Java. Aqui no Kotlin você encontrará essa notação em expressões lambdas e também na expressão `when`.

Em expressões `when` utilizamos a notação `->` para separar a condição do bloco de resultado, veja um exemplo:

```
when (numero){

    1 -> println("número é 1")
    2 -> println("número é 2")
    else -> println("número não é 1 nem 2")

}
```

Em expressões lambdas, utilizamos a notação `->` para separar os parâmetros do corpo da função. O método `setOnItemLongClickListener` está recebendo uma expressão lambda que recebe os parâmetros: `adapterView`, `view`, `position` e `id`, e logo após os parâmetros utilizamos a notação `->` para indicar o corpo da função.

Ainda precisamos resolver um problema para poder implementar essa lógica: para remover um item da lista vamos

utilizar o método `remove` do adaptador, no entanto esse método precisa do objeto que ele vai remover, ou seja, como parâmetro, precisamos passar exatamente o item da lista em que o usuário clicou!

Para resolver isso, podemos utilizar outro método do adaptador, o método `getItem`. Ele retorna um item específico da lista de acordo com a posição passada.

Por exemplo, se quiser retornar o primeiro item da lista, basta passar a posição 0: `getItem(0)`; para o segundo item, a posição 1: `getItem(1)` e assim por diante. Então, precisaremos saber em qual item exato o usuário clicou, mas isso não será um problema, porque o método `setOnItemClickListener` já nos passa essa informação através da variável `position`!

Tendo essas informações em mãos fica fácil. Primeiro vamos acessar exatamente o item clicado: `adapter.getItem(position)`. Vamos também guardar essa informação em uma variável pois precisaremos dela para utilizar no método `remove`, então o código ficaria: `val item = adapter.getItem(i)`. Na sequência, chamaremos o método `remove` do adaptador: `adapter.remove(item)`. Veja a implementação completa do código:

```
list_view_produtos.setOnItemClickListener{ adapterView: AdapterView<*>, view: View, position: Int, id: Long ->

    //buscando o item clicado
    val item = produtosAdapter.getItem(position)

    //removendo o item clicado da lista
    produtosAdapter.remove(item)

    //retorno indicando que o click foi realizado com sucesso
```

```
        true  
    }
```

Não se esqueça de que a definição desse método estará logo após a definição do método ouvinte do botão. Teste o aplicativo e veja se é possível adicionar e remover itens da lista!

Veja o código completo da `MainActivity` :

```
package br.com.livrokotlin.listadecompras  
  
import android.os.Bundle  
import android.support.v7.app.AppCompatActivity  
import android.widget.ArrayAdapter  
import kotlinx.android.synthetic.main.activity_main.*  
  
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        //Implementação do adaptador  
        val produtosAdapter = ArrayAdapter<String>(this, android.  
R.layout.simple_list_item_1)  
  
        //definindo o adaptador na lista  
        list_view_produtos.adapter = produtosAdapter  
  
        //definição do ouvinte do botão  
        btn_inserir.setOnClickListener {  
  
            //pegando o valor digitado pelo usuário  
            val produto = txt_produto.text.toString()  
  
            if (produto.isNotEmpty()) {  
                //enviado o item para a lista  
                produtosAdapter.add(produto)  
  
                //limpando a caixa de texto  
                txt_produto.text.clear()  
            }  
        }  
    }  
}
```

```
        }else{
            txt_produto.error = "Preencha um valor"
        }

    }

list_view_produtos.setOnItemLongClickListener{ adapterView: AdapterView<*>, view: View, position: Int, id: Long ->

    //buscando o item clicado
    val item = produtosAdapter.getItem(position)

    //removendo o item clicado da lista
    produtosAdapter.remove(item)

    //retorno indicando que o click foi realizado com sucesso
    true
}
}
```

6.7 RESUMINDO

Neste capítulo, aprendemos o básico de como se trabalhar com listas. Esse conteúdo é muito importante porque a grande maioria dos aplicativos utiliza listas em algum momento. Por enquanto, essa lista de compras está bem simples, mas no próximo capítulo vamos fazer a lista de compras 2.0!

Vamos trabalhar ainda em cima desse mesmo projeto e vamos implementar modificações significativas. Primeiro, vamos personalizar o layout dos itens da lista e, em seguida, guardaremos os dados em um banco de dados. Da forma como está agora, os dados estão salvos somente na memória RAM, isso quer dizer que assim que o aplicativo for fechado todos os dados serão perdidos.

Para resolver isso, devemos começar a guardar as informações em um banco de dados. Espero você no próximo capítulo!

CAPÍTULO 7

LISTA DE COMPRAS 2.0

Neste capítulo, vamos implementar algumas alterações no projeto feito no capítulo anterior e fazer a lista de compras 2.0! Faremos algumas atualizações no layout do App, que ficou muito simplificado, depois daremos um upgrade na lista. Em vez de simplesmente inserirmos o nome do produto vamos inserir também a quantidade, o valor do produto e uma foto!

Além disso, incluiremos um texto para exibir o total da compra. A imagem a seguir mostra o resultado final do aplicativo para você ver como ficará o resultado ao final deste capítulo:



Figura 7.1: Lista de compras final

7.1 PLANEJAMENTO

Vamos começar como sempre pelo planejamento. Nós já temos um aplicativo em funcionamento então o planejamento é somente das alterações. A primeira alteração que podemos fazer é modificar a lista para receber não só a informação de nome do produto, mas também a informação da quantidade, valor e foto.

Pensando desta maneira conseguimos enxergar que precisaremos de mais campos para o usuário inserir todas essas informações. Como agora o usuário precisará cadastrar mais coisas, talvez deixar tudo na mesma tela não seja uma boa ideia, pois a tela ficaria muito poluída. Que tal então dividirmos o aplicativo em duas telas?

Na tela principal, aparecerá a lista com todos os itens inseridos, o total da compra e um botão que dará acesso a uma outra tela somente de inserção de informações. Veja como ficaria o resultado final:

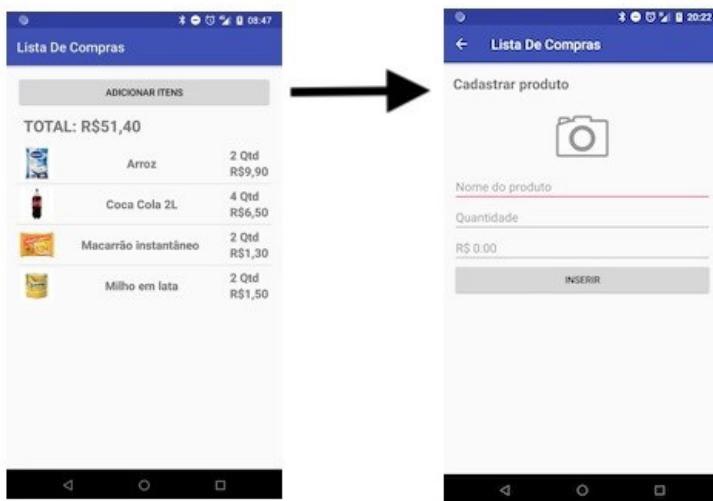


Figura 7.2: Telas do aplicativo

Desta forma, separamos as responsabilidades: uma tela servirá somente para exibir informações e a outra tela servirá somente para cadastrar informações.

Vamos precisar pensar também no layout da lista. Hoje a lista mostra simplesmente o nome do produto, uma única `String`, mas agora com novas informações a lista precisará se adequar a um novo layout. Aqui entra uma grande mudança em relação à lista feita no capítulo anterior, lá nossa lista era simples, agora vamos montar uma lista personalizada!

Quando me refiro a listas personalizadas aqui no Android, quero dizer que podemos personalizar o layout da lista com quantas informações desejarmos! Podemos utilizar imagens, tamanhos de fonte diferente nos textos, cores diferentes, planos de fundo etc. Enfim, quando se trata de personalização o Android nos dá total liberdade. Para o nosso caso, vamos personalizar a lista para mostrar a imagem do produto, seu nome, a quantidade e o valor.

Precisamos levar em consideração também que anteriormente quando estávamos guardando somente o nome do produto uma variável do tipo `String` dava conta, mas e agora, como faremos para agrupar todas essas informações? Para resolver esse problema podemos recorrer à Programação Orientada a Objetos! Que tal criarmos uma classe que tenha todos os atributos de que precisaremos, `nome`, `quantidade`, `valor` e `imagem`?

Essa solução se mostra muito adequada pois, além de resolver nosso problema no momento, nos dá a liberdade de evoluções futuras. Por ora, temos essas informações, mas e se depois quisermos guardar mais uma informação? Tendo uma classe para isso, bastaria criar um novo atributo!

Vamos ao trabalho!

7.2 CRIANDO UMA NOVA ACTIVITY DE CADASTRO

Vamos começar separando as telas. Da forma como está agora, o aplicativo faz a inserção e a visualização das informações tudo na mesma tela. Vamos mudar isso. O primeiro a se fazer é criar uma nova Activity , que será a tela de inserção de dados. Deixaremos a MainActivity para mostrar a lista, assim, quando o usuário abrir o aplicativo, ele já vê direto a lista, e se ele quiser cadastrar um novo item ele poderá acessar a tela de cadastro.

Clique com o botão direito em cima da pasta `java` e escolha a opção `New > Activity > Empty Activity` . Nomeie o arquivo como `CadastroActivity` e clique em `Finish` .

Isso vai criar uma nova Activity. Lembre-se de que, fazendo dessa forma, a IDE criará o arquivo Kotlin da Activity `CadastroActivity` , o arquivo `xml activity_cadastro` e também registrará essa Activity no arquivo `AndroidManifest` .

Agora vamos montar o layout dessa tela. O layout final deve ficar parecido com a imagem a seguir:



Figura 7.3: Layout cadastro

Podemos identificar pela imagem que esse layout pode ser feito com um layout linear e orientação vertical, pois os componentes estão um embaixo do outro. Podemos identificar também que precisaremos de um `ImageView` para exibir a foto, três `EditText` (para nome, quantidade e valor) e um botão no final para efetuar o cadastro. Temos também um `TextView` com o texto "Cadastrar produto" para informar o usuário do intuito dessa tela.

Abra o arquivo `activity_cadastro` para montarmos esse layout. A montagem não tem nada de novo, vamos simplesmente definir todos os componentes em um `LinearLayout` com orientação vertical.

Seguem as configurações de cada componente do layout:

- `LinearLayout` :

- android:orientation="vertical"
- android:padding="16dp"
- TextView :
 - android:text="Cadastrar produto"
 - android:textStyle="bold" (Para deixar o texto em negrito)
 - android:textSize="21sp"
 - android:layout_marginBottom="15dp"
- ImageView :
 - android:layout_width="100dp"
 - android:layout_height="100dp"
 - android:src="@android:drawable/ic_menu_camera" (ícone de câmera do Android)
 - android:id="@+id/img_foto_produto"
 - android:layout_gravity="center_horizontal"
- EditText :
 - android:layout_width="match_parent"
 - android:layout_height="wrap_content"
 - android:id="@+id/txt_produto"
 - android:hint="Nome do produto"
- EditText :
 - android:layout_width="match_parent"
 - android:layout_height="wrap_content"
 - android:id="@+id/txt_qtd"
 - android:inputType="number" (Para exibir o teclado numérico)
 - android:hint="Quantidade"
- EditText :
 - android:layout_width="match_parent"
 - android:layout_height="wrap_content"

- android:id="@+id/txt_valor"
- android:inputType="numberDecimal" (Para exibir o teclado numérico com opção de decimais)
- android:hint="R\$ 0.00"
- Button :
 - android:layout_width="match_parent"
 - android:layout_height="wrap_content"
 - android:text="inserir"
 - android:id="@+id btn_inserir"

A propriedade `inputType`, que foi usada nos `EditText`, define o tipo de entrada da caixa de texto. Na prática, o Android exibirá um tipo de teclado específico de acordo com o `inputType` escolhido. Por exemplo, se você tem um `EditText` em que o usuário vai digitar um endereço de e-mail, você deve especificar um `inputType` igual a `textEmailAddress`, assim o teclado que se abrirá para o usuário será um teclado que facilita a entrada de e-mails.

Existem diversos tipos de `inputType`. Os mais comuns e mais úteis no dia a dia são:

- `number` : Números inteiros
- `numberDecimal` : Números decimais
- `numberSigned` : Números com sinal
- `textPassword` : Senha em texto
- `numberPassword` : Senha numérica
- `phone` : Número de telefone
- `date` : Datas

Para uma lista completa sobre os `inputTypes` consulte a documentação oficial no link:
<https://developer.android.com/reference/android/text/InputType.html>

Vamos começar a implementar o código dessa tela. Começaremos pelo *container* principal, que será um layout linear, e depois vamos adicionando os componentes dentro dele:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".br.com.livrokotlin.listadecompras.CadastroActivity"
    android:orientation="vertical"
    android:padding="16dp">

</LinearLayout>
```

Agora vamos adicionando os componentes. Primeiro, vamos colocar um `TextView` com um título `Cadastrar Produto` para indicar ao usuário que esse é um formulário de cadastro de produto:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Cadastrar produto"
    android:textStyle="bold"
    android:textSize="21sp"
    android:layout_marginBottom="15dp" />
```

Em seguida, vamos utilizar o componente `ImageView` para exibir uma imagem padrão, que depois o usuário usará para personalizar a imagem do produto.

O componente ImageView

O componente `ImageView` tem como objetivo exibir uma imagem ao usuário, podendo ser um `Bitmap` ou um arquivo de imagem da pasta `drawable`. Na pasta `res` do projeto, há uma pasta chamada `drawable`, que é onde colocamos imagens para usar no nosso App. Então, para exibir uma imagem no `ImageView` primeiro devemos ter essa imagem na pasta `drawable`. Você pode fazer isso simplesmente copiando a imagem e colando na pasta pelo próprio Android Studio. Em seguida, você define a propriedade `src` do `ImageView` referenciando essa imagem, veja um exemplo:

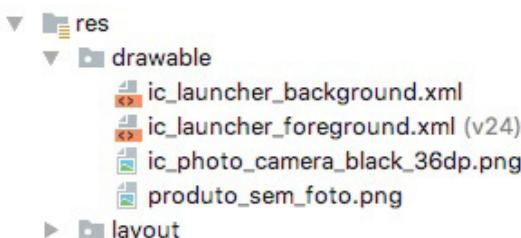


Figura 7.4: Pasta drawable

Temos 4 arquivos dentro da pasta, 2 PNGs e 2 XMLs. Os 4 são arquivos de imagens, mas vamos nos focar nos de tipo PNG, que são imagens mais comuns de trabalharmos no dia a dia. Vamos supor que eu queira exibir em um `ImageView` a imagem `produto_sem_foto.png`. Para isso, eu definiria a propriedade `src` do `ImageView` da seguinte maneira:

`android:src="@drawable/produto_sem_foto"` . Veja que eu não preciso passar a extensão da imagem, somente o nome dela. A tag completa desse exemplo ficaria assim:

```
<ImageView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:src="@drawable/produto_sem_foto" />
```

Voltando ao nosso projeto, vamos implementar também um `ImageView` abaixo do `TextView` do título. Nesse `ImageView`, vamos definir a imagem de um ícone de câmera para indicar ao usuário que ele deve carregar a imagem do produto nesse campo. Para isso, você deveria ter a imagem do ícone de câmera em sua pasta `drawable` , mas a plataforma Android tem alguns ícones "embutidos" que você pode usar simplesmente utilizando `@android:drawable/` mais o nome do recurso. O ícone que vamos utilizar se chama `ic_menu_camera` . Vamos montar um `ImageView` que exiba esse ícone:

```
<ImageView  
    android:layout_width="100dp"  
    android:layout_height="100dp"  
    android:src="@android:drawable/ic_menu_camera" />
```

Deixei um tamanho fixo de `100dp` por `100dp` pois acredito ser um tamanho suficiente para esse ícone, mas você pode testar outros valores e ver como fica a visualização. Vou adicionar um `id` e também deixar essa imagem centralizada:

```
<ImageView  
    android:layout_width="100dp"  
    android:layout_height="100dp"  
    android:src="@android:drawable/ic_menu_camera"  
    android:id="@+id/img_foto_produto"  
    android:layout_gravity="center_horizontal"/>
```

Agora vamos colocar os `EditText`s de entrada de dados, vamos colocar um para o nome do produto, um para a quantidade e um para o valor:

```
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/txt_produto"  
    android:hint="Nome do produto"/>  
  
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/txt_qtd"  
    android:inputType="number"  
    android:hint="Quantidade"/>  
  
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/txt_valor"  
    android:inputType="numberDecimal"  
    android:hint="R$ 0.00"/>
```

Por fim, vamos incluir um botão para salvar os dados:

```
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="inserir"  
    android:id="@+id/btn_inserir"/>
```

Confira agora como ficará o código completo:

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    tools:context="br.com.livrokotlin.listadecompras.CadastroActivity"
```

```
    android:orientation="vertical"
    android:padding="16dp"

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Cadastrar produto"
        android:textStyle="bold"
        android:textSize="21sp"
        android:layout_marginBottom="15dp" />

    <ImageView
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@android:drawable/ic_menu_camera"
        android:id="@+id/img_foto_produto"
        android:layout_gravity="center_horizontal"
    />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txt_produto"
        android:hint="Nome do produto"
    />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txt_qtd"
        android:inputType="number"
        android:hint="Quantidade"
    />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txt_valor"
        android:inputType="numberDecimal"
        android:hint="R$ 0.00"
    />

    <Button
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:text="inserir"
        android:id="@+id/btn_inserir"
    />

</LinearLayout>
```

Agora vamos já criar o esquema de navegação entre as telas, e para isso precisamos definir a hierarquia de telas. Sabemos que a tela principal será a `MainActivity` e a `CadastroActivity` será chamada a partir da `main`, então podemos dizer que a `CadastroActivity` é uma tela filha da `MainActivity`. Isso é importante para podermos configurar a hierarquia de telas corretamente, pois o Android se baseará nessas informações para criar o esquema de navegação.

Para definirmos que a `CadastroActivity` será filha da `MainActivity`, temos que incluir uma *meta tag* no `AndroidManifest`. Essa *meta tag* é simplesmente uma configuração do aplicativo que é definida por um `name` e um `value`. Aqui vamos definir uma *meta tag* com `name android.support.PARENT_ACTIVITY`, o que vai dizer ao sistema operacional que essa Activity é filha de outra Activity. Devemos dizer de quem ela é filha através da propriedade `value`, nesse caso ela será filha da `MainActivity`.

A definição dessa *meta tag* ficará assim:

```
<meta-data
    android:name="android.support.PARENT_ACTIVITY"
    android:value=".MainActivity" />
```

Inclua essa tag dentro da tag da Activity de cadastro no arquivo `AndroidManifest`.

A tag da activity ficará assim:

```
<activity android:name=".CadastroActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity" />
</activity>
```

Com essa configuração, o Android saberá qual o nosso esquema de navegação e então não precisaremos reinventar a roda, ele próprio já criará um botão de voltar na tela de cadastro que volta para a tela principal!

Agora precisaremos programar como essa tela de cadastro será aberta. No nosso layout inicial, pensamos em incluir um botão na tela principal e, ao clicar nele, a tela de cadastro se abrirá. Vamos manter essa ideia e programar esse botão.

Abra o arquivo `activity_main` para definirmos o botão no `xml` e aproveitar para ajeitar o layout dessa tela.

Primeiro, vamos fazer uma limpeza nesse arquivo, alguns itens não precisam mais estar aqui porque estão agora na tela de cadastro. Essa tela terá agora somente um botão, um texto para mostrar o total e a lista. Então vamos remover daqui os componentes `EditText` e `Button`, pois não são mais necessários nessa Activity. Deixe somente o `LinearLayout` e a `ListView`, seu arquivo ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.livrokotlin.listadecompras.MainActivity"

    android:orientation="vertical"
```

```
    android:padding="16dp">

    <ListView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/list_view_produtos"></ListView>

</LinearLayout>
```

Agora, adicione uma `TextView` em cima da lista. Essa `TextView` exibirá uma somatória com o total dos itens inseridos na lista.

Veja o código:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="TOTAL: 0,00"
    android:textSize="24sp"
    android:textStyle="bold"
    android:padding="10dp"
    android:id="@+id/txt_total"
/>
```

Em cima dessa `TextView`, adicione um botão com texto `Adicionar itens`, que será usado para direcionar o usuário à tela de cadastro. Veja o código do botão:

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Adicionar itens"
    android:id="@+id/btn_adicionar"
/>
```

Certo, agora já temos o layout dessa tela ajustado. No entanto, com essas modificações de layout o nosso código parou de compilar porque alguns componentes não existem mais na `MainActivity`, já que foram migrados para a

`CadastroActivity`. Vamos então ajustar o código.

Abra o arquivo `MainActivity` e localize a seguinte linha de código:

```
btn_inserir.setOnClickListener{  
    ...
```

Essa é a definição do *listener* do botão inserir, mas não faremos mais essa ação aqui, isso tudo ficará na tela de cadastro. Vamos transferir esse pedaço de código para lá.

Selecione todo o bloco desse listener e recorte (`Ctrl + X`). Com esse pedaço de código recortado, abra o arquivo `CadastroActivity` e cole-o no método `onCreate`.

Você vai perceber que a linha `produtosAdapter.add(produto)` começará a apresentar um erro, isso porque o adaptador não está nessa tela! Por ora, vamos apagar essa linha, faremos outra estratégia para enviar a informação cadastrada ao adaptador.

O código da `CadastroActivity` ficará assim:

```
import android.os.Bundle  
import android.support.v7.app.AppCompatActivity  
import kotlinx.android.synthetic.main.activity_cadastro.*  
  
class CadastroActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_cadastro)  
  
        //definição do ouvinte do botão  
        btn_inserir.setOnClickListener {
```

```
//pegando o valor digitado pelo usuário  
val produto = txt_produto.text.toString()  
  
if (produto.isNotEmpty()) {  
    //enviado o item para a lista  
  
    //limpando a caixa de texto  
    txt_produto.text.clear()  
  
} else{  
    txt_produto.error = "Preencha um valor"  
}  
  
}  
}  
}
```

Atenção ao:

```
import kotlinx.android.synthetic.main.activity_cadastro.*
```

Como estamos fazendo um `Ctrl + X`, é possível que a IDE faça esse import automaticamente referenciando a `activity_main`. Se for o caso, mude para `activity_cadastro`.

Agora volte à `MainActivity`. Vamos implementar o código do botão que abrirá a tela de cadastro. Para isso, dentro do método `onCreate` adicione um listener para o botão `btn_adicionar`. Dentro dele, precisaremos executar um comando para abrir a tela de cadastro.

Iniciando uma nova atividade (startActivity)

No Android podemos usar a função `startActivity` (inicia uma atividade) para abrir uma nova tela do nosso App. Tradicionalmente, o método `startActivity` recebe por parâmetro uma `Intent` que pode ser explícita ou implícita. Uma `Intent` implícita significa que deixaremos o próprio Android decidir qual Activity ele vai abrir. Por exemplo, para abrir uma URL no browser poderíamos fazer o seguinte código:

```
//criando uma Intent implícita para abrir o google.com  
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("http://google.  
com"))  
  
//iniciando a atividade  
startActivity(intent)
```

Nesse código, não estamos especificando qual a Activity que vai abrir a URL. O próprio Android vai resolver qual Activity abrir. Como nesse caso é uma URL, a Activity que se abrirá será a do browser.

Quando criamos uma `Intent` explícita, dizemos exatamente qual activity queremos abrir, nesse caso o objeto `Intent` deve receber dois parâmetros na sua criação: Um objeto `Context` e a `class` da activity que desejamos abrir. No nosso App gostaríamos de abrir explicitamente a tela `CadastroActivity`, então podemos criar uma `Intent` assim:

```
//Criando a Intent explícita  
val intent = Intent(this, CadastroActivity::class.java)  
  
//iniciando a atividade  
startActivity(intent)
```

Essa sintaxe `CadastroActivity::class.java` é um pouco estranha, não? Realmente, o que acontece aqui é que o objeto `Intent` recebe um tipo `Class` que é um tipo do Java e por isso

temos que passar desta forma. Vamos ver mais sobre `Intents` ainda neste capítulo.

Vamos fazer esse código dentro do listener do `btn_adicionar` :

```
btn_adicionar.setOnClickListener {  
    //Criando a Intent explícita  
    val intent = Intent(this, CadastroActivity::class.java)  
  
    //iniciando a atividade  
    startActivity(intent)  
}
```

Agora você pode testar o aplicativo para ver o que fizemos até aqui. A navegação entre as telas deve estar funcionando, e o layout de ambas as telas também já deve estar pronto.

7.3 PERSONALIZANDO A LISTVIEW

Vamos agora fazer a personalização da lista. Diferente da outra lista que só exibia um texto, agora teremos um layout bem mais sofisticado, mas para isso funcionar precisaremos fazer várias modificações no nosso código. Vamos fazer por partes.

O primeiro a se fazer é montar o layout dos itens. Não fizemos isso ainda porque na primeira versão deste projeto nós usamos um layout padrão para uma exibição de texto, que o próprio Android disponibiliza. Mas agora não, como nossa lista é muito particular, precisaremos montar o layout dos itens. Para isso, precisaremos de um arquivo `xml` que terá a definição do layout e posteriormente o usaremos no adaptador.

Pense nesse arquivo que criaremos como um molde que todos

os itens seguirão. O layout que precisamos montar é para obter o seguinte resultado:

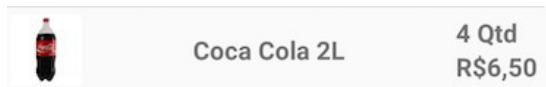


Figura 7.5: Item da lista

Perceba que nesse layout tem uma imagem à esquerda, um texto centralizado, e à direita dois textos, um embaixo do outro. Esse é o layout a que precisamos chegar e, uma vez construído, todos os itens da lista seguirão o mesmo padrão.

Então vamos criar um arquivo `xml` para montar esse layout. Clique com o botão direito na pasta `layout` e escolha a opção `New > Layout Resource file`. Na janela que se abrir preencha o campo `File name` (nome do arquivo) com `list_view_item` e o campo `Root Element` (elemento raiz) como `LinearLayout` e clique em `OK`.

Isso criará um novo arquivo na pasta, no qual vamos implementar o código do layout dos itens. Para montá-lo, vamos usar um `LinearLayout` horizontal onde colocaremos os elementos de `ImageView` para imagem, `TextView` para o nome do produto e outro `LinearLayout`, mas agora vertical, que abrigará os outros dois `TextView`. Observe a imagem a seguir, ela mostra o esqueleto do layout que precisaremos montar:

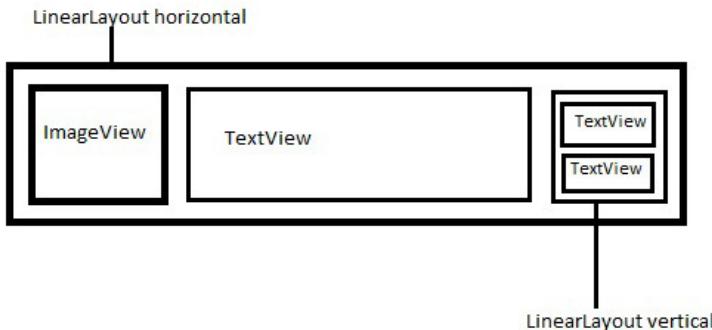


Figura 7.6: Esqueleto do item

Para a montagem desse layout só existe um ponto que ainda não abordamos, o peso ponderado do layout! O `LinearLayout` nos permite atribuir um peso para cada componente dentro dele através do atributo `layout_weight`. Esse atributo define um valor de importância para cada componente na questão de quanto espaço ele deve ocupar na tela. Vamos a um exemplo.

Imagine que você tenha um `LinearLayout` horizontal com dois componentes dentro. Se você quiser que cada componente ocupe exatamente a metade da tela, basta atribuir peso 1 para os componentes, desta forma o `LinearLayout` fará uma ponderação nos pesos e, como ambos terão pesos iguais, dividirá o espaço igualmente para cada componente.

Para esse atributo funcionar corretamente, devemos atribuir a largura ou altura do componente para `0dp`, então se eu

quero balancear componentes na horizontal eu devo atribuir o peso deles e deixar a largura igual a 0dp (`layout_width="0dp"`). No entanto, se eu estou平衡ando um componente na vertical, eu devo atribuir o seu peso e deixar a sua altura como 0dp (`layout_height="0dp"`). A imagem a seguir representa um linear layout com 2 componentes dentro balanceados com peso 1:



Figura 7.7: Ponderação de layout

Para montar o layout, vamos utilizar um `LinearLayout` com orientação horizontal, dentro do qual criaremos uma `ImageView` com tamanho fixo de 50x50. Em seguida, criaremos um `TextView` com peso 1, assim ele dominará o espaço restante. Por fim, criaremos um `LinearLayout` com orientação vertical e dentro dele 2 elementos `TextView`.

O código a seguir demonstra como criar esse layout. Abra o arquivo `list_view_item` e construa esse código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:minHeight="50dp">
```

```
        android:gravity="center_vertical"
        android:padding="5dp">

    <ImageView
        android:layout_width="50dp"
        android:layout_height="50dp"
        android:src="@android:drawable/ic_menu_camera"
        android:id="@+id/img_item_foto"
        />

    <TextView
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:id="@+id/txt_item_produto"
        android:textSize="18sp"
        android:textStyle="bold"
        android:gravity="center"
        />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="horizontal">

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:id="@+id/txt_item_qtd"
                android:textSize="18sp"
                android:textStyle="bold"
                />

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=" Qtd"
                android:textSize="18sp"
                android:textStyle="bold"
                />
        
```

```
</LinearLayout>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textStyle="bold"
    android:textSize="18sp"
    android:id="@+id/txt_item_valor"/>

</LinearLayout>

</LinearLayout>
```

Perfeito! Já temos o layout dos itens da nossa lista! Vamos seguir adiante para fazer a lista funcionar e ver se o resultado visual está bom ou precisará de algum ajuste - e caso precise voltaremos nesse arquivo para os ajustes visuais.

7.4 CRIANDO A CLASSE DE DADOS

O próximo passo agora é definir a estrutura de dados que esta lista vai representar, isto é, a estrutura das informações que ela vai exibir! Anteriormente, nossa estrutura de dados era um texto simples, então usamos o objeto `String` como estrutura de dados. Mas e agora? Com uma simples `String` conseguiremos representar todas as novas informações?

Nesse caso, uma `String` já não nos atenderá mais, então precisaremos de uma estrutura para essa demanda específica. Como o Kotlin é uma linguagem orientada a objetos, podemos criar uma classe que representará exatamente a estrutura de dados de que precisamos!

Vamos primeiro pensar no nome dessa classe, que deve ser claro o bastante para indicar exatamente o que essa classe

representa. O nome `Produto` me parece um nome adequado, uma vez que essa classe representará um produto inserido na lista. Vamos pensar também em quais atributos essa classe terá.

Certamente, ela terá um atributo para guardar o nome do produto, que podemos chamar de `nome`, e ele deve ser do tipo `String` pois o nome do produto é um conteúdo de texto. Precisaremos também de um atributo para a quantidade, que podemos chamar de `quantidade`, esse será do tipo `Int` pois a quantidade será um número inteiro.

Vamos precisar também de um atributo para o valor do produto, que podemos chamar de `valor` e será do tipo `Double` pois o valor de um produto não é um número inteiro (muitas vezes, um preço é um número decimal) então precisaremos de uma variável que suporte esse tipo de número. Por fim, vamos definir um atributo para guardar a foto do produto e chamaremos de `foto`. Esse atributo pode ser do tipo `Bitmap`, tipo de variável que serve para guardar elementos gráficos, como uma foto.

Para criação da classe, clique com o botão direito em cima da pasta `br.com.livrokotlin.listadecompras` e escolha a opção `New > Kotlin File/class`. Na janela que se abrir, preencha o campo `Name (nome)` como `Produto` e clique em `OK`.

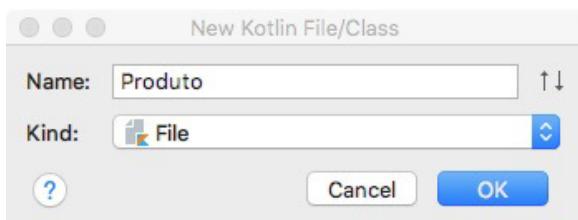


Figura 7.8: Classe produto

Isso criará um arquivo vazio na pasta. Abra-o para começar a implementação do código. Podemos dizer que a classe que vamos criar é uma classe de dados, isto é, que usaremos somente para guardar informações; não será uma classe que terá métodos ou funções, somente os atributos `nome` , `quantidade` , `valor` e `foto` . Por isso, podemos criá-la como uma `data class` e assim nosso código ficará muito reduzido!

Para criar a nossa classe, iniciaremos com as palavras `data class` , indicando que é uma classe de dados. Em seguida, digitaremos o nome da classe, `Produto` , depois implementaremos através do construtor os atributos de `nome` , `quantidade` , `valor` e `foto` . O código completo da classe será o seguinte:

```
data class Produto(val nome:String, val quantidade:Int, val valor:Double, val foto: Bitmap)
```

Faça também o `import` da classe `Bitmap` :

```
import android.graphics.Bitmap
```

A classe está pronta! Mas vamos pensar em uma situação: será que devemos obrigar o usuário a definir sempre uma foto para o produto? Acredito que esse atributo deva ser opcional, assim, se o usuário não definir nenhuma foto, usaremos alguma imagem padrão.

Pensando nisso, vamos mudar um pouco essa classe e deixar o atributo `foto` aceitando valores nulos, pois se nenhuma foto for atribuída essa variável deve ser nula, e também vamos deixar esse atributo como opcional na construção do objeto.

Para deixar o atributo `foto` aceitar valores nulos, basta incluir o sinal de interrogação na frente de sua definição: `val foto:`

Bitmap? . Agora vamos deixá-lo como atributo opcional; vamos definir que se o usuário não passar nenhum valor a essa variável ele assumirá o valor nulo, por padrão. Para isso, basta adicionar = null na definição da variável, assim: val foto: Bitmap? = null . Feito isso nossa classe agora ficará assim:

```
data class Produto(val nome:String, val quantidade:Int, val valor :Double , val foto: Bitmap? = null )
```

7.5 CUSTOMIZANDO O ADAPTADOR

Agora faremos talvez o passo mais importante, que é a customização do adaptador! Com todas essas alterações, o adaptador que estamos usando já não dá mais conta. Lembrando da analogia de que um adaptador é como um adaptador de tomada, desses que usamos para ligar uma tomada de 3 pinos em uma rede elétrica de 2 pinos, agora estaríamos lidando com uma situação totalmente nova, uma situação em que nenhum adaptador desses comprados em mercado serviria - imagine que agora nossa rede elétrica é de 4 pinos!

E você deve estar pensando "Pera lá, não existem redes de 4 pinos...". Sim, aí é que está, essa é a liberdade que temos quando trabalhamos com listas e adaptadores no Android. Podemos criar qualquer estrutura de dados que atenda a nossa necessidade e depois simplesmente criar um adaptador customizado para essa estrutura de dados! Essa é a situação que temos agora: imagine que a nossa classe `Produto` seja a rede elétrica de 4 pinos e, como não existem adaptadores para 4 pinos, vamos criar um adaptador customizado!

Vamos começar criando o arquivo para ele, para isso clique

com o botão direito em cima da pasta `br.com.livrokotlin.listadecompras` e escolha a opção New > Kotlin File/Class . Preencha o nome do arquivo como `ProdutoAdapter` , pois este será um adaptador exclusivo para a classe `Produto` .

No arquivo que se abriu, vamos começar a escrever nosso código, vamos iniciar criando a classe: `class ProdutoAdapter(contexto: Context)` . No construtor da classe eu já estou passando um parâmetro do tipo `Context` chamado `contexto` . Essa variável receberá o contexto em que esse adaptador está sendo usado.

Em seguida, faremos uma herança da classe `ArrayAdapter` , isso porque esse adaptador customizado ainda será um adaptador, então ele terá todas as características de um adaptador do tipo `ArrayAdapter` , mais as características novas que implementaremos. O código ficará o seguinte:

```
import android.widget.ArrayAdapter
import android.content.Context

class ProdutoAdapter(contexto: Context) : ArrayAdapter<Produto>()
{
}
```

No entanto, se você resgatar o código em que criamos o adaptador de `String` , você verá que a classe `ArrayAdapter` recebe 2 parâmetros, um contexto e o layout. Veja o código que utilizamos na `MainActivity` para relembrar:

```
val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1)
```

A palavra-chave `this` faz referência à instância do objeto em que está sendo utilizada. O construtor do `ArrayAdapter` recebe como primeiro parâmetro um objeto `Context`, que é base para o objeto `Activity`, então ao utilizar o `this` dentro de uma `Activity` estamos fazendo uma referência à própria `Activity`, mas também a um objeto `Context`.

Nesse código estamos passando o contexto através da variável `this` e o layout através do `android.R.layout.simple_list_item_1`. Isso significa que um `ArrayAdapter` precisa de pelo menos essas duas informações, então, na declaração da nossa classe, precisaremos passá-las. Para a variável de contexto está fácil pois já definimos uma variável `contexto`, mas o layout será customizado, e isso faremos um pouco mais adiante no código. Por isso, por enquanto passaremos `0` no construtor para informar que, por enquanto, não tem nenhum layout definido. O código ficará assim:

```
import android.widget.ArrayAdapter  
import android.content.Context  
  
class ProdutoAdapter(contexto: Context) : ArrayAdapter<Produto>(c  
ontexto,0) {  
  
}
```

Agora precisaremos implementar o comportamento desse adaptador, isto é, como ele fará para preencher as informações que queremos da forma correta.

A classe `ArrayAdapter` implementa o método chamado

`getView`, que é o método principal do adaptador porque a função dele é retornar a `View` de cada item da lista que será exibida na tela. O método `getView` é responsável por montar o item de exibição na tela, é esse método que pegará a informação e colocará no lugar certo, por isso ele é o método mais importante de um adaptador.

Para entender melhor como o método `getView` funciona, imagine uma linha de produção de veículos em que existe um monte de peças separadas, e na linha de montagem é feita a junção de todas as peças para, no final, sair um veículo pronto. É exatamente assim que funciona um adaptador com o método `getView`. Nesse caso, o `getView` seria a linha de montagem em que chegam todas as informações e ele será responsável por fazer a junção de tudo e, ao final, devolver um item pronto para exibição.

Ainda neste mesmo exemplo, imagine que esta fábrica vai produzir um total de 10 veículos por dia, dessa forma, na linha de produção serão montados exatamente 10 veículos, assim acontece também com o método `getView`: se temos 10 itens para exibir na lista, o `getView` será executado 10 vezes e montará cada um dos 10 itens!

A estrutura do método `getView` é a seguinte:

```
override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {  
}
```

Adicione também os imports na classe:

```
import android.view.View  
import android.view.ViewGroup
```

Dica: o Android Studio possui um recurso chamado `Auto Import`, que basicamente faz os imports automaticamente no nosso projeto conforme formos usando os componentes. Para habilitá-lo clique no menu `File > Settings` e, na caixa de pesquisa. Busque por `auto import`. Na janela de configuração, selecione `All` na opção `Insert imports on paste` e marque as opções: `Add unambiguous imports on the fly` e `Optimize imports on the fly`.

Pela estrutura do método, podemos perceber que ele recebe 3 variáveis:

- `position` : A posição do item que está prestes a ser montado na lista.
- `convertView` : A `View` que foi usada no último item para ser reutilizada. Antes de utilizar essa variável devemos sempre verificar se ela não está nula, pois, se for o primeiro item a ser criado, essa variável estará nula e não será possível reutilizá-la.
- `parent` : Uma `view` pai à qual eventualmente a `view` principal poderá estar atrelada.

Vamos começar por partes. Primeiro vamos declarar uma variável do tipo `View` que será responsável por guardar o layout desse item. Vou chamá-la de `v` para facilitar o código do restante do método.

```
val v:View
```

Agora vamos fazer uma verificação em cima da variável

`convertView` e, caso ela não seja nula, vamos preencher a variável `v` com a variável `convertView`; caso contrário, vamos inflar o layout que criamos. Essa ideia é de reaproveitamento da `View`: se eu inflar o layout na primeira vez, nas próximas ele já vai estar inflado para eu utilizar.

Chamamos "Inflar um layout" o processo de converter um arquivo `xml` para um objeto do tipo `View`. Imagine que, para fazer um bolo, você precise da massa de bolo e de uma forma untada. Para assá-lo, você pega a massa e coloca dentro da forma. Isso seria o "inflar o layout", pegar o `xml` e colocar dentro de um objeto.

Vamos então à verificação:

```
if(convertView != null){  
    v = convertView  
}else{  
    //inflar o layout  
}
```

Então, caso a `convertView` não seja nula, não precisaremos inflar o layout pois ele já estará inflado dentro da `convertView`. Agora no `else`, devemos inflá-lo. Para isso, existe uma classe no Android chamada `LayoutInflater`, que serve justamente para esse propósito.

Para usar o `LayoutInflater`, primeiro devemos criar o objeto a partir do contexto: `LayoutInflater.from(context)`. Em seguida, chamamos o método `inflate`.

O método `inflate` possui duas assinaturas: a primeira recebe dois parâmetros, que são respectivamente uma `View` chamada `resource` e uma `ViewGroup` chamada `root`:

```
inflate(resource:View, root:ViewGroup)
```

E a segunda recebe, além do `resource` e do `ViewGroup`, um Boolean chamado `attachToRoot`, que indicará se o layout passado na variável `resource` será anexado ao `ViewGroup`:

```
inflate(resource:View, root:ViewGroup, attachToRoot:Boolean)
```

Sabendo que o método `inflate` pode receber dois ou três parâmetros, vamos entender o que é exatamente cada um.

O primeiro parâmetro é o `resource`, que é o arquivo `xml` que você gostaria de "inflar", para nosso caso, será o arquivo `list_view_item` pois é ele que definimos como layout dos itens dessa lista.

O segundo parâmetro é um `ViewGroup` de que esse layout faz parte. Aqui temos a opção de simplesmente passar `null` a essa variável e indicar que nosso layout não faz parte de nenhum `ViewGroup`, nesse caso teríamos um código assim:

```
LayoutInflater.from(context).inflate(R.layout.list_view_item, null)
```

`ViewGroup` é um tipo especial de `View` que pode conter outras `Views` dentro. Um exemplo é o `LinearLayout`, dentro do qual podemos agregar várias outras `Views`. Em resumo, uma `ViewGroup` é um componente que agrupa outros componentes dentro dele.

Quando utilizamos o método `inflate` dessa forma, estamos dizendo que nosso layout não faz parte de nenhum `ViewGroup`, o que nesse caso não é bem uma verdade. Aqui, o `ViewGroup` do qual esse layout faz parte é a própria `ListView`, e apesar de esse código funcionar, se utilizarmos dessa maneira o nosso layout acabará não herdando as configurações gerais definidas na `ListView`.

Se repararmos, o próprio método `getView` já nos dá acesso a uma variável chamada `parent`, que é justamente a `ListView` que estamos montando, então podemos passar essa variável como `ViewGroup` do método `inflate`:

```
LayoutInflater.from(context).inflate(R.layout.list_view_item, parent)
```

Agora precisamos passar o último parâmetro `attachToRoot`, um valor booleano indicando se o nosso layout será anexado ao `ViewGroup`. No nosso caso é `false`, ou seja, não será diretamente anexado ao `ViewGroup`, mas, sim, herdará configurações gerais definidas na `ListView`. O código ficará assim:

```
LayoutInflater.from(context).inflate(R.layout.list_view_item, parent, false)
```

Colocando esse código no nosso projeto, teremos o seguinte:

```
...
else{
    //inflar o layout
    v = LayoutInflater.from(context).inflate(R.layout.list_view_item, parent, false)
}
```

Faça também o `import` da classe `LayoutInflater`:

```
import android.view.LayoutInflater
```

Imaginando que o método `getView` seja uma linha de produção de automóveis, inflar o layout seria preparar o chassi do veículo para receber os outros componentes.

Agora, o próximo passo é acessar os dados que serão exibidos na tela. Essa informação já está no adaptador e pode ser acessada pelo método `getItem` passando a posição do item que desejamos acessar. Para saber a posição do item não há problema, pois o método `getView` já nos dá acesso a uma variável chamada `position`, que guarda exatamente a informação de posição que precisamos. Então o código é muito simples:

```
val item = getItem(position)
```

Agora vamos acessar os elementos visuais. Na montagem do layout, colocamos três `TextViews` : `txt_item_produto` , `txt_item_qtd` e `txt_item_valor` para exibir o nome do produto, a quantidade e o valor, respectivamente. Também colocamos um `ImageView` , `img_item_foto` , para exibir a foto do produto.

Então vamos acessar cada um desses elementos através dos Ids. Para isso, vamos usar o método `findViewById` que já conhecemos, a única diferença é que agora ele será usado na variável `v` porque é nela que o layout está inflado. Veja:

```
val txt_produto = v.findViewById<TextView>(R.id.txt_item_produto)
val txt_qtd = v.findViewById<TextView>(R.id.txt_item_qtd)
val txt_valor = v.findViewById<TextView>(R.id.txt_item_valor)
val img_produto = v.findViewById<ImageView>(R.id.img_item_foto)
```

Imports necessários:

```
import android.widget.ImageView
import android.widget.TextView
```

Neste momento, temos acesso a duas coisas importantes:

1. os componentes da tela
2. a variável `item` que guarda o produto.

Sendo assim, agora basta atualizarmos a propriedade `text` dos `TextViews` com as informações da variável `item`:

```
txt_qtd.text = item.quantidade.toString()
txt_produto.text = item.nome
txt_valor.text = item.valor.toString()
```

Para atualizar a foto, devemos antes verificar se a variável não é nula, pois o usuário não é obrigado a definir uma imagem. E caso a foto for diferente de `null`, atribuiremos a ela a `img_produto` através do método `setImageBitmap`. Esse método serve para atribuir uma variável do tipo `Bitmap` a um `ImageView`, que é exatamente a situação que temos.

```
if (item.foto != null){
    img_produto.setImageBitmap(item.foto)
}
```

Por fim, vamos retornar a variável `v`. A variável `v` está carregando dentro de si todo o layout do item que será mostrado na lista, então tudo o que foi construído nesse método está sendo carregado dentro da variável `v` definida no início do método.

```
return v
```

Ainda pensando em uma linha de produção, o método `getItem` seria preparar o motor do veículo. O `findViewById` seria preparar a pintura e estética, e a atualização dos atributos seria a finalização, quando os componentes são juntados para criar o produto final.

Veja o código completo da classe:

```
import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ArrayAdapter
import android.widget.ImageView
import android.widget.TextView

class ProdutoAdapter(contexto: Context) : ArrayAdapter<Produto>(c
ontexto, 0) {

    override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
        val v:View
        if(convertView != null){
            v = convertView
        }else{
            v = LayoutInflater.from(context).inflate(R.layout.li
```

```

        t_view_item, parent, false)
    }

    val item = getItem(position)

    val txt_produto = v.findViewById<TextView>(R.id.txt_item_
produto)
    val txt_qtd = v.findViewById<TextView>(R.id.txt_item_qtd)
    val txt_valor = v.findViewById<TextView>(R.id.txt_item_va
lor)
    val img_produto = v.findViewById<ImageView>(R.id.img_item
_foto)

    txt_qtd.text = item.quantidade.toString()
    txt_produto.text = item.nome
    txt_valor.text = item.valor.toString()

    if (item.foto != null){
        img_produto.setImageBitmap(item.foto)
    }

    return v
}

}

```

Nosso adaptador está pronto! Mas tem mais uma pequena coisa que eu gostaria já de ajustar aqui, porque isso dará diferença no resultado final do nosso aplicativo. Definimos que vamos exibir o valor de cada produto de acordo com o que o usuário cadastrar, no entanto, o usuário vai cadastrar um número decimal, por exemplo: 10.50 . Para a exibição, seria muito interessante exibir esse número no padrão moeda brasileira R\$ 10,50 , então vamos aproveitar que já estamos mexendo no adaptador e fazer esse ajuste também.

Para fazer essa formatação de moeda temos a classe `NumberFormat` em que podemos definir N tipos de formatações

numéricas. Essa classe possui um método chamado `getCurrencyInstance`, que retorna uma instância de formatação do formato moeda! Então podemos usá-lo para construir um objeto de formatação e depois formatar a variável `valor`.

```
//obtendo a instância do objeto de formatação  
val f = NumberFormat.getCurrencyInstance()  
  
//formatando a variável no formato moeda  
txt_valor.text = f.format(item.valor)
```

Devemos nos atentar para um detalhe: o método `getCurrencyInstance` vai formatar de acordo com a linguagem em que o aparelho estiver configurado. Geralmente, os emuladores estão em inglês, então esse código formataria o valor em dólar e não em reais. Podemos definir qual linguagem de formatação usar passando um objeto `Locale` para o método `getCurrencyInstance`. Veja como definir a formatação em português do Brasil:

```
//obtendo a instância do objeto de formatação  
val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))  
  
//formatando a variável no formato moeda  
txt_valor.text = f.format(item.valor)
```

Faça os imports da classe `NumberFormat`:

```
import java.text.NumberFormat
```

E da classe `Locale`:

```
import java.util.Locale
```

Dessa forma, independentemente da linguagem do aparelho, a formatação será feita sempre em reais.

Agora sim nosso adaptador está pronto!

7.6 CADASTRANDO ITENS

Vamos agora resolver a tela de cadastro. Já temos todo o layout pronto, só precisaremos trabalhar no código. Mas antes precisamos pensar em um detalhe: onde as informações de produtos serão armazenadas?

No Android, cada Activity é independente, ou seja, por padrão elas não compartilham informações. Portanto, temos que pensar em como vamos armazenar essas informações para mostrar na tela principal. O ideal seria usar uma estrutura de banco de dados pois as informações ficariam gravadas permanentemente no disco e qualquer classe do aplicativo poderia acessá-las, mas vamos aprender a utilizar o banco de dados do Android somente no próximo capítulo, então aqui precisamos de uma solução mais simplificada.

Podemos armazenar as informações de produtos em uma variável global, que pode ser acessada de qualquer parte do aplicativo. Com uma variável global, eu posso gravar informações por meio da tela de cadastro e depois lê-las através da tela principal, o ponto negativo desta abordagem é que, como esses dados ficam somente em memória RAM, quando o aplicativo for fechado esses dados serão perdidos, mas isso resolveremos no próximo capítulo com a implementação de um banco de dados.

Em Kotlin, é muito fácil criar variáveis globais. Basta criar um arquivo simples em Kotlin e definir a variável lá, que ela automaticamente estará visível em todo o projeto! Vamos fazer isso então.

Crie um arquivo chamado `utils` no projeto. Para isso, basta

clicar com o botão direito na pasta `br.com.livrokotlin.listadecompras` e escolher a opção `New > Kotlin file/Class`.

Vamos chamar esse arquivo de `Utils` pois podemos usá-lo posteriormente para colocar funções ou outras variáveis que serão úteis ao nosso projeto.

Nesse arquivo, vamos definir a nossa variável global. Essa variável deve ser capaz de armazenar vários produtos cadastrados, então ela não pode ser simplesmente do tipo `Produto`, ela deve ser uma lista pois não sabemos quantos produtos o usuário deseja cadastrar.

Vamos definir então uma lista mutável, ou seja, uma lista que pode ter sua estrutura alterada incluindo ou removendo itens. Vamos chamar essa variável de `produtosGlobal` para dar a ideia de que ela armazenará produtos de forma global e vamos definir que ela será do tipo `mutableListOf<Produto>`, ou seja, uma lista mutável do tipo `Produto`. Veja como ficará a implementação:

```
val produtosGlobal = mutableListOf<Produto>()
```

E pronto! Já temos uma variável que pode ser acessada de forma global.

Partiremos agora para o código da tela de cadastro. Abra o arquivo `CadastroActivity`. Nessa classe, já tem um pedaço de código pronto que fizemos no começo do capítulo, no qual temos a definição do botão e uma verificação para saber se o usuário deixou o nome em branco. O código que temos até agora é o seguinte:

```
//definição do ouvinte do botão
btn_inserir.setOnClickListener {

    //pegando o valor digitado pelo usuário
    val produto = txt_produto.text.toString()

    if (produto.isNotEmpty()) {
        //enviado o item para a lista

    }else{
        txt_produto.error = "Preencha um valor"
    }

}
```

Vamos incrementar esse código com as mudanças necessárias. Primeiro, vamos pegar as informações de quantidade e valor - a foto do produto faremos por último. Vamos incluir então uma linha de código para pegar o valor e uma linha para pegar a quantidade:

```
//pegando os valores digitados pelo usuário
val produto = txt_produto.text.toString()
val qtd = txt_qtd.text.toString()
val valor = txt_valor.text.toString()
```

Agora vamos mudar um pouco a verificação também. Antes, verificávamos somente se o nome do produto não estava vazio; agora precisamos verificar as variáveis `qtd` e `valor`, pois ele será obrigado a informar todas essas informações. Vamos utilizar operador "E" (`&&`) para incluir as verificações no mesmo `if`:

```
if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()
()) {
    //enviado o item para a lista

} ...
```

Dessa forma, só vamos entrar nesse `if` se todas as variáveis estiverem preenchidas; se por acaso qualquer uma estiver vazia, o

código cairá no `else`. Já aproveitando, vamos ter que mexer nesse `else` também. Atualmente, o `else` considera que o nome do produto está vazio e então envia uma mensagem de erro.

Mas agora as coisas são diferentes. O usuário pode ter deixado de preencher qualquer uma das informações, ou pode não ter preenchido nenhuma, são várias situações diferentes que podem acontecer aqui e precisamos saber exatamente qual a caixa que ele deixou de preencher para definir a mensagem correta na caixa correta. Para isso, podemos abrir outros `ifs` aqui dentro. Que tal se fizermos uma verificação para cada caixinha individualmente e setarmos uma mensagem personalizada? O código ficaria assim:

```
...
}else{

    if (txt_produto.text.isEmpty()){
        txt_produto.error = "Preencha o nome do produto"
    }else{
        txt_produto.error = null
    }

    if (txt_qtd.text.isEmpty()){
        txt_qtd.error = "Preencha a quantidade"
    }else{
        txt_qtd.error = null
    }

    if (txt_valor.text.isEmpty()){
        txt_valor.error = "Preencha o valor"
    }else{
        txt_valor.error = null
    }
}
```

Nesse código é feita a verificação de cada caixinha. Se ela estiver vazia, colocamos uma mensagem personalizada, caso contrário definimos seu atributo `error` como `null` para não

aparecer mais erro naquela caixa de texto.

Apesar de fazer sentido, ficou um código um pouco extenso para uma coisa simples. Será que não conseguimos fazer a mesma coisa com menos código? A resposta é sim! Graças à linguagem Kotlin, conseguiremos reduzir esse código todo a 3 linhas!

Isso porque o Kotlin nos permite fazer um `if` em uma única linha. A estrutura é basicamente a mesma, no entanto, o `if` e o `else` são feitos na mesma linha. Esse tipo de `if` só faz sentido se for uma atribuição de valor condicional, que é nosso caso: aqui estamos atribuindo ao atributo `error` uma mensagem personalizada ou `null`, então conseguimos fazer essa atribuição condicional na mesma linha. Veja como fica no código:

```
...
}else{
    txt_produto.error = if (txt_produto.text.isEmpty()) "Preencha o nome do produto" else null
    txt_qtd.error = if (txt_qtd.text.isEmpty()) "Preencha a quantidade" else null
    txt_valor.error = if (txt_valor.text.isEmpty()) "Preencha o valor" else null
}
```

Uma solução bem elegante, não é?

Agora precisamos fazer o cadastro em si, ou seja, pegar essas informações e inserir naquela variável global que criamos. Mas precisamos ter em mente que a variável `produtosGlobal` que criamos é uma lista do tipo `Produto`, sendo assim só podemos inserir nela objetos do tipo `Produto`. Então, antes de mandar para a lista, precisamos criar esse objeto. Como já temos a classe pronta, basta criarmos um objeto com as informações que o

usuário inseriu. Vamos criá-lo:

```
val prod = Produto(producto, qtd.toInt(), valor.toDouble())
```

Nesse código estamos criando um objeto do tipo `Produto` e passando o nome, a quantidade e o valor. Perceba que, na variável `qtd`, utilizamos o método `toInt()` porque definimos essa variável como `Int`. O mesmo acontece com a variável `valor`, mas aqui utilizamos o método `toDouble()` porque definimos que esse atributo é `Double`. Por fim, não passamos a imagem porque ainda não a temos e, como definimos como parâmetro opcional, o código funcionará mesmo sem ela.

Feito isso, basta adicionar esse item à variável `produtosGlobal` com o método `add`:

```
produtosGlobal.add(prod)
```

Na sequência, vamos também limpar as caixinhas para que o usuário possa inserir outro produto:

```
txt_produto.text.clear()  
txt_qtd.text.clear()  
txt_valor.text.clear()
```

O código completo do botão ficará assim:

```
//definição do ouvinte do botão  
btn_inserir.setOnClickListener {  
  
    //pegando os valores digitados pelo usuário  
    val produto = txt_produto.text.toString()  
    val qtd = txt_qtd.text.toString()  
    val valor = txt_valor.text.toString()  
  
    //verificando se o usuário digitou algum valor  
    if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()) {  
        //salvando o produto no arraylist  
        produtosGlobal.add(Produto(produto, qtd.toInt(), valor.toDouble()))  
        //limpando as caixinhas  
        txt_produto.text.clear()  
        txt_qtd.text.clear()  
        txt_valor.text.clear()  
    }  
}
```

```

mpty()) {

    val prod = Produto(produto, qtd.toInt(), valor.toDouble()
)

    produtosGlobal.add(prod)

    txt_produto.text.clear()
    txt_qtd.text.clear()
    txt_valor.text.clear()

}else{

    txt_produto.error = if (txt_produto.text.isEmpty()) "Preencha o nome do produto" else null
    txt_qtd.error = if (txt_qtd.text.isEmpty()) "Preencha a quantidade" else null
    txt_valor.error = if (txt_valor.text.isEmpty()) "Preencha o valor" else null
}

}

```

Você pode testar o aplicativo agora, cadastrar itens e testar se as verificações funcionam. Mas ainda não verá nenhum item cadastrado pois ainda não arrumamos a tela principal para exibi-los.

7.7 EXIBINDO ITENS CADASTRADOS

Exibir os itens cadastrados na tela principal não será difícil porque já temos tudo o que precisamos pronto, é só começar a juntar as peças desse quebra-cabeça. Abra o arquivo `MainActivity` para começarmos os trabalhos.

Vamos começar mudando o adaptador. Vamos utilizar nosso adaptador personalizado que criamos no começo do capítulo.

Localize a linha da definição do adaptador onde está o seguinte código:

```
//Implementação do adaptador  
val produtosAdapter = ArrayAdapter<String>(this, android.R.layout.  
.simple_list_item_1)
```

Vamos mudar essa definição para uma instancia da classe `ProdutoAdapter` :

```
//Implementação do adaptador  
val produtosAdapter = ProdutoAdapter(this)
```

Em seguida, vamos acessar a variável `produtosGlobal` e inserir seus itens no `produtosAdapter`. A única consideração que temos que fazer é que o método `add` do adapter aceita somente 1 elemento, e a variável `produtosGlobal` é uma lista; para esses casos, temos o método `addAll` , que recebe uma lista como parâmetro e adiciona todos elementos:

```
produtosAdapter.addAll(produtosGlobal)
```

Só com isso já conseguimos visualizar os itens cadastrados! Teste o aplicativo e veja se eles aparecem na tela principal.

Aqui podem acontecer duas situações: a primeira é você cadastrar um item, clicar no botão `voltar` do aparelho, e na tela principal não aparecer item nenhum; a outra situação é você cadastrar um item e clicar no botão `voltar` da barra de superior do aplicativo, desta forma os itens cadastrados aparecerão normalmente.

Mas por que isso acontece? Tem diferença entre o botão `voltar` do aparelho e o botão `voltar` da barra superior do aplicativo? A resposta é sim, tem diferença!

A diferença é que, quando se volta através do botão do aparelho, a Activity anterior não é recriada, ou seja, o método `onCreate` não é executado. Dessa forma, ela simplesmente carrega a Activity do jeito como ela estava na memória. No entanto, quando voltamos através do botão superior, a Activity é recriada e o método `onCreate` é chamado. Sendo assim, agora entendemos por que quando voltamos com o botão do aparelho não aparecem os itens cadastrados - porque o código que preenche a lista está no `onCreate`, assim ele não é chamado.

Vamos resolver isso transferindo o código que preenche a lista para o método `onResume`, que é chamado sempre quando uma tela for aparecer, independente se ela está sendo criada ou se simplesmente está voltando. Se houver dúvidas sobre o funcionamento dos métodos `onCreate` e `onResume`, consulte no capítulo 4 a seção *Activities*.

Vamos então implementar o método `onResume` na classe `MainActivity` e transferir a linha de código `produtosAdapter.addAll(produtosGlobal)` para lá.

```
override fun onResume() {  
    super.onResume()  
    produtosAdapter.addAll(produtosGlobal)  
}
```

Mas tem um problema: você verá que a variável `produtosAdapter` não existe nesse método porque ela foi criada no método `onCreate`. E está certo manter a criação do adaptador lá no `onCreate` pois ele será criado uma única vez e depois só vamos atualizar seus itens. Mas para resolver nosso problema podemos acessar o `produtosAdapter` através da própria `list_view_produtos`:

```
override fun onResume() {  
    super.onResume()  
  
    val adapter = list_view_produtos.adapter as ProdutoAdapter  
  
    adapter.addAll(produtosGlobal)  
}
```

Ótimo! Teste o aplicativo novamente e veja se está tudo funcionando.

Se você cadastrou alguns itens, voltou para a tela principal, cadastrou mais itens, você percebeu que temos mais um pequeno problema a resolver. Da forma como está, você deve ter visto que algumas vezes a lista exibe itens duplicados. Isso está acontecendo porque o método `adapter.addAll(produtosGlobal)` sempre adiciona todos os itens na lista, não considerando possibilidade de ela já possuí-los, e por isso ficam itens duplicados. Para resolver isso, vamos limpar a lista antes de adicionar itens novos com o método `clear()` do adaptador. Este método simplesmente limpa a lista.

```
override fun onResume() {  
    super.onResume()  
  
    val adapter = list_view_produtos.adapter as ProdutoAdapter  
  
    adapter.clear()  
    adapter.addAll(produtosGlobal)  
}
```

Agora sim temos o funcionamento correto da lista! Faça alguns testes, veja se agora os itens param de duplicar.

7.8 SOMANDO OS ITENS DA LISTA

Agora que já temos o fluxo básico funcionando, conseguimos

cadastrar um item e conseguimos visualizá-lo na lista. Vamos implementar a funcionalidade de somatória na tela principal. No layout pensado inicialmente, existe uma `TextView` para exibir a soma dos itens cadastrados na lista. Veja novamente o layout:

ADICIONAR ITENS		
TOTAL: R\$51,40		
	Arroz	2 Qtd R\$9,90
	Coca Cola 2L	4 Qtd R\$6,50
	Macarrão instantâneo	2 Qtd R\$1,30
	Milho em lata	2 Qtd R\$1,50

Figura 7.9: Layout da lista

No layout que criamos lá no começo, já definimos essa `TextView` com id `txt_total`, então precisamos somente somar os itens da lista e atualizar esse `TextView`. Para fazer essa soma, na maioria das linguagens precisaríamos fazer uma estrutura de repetição e, dentro dela, ir acumulando o valor de cada item. Em Kotlin poderíamos fazer assim também, o código seria o seguinte:

```
var soma = 0.0
for (item in produtosGlobal){
    soma += item.valor * item.quantidade
}
```

Não é um código complicado, simplesmente aplicamos um `for` na variável `produtosGlobal` e, a cada item, fazemos a multiplicação do valor pela quantidade e acumulamos na variável `soma`. Mas podemos deixá-lo mais simplificado utilizando a função `sumByDouble` da lista. Através dessa função, podemos fazer uma soma de todos os itens da lista sem precisar de uma estrutura de repetição. Veja como ficaria o código:

```
val soma = produtosGlobal.sumByDouble { it.valor * it.quantidade }
```

E pronto! Somente essa linha substitui aquele `for`, então vamos utilizar desta forma. Tendo a variável `soma`, só precisamos atualizar a `TextView`, e já vamos aproveitar para formatar essa variável no formato da moeda:

```
val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))

txt_total.text = "TOTAL: ${f.format(soma)}"
```

Faça os `imports` das classes `NumberFormat`:

```
import java.text.NumberFormat
```

E `Locale`:

```
import java.util.Locale
```

O código completo do método `onResume`:

```
override fun onResume() {
    super.onResume()

    val adapter = list_view_produtos.adapter as ProdutoAdapter
    adapter.clear()
    adapter.addAll(produtosGlobal)
```

```
    val soma = produtosGlobal.sumByDouble { it.valor * it.quantidade }

    val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))
    txt_total.text = "TOTAL: ${f.format(soma)}"

}
```

Teste o aplicativo e veja se a soma está sendo feita corretamente!

7.9 ACESSANDO A GALERIA DE IMAGENS

Vamos resolver agora a última parte do nosso aplicativo, que é poder incluir uma imagem para o produto cadastrado. Vamos fazer isso acessando a galeria de imagens do celular e definindo a foto de acordo com a imagem escolhida pelo usuário.

Para acessar a galeria de imagens do aparelho, precisamos criar um objeto de intenção Intent e definir alguns parâmetros para ele. Mas, antes de começar o código em si, vamos entender primeiro o que é um objeto Intent e como ele funciona.

O objeto Intent

A Intent é um objeto de mensagem que pode ser usado para solicitar uma ação de outro componente de aplicativo. Embora os *intents* facilitem a comunicação entre componentes de diversos modos, vamos ver como uma Intent trabalha para iniciar uma Activity :

Iniciar uma atividade:

Uma Activity representa uma única tela em um aplicativo.

Como já visto, é possível iniciar uma nova instância de uma Activity passando uma Intent para o método `startActivity()`. A Intent descreve a atividade a iniciar e carrega todos os dados necessários.

Veja um exemplo de uma Intent para uma chamada telefônica:

```
//definindo a intent com ação para chamada telefônica  
val chamada = Intent(Intent.ACTION_DIAL)  
  
//definindo o parâmetro que a intent usará para chamada  
chamada.setData("tel:1155559900");  
  
//inicializando a activity de chamada telefônica  
startActivity(chamada);
```

Esse código descreve como efetuar uma chamada telefônica a partir de uma Intent . Esse tipo de solicitação é chamado de *Intent implícita* pois não estamos especificando exatamente qual Activity o Android vai abrir para efetuar a chamada, simplesmente estamos definindo a ação ACTION_DIAL e o sistema operacional resolverá isso para nós.

Em alguns casos, desejamos receber algum retorno da Activity que foi iniciada. Nossa caso será um exemplo disso: precisamos inicializar a Activity da galeria para o usuário escolher uma imagem; assim que o usuário tiver escolhido, precisamos que essa imagem seja enviada como resposta para o aplicativo, para podermos salvá-la.

Para esses casos, a inicialização da Activity deve ser feita por meio do método `startActivityForResult()` e para capturar o resultado devemos implementar o método

`onActivityResult` . Veja um exemplo de como esses métodos trabalham em conjunto:

```
fun abrirGaleria(){

    //definindo a ação de conteúdo
    val intent = Intent(Intent.ACTION_GET_CONTENT)

    //definindo filtro para imagens
    intent.type = "image/*"

    //inicializando a activity com resultado
    startActivityForResult(Intent.createChooser(intent, "Selecion
e uma imagem"), COD_IMAGE)

}
```

Faça o import da classe Intent :

```
import android.content.Intent
```

Atenção: esse código não fica dentro do método `onCreate` , é uma função separada, mas ainda no escopo da Activity.

O código anterior demonstra uma chamada ao método `startActivityForResult` para abrir a galeria de imagens do celular. Observe que, junto à chamada do método, é necessário passar um número inteiro que foi definido na variável `COD_IMAGE` . Esse número é de escolha do desenvolvedor, não existe um número predefinido. Esse código será usado no método `onActivityResult` para saber que a resposta veio desta requisição.

Crie também a variável `COD_IMAGE` no escopo da Activity,

geralmente no começo da classe, antes do método `onCreate`. Vou definir um valor de `101`, mas como disse, esse valor é de escolha do desenvolvedor. Veja como isso fica no código:

```
class CadastroActivity : AppCompatActivity() {  
  
    val COD_IMAGE = 101  
  
    ...  
    //restante do código
```

Veja agora a implementação do método `onActivityResult`:

```
override fun onActivityResult(requestCode: Int, resultCode: Int,  
data: Intent?) {  
    super.onActivityResult(requestCode, resultCode, data)  
  
    if (requestCode == COD_IMAGE && resultCode == Activity.RESULT  
_OK) {  
  
        if (data != null) {  
  
            //Neste ponto podemos acessar a imagem escolhida atra  
vés da variável "data"  
  
        }  
    }  
}
```

Aqui você precisará do `import` da classe `Activity`:

```
import android.app.Activity
```

O que precisamos é exatamente disso! Abra o arquivo `CadastroActivity` e implemente as funções `abrirGaleria` e `onActivityResult` assim como as definimos.

Em seguida, vamos definir que a função `abrirGaleria` seja

chamada quando o usuário clicar no objeto `ImageView`, para isso inclua o *listener* de clique do `ImageView` para chamar a função `abrirGaleria` no método `onCreate` da `Activity`:

```
img_foto_produto.setOnClickListener {  
    abrirGaleria()  
}
```

Aqui você já consegue testar o aplicativo. Você verá que conseguimos abrir a galeria e selecionar uma imagem, mas ela ainda não aparece no nosso aplicativo. Vamos continuar o código para trazer a imagem escolhida para dentro do App.

O método `onActivityResult` nos dá acesso a 3 variáveis: `requestCode`, `resultCode` e `data?`. O `requestCode` é o código da requisição, um número inteiro pelo qual podemos diferenciar as requisições feitas. O `resultCode` é o código do resultado da `Activity`, ele pode ter o conteúdo igual a `Activity.RESULT_OK`, o que significa que o usuário não cancelou a ação, ou pode ser igual a `Activity.RESULT_CANCELED`, o que significa que o usuário cancelou a ação. Por fim, temos a variável `data?`, que pode ser nula, então devemos sempre fazer a verificação de nulo nela. Ela carrega o conteúdo que foi retornado da `Activity`, no nosso caso, ela vai carregar a URI da imagem.

Como precisamos guardar essa imagem em um `Bitmap`, podemos ler essa URI e transformar para `Bitmap`. Já vamos aproveitar e exibir a imagem na tela, isso será feito com o código a seguir:

```
//lendo a URI com a imagem  
val inputStream = contentResolver.openInputStream(data.getData())  
;
```

```
//transformando o resultado em bitmap  
imageBitMap = BitmapFactory.decodeStream(inputStream)  
  
//Exibir a imagem no aplicativo  
img_foto_produto.setImageBitmap(imageBitMap)
```

Vamos então implementar esse código no nosso aplicativo. Primeiro, defina a variável `imageBitMap` no escopo da classe `CadastroActivity`:

```
var imageBitMap: Bitmap? = null
```

Você precisará importar a classe `Bitmap`:

```
import android.graphics.Bitmap
```

Agora complete o código do método `onActivityResult`:

```
override fun onActivityResult(requestCode: Int, resultCode: Int,  
data: Intent?) {  
    super.onActivityResult(requestCode, resultCode, data)  
  
    if (requestCode == COD_IMAGE && resultCode == Activity.RESULT  
_OK) {  
  
        if (data != null) {  
  
            //lendo a uri com a imagem  
            val inputStream = contentResolver.openInputStream(dat  
a.getData());  
  
            //transformando o resultado em bitmap  
            imageBitMap = BitmapFactory.decodeStream(inputStream)  
  
            //Exibir a imagem no aplicativo  
            img_foto_produto.setImageBitmap(imageBitMap)  
        }  
    }  
}
```

Faça também o `import` da classe `BitmapFactory`:

```
import android.graphics.BitmapFactory
```

Agora você pode testar o aplicativo novamente e você verá que já conseguimos procurar uma imagem e ela é exibida no aplicativo! Mas ainda falta um detalhe: se você salvar um item, você vai ver que a imagem ainda não está sendo salva, isso porque não estamos salvando a variável `imageBitMap`. Vamos fazer isso.

Localize no método `onCreate` a seguinte linha de código:

```
val prod = Produto(produto, qtd.toInt(), valor.toDouble())
```

Essa linha monta o objeto que está sendo salvo. Vamos modificá-la para incluir a variável `imageBitMap`:

```
val prod = Produto(produto, qtd.toInt(), valor.toDouble(), imageBitMap)
```

E pronto! Agora você pode testar o aplicativo e ver o fluxo completo em funcionamento.

7.10 RESUMINDO

Neste capítulo, vimos alguns recursos muito importantes do Android e muitas coisas legais da linguagem Kotlin. Vimos como montar uma lista personalizada com um adaptador customizado, trabalhar com mais de uma `Activity`, como podemos simular um processo de cadastro, acessar a galeria de imagens e muito mais!

No próximo capítulo, vamos construir um aplicativo utilizando banco de dados, assim conseguiremos guardar informações no disco para que não se percam quando o aplicativo for fechado, ou

até mesmo quando o telefone for desligado. Espero você no próximo capítulo.

CAPÍTULO 8

PERSISTÊNCIA DE DADOS COM SQLITE

Neste capítulo, vamos entender como funciona a persistência de dados em um banco de dados SQLite, para isso, vamos aproveitar nosso aplicativo de lista de compras. Nosso aplicativo de lista de compras está bem legal, porém você deve ter percebido que, toda vez que fechamos e abrimos de novo, os itens que estavam inseridos são perdidos. Isso acontece porque não estamos utilizando nenhuma forma de persistência de dados.

O objetivo deste capítulo é armazenar e recuperar dados de um banco de dados SQLite que já está disponível na plataforma do Android. Ao contrário de outros bancos de dados, o SQLite não utiliza um servidor, ele armazena a estrutura de tabelas e dados em um arquivo no disco. Quando criamos um banco de dados para o nosso aplicativo, ele estará acessível em todas as nossas classes, mas não em outro aplicativo. Quando o aplicativo é desinstalado, seus dados também são removidos.

8.1 A BIBLIOTECA ANKO SQLITE

Aproveitarei esta seção para introduzir a biblioteca Anko. O objetivo dessa biblioteca é deixar o desenvolvimento de aplicativos

com Kotlin mais rápido e mais fácil. Se o Kotlin por si só já tem essa proposta, o Anko vem para ajudar ainda mais nesse quesito. A biblioteca Anko também foi desenvolvida pela equipe da JetBrains, a mesma empresa criadora da linguagem Kotlin, e por isso ela tem uma integração incrível com a linguagem.

O Anko pode ser dividido em quatro partes diferentes:

- **Anko Commons**: uma caixa de ferramentas para diversas funções para as tarefas mais comuns do Android.
- **Anko Layouts**: criação de layouts dinâmicos de forma rápida e eficiente.
- **Anko SQLite**: ferramentas que facilitam a criação e manipulação de banco de dados SQLite.
- **Anko Coroutines**: utilitários baseados na biblioteca `kotlinx.coroutines`.

Por hora, vamos utilizar o Anko SQLite para manipulação do banco de dados. Nos próximos capítulos exploraremos algumas outras ferramentas do Anko, mas vale muito a pena conferir a documentação oficial e todo o poder da biblioteca, em sua página no GitHub: <https://github.com/Kotlin/anko>

Usaremos a biblioteca Anko para manipulação de dados com SQLite porque é uma biblioteca fantástica e facilita muito o dia a dia do desenvolvedor, mas não é a única opção para trabalhar com persistência de dados locais no Android. Na hora de decidir qual arquitetura seu aplicativo usará, temos alguns caminhos que podem ser seguidos:

Utilizar a classe `SQLiteOpenHelper` disponível na plataforma Android

A plataforma Android nos disponibiliza uma classe chamada `SQLiteOpenHelper`. Com sua implementação, podemos criar um banco de dados, tabelas, gerenciar versões e posteriormente utilizá-la para inserir, deletar, atualizar e buscar dados do banco de dados. A grande vantagem de se implementar diretamente a classe `SQLiteOpenHelper` é que você terá o controle total do seu banco de dados; a desvantagem é que é uma abordagem muito mais trabalhosa em relação a código.

O seguinte artigo mostra como implementar um banco de dados utilizando a classe `SQLiteOpenHelper` :
<https://developer.android.com/training/data-storage/sqlite.html>

Utilizar alguma solução ORM

A sigla ORM vem do inglês *Object Relational Mapper*, que podemos traduzir como "mapeamento de objeto relacional". Na prática, a função do ORM é abstrair ao máximo as interações com o banco de dados através de seus objetos relacionais. A ideia é fazer o programador focar seu tempo somente em desenvolver o código e nas lógicas de negócio, e não se preocupar com o banco de dados. A grande vantagem é a facilidade de interação com os dados e a alta produtividade que utilizar um ORM pode trazer ao desenvolvimento do projeto; a desvantagem é que exige uma certa maturidade do programador e perfeito entendimento dos conceitos de Orientação a Objetos para conseguir implementar a solução de forma eficaz.

Durante o evento Google I/O 2017 foi anunciada a biblioteca `Room`, que é a solução ORM do Google para Android.

O seguinte artigo mostra como utilizar a biblioteca Room para

interações com banco de dados:
<https://developer.android.com/training/data-storage/room/index.html>

Para este projeto, vamos usar a biblioteca Anko SQLite. Ela não é uma solução de ORM, mas também facilita muito a manipulação de dados em um banco de dados. Na verdade, o Anko SQLite é uma camada de abstração da classe `SQLiteOpenHelper` feita em Kotlin. A grande diferença é que o Anko utiliza recursos da linguagem Kotlin para facilitar a programação do banco de dados. Ainda teremos tudo sob nosso controle, como se tivéssemos utilizando a classe `SQLiteOpenHelper`, mas com a vantagem de um código muito mais intuitivo e fácil de implementar.

Adicionando Anko SQLite ao projeto

Para utilizar o Anko, devemos adicionar suas dependências ao nosso projeto nos arquivos de `build.gradle`.

Primeiro, localize o arquivo `build.gradle` do projeto, ele estará marcado com a frase (Project: Lista de Compras) ao lado:



Figura 8.1: build.gradle

Primeiro, vamos definir uma variável para guardar a versão do Anko. No momento da escrita deste livro, a biblioteca está na versão 0.10.4, então faremos uma variável para guardar essa versão

e, caso seja atualizada, basta voltar a essa variável e atualizar seu valor.

Abra o arquivo `build.gradle` e adicione, logo abaixo da definição da versão do Kotlin, a seguinte variável:

```
ext.anko_version='0.10.4'
```

The screenshot shows the `build.gradle` file for a project. A yellow highlight covers the entire code block. A red arrow points from the text `ext.anko_version='0.10.4'` back up towards the top of the file, indicating its location relative to the other configuration blocks. The code includes sections for `buildscript`, `allprojects`, and a task definition, with various repositories and classpath entries.

```
buildscript {
    ext.kotlin_version = '1.2.21'
    ext.anko_version='0.10.4' ←
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:3.0.1"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Figura 8.2: Anko Version

A seta indica o local de criação da variável.

Agora devemos incluir as dependências no arquivo `build.gradle` do módulo. Ele estará logo abaixo do arquivo do projeto. Você localizará visualmente pela frase `(Module: app)` ao lado do nome do arquivo.

Com o arquivo aberto, navegue até a seção de dependências e adicione a seguinte linha:

```
implementation "org.jetbrains.anko:anko-sqlite:$anko_version"
```

A configuração `compile` está obsoleta e foi substituída pela configuração `implementation` junto com o Android Plugin 3.0.0: <http://d.android.com/r/tools/update-dependency-configurations.html>

Veja na imagem a dependência do Anko SQLite:

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'com.android.support:appcompat-v7:26.1.0'  
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'com.android.support.test:runner:1.0.1'  
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'  
  
    //dependencia do Anko SQLite  
    implementation "org.jetbrains.anko:anko-sqlite:$anko_version"  
}
```

Figura 8.3: Dependência Anko SQLite

Feito isso, clique no botão `Sync Now` (sincronizar agora) e o Android baixará a biblioteca e a adicionará ao projeto.

Dica: a biblioteca Anko possui diversas funções que são úteis ao dia a dia do desenvolvimento Android. Uma delas é o `startActivity` que vimos no capítulo anterior. Lá utilizamos o `startActivity` para abrir a tela de cadastro com o seguinte código:

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

Com a biblioteca Anko adicionada ao nosso projeto, poderíamos reescrever o mesmo código da seguinte forma:

```
startActivity<CadastroActivity>()
```

Isso mesmo, com uma única linha e com um código muito mais claro!

Você precisará do seguinte `import` para implementação desse código:

```
import org.jetbrains.anko.startActivity
```

8.2 CRIANDO O BANCO DE DADOS

A primeira coisa a fazer é a criação do banco de dados e das tabelas necessárias. No nosso caso, precisaremos de uma única tabela para guardar as informações de produtos. A biblioteca Anko provê uma classe chamada `ManagedSQLiteOpenHelper`, que nada mais é do que uma abstração em cima da classe `SQLiteOpenHelper`. Seu intuito é fazer o gerenciamento do

banco de dados, desde a criação até a manipulação de dados. Para criação do nosso banco de dados, faremos uma classe que estende a classe `ManagedSQLiteOpenHelper`.

A classe `ManagedSQLiteOpenHelper` recebe 4 parâmetros em seu construtor, sendo 3 obrigatórios e 1 opcional. O primeiro é um `Context`, com o contexto da aplicação (obrigatório); depois uma `String` com o nome do banco de dados (obrigatório); um objeto do tipo `CursorFactory`, caso o programador queira implementar uma lógica de cursores personalizada (opcional); e um `Integer` com a versão do banco de dados (obrigatório).

Então vamos criar uma classe para nosso banco de dados chamada `ListaComprasDatabase`. Para isso, primeiro crie o arquivo `ListaComprasDatabase` dentro do pacote `br.com.livrokotlin.listadecompras`.

Agora vamos implementar o código da classe, que deve ter um construtor que recebe um contexto do tipo `Context` e deve estender a classe `ManagedSQLiteOpenHelper`. Observe a implementação desse código.

```
class ListaComprasDatabase(context: Context) : ManagedSQLiteOpenHelper(ctx = context, name = "listaCompras.db", version = 1) {  
}
```

Adicione também os seguintes imports:

```
import android.content.Context  
import org.jetbrains.anko.db.ManagedSQLiteOpenHelper
```

Nessa classe devemos implementar 2 métodos: `onCreate` e `onUpgrade`.

O método `onCreate` é responsável pela criação das tabelas no banco de dados. Quando acessamos essa classe, ela automaticamente executa o `onCreate` e verifica se o banco de dados já está criado ou não. Caso o banco de dados não esteja criado, ele executa o código do `onCreate`. Por isso, devemos colocar aqui todo o código referente à criação das tabelas, com a seguinte estrutura: `fun onCreate(db: SQLiteDatabase)`, sendo que a variável `db` é o banco de dados em si, pronto para manipulação.

O outro método que precisamos implementar é o `onUpgrade`. Diferente do anterior, o `onUpgrade` é executado somente quando há uma alteração na versão do banco de dados. Ele possui a seguinte estrutura: `onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int)`, sendo que a variável `db` é o banco de dados em si para manipulação; e as variáveis `oldVersion` e `newVersion` são números inteiros que guardam a versão do banco de dados. Desta forma, o programador conseguirá gerenciar exatamente as atualizações de acordo com cada versão do banco.

Quando estendemos a classe `ManagedSQLiteOpenHelper`, passamos um `Int` com o número de versão. Geralmente começamos com `1` e, quando mudamos esse número de versão, o `onUpgrade` entra em ação. Então nesse método colocamos todas as alterações de banco de dados, novas tabelas, exclusão de tabelas, criação de novas colunas etc.

Imagine que você lançou um aplicativo na Play Store e ele utiliza um banco de dados com 2 tabelas. Depois de um tempo, você lançou uma atualização, na qual você incluiu uma nova tabela

no banco de dados. Nesse cenário, o que acontece com os usuários que já tinham o aplicativo instalado? Será que eles terão que desinstalar o aplicativo, consequentemente perdendo os dados salvos, e instalar novamente para ter a nova versão de banco de dados? Isso me parece uma má ideia! Por isso, existe o `onUpgrade` e as versões de banco de dados, dessa forma o programador implementa as alterações dentro do método `onUpgrade`, incrementa a versão do banco e, quando o usuário atualizar o aplicativo, ele receberá somente as atualizações.

Vamos fazer a implementação de ambos os métodos:

```
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import org.jetbrains.anko.db.ManagedSQLiteOpenHelper

class ListaComprasDatabase(context: Context) : ManagedSQLiteOpenHelper(ctx = context, name = "listaCompras.db", version = 1) {

    override fun onCreate(db: SQLiteDatabase) {
        // Criação de tabelas
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // Atualização do banco de dados
    }
}
```

Agora vamos implementar o código de criação da tabela em si. Para nosso projeto, precisaremos de uma tabela que reflita a estrutura da classe `Produto`. Então, ela deverá ter colunas para o nome do produto, a quantidade, o valor e sua foto. Além disso,

também devemos prever uma coluna de `id`, para que possamos diferenciar um registro do outro no banco de dados. Esse `id` deverá ser um número inteiro e autoincremental, de modo que, quando inserirmos um novo registro, o próprio banco de dados criará um `id` para ele incrementalmente.

Também é importante que o `id` seja uma chave primária. No contexto de banco de dados, chaves primárias são campos que não se repetem na mesma tabela. Nesse caso, não podemos ter dois produtos com o mesmo `id`, e por isso definir o campo como chave primária vai garantir que o `id` não se repetirá.

Para a criação das tabelas, usaremos o método `createTable` da variável `db`. Esse método cria uma tabela no banco de dados de acordo com alguns parâmetros passados, que serão 3:

- `tableName: String` : uma `String` com o nome da tabela que será criada.
- `ifNotExists: Boolean = false` : um booleano informando que a criação da tabela seria feita somente se ela ainda não existe; este campo é falso por padrão. Devemos passar `true` se quisermos que a tabela seja criada se ela não existir ainda.
- `vararg columns: Pair<String, SqlType>` : são as colunas que a tabela terá, no formato sendo do tipo `Pair` com o nome da coluna e seu tipo.

Portanto, vamos criar uma tabela chamada `produtos` com colunas de `id`, `nome`, `quantidade`, `valor` e `foto`.

A implementação ficará da seguinte maneira:

```
override fun onCreate(db: SQLiteDatabase) {  
  
    // Criação de tabelas  
  
    db.createTable("produtos", true,  
        "id" to INTEGER + PRIMARY_KEY + UNIQUE,  
        "nome" to TEXT,  
        "quantidade" to INTEGER,  
        "valor" to REAL,  
        "foto" to BLOB  
    )  
  
}
```

Esse código precisará do `import` da biblioteca Anko SQLite:

```
import org.jetbrains.anko.db.*
```

Quando fazemos um `import` utilizando `*`, estamos importando todas as classes daquele determinado pacote. Se fôssemos importar individualmente, precisaríamos dos seguintes `imports`:

```
import org.jetbrains.anko.db.ManagedSQLiteOpenHelper  
import org.jetbrains.anko.db.createTable  
import org.jetbrains.anko.db.INTEGER  
import org.jetbrains.anko.db.PRIMARY_KEY  
import org.jetbrains.anko.db.UNIQUE  
import org.jetbrains.anko.db.TEXT  
import org.jetbrains.anko.db.REAL  
import org.jetbrains.anko.db.BLOB
```

Os tipos de dados que podemos usar para criar as colunas são:

- `INTEGER` : para colunas que guardam valores inteiros;
- `REAL` : para colunas que guardam valores reais, ou seja, com números quebrados;

- TEXT : para guardar valores em texto;
- BLOB : para guardar qualquer valor de modo binário.

Agora criaremos mais alguns códigos nessa classe, com o intuito de facilitar seu uso. Ao final, quando formos fazer qualquer interação com o banco, simplesmente usaremos:

```
database.use {  
    //aqui faremos as interações com o banco de dados  
}
```

Como ter uma instância única (Singleton)

Para conseguirmos esse efeito, teremos que fazer um Singleton da nossa classe e dar acesso à instância através do contexto. Singleton significa instância única. É a ideia de ter uma instância única dessa classe juntamente com a linguagem Kotlin que vai trazer toda a facilidade de uso nas manipulações com o banco de dados. Se nossa classe não fosse um Singleton, teríamos que criar a instância dela toda vez que fôssemos usar e não teríamos como simplesmente usar o comando `database.use { }`.

O padrão Singleton pode ser particularmente útil em algumas situações. Para acesso a bancos de dados é comum termos classes Singleton porque não é necessário ficar criando novas instâncias da classe toda vez que for fazer um acesso ao banco. Como o banco geralmente é único, a instância de acesso também pode ser única.

Vamos a um exemplo prático. Imagine que seu aplicativo possua uma classe `Calc` com um método que efetue a soma de dois números:

```
class Calc{  
    fun soma(a: Int, b:Int) = a + b
```

}

Para usar essa classe basta instanciá-la e chamar o método soma :

```
val c = Calc() // criando a instância  
c.soma(1,3) //chamando o método soma
```

Esse é um código bem trivial e não temos nenhum problema com ele. Dessa maneira, cada vez que criamos uma instância desse objeto `Calc()`, teremos uma instância diferente, ou seja, outro espaço na memória RAM foi alocado. Mas em alguns casos não precisamos e nem queremos esse efeito; queremos que uma classe possua uma instância única, e toda vez que formos usá-la simplesmente acessamos a que já existe, em vez de criar instâncias novas sem necessidade.

Para criar uma classe Singleton devemos seguir a seguinte lógica: criar uma variável estática do mesmo tipo da classe, geralmente chamada de `instance` , e implementar um método estático que retorna essa variável, geralmente chamado de `getInstance` . O método `getInstance` verifica se a instância da classe já existe e, caso sim, retorna a instância existente; caso contrário, cria uma instância na hora.

Em Kotlin, por definição, não temos variáveis estáticas, mas existem os blocos `companion object` que, no final das contas, são a mesma coisa. Quando tenho variáveis e métodos dentro de blocos `companion object` , consigo acessá-los sem a necessidade de criar uma instância da classe. Então usaremos esse recurso para criar classes Singletons.

Voltando ao exemplo da classe `Calc` , se fôssemos fazer um Singleton dela poderíamos fazer assim:

```
class Calc{  
  
    companion object {  
        //variável de instância  
        private var instance: Calc? = null  
  
        //método que verifica se a instância é nula  
        fun obterInstancia(): Calc {  
            if (instance == null) {  
                instance = Calc()  
            }  
            return instance!!  
        }  
    }  
  
    fun soma(a: Int, b:Int) = a + b  
}
```

E na hora de usar:

```
val c = Calc.obterInstancia() // obtendo a instância  
c.soma(1,3) //chamando o método soma
```

Desta forma, se eu chamar 10 vezes a instância desse objeto, nas 10 vezes eu estaria acessando exatamente a mesma instância.

Para entender um pouco mais sobre o padrão Singleton, recomendo a leitura adicional do artigo:
<https://www.devmedia.com.br/padrao-de-projeto-singleton-em-java/26392>

Aplicando o padrão Singleton na nossa lista de compras

Vamos por partes, primeiro vamos implementar o padrão Singleton na classe, para isso faremos uso do companion object

do Kotlin, utilizamos `companion object` quando queremos definir uma função ou um objeto estático na classe.

A implementação do código é a seguinte:

```
//Padrão Singleton
companion object {
    private var instance: ListaComprasDatabase? = null

    @Synchronized
    fun getInstance(ctx: Context): ListaComprasDatabase {
        if (instance == null) {
            instance = ListaComprasDatabase(ctx.getApplicationContext())
        }
        return instance!!
    }
}
```

A anotação `@Synchronized` protege o método contra acesso concorrente de múltiplas threads.

E para dar acesso à instância, criaremos uma função de extensão:

```
// Acesso a propriedade pelo contexto
val Context.database: ListaComprasDatabase
    get() = ListaComprasDatabase.getInstance(getApplicationContext())
```

Atenção: essa função fica fora da classe, mas pode mantê-la no mesmo arquivo.

Com isso, conseguiremos acessar o banco de dados de qualquer parte do aplicativo simplesmente usando:

```
database.use {  
}  
}
```

Por fim, a nossa classe ficará assim:

```
import android.content.Context  
import android.database.sqlite.SQLiteDatabase  
import org.jetbrains.anko.*  
  
class ListaComprasDatabase(context: Context) : ManagedSQLiteOpenHelper  
(ctx = context, name = "listaCompras.db", version= 1) {  
  
    //singleton da classe  
    companion object {  
        private var instance: ListaComprasDatabase? = null  
  
        @Synchronized  
        fun getInstance(ctx: Context): ListaComprasDatabase {  
            if (instance == null) {  
                instance = ListaComprasDatabase(ctx.getApplicationContext())  
            }  
            return instance!!  
        }  
    }  
  
    override fun onCreate(db: SQLiteDatabase) {  
        // Criação de tabelas  
  
        db.createTable("produtos", true,  
            "id" to INTEGER + PRIMARY_KEY + UNIQUE,  
            "nome" to TEXT,  
            "quantidade" to INTEGER,  
            "valor" to REAL,  
            "foto" to BLOB  
        )  
    }  
  
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {  
        // Atualização do banco de dados
```

```
        }
    }

// Acesso à propriedade pelo contexto
val Context.database: ListaComprasDatabase
    get() = ListaComprasDatabase.getInstance(getApplicationContext())
```

Você pode conferir a implementação dessa classe na documentação do Anko SQLite: <https://github.com/Kotlin/anko/wiki/Anko-SQLite>

A tabela `produtos` que criamos no banco de dados deve refletir o modelo de dados da classe `Produto`, por isso criamos a tabela com base nas propriedades da classe. No entanto, a tabela possui uma coluna de `id` que a classe não possui, por isso devemos voltar na classe `Produto` e adicionar uma propriedade para o `id` do tipo `Int`, pois é o mesmo tipo que definimos no banco de dados.

Abra a classe `Produto` e inclua a propriedade `id` como no código a seguir:

```
data class Produto(val id:Int, val nome:String, val quantidade:Int, val valor:Double, val foto: Bitmap? = null)
```

Feita essa alteração, o código vai parar de compilar e gerará um erro na `CadastroActivity` na linha `val prod = Produto(produto, qtd.toInt(), valor.toDouble(), imageBitmap)`. Isso porque agora a classe tem um atributo a mais que não está sendo passado. Não se preocupe com esse erro, vamos resolvê-lo mais adiante.

Feitas essas alterações, a estrutura do nosso banco de dados estará pronta. Observe que, se o banco de dados tivesse mais

tabelas, eu poderia criar todas no método `onCreate` chamando a função `createTable` para cada uma. Com essa estrutura pronta, quando formos acessar o banco de dados usaremos `database.use { }` e entre as chaves faremos as ações relacionadas ao banco de dados.

Esse modelo parece complexo, mas no final das contas é bem simples porque, como a estrutura da classe se manterá a mesma, em um outro aplicativo bastaria seguir a mesma estrutura e modificar os parâmetros relacionados ao projeto.

8.3 INSERINDO DADOS NO BANCO DE DADOS

Com o banco de dados pronto, podemos agora passar a inserir nele os valores da lista. Para inserir dados, usaremos o método `insert`, que recebe como parâmetro o nome da tabela e uma lista de chave-valor com os dados a serem inseridos.

Por exemplo, vamos supor que temos uma tabela `usuarios` com as colunas de `nome` e `email` e queremos inserir um novo usuário nessa tabela, com o nome de "João" e e-mail de "`joão@teste.com`". Para inseri-lo, faríamos o seguinte código:

```
database.use{  
  
    insert("usuarios",  
        "nome" to "João",  
        "email" to "joão@teste.com")  
  
}
```

A função `insert` necessita do seguinte `import`:

```
import org.jetbrains.anko.db.insert
```

Perceba como o código ficou simples, bastou passar o nome da tabela e, em seguida, o nome das colunas e seus valores. Repare também que eu não precisei passar a coluna `id`, isso porque o próprio banco de dados vai gerar um `id` sequencial.

O que precisamos aqui é pegar esse `id` que foi gerado e guardar em uma variável para poder usar mais adiante. Para isso, basta utilizar o retorno do método `insert`. Esse método retorna o `id` do registro inserido, ou retorna `-1` caso ocorra algum erro na inserção. Então, podemos fazer:

```
val idUsuario = insert("usuarios",
    "nome" to "João",
    "email" to "joão@teste.com")
```

Vamos implementar a função de `insert` na tela de `CadastroActivity`, no clique do botão. Vamos seguir a seguinte lógica: após o usuário clicar no botão, vamos continuar verificando se os campos estão preenchidos e, caso sim, vamos já inserir no banco de dados; então, somente se a inserção no banco de dados ocorrer corretamente, vamos inserir esse item na lista.

Lembre-se de que para saber se o item foi inserido corretamente podemos verificar o valor retornado pelo método `insert`: se ele for diferente de `-1` é porque a inserção foi feita com sucesso; caso contrário, exibiremos uma mensagem padrão.

Veja como fica em código:

```
val idProduto = insert("Produtos",
    "nome" to produto,
    "quantidade" to qtd,
    "valor" to valor.toDouble(),
    "foto" to imageBitmap
)
```

```
if (idProduto != -1L) {  
  
    toast("Item inserido com sucesso")  
  
} else {  
    toast("Erro ao inserir no banco de dados")  
}
```

Faça também o `import` da função `toast`:

```
import org.jetbrains.anko.toast
```

Enviando mensagens de aviso ao usuário

Toast: observe que no `else` utilizei o comando `toast`, esse comando exibe uma pequena mensagem na tela que desaparecerá sozinha depois de alguns segundos. É possível utilizar o comando `toast` dessa forma, `toast("Mensagem")`, porque estamos usando a biblioteca Anko, caso contrário, a forma padrão de se escrever um Toast é:

```
Toast.makeText(context, "Mensagem", Toast.LENGTH_SHORT).show()
```

Das duas formas, o resultado será o seguinte:



Figura 8.4: Mensagem toast

O Toast é muito útil para enviar pequenas mensagens de aviso ao usuário. Esse tipo de mensagem não necessita de nenhuma confirmação do usuário, ela simplesmente aparece na tela e desaparece depois de alguns segundos.

Alert: outra forma de enviar mensagens ao usuário é através do comando `alert`. Diferente do `toast`, o `alert` envia uma mensagem e espera uma ação do usuário antes de desaparecer da tela. A forma mais simples de se usar o `alert` é a seguinte:

```
alert("Mensagem").show()
```

Import necessário para função `alert`:

```
import org.jetbrains.anko.alert
```

Esse comando resultará no seguinte:

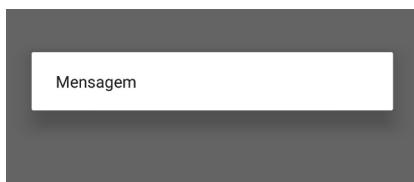


Figura 8.5: Alert simples

A vantagem do `alert` é que ele nos dá mais opções de customizações, por exemplo, eu poderia criar uma caixa de alerta com título e mensagem:

```
alert("Mensagem", "Titulo").show()
```

E o resultado seria o seguinte:

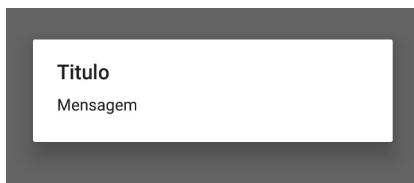


Figura 8.6: Alert com título

Com o `alert`, podemos também fazer mensagens em que o usuário precisa confirmar alguma ação através de dois botões: `Ok` e `Cancel`. Para configurá-los, utilizamos o comando da seguinte maneira:

```
alert("Mensagem", "Titulo") {  
    //Botão de OK  
    yesButton {  
        //Ação caso escolheu a opção OK  
    }  
  
    //Botão de cancel  
    noButton {  
        //Ação caso escolheu a opção CANCEL  
    }  
  
}.show()
```

E o resultado será:

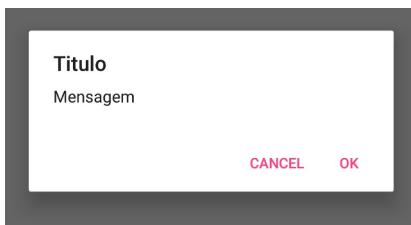


Figura 8.7: Alert com botões

Os textos "OK" e "CANCEL" são configurados automaticamente de acordo com a linguagem do aparelho. Caso a linguagem do aparelho esteja em português, esses textos serão respectivamente "OK" e "CANCELAR".

Em resumo, as duas formas são muito úteis e em cada situação podemos escolher entre uma ou outra. O `alert` possui mais opções de configurações em relação ao `toast`, mas em alguns

casos simplesmente queremos enviar um aviso ao usuário sem a necessidade de nenhuma ação posterior, nesse caso o `toast` é a melhor opção.

Inserindo informações

Voltando ao nosso projeto, então somente depois que o registro for inserido no banco de dados atualizaremos a lista, não queremos correr o risco de exibir para o usuário um item que não foi realmente salvo.

Vamos começar a implementação desse código na `CadastroActivity`. Abra o arquivo `CadastroActivity` e navegue até o listener do clique do botão.

Vamos mexer dentro do `if` de verificação se as caixas de texto estão vazias porque é aqui onde um item é salvo. Como vamos usar outra estratégia para salvar os dados, um banco de dados, o código antigo não faz mais sentido, por isso apague essas duas linhas de código:

```
val prod = Produto(produto, qtd.toInt(), valor.toDouble(), imageB  
itMap)  
  
produtosGlobal.add(prod)
```

Não usaremos mais essa variável `produtosGlobal` e também não precisaremos mais criar o objeto `prod` aqui, isso porque agora vamos inserir essas informações direto no banco de dados. O seu `if` deve ficar assim:

```
if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()  
) {  
  
    //Aqui implementaremos o código para inserir o produto no ban  
co de dados
```

```
txt_produto.text.clear()
txt_qtd.text.clear()
txt_valor.text.clear()

}
```

Agora vamos fazer o código de inserção no banco de dados. Primeiro, vamos montar o comando `insert` e, em seguida, verificar o retorno da função para ver se o item foi inserido com sucesso no banco. Para isso insira o seguinte código dentro do `if :`

```
database.use {

    val idProduto = insert("Produtos",
        "nome" to produto,
        "quantidade" to qtd,
        "valor" to valor.toDouble(),
        "foto" to imageBitmap
    )

    if (idProduto != -1L) {
        toast("Item inserido com sucesso")

        txt_produto.text.clear()
        txt_qtd.text.clear()
        txt_valor.text.clear()
    } else {
        toast("Erro ao inserir no banco de dados")
    }
}
```

Esse código vai precisar dos seguintes imports :

```
import org.jetbrains.anko.db.insert
```

```
import org.jetbrains.anko.toast
```

Só temos mais uma coisa para nos preocuparmos agora: para gravar a foto no banco de dados, estamos enviando diretamente o objeto `Bitmap` e, apesar de não ocorrer nenhum erro de compilação, com certeza teremos um erro na execução desse projeto, porque o tipo `BLOB` que usamos na coluna de foto do banco de dados não suporta um `Bitmap` diretamente. Devemos enviar um vetor de bytes em Kotlin um objeto do tipo `ByteArray`.

Por isso, antes de enviar ao banco, devemos transformar esse `Bitmap` em `ByteArray` e depois, quando resgatarmos as informações do banco, devemos fazer o processo contrário, transformar o `ByteArray` em `Bitmap`.

Extensões

Vamos implementar essas duas funções como extensões, esse é um recurso muito legal da linguagem Kotlin. Extensão é um recurso com o qual eu posso adicionar uma funcionalidade em um objeto sem precisar utilizar uma herança de classes, isto é, eu não preciso ter uma nova classe que herda a classe pai para adicionar funcionalidades à classe pai, basta criar uma extensão!

Para o nosso projeto, vamos criar 2 extensões, uma para classe `Bitmap` e uma para classe `ByteArray`. A extensão da classe `Bitmap` terá a função de devolver um `ByteArray` e a extensão da classe `ByteArray` vai fazer o contrário, devolver um `Bitmap`. Desta forma, vamos poder usar as extensões em qualquer objeto do tipo `Bitmap` ou `ByteArray`.

No nosso projeto, já existe um arquivo chamado `Utils.kt`.

Vamos utilizá-lo para criar as extensões dentro dele, então abra o arquivo.

Vamos fazer primeiro a extensão do `Bitmap`. Ela se chamará `toByteArray()` para deixar claro que ela transformará um `Bitmap` em `ByteArray`. Para declarar a extensão, devemos criar uma função com o nome da classe que receberá a extensão como prefixo. Vamos à prática, vou criar uma função de extensão para classe `Bitmap` com o nome `toByteArray` e ela retornará um objeto `ByteArray`:

```
fun Bitmap.toByteArray(): ByteArray {  
}
```

Agora vamos implementar o conteúdo dessa função. Nesse contexto, o uso da palavra `this` representa o `Bitmap` que chamou a função. Então, quando usamos `this` aqui estamos nos referindo ao próprio objeto.

Essa função será dividida em 3 partes: primeiro, vamos declarar um objeto do tipo `ByteArrayOutputStream`, que será responsável por fazer a transformação do `Bitmap` em `ByteArray`:

```
val stream = ByteArrayOutputStream()
```

Em seguida, faremos a compressão da imagem dentro do objeto `stream`. Para isso, vamos passar 3 parâmetros: o formato da imagem, a qualidade da compressão, sendo 0 como qualidade máxima e 100 como o máximo de compressão, e por fim o objeto `stream`:

```
//comprimindo a imagem  
this.compress(android.graphics.Bitmap.CompressFormat.PNG, 0, stre
```

am)

Por último, vamos retornar o resultado da função `toByteArray` do objeto `stream`:

```
return stream.toByteArray()
```

Confira a implementação completa:

```
fun Bitmap.toByteArray(): ByteArray {  
  
    val stream = ByteArrayOutputStream()  
  
    //comprimindo a imagem  
    this.compress(android.graphics.Bitmap.CompressFormat.PNG, 0,  
    stream)  
  
    //transformando em um array de caracteres  
    return stream.toByteArray()  
  
}
```

Você precisará dos seguintes imports:

```
import android.graphics.Bitmap  
import java.io.ByteArrayOutputStream
```

A próxima extensão é mais simples, ela será um objeto do tipo `ByteArray` e devemos retornar um objeto do tipo `Bitmap`. Para isso, existe o método estático `decodeByteArray` da classe `BitmapFactory`, que faz a decodificação de um `ByteArray` para `Bitmap`:

```
fun ByteArray.toBitmap() :Bitmap{  
  
    return BitmapFactory.decodeByteArray(this, 0, this.size);  
}
```

Aqui você precisará do seguinte import :

```
import android.graphics.BitmapFactory
```

Implemente essas duas funções no arquivo `Utils.kt`. A ideia de implementar essas funções dentro desse arquivo é simplesmente para deixar nosso código um pouco mais modularizado. Como essas duas funções são de extensão, elas poderiam estar em qualquer arquivo do projeto, no entanto, isso dificultaria sua manutenção.

Agora volte à `CadastroActivity` e localize a função de `insert` no banco de dados. Precisamos mudá-la para enviar um `ByteArray` ao banco, e não mais um `Bitmap` diretamente.

```
database.use {  
  
    val idProduto = insert("Produtos",  
        "nome" to produto,  
        "quantidade" to qtd,  
        "valor" to valor.toDouble(),  
        "foto" to imageBitmap?.toByteArray()  
    //Acrecentamos a chamada a função de extensão  
    )  
  
    ...  
    //restante do código
```

E pronto! A parte de inserir no banco de dados está completa! Confira o código completo do listener do botão:

```
//definição do ouvinte do botão  
btn_inserir.setOnClickListener {  
  
    //pegando o valor digitado pelo usuário  
    val produto = txt_produto.text.toString()  
    val qtd = txt_qtd.text.toString()  
    val valor = txt_valor.text.toString()  
  
    //verificando se o usuário digitou algum valor  
    if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()) {
```

```

database.use {

    val idProduto = insert("Produtos",
        "nome" to produto,
        "quantidade" to qtd,
        "valor" to valor.toDouble(),
        "foto" to imageBitmap?.toByteArray()
    //Aumentamos a chamada à função de extensão
    )

    if (idProduto != -1L) {
        toast("Item inserido com sucesso")
        txt_produto.text.clear()
        txt_qtd.text.clear()
        txt_valor.text.clear()
    } else {
        toast("Erro ao inserir no banco de dados")
    }
}

} else {

    txt_produto.error = if (txt_produto.text.isEmpty()) "Preencha o nome do produto" else null
    txt_qtd.error = if (txt_qtd.text.isEmpty()) "Preencha a quantidade" else null
    txt_valor.error = if (txt_valor.text.isEmpty()) "Preencha o valor" else null
}
}

```

8.4 SELECIONANDO REGISTROS DO BANCO

DE DADOS

A essa altura, nosso aplicativo já cadastra as informações no banco de dados! Mas você deve ter percebido que, ao fechar o aplicativo e abrir novamente, os dados não se mantêm, a lista vem vazia! O que houve? Será que a inserção no banco de dados não foi bem-sucedida?

Não é nada disso, os dados estão sim sendo armazenados no banco de dados, no entanto não estamos fazendo a busca das informações na hora de abrir o aplicativo!

O que devemos fazer é acessar o banco de dados e buscar todos os registros cadastrados quando o aplicativo for aberto, assim, quando o usuário abrir o aplicativo ele encontrará todos os itens que ele previamente cadastrou.

Para buscar dados do banco, utilizamos o método `select`, que, essencialmente, faz uma busca nos registros da tabela especificada. Veremos como podemos usar o `select` para algumas das buscas mais comuns.

Para fazer uma busca em todos os registros de uma tabela, basta usar o comando `select` e passar o nome da tabela em `String`. Por exemplo:

```
select("produtos")
```

Realmente simples! Caso precise filtrar a consulta, podemos usar o comando `whereArgs` e passar os argumentos que queremos. Vamos supor que você queira trazer todos os produtos com quantidade maior que 5, o comando ficaria:

```
select("produtos").whereArgs( "quantidade > {qtd}", "qtd" to
```

No entanto, até esse ponto esses comandos ainda não executam no banco de dados, ou seja, estamos simplesmente montando a consulta. Precisamos executar essa consulta no banco, e para isso existe a função `exec`. Ela é responsável por executar, de fato, a consulta no banco de dados e, a partir daí, podemos trabalhar seus resultados.

Veja no exemplo:

```
select("produtos").whereArgs( "quantidade > {qtd}", "qtd" to  
5 ).exec {  
  
    //Aqui conseguimos trabalhar com o resultado da consulta  
    ao banco  
  
}
```

No nosso projeto, precisamos de uma busca em todos os registros do banco de dados, então nosso comando será simplesmente:

```
select("produtos").exec {  
  
}
```

Precisaremos também tratar o resultado da busca dentro da função `exec`. Para isso, existem 3 tipos de funções para converter o resultado da busca:

- `parseSingle` : converte exatamente 1 linha; usamos essa função quando sabemos que nossa consulta retornará exatamente 1 registro.
- `parseOpt` : converte 1 ou nenhuma linha; diferente do `parseSingle`, o `parseOpt` pode ter um retorno nulo

caso não tenha nenhum registro.

- `parseList` : converte vários registros em uma lista, para consultas que retornaram mais de 1 resultado.

Esses 3 métodos recebem como parâmetro um objeto `rowParser` , que tem como objetivo converter cada coluna da tabela em um tipo do Kotlin.

Entenda o objeto `rowParser` como um conversor, ele vai pegar os resultados do banco de dados e converter para algum tipo de objeto que faça sentido para nosso aplicativo. No nosso caso, o que faz sentido é a criação de um objeto do tipo `Produto` , então será esse `rowParser` a ser implementado que converterá uma linha do banco de dados em um objeto do tipo `Produto` !

A implementação do `rowParser` é muito simples, basta declararmos variáveis na sequência que retornará a consulta, e depois podemos usá-las para criação do objeto.

Observe o código para criação do nosso `rowParser` :

```
val parser = rowParser {  
    id: Int, nome: String,  
    quantidade: Int,  
    valor: Double,  
    foto: ByteArray? ->  
    //Colunas do banco de dados  
  
    //Montagem do objeto Produto com as colunas do banco  
    Produto(id, nome, quantidade, valor, foto?.toBitmap() )  
}
```

Aqui encontramos novamente a notação de flecha `->` que, nesse contexto, está fazendo a separação dos parâmetros recebidos do corpo da função.

Em seguida, podemos utilizar o método `parseList` pois a nossa consulta retornará mais de um registro, retornará uma lista:

```
val listaProdutos = parseList(parser)
```

Temos uma lista de produtos com todos os registros que estão salvos no banco de dados! A partir daí, podemos inserir essa lista inteira no adaptador e assim exibir todos os itens para o usuário. Para adicionar uma lista inteira ao adaptador, podemos usar o método `addAll`, que funciona de forma semelhante ao `add`, porém não recebe um único item, mas sim uma lista ou um array.

Veja no código:

```
adapter.clear() //Limpa os itens antigos do adapter  
adapter.addAll(listaProdutos)
```

Agora vamos implementar esse código no método `onResume` da `MainActivity`, porque é lá que está o código para preencher a lista. Então devemos modificar o código atual para acessar o banco de dados e preencher a lista a partir do banco.

O código do método ficará assim, observe que o código está todo comentado e algumas coisas ainda se mantêm em relação ao código anterior:

```
database.use {
```

```

//Efetuando uma consulta no banco de dados
select("produtos").exec {

    //Criando o parser que montará o objeto produto
    val parser = rowParser {

        id: Int, nome: String,
        quantidade: Int,
        valor: Double,
        foto: ByteArray? ->
        //Colunas do banco de dados

        //Montagem do objeto Produto com as colunas do banco
        Produto(id, nome, quantidade, valor, foto?.toBitmap() )
    }

    //criando a lista de produtos com dados do banco
    var listaProdutos = parseList(parser)

    //limpando os dados da lista e carregando as novas informações
    adapter.clear()
    adapter.addAll(listaProdutos)

    //efetuando a multiplicação e soma da quantidade e valor
    val soma = listaProdutos.sumByDouble { it.valor * it.quantidade }

    //formatando em formato moeda
    val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))
    txt_total.text = "TOTAL: ${f.format(soma)}"

}

}

```

Para esse código, você usará os seguintes imports :

```

import org.jetbrains.anko.db.parseList
import org.jetbrains.anko.db.rowParser
import org.jetbrains.anko.db.select

```

Agora execute novamente o aplicativo e veja se ele carrega os dados que foram cadastrados no banco!

8.5 REMOVENDO UM REGISTRO

Para remover um registro do banco de dados, usamos o método `delete`, passando o nome da tabela e os argumentos para remoção. Isto é, quando vamos deletar um registro do banco temos que especificar qual registro é esse, e isso pode ser feito de várias formas. Por exemplo, imagine que eu queira deletar todos os produtos com quantidade maior que 10, faria um código assim:

```
delete("produtos", "quantidade > {qtd}", "qtd" to 10)
```

Perceba que a `String` que eu passei de argumento foi `quantidade > {qtd}`, em que `quantidade` é o nome da coluna da tabela e `{qtd}` indica que o valor aqui será trocado por um argumento de mesmo nome `qtd`, por isso, no terceiro parâmetro eu passo o argumento com o valor: `"qtd" to 10`.

No nosso projeto, vamos deletar um item com base no id do produto. Quando o usuário clicar e segurar algum item da lista, vamos deletá-lo com base em seu id, então nosso código ficará assim:

```
delete("produtos", "id = {id}", "id" to idProduto)
```

Claro que a variável `idProduto` deve guardar o `id` do produto que será deletado da lista. Para deixar o código um pouco melhor, vamos separar o ato de deletar em uma função dentro da `MainActivity`. Essa função deve receber uma variável `idProduto:Int` com o `id` do produto e executar o comando de `delete` no banco.

A implementação da função ficará assim:

```
fun deletarProduto(idProduto:Int) {
```

```
database.use {  
    delete("produtos", "id = {id}", "id" to idProduto)  
}  
}
```

Agora vamos chamar essa função no listener de clique longo da lista. Localize esse listener que foi implementado no método `onCreate`. Dentro do listener, basta chamar a função `deletarProduto` passando o `id` do item que foi selecionado:

```
deletarProduto(item.id)
```

Confira como ficará o código completo do listener:

```
//definição do ouvinte da lista para clicks longos  
list_view_produtos.setOnItemLongClickListener{ adapterView: AdapterView<*>, view1: View, i: Int, l: Long ->  
  
    //buscando o item clicado  
    val item = adapter.getItem(i)  
  
    //removendo o item clicado da lista  
    adapter.remove(item)  
  
    //deletando do banco de dados  
    deletarProduto(item.id)  
  
    true  
}
```

Podemos ainda colocar um `toast` para avisar o usuário de que o item foi deletado com sucesso.

```
...  
  
//deletando do banco de dados  
deletarProduto(item.id)  
  
toast("item deletado com sucesso")
```

```
    true  
}
```

8.6 ATUALIZANDO UM REGISTRO

Nosso projeto está completo assim como planejamos. No entanto, não temos um modo de atualizar algum registro que já foi cadastrado. Caso o usuário perceba que digitou algum dado errado, ele terá que apagar o registro e cadastrar de novo.

Esta seção é destinada à atualização de um registro no banco de dados, mas eu não farei a atualização direto no projeto em que víhamos trabalhando. Vou explicar como podemos atualizar um registro cadastrado no banco de dados e deixarei de desafio para você implementar a funcionalidade de editar um registro!

Para atualizar um registro do banco, a biblioteca nos disponibiliza a função `update`, que vai executar uma atualização no banco de dados de acordo com os parâmetros passados a ela. Podemos dividir a função `update` em 3 partes, sendo a primeira onde passamos o nome da tabela que será atualizada e os valores a serem atualizados, por exemplo:

```
update("Usuarios", "nome" to "Alice")
```

Nesse exemplo, a tabela se chama `Usuarios` e estamos atualizando a coluna `nome` com o valor `Alice`. Mas se executarmos esse comando dessa forma, todos os registros do banco de dados seriam atualizados com o nome `Alice` e muito provavelmente não é isso que você vai querer!

Vamos à segunda parte do comando, a passagem dos

argumentos. Os argumentos vão dizer qual registro queremos atualizar. Não são todos, mas um específico. Então especificamos qual registro vamos atualizar através dos argumentos que pode ser um `id` específico, um nome específico ou até mesmo um conjunto de dados. Para passagens desses argumentos, usamos a função `whereArgs` na qual podemos especificar um ou mais argumentos. Veja um exemplo:

```
update("Usuarios", "nome" to "Alice").whereArgs(" id = {id}"  
, "id" to 4)
```

Aqui estamos atualizando com base no `id`. Perceba que a função `whereArgs` recebe 2 argumentos: primeiro, o argumento em si, nesse caso `id = {id}` e aqui o `{id}` é um curinga que será substituído com o valor passado no segundo argumento; já no segundo, passamos o valor do curinga: `"id" to 4`, nesse caso estamos definindo que o `id` que será atualizado é o que possui valor `4`.

Por fim, devemos chamar a função `exec()`, que executa de fato a atualização no banco de dados, retornando um `Int` com a quantidade de registros afetados pela atualização.

Agora que você sabe como funciona a função `update`, que tal tentar implementar no projeto de lista de compras? Será um bom desafio para seu aprendizado.

Se você quiser, pode conferir a minha solução no GitHub: <https://github.com/kassiano/livrokotlin> O código estará todo comentado.

8.7 RESUMINDO

Chegamos ao final de mais um capítulo! Foi um capítulo denso, com muito conteúdo e aprendizagem. Aqui, exploramos a biblioteca Anko para manipular um banco de dados local em SQLite. No próximo capítulo, ainda vamos explorar mais a biblioteca Anko para fazer requisições assíncronas na internet. Até lá!

CAPÍTULO 9

CALCULADORA DE BITCOIN

Neste capítulo, vamos construir um App bem bacana, uma calculadora de bitcoins! Não se preocupe se você não possui conhecimentos profundos sobre a tecnologia do bitcoin ou como ele funciona, isso não será necessário nesse projeto.

Nesse projeto, vamos acessar um serviço externo para buscar os valores de cotação atual do bitcoin e assim construir um aplicativo que converte um valor entre reais e bitcoins com base na cotação atual. O objetivo principal é consultar um serviço externo da internet, você poderá ampliar esse conhecimento para acessar outros serviços como previsão do tempo, APIs dos Correios ou qualquer outro serviço REST.

A sigla REST vem de *Representational State Transfer*, em português "Transferência de Estado Representacional", que é uma abstração da arquitetura da World Wide Web. Mais precisamente, é uma arquitetura que define um conjunto de restrições e propriedades baseados em HTTP. Serviços Web compatíveis com a arquitetura REST fornecem interoperabilidade entre sistemas na internet. Desta forma, podemos integrar diferentes sistemas através de APIs REST. Imagine um sistema escrito em Java que

acesse um banco de dados MySQL para gravar e recuperar informações de um e-commerce. Imagine também que precisamos fazer um aplicativo em Kotlin que interaja com esse sistema para enviar e receber informações. Nesse cenário, se o sistema escrito em Java possuir um serviço REST implementado, a integração com o aplicativo em Kotlin fica transparente a ponto de não precisarmos saber como o sistema em Java foi implementado. Não precisamos saber nem da linguagem de programação utilizada nem de como ele faz para acessar o banco de dados.

Um serviço REST implementa as seguintes operações:

- **GET** : obtém os dados de um determinado recurso.
- **POST** : cria um novo recurso no servidor.
- **PUT** : substitui os dados de um determinado recurso.
- **DELETE** : exclui um determinado recurso.
- **PATCH** : atualiza parcialmente um determinado recurso.
- **HEAD** : é similar ao **GET** , porém utilizado apenas para se obter os cabeçalhos de resposta, sem os dados em si.
- **OPTIONS** : obtém quais manipulações podem ser realizadas em um determinado recurso.

As operações mais importantes de um serviço REST são: **GET** , **POST** , **PUT** e **DELETE** pois estão diretamente relacionadas com uma operação de CRUD no banco de dados.

9.1 PLANEJAMENTO

Vamos começar pensando em como será esse aplicativo. Será um App de uma única Activity que, ao ser aberto, acessará um serviço externo na internet e buscará a cotação atual do bitcoin,

mostrando esse valor na tela. Na mesma tela, terá um campo de texto para o usuário informar um valor em reais e um botão para efetuar o cálculo. Ao clicar no botão, o App converterá o valor informado pelo usuário em bitcoins de acordo com a cotação atual.

Ao final, teremos um resultado assim:



Figura 9.1: Calculadora de bitcoins

Para começar nossa calculadora de bitcoin, crie um novo

projeto no Android Studio com as seguintes configurações:

- Name: Calculadora de Bitcoin
- Company Domain: livrokotlin.com.br
- SDK Mínimo: Android 5.0
- Tipo de Activity: Empty Activity

Marque também a opção `Include Kotlin support`.

9.2 COMPONENTIZANDO O LAYOUT

Nosso layout não é muito complexo, mas vamos dividi-lo em algumas partes para facilitar o entendimento e também o desenvolvimento. Podemos dividir nossa tela em alguns blocos independentes, entre bloco de título, bloco de cotação, bloco de entrada de dados, e bloco de saída de dados.

Pensando dessa forma, podemos separar nosso layout para componentizá-lo, ou seja, criar componentes de layout reutilizáveis. Então em vez de criar o layout de uma tela toda no mesmo arquivo, você pode criar vários arquivos, cada um com uma parte da tela, e juntar tudo no arquivo principal.

O objetivo principal de separar o layout dessa forma é poder reutilizar um mesmo componente em outras telas. Por exemplo, vamos criar nesse aplicativo um bloco de cotação, que será um arquivo `xml` separado que exibirá a cotação atual do bitcoin. Esse mesmo arquivo pode ser usado em outra tela para mostrar a mesma informação, ou seja, aqui eu tenho um benefício de reutilização de código.

Vamos trabalhar dessa forma nesse projeto, nossa tela será

componentizada da seguinte maneira:

Bloco título:



Figura 9.2: Bloco título

Esse componente terá um `TextView` com o título da aplicação centralizado na tela.

Bloco de cotação:



Figura 9.3: Bloco de cotação

Esse componente terá dois `TextViews` alinhados horizontalmente, o `TextView` da direita será atualizado com o valor da cotação do bitcoin.

Bloco de entrada de dados:

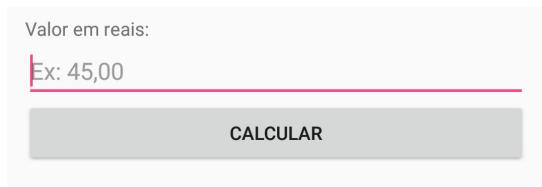


Figura 9.4: Bloco de entrada de dados

Esse componente terá o `EditText` em que o usuário vai digitar o valor para conversão e também terá o botão. Nesse bloco reunimos os componentes com que o usuário vai interagir para

entrar com informações no aplicativo.

Bloco de saída de dados:



Figura 9.5: Bloco de saída de dados

Esse componente terá os `TextViews` para exibição do resultado calculado, ou seja, a saída que o aplicativo vai gerar.

Vamos criar cada componente. Primeiro, crie um arquivo na pasta `layout` para cada um desses componentes com os seguintes nomes:

- `bloco_cotacao.xml`
- `bloco_entrada.xml`
- `bloco_saida.xml`
- `bloco_titulo.xml`

Sua pasta `layout` ficará com a seguinte estrutura:

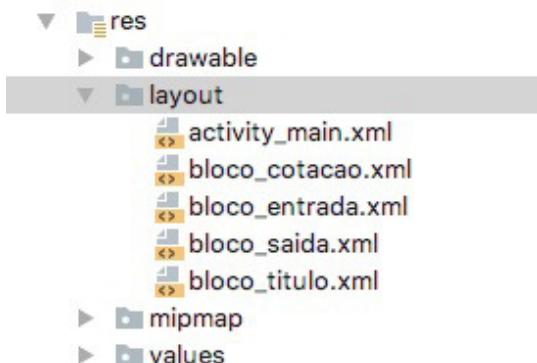


Figura 9.6: Estrutura layout

Agora vamos trabalhar individualmente em cada arquivo. Vamos começar pelo título. Para o título, teremos um `TextView` com o texto "Calculadora de bitcoin" com tamanho `18sp`, centralizado e com estilo de texto em negrito.

O arquivo `bloco_titulo.xml` ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Calculadora de bitcoin"
    android:gravity="center"
    android:textSize="18sp"
    android:textStyle="bold">
</TextView>
```

Agora vamos trabalhar no arquivo `bloco_cotacao.xml`. Ele deverá ter duas `TextViews` dispostas uma ao lado da outra, a da esquerda terá um texto fixo indicando ao que se refere aquele campo, e a da direita terá um texto atualizável conforme a cotação atual do bitcoin. Abra o arquivo `bloco_cotacao.xml` e implemente o seguinte código:

```
<?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="wrap_content"
    android:orientation="horizontal"
    android:layout_marginTop="20dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Cotação atual: " />

    <TextView
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:id="@+id/txt_cotacao"
        android:text="R$ 0,00"
        android:textStyle="bold" />

    </LinearLayout>
```

Na sequência, trabalharemos o arquivo `bloco_entrada.xml`. Ele terá como componentes principais um `EditText` e `Button` para permitir a entrada de valores e o disparo da ação de calcular. Abra esse arquivo e implemente o seguinte código:

```
<?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="wrap_content"
    android:layout_marginTop="20dp"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Valor em reais:" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/txt_valor"
        android:hint="Ex: 45,00" />

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/btn_calcular"
        android:text="calcular" />

</LinearLayout>
```

Por fim, vamos implementar o código do arquivo `bloco_saida.xml`. Esse arquivo terá simplesmente dois `TextViews` para exibir o resultado da conversão. Abra o arquivo

`bloco_saida.xml` e implemente o seguinte código:

```
<?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="wrap_content"
    android:layout_marginTop="20dp"
    android:orientation="vertical"
    android:gravity="center">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="Quantidade de bitcoins: " />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:id="@+id/txt_qtd_bitcoins"
        android:text="0.00000000" />

</LinearLayout>
```

Muito bom! Agora temos todas as partes do nosso layout separadas em arquivos. Talvez para esse projeto seja um pouco de exagero essa separação toda, mas você começa a enxergar todas as possibilidades futuras e claro, cada caso é um caso, cabe a você programador pensar na melhor estrutura para seu projeto. Imagine um aplicativo que terá uma tela para cadastro de usuários e posteriormente uma tela para edição desse usuário. Nesse caso, o formulário de cadastro tem os mesmos campos do formulário de edição, porém estão em telas diferentes. Você poderia criar o formulário em um arquivo `xml` separado e reutilizar nas duas telas!

Das vantagens de separar o layout dessa maneira, posso

destacar como principal a reutilização de componentes. Se em outra tela do aplicativo eu precisasse de um bloco para exibir a cotação atual do bitcoin, eu poderia simplesmente reutilizar o mesmo arquivo nas duas telas. Outra vantagem é que isso facilita a manutenção do código. Se eu precisasse fazer uma mudança de layout no bloco de cotação, por exemplo, essa mudança estaria isolada e centralizada neste arquivo. Assim, todas as telas que o utilizam já estariam atualizadas por consequência.

Esse mesmo ponto também pode ser uma desvantagem. Imagine que eu quero ter o bloco de cotação em duas telas diferentes, mas em uma tela quero o texto em negrito e na outra quero o texto normal, sem negrito. Nesse caso, ter o arquivo centralizado não me ajudaria muito. Eu teria que pensar em outra estratégia ou então simplesmente não separar os arquivos.

Vamos agora juntar todas as peças desse nosso quebra-cabeça. Agora que já temos todas as partes programadas vamos juntá-las para formar a nossa tela completa. Essa junção acontecerá no arquivo `activity_main.xml` pois ele é o arquivo principal dessa Activity.

Para fazer essa junção, utilizamos a tag `<include />` passando o arquivo como parâmetro na propriedade `layout`. A tag `include` faz a inclusão de um arquivo dentro de outro, o que é exatamente o que precisamos. Então vamos incluir cada pedacinho da nossa tela para criar uma tela completa.

Mas antes de começar os "includes" vamos preparar nosso arquivo `activity_main.xml` para podermos agrupar todos os componentes dentro dele. Abra o arquivo e crie um `LinearLayout` principal com orientação vertical, dessa forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.livrokotlin.calculadordadebitcoin.MainActivity"
    android:padding="16dp"
    android:orientation="vertical">

</LinearLayout>
```

Agora, dentro da tag do `LinearLayout` vamos incluir o primeiro arquivo, o de título:

```
<include layout="@layout/bloco_titulo"/>
```

Na sequência, inclua os outros arquivos:

```
<include layout="@layout/bloco_cotacao"/>

<include layout="@layout/bloco_entrada"/>

<include layout="@layout/bloco_saida"/>
```

Ao final, o código completo ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="br.com.livrokotlin.calculadordadebitcoin.MainActivity"
    android:padding="16dp"
    android:orientation="vertical">

    <include layout="@layout/bloco_titulo"/>

    <include layout="@layout/bloco_cotacao"/>
```

```
<include layout="@layout/bloco_entrada"/>

<include layout="@layout/bloco_saida"/>

</LinearLayout>
```

Perceba que o código da nossa tela ficou bem enxuto: temos uma `LinearLayout` servindo de contêiner e dentro dele fomos incluindo cada pedacinho da tela através da tag `<include />`.

E pronto! Dessa forma, nosso código final ficou muito simplificado. É claro que, como nosso projeto é de uma única tela, não temos tantas vantagens nessa componentização do layout, mas em projetos maiores isso já se torna relevante e, ressalto, cabe a você como programador, avaliar qual a melhor situação para separar seu layout em diversos arquivos.

9.3 API DE DADOS - MERCADO BITCOIN

Vamos começar a programar nosso projeto já pelo ponto principal, que é a busca da cotação do bitcoin na internet. Para isso, precisamos acessar uma API externa que nos retorne esse valor. Existem diversas APIs abertas na internet que nos retornam a cotação do bitcoin, aqui vamos usar a API do Mercado Bitcoin que é uma das maiores casas de câmbio de bitcoin do Brasil. Eles possuem uma API de dados aberta que podemos acessar para fazer a consulta.

O termo API significa "Application Programming Interface" (Interface de programação de aplicações). Na prática, uma API é uma interface para integração entre sistemas. No contexto de internet, uma API geralmente está associada a um serviço REST, mas não exclusivamente. Muitas empresas e serviços online disponibilizam APIs para interação com seu sistema interno. Na Web você encontra facilmente APIs para busca de CEP, previsão do tempo, cotações de moedas, cotação de preço de ações etc.

A API do Mercado Bitcoin pode ser acessada fazendo uma requisição GET para a seguinte URL:
<https://www.mercadobitcoin.net/api/COIN/METHOD/>. Onde COIN e METHOD são parâmetros que devemos passar. O parâmetro COIN é a criptomoeda cujo valor queremos consultar. Como o Mercado Bitcoin trabalha com outras moedas além do bitcoin, ele nos dá essa opção para configurarmos qual moeda estamos consultando. As opções que podemos usar aqui são:

- **BTC** : Bitcoin
- **LTC** : Litecoin
- **BCH** : BCash

Como estamos consultando o preço do bitcoin, passaremos **BTC** no parâmetro **COIN**.

Já no parâmetro **METHOD** temos as seguintes opções:

- **ticker** : resumo de operações executadas

- `orderbook` : livro de negociações, ordens abertas de compra e venda
- `trades` : histórico de operações executadas

Para nosso caso, a opção `ticker` é a que melhor se encaixa, porque nela teremos um retorno resumido com os valores atuais da cotação da moeda. Nossa URL de requisição ficará assim então: <https://www.mercadobitcoin.net/api/BTC/ticker/>.

A chamada a essa API nos retornará as seguintes informações:

- `high` : Maior preço unitário de negociação das últimas 24 horas.
- `low` : Menor preço unitário de negociação das últimas 24 horas.
- `vol` : Quantidade negociada nas últimas 24 horas.
- `last` : Preço unitário da última negociação.
- `buy` : Maior preço de oferta de compra das últimas 24 horas.
- `sell` : Menor preço de oferta de venda das últimas 24 horas.
- `date` : Data e hora da informação em Era Unix .

Esses dados serão retornados em um objeto JSON, veja um exemplo de retorno:

```
{  
  "ticker": {  
    "high": "14481.47000000",  
    "low": "13706.00002000",  
    "vol": "443.73564488",  
    "last": "14447.01000000",  
    "buy": "14447.00100000",  
    "sell": "14447.01000000",  
    "date": 1502977646  
  }  
}
```

```
    }  
}
```

A informação que estamos buscando está na propriedade `last` que contém o valor da última cotação do bitcoin.

Você pode consultar a documentação completa da API em: <https://www.mercadobitcoin.com.br/api-doc/>.

Agora vamos testar o acesso a API. Para isso, precisaremos fazer uma consulta `GET` na URL da API. Se você copiar a URL que vimos anteriormente e acessar (<https://www.mercadobitcoin.net/api/BTC/ticker>) e colar na barra de endereços do seu navegador de internet você verá na tela o retorno da API em formato JSON:



Figura 9.7: Retorno API

A String retornada pela API parece um pouco confusa se você não conhece o formato JSON, mas na verdade ela é bem simples. O formato JSON é um tipo de formatação de dados baseado em texto e independente de linguagem de programação, e por isso é muito utilizado para troca de informação entre sistemas e também em APIs.

Um objeto JSON é representado por chaves `{ }` e suas propriedades são um conjunto de nome-valor. Por exemplo,

vamos supor que queremos representar uma pessoa de nome igual a João e idade igual a 35. Em JSON essa representação é: { "nome": "João", "idade": 35} . Se João possui também um endereço na Rua 9 de setembro, número 100 esse endereço pode ser representado por um objeto interno:

```
{  
    "nome": "João",  
    "idade": 35,  
    "endereco": {  
        "rua": "9 de setembro",  
        "num": 100  
    }  
}
```

Se quisermos representar um conjunto de dados colocamos esse conjunto entre colchetes []. Por exemplo, um conjunto de pessoas com nome e idade:

```
[  
    { "nome": "João", "idade" : 35},  
    { "nome": "Maria", "idade" : 30},  
    { "nome": "José", "idade" : 40}  
]
```

JSON (*JavaScript Object Notation* - Notação de Objetos JavaScript) é uma formatação leve de troca de dados. Para seres humanos, é fácil de ler e escrever. Para máquinas, é fácil de interpretar e gerar. Leia mais sobre o formato JSON em: <https://www.json.org/json-pt.html>

9.4 BUSCANDO OS DADOS DA API

Vamos começar a botar a mão na massa. Já conhecemos a estrutura da API que vamos acessar e como ela nos retornará a informação, agora é hora de escrever o código que fará esse acesso.

A primeira coisa é dar a permissão de internet ao nosso aplicativo. Como nosso aplicativo fará uma consulta à internet, ele precisa dessa permissão definida no arquivo `AndroidManifest`, caso contrário, o Android bloqueará qualquer acesso à internet vindo do nosso App. Para definir isso, abra o arquivo `AndroidManifest` e adicione a seguinte permissão dentro da tag `<manifest>` e antes da tag `<application>`:

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

Agora vamos configurar a biblioteca Anko no nosso projeto, ela nos ajudará com funções assíncronas para acesso à internet. Abra o arquivo `build.gradle` do projeto e adicione a seguinte variável abaixo da variável `ext.kotlin_version`.

```
ext.anko_version='0.10.4'
```

Agora precisamos adicionar a dependência da biblioteca ao nosso projeto, isso é feito no arquivo `build.gradle` do módulo em `dependencies`. Abra o arquivo `build.gradle` do módulo e adicione a dependência do Anko:

```
implementation "org.jetbrains.anko:anko-commons:$anko_version"
```

Sincronize o projeto clicando no link `Sync Now`.

Com a permissão configurada e a biblioteca adicionada, vamos ao código. Uma informação que precisamos guardar é a URL da API que vamos acessar, então vamos criar uma variável que guarde essa URL e, quando precisarmos acessar a API, simplesmente

usamos a variável. Precisamos também de uma variável para guardar o valor da cotação do bitcoin que a API retornar, então vamos começar nosso código criando essas duas variáveis.

No arquivo `MainActivity`, vamos definir duas variáveis no escopo da classe, uma para guardar a URL da API e uma para guardar o valor da cotação:

```
val API_URL = "https://www.mercadobitcoin.net/api/BTC/ticker/"  
var cotacaoBitcoin:Double = 0.0
```

Agora vamos criar uma função chamada `buscarCotacao`, que será responsável por acessar a API e preencher a variável `cotacaoBitcoin`.

```
fun buscarCotacao() {  
}
```

Em seguida, faça uma chamada à função `buscarCotacao` dentro do método `onCreate` para que a cotação seja atualizada assim que o aplicativo for aberto. Ao final, sua classe `MainActivity` estará assim:

```
class MainActivity : AppCompatActivity() {  
  
    val API_URL = "https://www.mercadobitcoin.net/api/BTC/ticker/"  
  
    var cotacaoBitcoin:Double = 0.0  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        buscarCotacao()  
    }  
}
```

```
fun buscarCotacao(){  
}  
}
```

Agora precisaremos fazer uma chamada `GET` à URL da API. Faremos com que nosso aplicativo faça essa chamada internamente. Ele receberá o retorno da API em JSON e em seguida trataremos essa informação.

Aqui no Kotlin podemos fazer esse tipo de consulta facilmente através do objeto `URL` e a função `readText`. O objeto `URL` receberá a URL da API em seu construtor e, em seguida, podemos chamar o método `readText` que fará de fato o acesso à API e nos retornará uma `String` com o resultado da consulta. A construção do objeto `URL` ficará assim:

```
//Acessar a API e buscar seu resultado  
val resposta = URL(API_URL).readText()
```

Faça o `import` da classe `URL` :

```
import java.net.URL
```

Aqui resolvemos com uma única linha de código! Mas há muita coisa acontecendo nessa linha. Criamos uma instância da classe `URL` passando como parâmetro o link da API: `URL(API_URL)`, depois executamos o método `readText()` para efetuar a chamada à URL da API, por fim, guardamos o resultado na variável `resposta`. Veja quanta coisa está acontecendo em uma única linha de código!

Vamos colocar esse código dentro da função `buscarCotacao`,

assim toda vez que precisarmos fazer uma busca bastará chamar essa função:

```
fun buscarCotacao(){

    //Acessar a API e buscar seu resultado
    val resposta = URL(API_URL).readText()

}
```

Somente com esse código, na teoria, nosso aplicativo já faria a consulta à URL da API e retornaria o resultado na variável `resposta`. Mas na prática ainda existe um problema para resolver, vamos rodar o aplicativo para ver.

Ao executar o aplicativo você encontrará um erro, que será exibido na janela de Logcat do Android Studio:

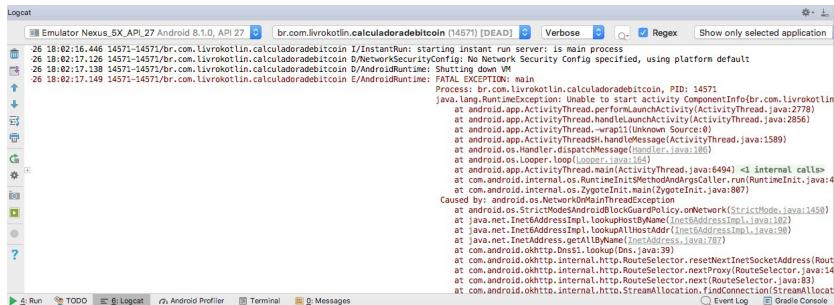


Figura 9.8: Erro Network

Geralmente os erros mostrados são assim, um texto gigante em vermelho com todas as informações sobre o erro. Desse texto todo, o mais importante é analisarmos a causa do erro. Localize uma linha iniciada por "Caused by: " (Causado por), é ali que ele nos dirá porque está ocorrendo tal erro. Vamos analisar a causa do erro do nosso aplicativo:

Caused by: android.os.NetworkOnMainThreadException

Analisando essa linha, vemos que a causa do erro que encontramos é `NetworkOnMainThreadException`. Esse erro diz que ocorreu um erro de chamada à rede (Network) a partir da Thread principal (`MainThread`). Na prática, esse erro diz que estamos fazendo uma chamada à rede na execução do processo principal do nosso aplicativo e isso não é permitido na plataforma Android.

O que acontece é que quando fazemos uma chamada à internet, não temos controle de quanto tempo essa chamada vai demorar para responder; isso depende da rede, do servidor, ou seja, coisas das quais não temos controle. Uma chamada dessas pode ser extremamente rápida ou pode demorar alguns segundos e aí que está o problema.

Imagine que esse servidor demore 5 segundos para responder. Nesses 5 segundos, o aplicativo estará travado na execução da linha de código que faz essa chamada. Para evitar que o aplicativo fique travado, devemos fazer essa chamada em paralelo, uma chamada assíncrona.

Quando fazemos uma chamada assíncrona, o seu processamento acontece em paralelo com o processamento principal do nosso aplicativo, assim se demorar 5, 10 segundos nosso aplicativo não ficará travado.

Executando uma tarefa assíncrona

Para executar uma tarefa assíncrona, o Android possui uma classe chamada `AsyncTask`. Para utilizá-la podemos criar uma classe que herde dela, veja um exemplo:

```
class tarefaAssincrona: AsyncTask<Void, Void, Void>() {  
  
    override fun doInBackground(vararg p0: Void?): Void {  
        //O código que estiver aqui executará em segundo plano  
    }  
  
    override fun onPostExecute(result: Void?) {  
        super.onPostExecute(result)  
        //Quando termina o doInBackground e volta ao processo principal  
    }  
  
}
```

E então executamos a tarefa: `tarefaAssincrona().execute()`. Se fôssemos usar a linha de código que faz a chamada dessa forma, a API deveria estar dentro do método `doInBackground`, mas implementação é um tanto complexa e por isso vamos utilizar a biblioteca Anko para facilitar as coisas.

Com o Anko, para executar uma tarefa assíncrona basta chamar a função `doAsync`, veja um exemplo:

```
doAsync {  
  
    //O código que estiver aqui executará em segundo plano  
  
}
```

Só mais um detalhe a considerar: quando estamos executando uma tarefa em segundo plano através do `AsyncTask` ou através do `doAsync`, estamos executando um código em um processo paralelo, e nesse processo não conseguimos acessar nenhum componente visual do aplicativo porque todas as interações visuais acontecem no processo principal.

A classe `AsyncTask` resolve isso através da implementação do

método `onPostExecute`, que é executado na sequência do `doInBackground`. No contexto do `onPostExecute` o aplicativo já está no processo principal e pode atualizar componentes visuais.

Já o Anko resolve isso através da função `uiThread`, e todo código dentro da `uiThread` necessariamente será executado no processo principal. Veja um exemplo:

```
doAsync {  
    //O código que estiver aqui executará em segundo plano  
  
    uiThread {  
        //Aqui o código volta a ser executado no processo principal  
    }  
}
```

A classe `AsyncTask` é uma classe feita em Java e por isso ela herda a complexidade de um código em Java, já a biblioteca Anko é feita e pensada para o Kotlin, por isso ela consegue ser muito mais simples. Trabalhando com Kotlin não vejo vantagem alguma em utilizar a classe `AsyncTask` a não ser se você precise manter um código legado em Java, mas caso contrário utilize a função `doAsync` juntamente com a função `uiThread`.

Vamos então implementar esse código, coloque o código da função `buscarCotacao` em um bloco `doAsync`:

```
fun buscarCotacao() {  
  
    doAsync {  
        //Acessar a API e buscar seu resultado  
        val resposta = URL(API_URL).readText()  
    }  
}
```

Você precisará importar a função `doAsync` da biblioteca Anko:

```
import org.jetbrains.anko.doAsync
```

Execute o aplicativo novamente e veja que não dará mais o erro de `NetworkOnMainThreadException`.

Tratando o retorno da API

Nesse momento, nosso App já se conecta à API e busca as informações para nós, mas essas informações chegam todas juntas em uma `String`. Para visualizar esse retorno, vamos fazer um `alert` com o conteúdo da variável `resposta`:

```
fun buscarCotacao() {  
  
    doAsync {  
        //Acessar a API e buscar seu resultado  
        val resposta = URL(API_URL).readText()  
  
        uiThread {  
            alert(resposta).show()  
        }  
    }  
}
```

Adicione também os imports das funções `alert` e `uiThread`:

```
import org.jetbrains.anko.alert  
import org.jetbrains.anko.uiThread
```

Executando o aplicativo, você terá algo assim:

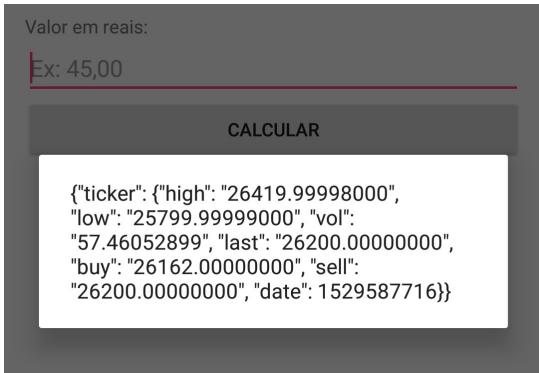


Figura 9.9: Retorno API

Vamos lembrar o JSON que a API do Mercado Bitcoin nos retorna:

```
{  
    "ticker": {  
        "high": "29440.00000000",  
        "low": "26900.00000000",  
        "vol": "236.79181650",  
        "last": "27089.99999000",  
        "buy": "27000.00001000",  
        "sell": "27090.00000000",  
        "date": 1522091000  
    }  
}
```

Analisando a estrutura de retorno, podemos ver que os dados são retornados em um objeto principal e dentro dele tem uma propriedade chamada `ticker`, que também é um objeto. Dentro do objeto `ticker` temos todas as informações de que precisamos.

No Android, existe uma classe chamada `JSONObject` em que conseguimos transformar uma `String` no formato JSON em um objeto que eu consigo trabalhar no código. Através desse objeto posso acessar seus objetos internos e suas propriedades.

Portanto, vamos tratar o retorno da API. Primeiro vamos criar um objeto `JSONObject` com a variável `resposta` :

```
JSONObject(resposta)
```

Você precisará do `import` :

```
import org.json.JSONObject
```

Através desse objeto eu consigo acessar o objeto interno `ticker` :

```
JSONObject(resposta).getJSONObject("ticker")
```

E através do `ticker` conseguimos acessar suas propriedades internas, `high`, `low`, `vol`, `last` etc. Para acessar uma propriedade diretamente devemos saber o tipo de dado que ela possui para poder utilizar o método correto. Por exemplo, se uma propriedade possuir um valor em texto, eu consigo acessá-lo através do método `getString`, se for um valor inteiro, eu consigo acessá-lo pelo método `getInt`. Veja todas as possibilidades:

- `getString` : para acessar valores em texto
- `getInt` : para acessar números inteiros
- `getDouble` : para acessar números com casas decimais
- `getBoolean` : para valores booleanos, verdadeiro ou falso
- `getJSONArray` : para conjunto de dados, arrays
- `getJSONObject` : para objetos JSON

No nosso caso, vamos direto em busca da propriedade `last`, que contém o valor atualizado da cotação do bitcoin. Esse valor é um número decimal, então usaremos o método `getDouble`. Veja como fica o código:

```
JSONObject(resposta).getJSONObject("ticker").getDouble("last")
```

O método `getDouble` já faz a conversão de `String` para `Double` também.

Essa linha de código nos retorna um valor `Double` com a cotação do bitcoin, vamos então guardá-lo na variável `cotacaoBitcoin`, que foi previamente criada.

```
cotacaoBitcoin = JSONObject(resposta).getJSONObject("ticker").getDouble("last")
```

Podemos visualizar a cotação através do comando `alert` que inserimos no nosso código anteriormente:

```
alert("$cotacaoBitcoin").show()
```

Como somente o `alert` terá alguma interação com o usuário, podemos deixar só ele na função `uiThread`, por enquanto. Nossa código está assim:

```
fun buscarCotacao(){  
    doAsync{  
        //Acessar a API e buscar seu resultado  
        val resposta = URL(API_URL).readText()  
  
        cotacaoBitcoin = JSONObject(resposta).getJSONObject("ticker").getDouble("last")  
  
        uiThread {  
            alert("$cotacaoBitcoin").show()  
        }  
    }  
}
```

Agora, antes de atualizar a `TextView` que exibirá a cotação, seria interessante formatarmos esse valor para o formato de dinheiro, utilizando a classe `NumberFormat`, que já nos é conhecida. Vamos então criar uma variável chamada `cotacaoFormatada` e formatar a cotação do bitcoin para o formato de dinheiro:

```
val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))

val cotacaoFormatada = f.format(cotacaoBitcoin)
```

Adicione também os `imports`:

```
import java.text.NumberFormat
import java.util.*
```

Agora podemos atualizar a `TextView` que exibirá a cotação do bitcoin: `txt_cotacao.setText("$cotacaoFormatada")`. Essa linha também fica dentro da função `uiThread`. Podemos até apagar o comando `alert`, estávamos usando-o somente para "debugar" o código.

Ao final, a função ficará assim:

```
fun buscarCotacao(){

    //iniciar uma tarefa assíncrona
    doAsync {

        //Acessar a API e buscar seu resultado
        val resposta = URL(API_URL).readText()

        //Acessando a cotação da String em Json
        cotacaoBitcoin = JSONObject(resposta).getJSONObject("ticker")
            .getDouble("last")

        //Formatação em moeda
    }
}
```

```

        val f= NumberFormat.getCurrencyInstance(Locale("pt", "br"))
    ))
    val cotacaoFormatada = f.format(cotacaoBitcoin)

    uiThread {
        //Atualizando a tela com a cotação atual
        txt_cotacao.setText("$cotacaoFormatada")
    }
}

Adicione também o import
kotlinx.android.synthetic.main.bloco_cotacao.* para
termos acesso aos componentes de UI declarados no XML.

```

Ao executar o aplicativo você verá o valor atualizado na tela:



Figura 9.10: Cotação bitcoin

9.5 EFETUANDO O CÁLCULO DE CONVERSÃO

Agora que já temos o valor atualizado da cotação do bitcoin, vamos para a segunda parte do nosso App e efetuar o cálculo de

conversão. Esse cálculo é muito simples de ser feito: vamos dividir o valor informado pelo usuário pelo valor da cotação e assim obtemos a quantidade de bitcoins a que aquele valor corresponde.

Por exemplo, vamos supor que a cotação do bitcoin esteja em R\$ 30.000 e o usuário informe um valor de R\$ 1.500, então dividiremos 1500 por 30000 que resultará em 0,05000000 ou seja uma fração de bitcoin. Nessa conversão temos que considerar também 8 casas decimais porque é a quantidade de casas decimais com que o bitcoin trabalha.

Para efetuar esse cálculo, vamos criar uma função no nosso código chamada `calcular`, assim separamos um pouco a lógica de cálculo de outras partes do sistema. Abra a `MainActivity` e implemente a função `calcular` logo abaixo da função `buscarCotacao`:

```
fun calcular(){  
}
```

Vamos já fazer um listener para o botão `calcular` chamar a função que acabamos de criar. Essa implementação deve ser feita dentro do método `onCreate` porque queremos registrar esse listener assim que o aplicativo for aberto.

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    buscarCotacao()  
  
    //listener do botão calcular  
    btn_calcular.setOnClickListener {  
        calcular()  
    }  
}
```

```
Adicione          o           import
kotlinx.android.synthetic.main.bloco_entrada.*
```

Agora vamos voltar à implementação da função `calcular`. O primeiro a se fazer nessa função é verificar se a caixa de texto está vazia, porque se o usuário não preencher nada e clicar em `calcular` não vamos efetuar cálculo nenhum, mas sim avisá-lo de que ele deve preencher um valor. Para isso, vamos fazer uma verificação se a propriedade `text` do `txt_valor` está vazia e, caso sim, utilizaremos a propriedade `error` para mostrar uma mensagem de erro ao usuário:

```
fun calcular(){
    if(txt_valor.text.isEmpty()) {
        txt_valor.error = "Preencha um valor"
        return
    }
}
```

Adiante no código podemos pegar o valor digitado pelo usuário. Nesse momento garantimos que ele digitou algum valor porque, caso contrário, ele cairá no `if` feito:

```
//valor digitado pelo usuário
val valor_digitado = txt_valor.text.toString()
```

Como vamos efetuar um cálculo com esse valor devemos transformá-lo para `Double` porque é um valor que pode ter casas decimais. Para isso, vamos simplesmente usar a função `toDouble()`:

```
val valor_digitado = txt_valor.text.toString().toDouble()
```

Aqui temos que pensar em mais um detalhe, o usuário pode usar tanto vírgula quanto ponto para separar casas decimais e

devemos tratar isso no código, porque variáveis Double utilizam ponto para separação de casas decimais. Caso o usuário separe as casas decimais com vírgula, nosso App lançará um erro ao executar o `toDouble()`.

Para tratar isso, podemos substituir as vírgulas por ponto caso o valor que o usuário digite tenha vírgulas. Para substituir caracteres, a classe String possui o método `replace`. Esse método recebe dois parâmetros em String, que são respectivamente o texto antigo e o texto a ser substituído: `replace(oldValue, newValue)`.

Vamos então utilizar o `replace` para trocar as vírgulas por pontos e então chamar o método `toDouble`:

```
val valor_digitado = txt_valor.text.toString().replace(",",".").  
toDouble()
```

Agora podemos fazer de fato a divisão do `valor_digitado` pela `cotacaoBitcoin` e vamos guardar o resultado desse cálculo na variável `resultado`:

```
val resultado = valor_digitado / cotacaoBitcoin
```

Antes de avançar, vamos pensar em um problema que essa linha de código pode dar: se por algum motivo a API do mercado bitcoin não responder e a variável de cotação ficar com valor igual a 0, como fica nossa divisão? Nesse caso, teríamos um resultado `NaN` (*Not a Number*) porque não é possível efetuar uma divisão por zero. Para evitar isso, vamos fazer uma verificação e, caso a `cotacaoBitcoin` for maior que 0, vamos fazer a divisão; caso contrário, vamos definir o `resultado` como 0:

```
val resultado = if(cotacaoBitcoin > 0) valor_digitado / cotacao  
Bitcoin else 0.0
```

Por fim, vamos atualizar a `TextView` que exibirá o resultado do cálculo, e vamos formatar o resultado para exibir exatamente 8 casas decimais.

```
txt_qtd_bitcoins.text = "%.8f".format(resultado)
```

Faça o `import` do bloco_saida :

```
import kotlinx.android.synthetic.main.bloco_saida.*
```

Ao final, sua função `calcular` deve ficar assim:

```
fun calcular(){

    if(txt_valor.text.isEmpty()) {
        txt_valor.error = "Preencha um valor"
        return
    }

    //valor digitado pelo usuário
    val valor_digitado = txt_valor.text.toString()
        .replace(",",".")
        .toDouble()

    //calculando o resultado
    //caso o valor cotação seja maior que zero, efetuamos o cálculo
    //caso contrário devolvemos 0
    val resultado = if(cotacaoBitcoin > 0) valor_digitado / cotacaoBitcoin else 0.0

    //atualizando a TextView com o resultado formatado com 8 casas decimais
    txt_qtd_bitcoins.text = "%.8f".format(resultado)

}
```

Teste o aplicativo e veja o resultado final. Pronto, temos um

App muito bacana para ver a cotação e fazer conversão de bitcoins! A partir daqui fica o desafio para você implementar melhorias nesse App. Por que não buscar a cotação de várias exchanges em vez de uma só e talvez dizer ao usuário qual exchange tem o melhor preço no momento? Criar funcionalidades diferentes é uma oportunidade de aprendizado enorme.

9.6 RESUMINDO

Neste capítulo, vimos algumas coisas muito interessantes e relevantes ao contexto de aplicativos: quantos aplicativos você conhece que buscam alguma informação da internet? Não é exagero dizer que a maioria! A maioria dos aplicativos hoje em dia faze acessos a alguma API na internet para buscar informações, então é muito relevante saber como se faz isso. E ao final, temos ainda um aplicativo muito útil para quem quer acompanhar o mercado de bitcoins.

A partir daqui, você pode ampliar seu conhecimento e criar aplicativos que acessam outros serviços, por exemplo, você pode criar um aplicativo que mostre a previsão do tempo, um aplicativo que converte dólar para reais, um aplicativo que acompanha a cotação da bolsa de valores e por aí vai.

CAPÍTULO 10

NOTIFICAÇÕES, PERMISSÕES, LOCALIZAÇÃO E PUBLICAÇÃO

Neste capítulo, vamos explorar alguns recursos muito úteis ao dia a dia do desenvolvimento de aplicativos. Não trabalharemos um projeto, mas sim recursos isoladamente e depois você será capaz de juntar todo esse conhecimento na produção de um projeto próprio.

Vamos começar explorando o sistema de notificações do Android. Em seguida, vamos ver como funcionam as permissões e como podemos trabalhar com localização através do GPS do aparelho e, por fim, como funciona o processo de publicação de um App na Play Store.

10.1 NOTIFICAÇÕES

Uma notificação é uma mensagem que seu aplicativo envia ao usuário, essa mensagem pode ter um conteúdo somente informativo ou pode indicar alguma ação ao usuário. Além disso,

as notificações não são exibidas dentro do aplicativo, mas sim na área de notificações do Android, que reunirá as notificações de todos os Apps. Veja um exemplo:

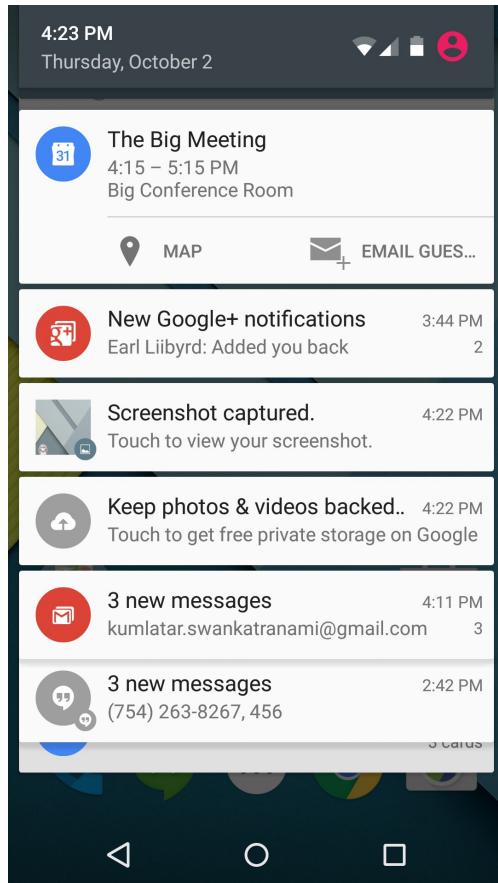


Figura 10.1: Notificações

Para criar uma notificação, primeiro utilizamos o objeto `NotificationCompat.Builder` e depois chamamos o método `build()` que retornará um objeto `Notification`, que é a

notificação em si. Para construir uma notificação simples, devemos definir pelo menos 3 atributos:

- Um ícone pequeno, através do método `setSmallIcon()`
- Um título para a notificação, através do método `setContentTitle()`
- Um texto de detalhes, definido através do método `setContentText()`

Vamos construir esse código, primeiro precisamos instanciar um objeto `NotificationCompat.Builder`:

```
val nBuilder = NotificationCompat.Builder(this)
```

Você precisará importar a classe `NotificationCompat`:

```
import android.support.v4.app.NotificationCompat
```

Na implementação desse código, o Android Studio vai "reclamar" colocando um tachado na chamada `Builder(this)` dizendo que essa forma de se construir o objeto está obsoleta, mas não se preocupe ainda, adequaremos esse código um pouco mais adiante.

Agora precisamos definir um ícone pequeno, para isso usaremos o método `setSmallIcon()` e como parâmetro eu vou passar o ícone padrão do App chamado `ic_launcher`, que se encontra na pasta `mipmap` dentro de `res`:

```
nBuilder.setSmallIcon(R.mipmap.ic_launcher)
```

Agora vamos definir um título para a notificação com o método `setContentTitle`, que recebe uma `String` com o título por parâmetro:

```
nBuilder.setContentTitle("Notificação Android!")
```

O próximo passo é definir um conteúdo de texto para a notificação, para isso vamos usar o método `setContentText` e passar uma `String` por parâmetro:

```
nBuilder.setContentText("Exemplo de notificação em Kotlin !")
```

Por fim, vamos construir o objeto `Notification` através do método `build`:

```
val notificacao = nBuilder.build()
```

Veja como fica esse trecho de código completo:

```
val nBuilder = NotificationCompat.Builder(this)

/*Definindo um ícone pequeno, nesse caso estou definindo o próprio ícone do App
que está sendo acessado por `R.mipmap.ic_launcher`*/
nBuilder.setSmallIcon(R.mipmap.ic_launcher)

//Definindo o título da notificação
nBuilder.setContentTitle("Notificação Android!")

//Definindo o conteúdo da notificação
nBuilder.setContentText("Exemplo de notificação em Kotlin !")

//construindo o objeto Notification
val notificacao = nBuilder.build()
```

Agora, para enviar a notificação, vamos usar o objeto `NotificationManager` que é o gerenciador de notificações do sistema. Teremos acesso a esse objeto através do código `getSystemService(Context.NOTIFICATION_SERVICE)`. O método `getSystemService` retorna o serviço do sistema especificado, algumas coisas no Android são gerenciadas pelo

próprio sistema operacional, como alarme, localização, notificações, entre outros. Então, quando usamos o `getSystemService` devemos passar como parâmetro qual serviço estamos querendo acessar. No caso, estamos acessando o serviço de notificação, por isso passamos `Context.NOTIFICATION_SERVICE` como parâmetro.

Para saber mais sobre os serviços do sistema acesse <https://developer.android.com/reference/android/content/Context>

Então nosso código ficará assim:

```
val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
```

Faça o import das classes `NotificationManager` e `Context`:

```
import android.app.NotificationManager  
import android.content.Context
```

E então chamamos o método `notify` do `NotificationManager` passando um ID para a notificação e o objeto `Notification` criado anteriormente. Esse ID de notificação servirá para gerenciar diferentes notificações no aplicativo.

Imagine, em um aplicativo de mensagens, quando um contato diferente envia uma notificação, ela deve ter um ID único para não interferir em notificações de outros contatos. Veja no código:

```
//enviando a notificação em si  
notificationManager.notify(1 /* ID da notificação */ , notificac  
ao)
```

Para melhorar um pouco o processo de desenvolvimento, podemos separar esse código em uma função que receba o título e o texto da notificação. Essa separação ajudará mais adiante no desenvolvimento porque quando precisar enviar uma notificação ao usuário bastará chamar a função:

```
fun notificacaoSimples(title:String, message:String){  
  
    val nBuilder = NotificationCompat.Builder(this)  
  
    /*Definindo um ícone pequeno, nesse caso estou definindo  
o próprio ícone do App  
    que está sendo acessado por `R.mipmap.ic_launcher`*/  
    nBuilder.setSmallIcon(R.mipmap.ic_launcher)  
  
    //Definindo o título da notificação  
    nBuilder.setContentTitle(title)  
  
    //Definindo o conteúdo da notificação  
    nBuilder.setContentText(message)  
  
    //construindo o objeto Notification  
    val notificacao = nBuilder.build()  
  
    //Acessando o serviço de notificação do sistema  
    val notificationManager = getSystemService(Context.NOTIFI  
CATION_SERVICE) as NotificationManager  
  
    //enviando a notificação em si  
    notificationManager.notify(1 , notificacao)  
  
}
```

Essa função ao ser chamada enviará uma notificação simples,

veja um exemplo de chamada dessa função:

```
notificacaoSimple("Título", "Olá, você está sendo notificado")
```

Para melhorar ainda mais, poderíamos criar uma função de extensão da Activity para notificação, assim em qualquer Activity do projeto bastará chamar a função notificacaoSimple ! Para essa alteração basta adicionar o prefixo Activity ao nome da função:

```
fun Activity.notificacaoSimple(title:String, message:String){  
    ...  
    //Código da função  
}
```

Executando esse código, teremos uma notificação na tela:

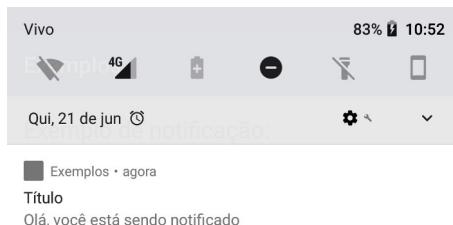


Figura 10.2: Notificação Simples

Observe: para executar esse código, você precisará chamar a função `notificacaoSimples` a partir de alguma ação, que pode ser o término de uma atividade, um botão, um menu etc. Lembre-se de que uma notificação está avisando o usuário de algo que esteja acontecendo ou já aconteceu no aplicativo.

Canais de notificação

Até a chegada do Android Oreo 8.0, esse código funcionaria perfeitamente e se você o executasse em um aparelho com sistema operacional anterior ao 8.0 ele enviaria a notificação sem nenhum problema. No entanto, o Android 8.0 juntamente com a API 26 implementou um novo recurso chamado *canais de notificação*, que permite ao usuário decidir o comportamento das notificações individuais de um mesmo aplicativo.

Na prática, com a criação dos canais de notificação o usuário pode decidir o comportamento das notificações de partes específicas do aplicativo. Por exemplo, o WhatsApp nos permite configurar individualmente as notificações de mensagens individuais, de grupos, de envio de mídias, de falhas etc.

Para nós, desenvolvedores, com essa alteração na API o `NotificationCompat.Builder` precisa receber, além do contexto, um canal de notificação, que posteriormente precisa ser criado no código. Vamos alterar um pouco o código da notificação para implementar um canal de notificação padrão, mas também faremos de forma que a notificação continue funcionando se o

sistema operacional não suportar essa funcionalidade.

A primeira alteração será na criação do `NotificationCompat.Builder`. Vamos passar, além do contexto, uma `String` indicando o `id` do canal de notificação. Essa alteração é necessária porque a API 26 agora nos obriga a indicar o canal de notificação na criação do objeto.

Como `id` do canal de notificação vou passar "default". Usaremos esse mesmo `id` mais adiante para criação do canal de notificação.

```
val nBuilder = NotificationCompat.Builder(this, "default")
```

Agora precisamos criar o canal em si. Mas vamos criar o canal de notificação somente se o sistema operacional suportar essa funcionalidade, então vamos fazer uma verificação na versão do sistema operacional em funcionamento. Para isso, podemos verificar a constante `Build.VERSION.SDK_INT` que nos retornará o número da API daquele sistema operacional, com o qual conseguimos verificar se ele suporta os canais de notificação.

Com o número da API, vamos verificar se ele é maior ou igual à API do Android Oreo 8.0. Poderíamos aqui fazer a verificação simplesmente comparando com o número 26 que é o número da API do Oreo, mas podemos usar a constante `Build.VERSION_CODES.O` para o código ficar mais legível.

A classe `Build.VERSION_CODES` é uma classe estática que contém uma constante indicando as diversas versões do Android, veja algumas das versões que você encontra:

- `Build.VERSION_CODES.LOLLIPOP` (Android Lollipop)

5.0.x - API 21)

- Build.VERSION_CODES.LOLLIPOP_MR1 (Android Lollipop 5.1.x - API 22)
- Build.VERSION_CODES.M (Android Marshmallow 6.x - API 23)
- Build.VERSION_CODES.N (Android Nougat 7.x - API 24)
- Build.VERSION_CODES.N_MR1 (Android Nougat 7.1.x - API 25)
- Build.VERSION_CODES.O (Android Oreo 8.x - API 26)

Confira uma lista completa em:

https://developer.android.com/reference/android/os/Build.VERSION_CODES

A verificação fica assim:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
    //Criação do canal de notificação  
}
```

Faça o import da classe Build :

```
import android.os.Build
```

Para criação do canal de notificação, usaremos a classe `NotificationChannel`, cujo construtor recebe 3 parâmetros:

- `id` : uma String que represente o id desse canal; vamos passar o mesmo id que usamos no Builder da notificação.
- `name` : uma String com o nome desse canal de notificação; esse nome deve representar o canal em si pois é esse nome que o usuário verá quando estiver configurando

as notificações do App.

- `importance` : um `Int` que representa a importância desse canal; temos algumas opções de constantes para usar aqui:
 - `NotificationManager.IMPORTANCE_DEFAULT` : importância padrão
 - `NotificationManager.IMPORTANCE_HIGH` : alta importância
 - `NotificationManager.IMPORTANCE_LOW` : baixa importância
 - `NotificationManager.IMPORTANCE_MAX` : importância máxima
 - `NotificationManager.IMPORTANCE_MIN` : importância mínima
 - `NotificationManager.IMPORTANCE_NONE` : sem importância
 - `NotificationManager.IMPORTANCE_UNSPECIFIED` : não especificada

Criaremos um objeto `NotificationChannel` e, em seguida, utilizaremos o método `createNotificationChannel` do `NotificationManager` para criar o canal. O código ficará assim:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
  
    //criação do objeto  
    val channel = NotificationChannel("default",  
        "Canal de notificação teste",  
        NotificationManager.IMPORTANCE_DEFAULT)  
  
    //criar o canal de notificação  
    notificationManager.createNotificationChannel(channel)  
}
```

Não esqueça do import de NotificationChannel:

```
import android.app.NotificationChannel
```

Vamos incluir esse trecho de código na função notificacaoSimples que criamos anteriormente. Esse pedaço de código vai logo após a criação do notificationManager porque o canal será registrado nesse mesmo objeto.

```
fun Activity.notificacaoSimples(title:String, message:String){  
    //Restante do código  
    ...  
  
    val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager  
  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
  
        //criação do objeto  
        val channel = NotificationChannel("default", "Canal de notificação teste", NotificationManager.IMPORTANCE_DEFAULT)  
  
        //criar o canal de notificação  
        notificationManager.createNotificationChannel(channel  
    )  
}  
  
    //enviando a notificação em si  
    notificationManager.notify(1 , notificacao)  
}
```

Veja como aparece esse canal de notificação nas configurações do Android:

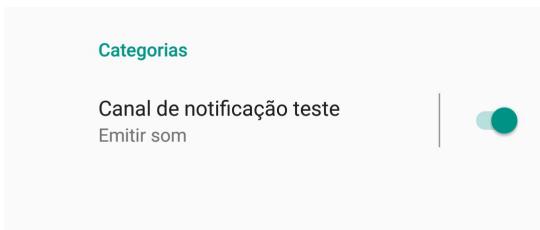


Figura 10.3: Canal de notificação

10.2 SOLICITAÇÃO DE PERMISSÕES

Até a chegada do Android 6.0 (API 23), os usuários concediam as permissões do App na hora da sua instalação. Já com a API 23, as permissões devem ser concedidas em tempo de execução. Para nós, desenvolvedores, foi uma mudança significativa porque precisamos solicitar a permissão de algum recurso pelo código em tempo de execução, isto é, quando o recurso for ser utilizado.

Por exemplo, imagine que você esteja desenvolvendo um aplicativo que acesse a câmera do aparelho para capturar uma foto ou um vídeo. Nesse caso, você deverá solicitar ao usuário a permissão para acessar a câmera somente no momento em que o aplicativo for utilizar a câmera, e também deverá tratar o caso de o usuário não conceder a permissão.

Antes de solicitar a permissão de fato ao usuário, devemos verificar se ele já não concedeu essa permissão ao App. Toda vez que seu App for utilizar um recurso que requer permissão, o App deve verificar se a permissão está concedida. Para verificar o estado de uma determinada permissão, a API nos disponibiliza o método:

`ContextCompat.checkSelfPermission` que recebe como parâmetro o contexto a permissão que você deseja verificar.

Atenção: todas as permissões ainda precisam estar declaradas no arquivo `AndroidManifest`. Para o exemplo a seguir você precisará adicionar a seguinte linha ao seu arquivo de manifest:

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

Essa linha deve estar fora da tag de `<application />`. Geralmente, declaramos as permissões logo no início do arquivo.

Veja o seguinte exemplo para verificar o estado da permissão `READ_CONTACTS` (leitura de contatos):

```
ContextCompat.checkSelfPermission(this, Manifest.permission.READ_CONTACTS)
```

Se a permissão estiver concedida, esse código nos retornará: `PackageManager.PERMISSION_GRANTED` ; caso contrário retornará: `PackageManager.PERMISSION_DENIED` .

Então, se eu quero ver se uma determinada permissão está concedida, basta colocar esse código em uma estrutura de decisão:

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_CONTACTS)
    == PackageManager.PERMISSION_GRANTED) {
    //Permissão concedida
} else {
    //Permissão não está concedida
```

```
}
```

Neste trecho de código você precisará importar as seguintes classes:

```
import android.Manifest  
import android.support.v4.content.ContextCompat  
import android.content.pm.PackageManager
```

Caso você verifique que a permissão está concedida, basta continuar com o código que usará tal recurso. Caso não esteja concedida, precisaremos perguntar ao usuário. Para isso, a API nos disponibiliza o método `ActivityCompat.requestPermissions`, que recebe como parâmetro o contexto, um array com as permissões (nesse caso podemos requisitar mais de uma ao mesmo tempo) e um ID de requisição.

O ID da requisição é um número inteiro gerenciado pelo desenvolvedor, ele será importante mais adiante quando formos pegar a resposta do usuário na função de callback e trataremos cada requisição pelo seu ID. Sendo assim, cada requisição deve ter um ID diferente. É uma boa prática defini-los como variáveis imutáveis, por isso o código a seguir usará uma `val` que chamei de `ID_REQUISICAO_READ_CONTACTS`. Geralmente, definimos essas variáveis no escopo da Activity. Veja um exemplo:

```
val ID_REQUISICAO_READ_CONTACTS = 1
```

E agora a chamada da função `requestPermissions` requisitando a permissão `READ_CONTACTS` (leitura de contatos):

```
ActivityCompat.requestPermissions(this,  
    arrayOf(Manifest.permission.READ_CONTACTS),  
    ID_REQUISICAO_READ_CONTACTS)
```

Vale lembrar que esse código fica no `else` do `if` feito

acima, pois só precisaremos requisitar uma permissão para o usuário se ela ainda não estiver concedida.

Para esse trecho de código você precisará importar a classe `ActivityCompat`:

```
import android.support.v4.app.ActivityCompat
```

Aqui estamos chamando a função `requestPermissions` passando primeiro o contexto como `this`, depois o array com as permissões: `arrayOf(Manifest.permission.READ_CONTACTS)`. Perceba que nesse caso estamos requisitando uma única permissão, mas mesmo assim é necessário passar dentro de um array, por isso usamos a função `arrayOf` para criação de um array. E por último, passamos o ID da requisição através da constante `ID_REQESTAO_READ_CONTACTS`.

Quando esse código for executado, o aplicativo exibirá uma caixa de diálogo em que é perguntado ao usuário se ele permite ao aplicativo acesso àquele recurso. Neste momento, o usuário pode tomar duas decisões: permitir ou não permitir. Em qualquer caso, devemos capturar a resposta.

Para isso, devemos implementar na Activity o método `onRequestPermissionsResult`, que retornará todas as requisições de permissão. A estrutura do método é a seguinte:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
}
```

Esse método possui 3 variáveis:

- `requestCode` : o código da requisição; este é o código que enviamos quando vamos fazer a requisição e com ele podemos controlar com qual requisição estamos lidando.
- `permissions` : as permissões requisitadas.
- `grantResults` : a resposta do usuário, se a permissão foi concedida ou não.

Sendo assim, devemos primeiro verificar qual requisição estamos tratando a partir de seu código:

```
if(requestCode == ID_REQUSICAO_READ_CONTACTS){  
    //Sei que aqui é a resposta da requisição de leitura de contatos feita anteriormente  
}  
}
```

Em seguida, dentro do `if`, verificamos se o usuário concedeu a permissão ou não:

```
if(grantResults.size > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {  
    //A permissão foi concedida, o aplicativo pode utilizar o recurso  
}  
} else {  
    //A permissão não foi concedida, aqui você deverá desabilitar a funcionalidade que utiliza tal recurso  
}
```

Ao final, o código do `onRequestPermissionsResult` ficará assim:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults: IntArray) {  
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
```

```
if(requestCode == ID_REQUSICAO_READ_CONTACTS) {  
  
    if(grantResults.size > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {  
  
        //A permissão foi concedida, o aplicativo pode utilizar o recurso  
  
    } else {  
  
        //A permissão não foi concedida, aqui você deverá desabilitar a funcionalidade que utiliza tal recurso  
    }  
  
}  
  
}
```

Essa função é responsável por receber a resposta do usuário, por isso aqui estamos validando primeiro se o código da requisição é o mesmo que foi requisitado: `if(requestCode == ID_REQUSICAO_READ_CONTACTS)`. Caso sim, estamos verificando se o usuário concedeu a permissão ou não.

Para testar esse código, você pode criar um botão e implementar essa lógica através do clique deste botão como na seguinte imagem de exemplo:

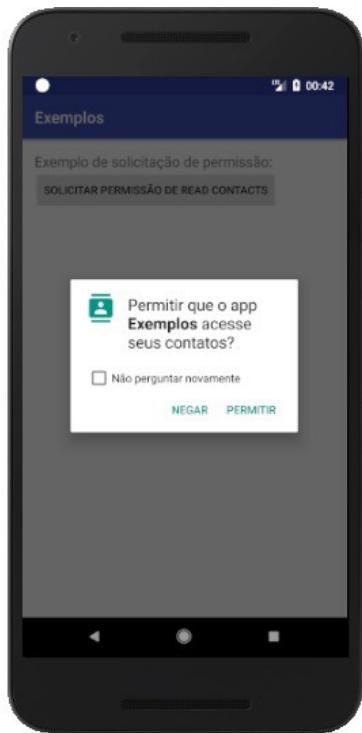


Figura 10.4: Requisitando permissão

A qualquer momento, o usuário pode acessar as configurações do sistema e tirar qualquer permissão que já foi concedida e, nesse caso seu App pedirá a permissão novamente.

10.3 LOCALIZAÇÃO POR GPS

O recurso de geolocalização está presente hoje em muitos

aplicativos, desde mapas e navegação até aplicativos de relacionamentos que utilizam a localização do usuário para conectar a pessoas próximas. Existe ainda uma gama de possibilidades que podem ser exploradas com geolocalização para melhorar a experiência do usuário com seu aplicativo.

Obter a localização do usuário na plataforma Android nunca foi uma tarefa complexa, muito pelo contrário, sempre foi uma tarefa fácil e com as atualizações de API isso só foi melhorado.

A classe LocationManager

Uma coisa importante que devemos entender para usar o LocationManager são os *locations providers* (provedores de localização). O location provider é basicamente a forma como a localização do usuário é obtida, isto é, uma interface que retornará a localização do usuário com base em alguma métrica.

Aqui no Android temos 3 tipos de provedores de localização que podemos utilizar:

- **GPS_PROVIDER** : esse provedor determina a localização do usuário com base no GPS do aparelho. Dependendo do local e condições externas, esse provedor pode demorar algum tempo para determinar a localização do usuário. A utilização desse provedor requer a permissão `ACCESS_FINE_LOCATION` .
- **NETWORK_PROVIDER** : este provedor utiliza a rede de antenas da rede móvel ou dados do access point de conexões Wi-Fi para determinar a localização do usuário. Esse provedor não é tão preciso quanto a localização exata

do usuário, porém ele tem uma resposta rápida à solicitação. A utilização desse provedor requer a permissão `ACCESS_COARSE_LOCATION` ou você pode usar também a permissão `ACCESS_FINE_LOCATION`.

- `PASSIVE_PROVIDER` : este é um provedor especial porque ele se aproveita da localização obtida por outros aplicativos para determinar a localização do usuário, então de fato ele não espera dados do GPS nem da rede, ele simplesmente busca dados que já foram gerados pelos outros provedores em algum outro aplicativo. Esse provedor requer a permissão `ACCESS_FINE_LOCATION`.

A utilização de um ou outro depende muito do tipo de aplicativo que você está desenvolvendo, mas uma solução comum é combinar a utilização dos provedores. Por exemplo, quando seu aplicativo precisar determinar a primeira localização do usuário ele pode usar um provedor de rede (`NETWORK_PROVIDER`) ou um provedor passivo (`PASSIVE_PROVIDER`), assim você consegue um resultado rápido e já exibe ao usuário; em paralelo você também pode requisitar a localização do provedor de GPS (`GPS_PROVIDER`) e, quando ele responder, você atualiza a informação para o usuário! Não existe uma estratégia perfeita, cada aplicativo vai demandar uma estratégia diferente e cabe a você pensar em qual o melhor provedor de localização para utilizar naquele momento.

Vamos fazer um pequeno exemplo de como utilizar a classe `LocationManager`. Para a implementação desse código, primeiro vamos acessar o serviço de localização do Android; em seguida, criaremos um listener (ouvinte) que tratará as alterações de

localização do usuário e depois requisitaremos as atualizações com base em um provedor.

Vamos começar acessando a instância da classe `LocationManager`. Vamos utilizar o método `getSystemService` para ter acesso aos serviços do sistema e vamos passar como parâmetro a constante `Context.LOCATION_SERVICE`. O retorno disso será um objeto do tipo `LocationManager`:

```
val locationManager = getSystemService(Context.LOCATION_SERVICE)  
as LocationManager
```

Você precisará importar a classe `LocationManager`:

```
import android.location.LocationManager
```

Dica: se você estiver utilizando a biblioteca Anko no seu projeto, você não precisará instanciar essa variável porque a biblioteca implementa uma propriedade de extensão da classe `Context`, chamada `locationManager`, que abstrai essa chamada. Então, se estiver usando a biblioteca Anko basta utilizar a propriedade `locationManager` diretamente porque ela estará disponível na própria Activity.

Agora precisaremos criar um listener que tratará qualquer mudança enviada pelo `locationManager`. Para esse fim, o Android possui uma interface chamada `LocationListener` com a assinatura dos métodos responsáveis por gerenciar as mudanças de localização.

O primeiro método é o `onLocationChanged` com a seguinte assinatura:

```
override fun onLocationChanged(location: Location?)
```

O método `onLocationChanged` é invocado quando a localização do usuário for encontrada ou sofrer alguma movimentação.

O próximo método que a interface implementa é o `onStatusChanged` e possui a seguinte assinatura:

```
override fun onStatusChanged(provider: String?, status: Int, extras: Bundle?)
```

Este método é invocado quando um status de provedor é alterado. Os seguintes status são possíveis:

- `LocationProvider.AVAILABLE` : disponível
- `LocationProvider.OUT_OF_SERVICE` : fora de serviço
- `LocationProvider.TEMPORARILY_UNAVAILABLE` : temporariamente indisponível

O próximo método da interface é o `onProviderDisabled` :

```
override fun onProviderDisabled(provider: String?)
```

Este método é invocado quando um provedor é desabilitado.

Por fim, temos o método `onProviderEnabled` com a seguinte assinatura:

```
override fun onProviderEnabled(provider: String?)
```

Este método é invocado quando um provedor é habilitado.

Precisamos criar uma classe que implemente essa interface

para que possamos obter os dados de localização do usuário. Não é uma má ideia criar uma classe interna dentro da própria Activity, pois assim temos acesso aos componentes de UI e fica mais fácil a atualização da visualização do usuário. No seguinte exemplo, crio uma classe chamada `MyLocationListener` e implemento a interface `LocationListener` :

```
class MyLocationListener : LocationListener{  
  
    override fun onLocationChanged(location: Location?) {  
    }  
  
    override fun onStatusChanged(provider: String?, status: Int,  
extras: Bundle?) {  
  
    }  
  
    override fun onProviderDisabled(provider: String?) {  
  
    }  
    override fun onProviderEnabled(provider: String?) {  
  
    }  
}
```

Você precisará do `import` da interface `LocationListener` :

```
import android.location.LocationListener
```

Agora precisamos requisitar as atualizações de localização e faremos isso através da variável `locationManager` chamando o método `requestLocationUpdates`. Na chamada desse método devemos passar por parâmetro o provedor que vamos utilizar, o tempo de atualização (um número `Long` correspondente ao tempo em segundos que o listener levará para receber as atualizações), uma distância `Float` correspondente à distância mínima que o dispositivo se deslocará para enviar uma atualização

ao listener e, por fim, o próprio `LocationListener`.

Primeiro vamos separar esses parâmetros em algumas variáveis:

```
val tempoAtualizacao:Long = 0  
val distanciaAtualizacao:Float = 0f  
val locationListener = MyLocationListener()
```

As variáveis `tempoAtualizacao` e `distanciaAtualizacao`, ambas com valor 0, significa que esse listener vai reagir a qualquer mudança na localização do usuário, tanto em tempo quanto em distância.

Agora vamos requisitar atualizações de um provedor de rede (`NETWORK_PROVIDER`):

```
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,  
    tempoAtualizacao,  
    distanciaAtualizacao,  
    locationListener)
```

Podemos requisitar também usando o provedor de GPS (`GPS_PROVIDER`):

```
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,  
    tempoAtualizacao,  
    distanciaAtualizacao,  
    locationListener)
```

E usando o provedor passivo (`PASSIVE_PROVIDER`):

```
locationManager.requestLocationUpdates(LocationManager.PASSIVE_PROVIDER,  
    tempoAtualizacao,  
    distanciaAtualizacao,  
    locationListener)
```

Como a localização do usuário é um dado pessoal, seu aplicativo precisará requisitar a permissão do usuário para utilizá-la. Você precisará da permissão de ACCESS_FINE_LOCATION e, se seu aplicativo for direcionado à versão 5.0 ou superior, você deverá adicionar a permissão de uso de hardware android.hardware.location.gps .

Se você for utilizar somente o provedor de rede (NETWORK_PROVIDER), você precisará somente da permissão ACCESS_COARSE_LOCATION , no entanto, se você utilizar a ACCESS_FINE_LOCATION , ela também funcionará, porque ela embute a outra permissão. Resumindo, a permissão ACCESS_FINE_LOCATION funciona para os dois casos, a ACCESS_COARSE_LOCATION funciona somente para provedor de rede.

Todas devem estar declaradas no AndroidManifest .

```
<manifest ... >
    <uses-permission android:name="android.permission.ACCESS_FINE
.LOCATION" />
    ...
    <!-- Necessário se seu aplicativo for direcionado ao Android
5.0 (API 21) ou superior. -->
    <uses-feature android:name="android.hardware.location.gps" />
    ...
</manifest>
```

Além disso, você deverá implementar o código de requisição dessas permissões em tempo de execução. Veja a seção anterior **Solicitação de permissões**.

Trabalhar com o LocationManager definitivamente não é nada complexo, no entanto trabalhar com dados de GPS pode ser um desafio, porque requer que o desenvolvedor pense em formas

de otimizar seu uso, uma vez que este recurso pode consumir muito a bateria do aparelho.

O Google disponibiliza uma API para trabalhar com localização chamada *Fused Location Provider*, que pode ser acessada através da biblioteca Google Play Services. Ela encapsula otimizações necessárias para se trabalhar com dados de GPS e por isso vem sendo uma boa alternativa. Não vou implementar o código dessa biblioteca neste livro, mas você pode conferir sua documentação oficial e tentar fazer uma aplicação simples utilizando-a.

Veja o link da documentação em
<https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient>

10.4 PUBLICAÇÃO NA PLAY STORE

Ter um aplicativo publicado na Play Store não é uma tarefa complexa, muito pelo contrário, a loja do Google é muito amigável e o processo de publicação é bem tranquilo! Mas antes de publicarmos, precisamos preparar nosso aplicativo para publicação - precisamos gerar o APK de release (lançamento).

O APK é o arquivo compilado do nosso aplicativo que podemos gerar a qualquer momento do desenvolvimento e esse APK pode ser instalado em um sistema Android se o aplicativo suportar a versão do aparelho. Vamos começar vendo como gerar o APK do seu aplicativo.

Gerando o APK

O Android Studio nos ajuda muito na geração do APK, você pode fazê-lo por meio do menu Build > Generate APK(s) :

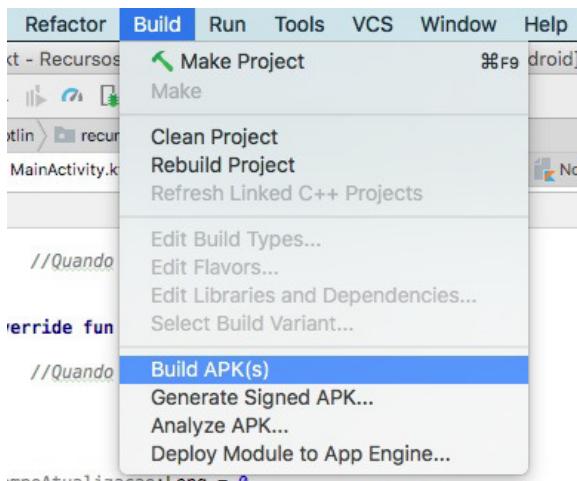


Figura 10.5: Gerar APK

Esse menu vai gerar um APK não assinado, ou debug, ou release, dependendo da configuração em que estiver o Android Studio. Geralmente, essa opção vai gerar um APK de debug que é basicamente a versão de desenvolvimento - ainda não é a finalizada. A versão de release é a versão de lançamento, ou seja, a finalizada e pronta para os usuários instalarem em seus aparelhos.

Você consegue mudar entre debug e release na janela Build Variants que fica no canto inferior esquerdo do Android Studio, ou no menu View > Tool Windows > Build Variants :

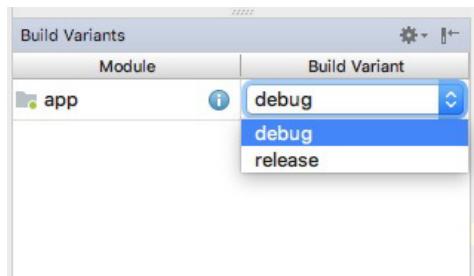


Figura 10.6: Janela Build Variants

Os arquivos gerados poderão ser encontrados na pasta: `app/build/outputs/apk/`. Lá você, encontrará uma pasta `debug` e uma pasta `release` e os APKs estarão em suas respectivas pastas.

Mas esses arquivos ainda não são os que serão publicados na Play Store por um motivo: eles não estão assinados digitalmente! Como eu tinha dito, essa opção gera um APK não assinado, isto é, ele não foi assinado digitalmente com um certificado. Um certificado digital é uma garantia de autenticidade. Quando criamos um APK assinado digitalmente, estamos garantindo a autenticidade daquele arquivo. Todo esse processo será feito pelo próprio Android Studio. Com ele conseguimos gerar um certificado digital e criar o arquivo assinado.

Se esse APK não assinado não pode ser enviado à Play Store, para que ele serve, então? A resposta é: para todo o resto que não seja a publicação na Play Store ou que não exija uma assinatura digital! Você pode enviar esse APK para um amigo instalar no celular dele, você pode instalar no seu celular, você pode disponibilizar para download fora da loja etc.

Um exemplo de aplicativo disponível na internet fora da loja oficial é o Libreflix, um projeto open-source cujo aplicativo ajudei a construir. Você pode conferir o Libreflix em <https://libreflix.org/apps>

Então, para enviar um APK para a Play Store, devemos ter um arquivo assinado. O processo de criação desse arquivo também é relativamente simples, mas antes de ver como criar um APK assinado vamos ver algumas configurações importantes sobre o controle de versão do nosso App.

São duas configurações importantes a serem observadas: `versionCode` e `versionName`, que são respectivamente o número da versão do código e o nome dessa versão. O `versionCode` é um número inteiro incremental, começamos em 1 e, conforme ocorrem atualizações do aplicativo, vamos incrementando para 2, 3, 4 etc.

O `versionName` é o nome da versão que vai aparecer para os usuários na Play Store. Geralmente, chamamos a primeira versão de "1.0" e conforme há atualizações vamos modificando o nome da versão: "1.1", "1.2" ... "2.0" etc. Aqui o desenvolvedor tem a liberdade de controlar o nome da versão de acordo com as métricas que ele preferir.

Uma sugestão é nomear mudanças menores, como correções de bugs, como incremento de uma mesma versão. Por exemplo, imagine que você lançou um aplicativo com versão "1.0" e uma semana depois precisou fazer uma correção de bug e lançará uma

nova versão. Nesse caso, você pode nomeá-la como "1.1" e lançar a atualização. Assim, se você precisar fazer novas correções ou mudanças menores, você vai atualizando para "1.2", "1.3" etc. Quando você implementar funcionalidades novas, você pode mudar a versão para "2.0" e continuar seguindo a mesma lógica. É comum também utilizar 3 unidades de separação, isto é, em vez de chamar a versão de "1.0" você pode chamá-la de "1.0.0".

Essas configurações são feitas no arquivo `build.gradle` do App. Abrindo o arquivo `build.gradle`, você as encontrará dentro da seção `defaultConfig`:



```
android {
    compileSdkVersion 26
    defaultConfig {
        applicationId "br.com.livrokotlin.recursosandroid"
        minSdkVersion 26
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

Figura 10.7: Versão do código e nome da versão

Depois de conferido, e feitas as mudanças necessárias, vamos à criação do APK assinado. Para isso, acesse o menu `Build > Generate Signed APK`:

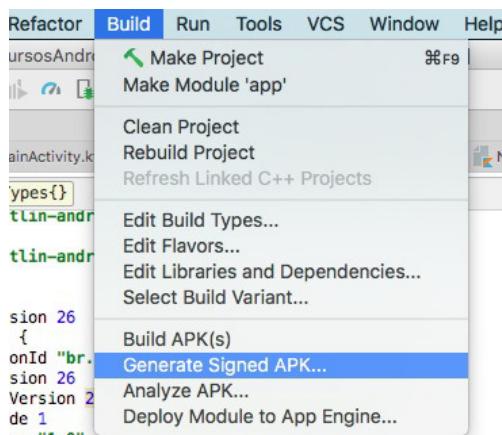


Figura 10.8: Gerar APK assinado

Ao clicar nessa opção, você cairá em uma janela para selecionar o módulo cuja APK você quer gerar:

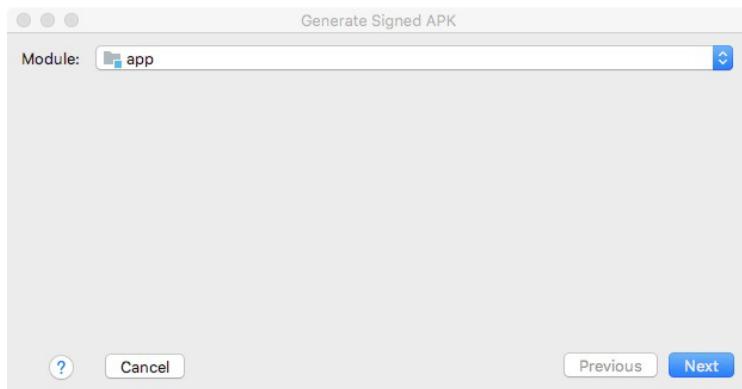


Figura 10.9: Selecionar módulo

Se você não estiver trabalhando com mais de um módulo no mesmo projeto, só terá uma opção para seleção nessa tela. Clicando em **Next** você cairá nesta tela:

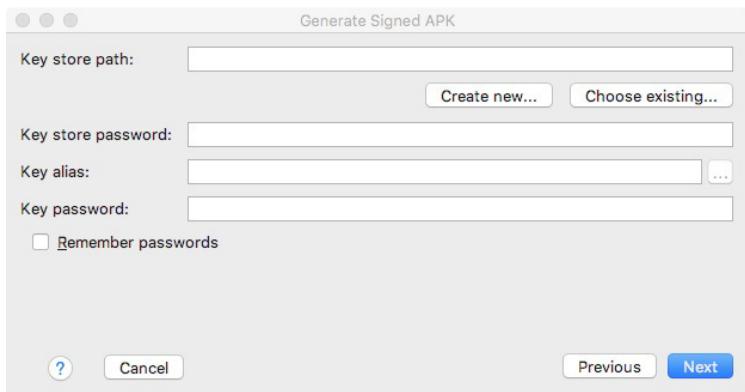


Figura 10.10: Selecionar arquivo de assinatura

Nessa tela, você pode criar um novo arquivo de assinatura ou escolher um existente. Se for um aplicativo novo, crie um novo arquivo através do botão `Create new...` para assiná-lo; se for um arquivo que já foi assinado você deve selecionar o mesmo arquivo usado anteriormente através do botão `Choose existing...`. Vamos seguir os passos de criação de um novo arquivo clicando no botão `Create new...`.

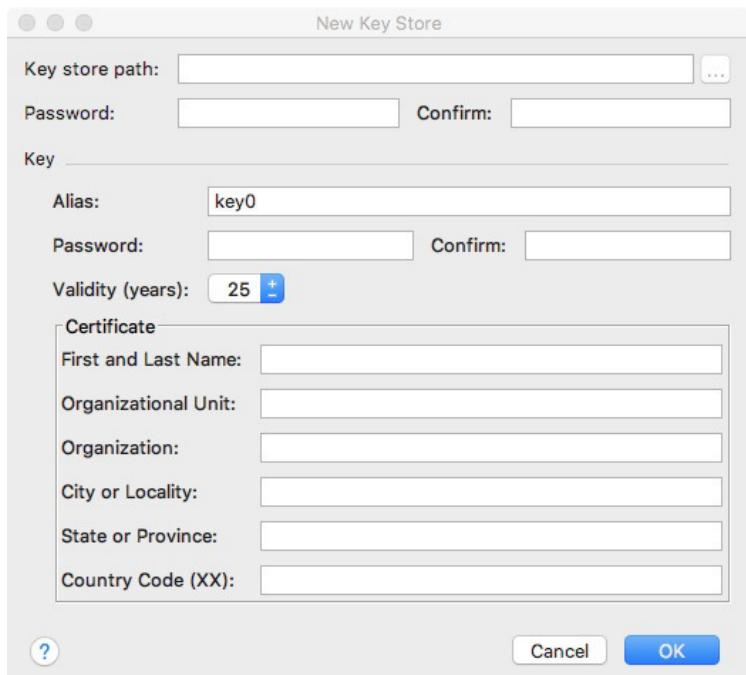


Figura 10.11: Novo arquivo de chave

Aqui você deve preencher alguns campos:

- Key store path : caminho para salvar o arquivo.
- Password : senha para o arquivo.

Key (Chave):

- Alias : um "apelido" para essa chave; esse nome é de escolha do desenvolvedor. Eu geralmente deixo o Alias com o mesmo nome do App.
- Password : senha para a chave que está sendo criada. Essa senha pode ser diferente da senha anterior, do arquivo, ou você pode manter a mesma senha para não confundir, no

entanto, para melhor segurança é bom ter senhas diferentes.

- **Validity** : validade desse certificado em anos.
- **First and Last Name** : primeiro e último nome, pode ser o seu nome.
- **Organizational Unit** : nome da sua unidade organizacional (campo opcional).
- **Organization** : nome da sua organização (campo opcional).
- **City or Locality** : cidade ou localização (campo opcional).
- **State or Province** : Estado ou província (campo opcional).
- **Country Code(XX)** : código do país (campo opcional).

ATENÇÃO: guarde muito bem esse arquivo, o Alias que você definiu e a senha, porque se você assinar um APK com ele e enviar para a loja, qualquer atualização desse aplicativo deverá ser assinada com esse mesmo arquivo. Caso você o perca, você não consegue mais atualizar seu App na loja, então muito cuidado com esse arquivo e suas senhas.

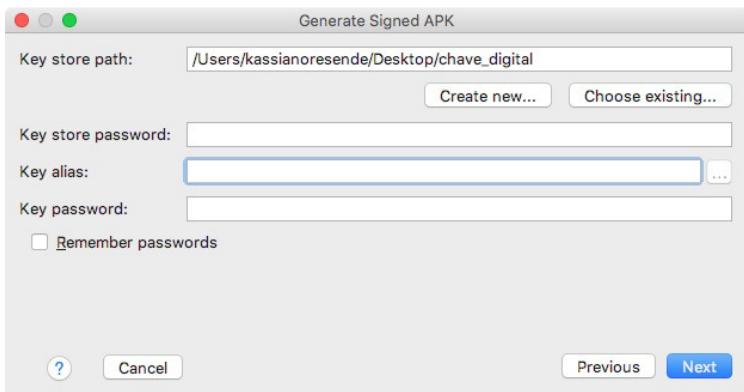


Figura 10.12: Senhas da chave

Depois da criação do arquivo, você precisará completar essa tela com as senhas e o Alias configurado no passo anterior. Ao clicar em **Next** :

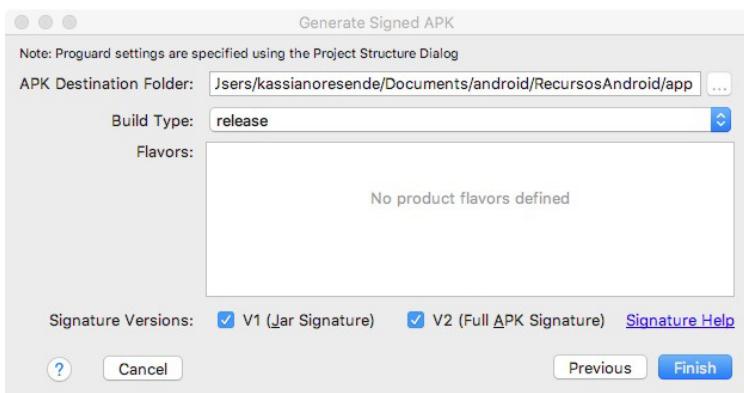


Figura 10.13: Finalizando o APK assinado

Nessa última tela, você pode escolher o **Build type** (tipo de compilação) entre **release** (lançamento) ou **debug** (desenvolvimento). Como vamos enviar à loja, este é um build de

release. Em Signature Versions (versões de assinatura) selecione as duas opções v1 e v2 . O tipo de assinatura v2 é relativamente novo no Android, antes o padrão era a v1 ; a v2 veio junto com a versão 7.0 do Android, e de acordo com a documentação do Android devemos escolher os dois tipos! Por fim, clique em Finish .

E seu APK assinado estará dentro da pasta app/release . Você usará esse arquivo um pouco mais adiante para enviar seu App para a Play Store.

Play Store

Para ter seu aplicativo publicado, você precisará de uma conta de desenvolvedor criada na Play Store. Para criar a conta, acesse o link: <https://play.google.com/apps/publish>

Na primeira tela, você deve efetuar login com alguma conta do Google. Em seguida, você será direcionado à seguinte tela:

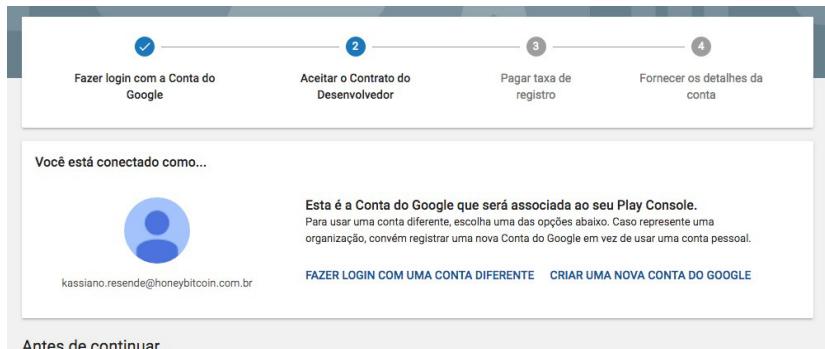


Figura 10.14: Criação de conta desenvolvedor

Nessa tela, você deve conferir os dados, ler o contrato do

desenvolvedor e aceitá-lo para continuar.

Antes de continuar...

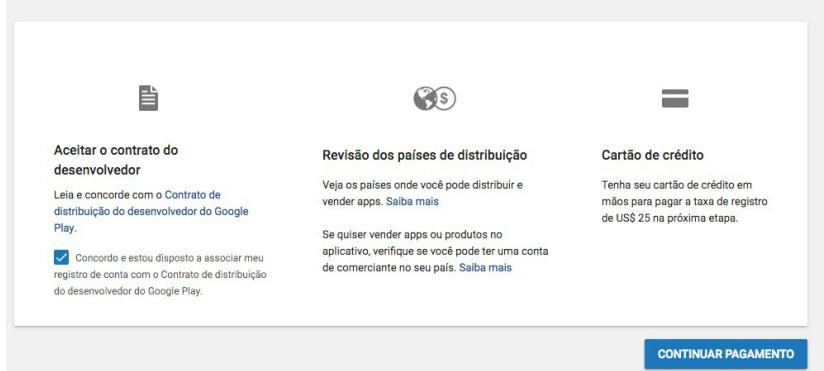


Figura 10.15: Criação de conta desenvolvedor

O próximo passo é efetuar o pagamento, a taxa cobrada é no valor de US\$25,00. Essa taxa é paga uma única vez e não necessita de renovação.

X Conclua sua compra :

Google Play US\$ 25,00
Developer Registration Fee

Adic novo cartão de crédito

Número do cartão # MM / AA CVC

O número do cartão é obrigatório

Nome do titular do cartão

O nome do titular do cartão é obrigatório

Endereço de faturamento

Ao continuar, você cria uma conta do Google Payments e concorda com [Termos de Serviço - Comprador \(BR\)](#) e [Aviso de privacidade](#).

PAGAR

Figura 10.16: Criação de conta desenvolvedor

Depois do pagamento, você preencherá mais alguns detalhes sobre sua conta de desenvolvedor e pronto!

Já logado na Play Store como desenvolvedor você poderá publicar novos aplicativos, gerenciar seus aplicativos já publicados, acessar relatórios etc.

Para publicar um aplicativo, você deve clicar no botão PUBLICAR UM APP PARA ANDROID NO GOOGLE PLAY e ir até a tela

Criar app , escolher o idioma padrão e preencher um título para o App:

Criar app

Idioma padrão *

Português (Brasil) – pt-BR

Título *

0/50

CANCELAR CRIAR

Figura 10.17: Criar novo App

Você caíra na tela de Detalhes do app , na qual você vai preencher uma descrição detalhada, enviar prints de tela, escolher a categoria do aplicativo etc.

Todos os apps

Versões de apps

Instant Apps Android

Biblioteca de artefatos

Catálogo de dispositivos

Assinatura de apps

Detalhes do app

Classificação de conteúdo

Preços e distribuição

Produtos no aplicativo

Serviço de tradução

Detalhes do produto

Título *

Português (Brasil) – pt-BR

Breve descrição *

Português (Brasil) – pt-BR

Descrição completa *

Português (Brasil) – pt-BR

POR
TUGU
S (BRAS
IL) – pt-BR

Gerenciar traduções

Os campos marcados com * devem ser preenchidos antes de publicar.

Teste

5/50

0/80

SALVAR RASCUNHO

Figura 10.18: Detalhes do App

Depois de preencher todos, o próximo passo é enviar o APK para a loja. Para isso, você deve clicar no link **Versões de apps** do menu lateral e na tela que se abrirá você terá basicamente 3 opções: criar uma versão de produção, uma versão Beta ou uma versão Alfa.

As versões Alfa e Beta são versões para testes que podem ser abertos ou fechados de acordo com o que o desenvolvedor configurar, a versão de produção é a final. Seguindo um fluxo de desenvolvimento padrão, você lançará uma versão Alfa primeiro, realizar todos dos testes e, se estiver tudo certo, promover essa versão para Beta ou para produção direto. Mas nada o impede de lançar o App direto em produção também; o fato de ter a possibilidade de realização de testes Alfa e Beta é muito legal, mas não é obrigatório que você siga esse fluxo.



Figura 10.19: Versões do App

No próximo passo, veremos direto como é o lançamento em produção, mas o fluxo é praticamente o mesmo se fosse Alfa ou Beta. Clicando em **Gerenciar produção** você cairá numa tela de

Boas-vindas com um botão Criar versão . Clicando nele, você entrará na seguinte tela:

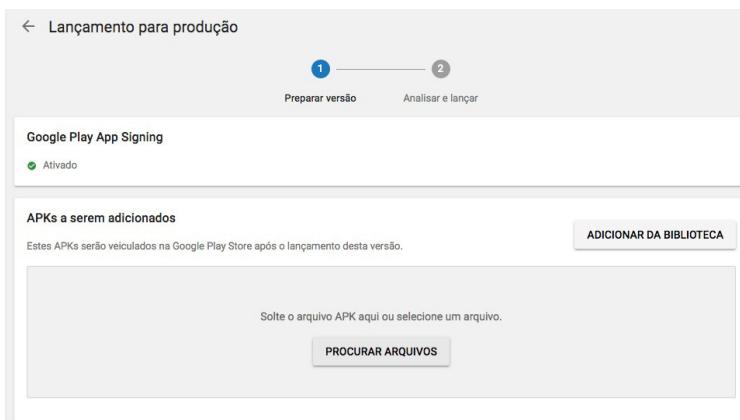


Figura 10.20: Gerenciar produção

É nessa tela que você vai enviar seu APK, basta procurar o arquivo e fazer o upload! Depois você deve ainda preencher a classificação de conteúdo e o preço e distribuição, esses links se encontrarão no menu lateral.

Depois de preencher todos os dados requisitados você poderá enviar seu aplicativo para análise e, se estiver tudo certo, seu aplicativo será publicado na loja!

Uma coisa legal do Android é que não tem muita burocracia para enviar um aplicativo, a análise não vê questões de layout para aprovar ou reprovar um App, também não analisa se seu código está bem escrito, seu código pode até quebrar que mesmo assim seu aplicativo será aprovado.

Aqui há uma grande diferença da Play Store com relação a Apple Store, que é muito mais rigorosa nessas questões, mas é

simplesmente uma outra visão. Aqui a filosofia é: se seu aplicativo tiver um visual ruim, ou estiver mal feito de alguma forma, quem vai ser o juiz é o usuário final. Ele que vai dizer se seu aplicativo é bom ou não e se ele tiver um layout ruim, tiver problemas no código, tiver problemas de navegação, provavelmente o usuário vai desinstalar o App e nunca mais instalar novamente - e dependendo da frustração que ele tiver ainda vai avaliá-lo negativamente. Então, apesar de a Play Store não nos impedir de enviar um aplicativo ruim, não é uma boa ideia enviar um App mal feito, pois os usuários são rigorosos.

Uma dica valiosa é sempre seguir os guidelines de interface com o usuário, lá você encontrará informações e dicas valiosas para criar interfaces que, além de visualmente bonitas, causarão uma boa experiência ao usuário com seu aplicativo. Você pode acessar os guidelines de interface do Android através do link:
https://developer.android.com/guide/practices/ui_guidelines/index.html

CAPÍTULO 11

CONCLUSÃO

Iniciamos nossa jornada falando um pouco dessa nova linguagem chamada Kotlin, vimos algumas diferenças e melhorias da linguagem principalmente em comparação com Java e vimos na prática porque o Kotlin ganhou tão rapidamente as graças da comunidade de desenvolvedores. Se você é novo no mundo de desenvolvimento Android, com certeza o Kotlin é a melhor opção para começar e espero que este livro o tenha ajudado a dar os primeiros passos em uma carreira de desenvolvimento Android. Particularmente, acho a plataforma Android incrível de se trabalhar: as coisas são muito bem resolvidas e o suporte a bibliotecas também é fantástico.

Para você que é iniciante no Android ou você que já é veterano e tem um bom background de Java, este livro é só o pontapé inicial nos seus estudos com a linguagem Kotlin. A abordagem que eu utilizei foi muito prática em forma de pequenos projetos que exploraram os recursos da linguagem assim como os recursos básicos do Android. Vou deixar aqui algumas dicas pra você continuar seus estudos.

A primeira dica é: estude Orientação a Objetos! E a melhor linguagem para se estudar Orientação a Objetos é Java. Então é uma boa ideia conhecer bem a linguagem ,mesmo que você não vá

utilizar o Java no desenvolvimento dos seus Apps, primeiro pela questão da Orientação a Objetos, que é fundamental para o seu desenvolvimento como programador; e segundo porque, trabalhando como programador Android, você vai se deparar com muitos códigos em Java, talvez tenha que dar manutenção em Apps legados etc. Além disso, saber Java vai lhe dar muito mais poder no Kotlin com o tempo, você vai ter uma visão muito mais ampliada sobre programação como um todo.

Aqui na Casa do Código você encontra vários livros sobre Java e Orientação a Objetos. Recomendo um chamado *Desbravando Java e Orientação a Objetos*, do Rodrigo Turini, um livro com uma abordagem simples e efetiva sobre o tema.

A outra dica para evoluir na carreira de desenvolvedor Android é conhecer as bibliotecas de produtividade disponíveis. A primeira que eu recomendo é a biblioteca Anko do Kotlin. Neste livro eu mostrei algumas coisas muito legais do Anko, no entanto ela possui diversos outros recursos interessantes que não foram abordados aqui. Você pode conferir a documentação oficial do Anko em <https://github.com/Kotlin/anko>.

A próxima biblioteca que você deve conhecer é a Picasso, essa biblioteca é para trabalhar com imagens. O grande problema que ela resolve é o carregamento de imagens a partir de uma URL. Sem o uso do Picasso, esse problema poderia ser resolvido da seguinte maneira: fazer um algoritmo para o download da imagem, salvá-la de alguma forma, carregar a imagem baixada no objeto `ImageView`.

Com o Picasso, basta passar a URL da imagem e o `ImageView` em que ela deve ser carregada e a biblioteca faz todo o trabalho.

Isso é resolvido com uma única linha de código, por exemplo:

```
Picasso.get().load("http://i.imgur.com/DvpvklR.png").into(imageView);
```

Além disso, o Picasso pode fazer algumas animações muito legais. Você pode conferir a documentação do Picasso em <http://square.github.io/picasso/>

Mais uma biblioteca importante de conhecer é o Retrofit. Essa biblioteca nos ajuda a consumir serviços REST e, apesar de o Kotlin por si só já possuir recursos interessantes que vimos neste livro, ele não resolve todos os problemas, e possivelmente você vai ver a necessidade de utilização de uma biblioteca com melhor suporte a requisições REST.

Aqui o Retrofit se encaixa muito bem, apesar de ser uma biblioteca em Java. Como sabemos, o Kotlin possui total interoperabilidade, ou seja, é totalmente possível utilizá-la no seu aplicativo em Kotlin. Você pode conferir a documentação do Retrofit em <http://square.github.io/retrofit/>.

Um bom lugar para se procurar é o site Android Arsenal, que reúne diversas bibliotecas de produtividade para Android: <https://android-arsenal.com/>

Uma última dica importante: estude UX! UX é "Experiência do Usuário" do inglês *User Experience*. Quando falamos de UX, não estamos falando apenas em deixar as coisas bonitas, mas sim de toda a experiência que o usuário terá com seu aplicativo, desde a instalação a partir da Play Store até a utilização no dia a dia. Pensar em UX é pensar no sentimento que o usuário terá utilizando seu aplicativo.

Nenhum App de sucesso hoje tem uma UX ruim, muito pelo contrário, eles só chegaram lá porque a experiência que o usuário tem com aquele App é tão boa que engaja os usuários a fazerem comentários positivos, a recomendarem aos amigos de forma espontânea, a criarem posts em redes sociais etc. Na minha opinião, todo desenvolvedor Mobile hoje deveria se preocupar com UX trabalhar em perfeita sintonia com o time de design e marketing.

Agora é botar em prática todo o conhecimento adquirido durante nossa jornada neste livro. Crie seus próprios projetos, publique seus projetos no GitHub, converse com outros desenvolvedores, se envolva em projetos open-source, participe de eventos e esteja sempre antenado com as novidades.

Um blog cujo conteúdo sempre vejo é o da Movile, primeiro porque é uma empresa muito relevante quando se fala de Mobile aqui no Brasil, e segundo porque eles já vêm usando Kotlin e suas soluções: <https://movile.blog/>

O blog da Caelum também tem um conteúdo bem interessante e diversos artigos sobre Kotlin: <http://blog.caelum.com.br/>

Deixarei também minhas redes sociais e algumas formas de contato:

No Linkedin: <https://www.linkedin.com/in/kassiano-resende-b2349956/>

No GitHub: <http://github.com/kassiano>

E-mail: kassiano.resende@gmail.com

Fique à vontade para entrar em contato através dessas

plataformas, mostrar seus projetos e trocar uma ideia.

Muito obrigado pela companhia, tenha muito sucesso na sua carreira e até a próxima =D