PROGRAMAÇÃO IV

AULA 1

Prof. Ricardo Rodrigues Lecheta

CONVERSA INICIAL

Olá! Seja bem-vindo à aula da disciplina Programação IV.

Nesta aula vamos estudar os conceitos fundamentais do desenvolvimento de aplicativos para Android e aprender a instalar o Android Studio (ferramenta de desenvolvimento). Vamos também criar nosso primeiro aplicativo para Android utilizando o emulador.

A aula está dividida nos seguintes tópicos:

1. História e conceitos fundamentais do Android
2. Criando um projeto no Android Studio
3. Entendendo o código-fonte que foi gerado
4. Trabalhando com textos e imagens
5. Entendendo o que é API Level

O mercado de aplicativos está bem aquecido e carente de bons profissionais. O Android e o iOS são os dois principais sistemas operacionais disponíveis nos atuais smartphones.

O objetivo deste curso é fornecer uma base sólida para que você aprenda os conceitos básicos do desenvolvimento de aplicativos para Android, a fim de que consiga iniciar sua carreira nesse mercado, que eu acredito que é muito promissor, e evoluir.

TEMA 1 – HISTÓRIA DO ANDROID

O Android é o sistema operacional móvel mais utilizado no mundo. Foi criado pelo Google e por gigantes do mundo da tecnologia, dentre eles Samsung, LG, Sony, Asus, Intel etc.

Não temos como entender o que significa o Android sem voltar um pouco no tempo. Antigamente, existiam os famosos BlackBerry (RIM) e Pocket PC (Microsoft), que revolucionaram o mundo *mobile* na época e que reinaram no início dos anos 2000. Ainda nessa época, tivemos os famosos celulares com Java J2ME (Java Platform, Micro Edition), que foram ganhando muita popularidade com o tempo. Diversos fabricantes utilizavam o J2ME como base para seus aplicativos (Nokia, Motorola, Sony Ericsson, LG, dentre outros).

A ideia do J2ME era usufruir do grande lema da linguagem Java (Write Once Run Everywhere – escreva uma vez, execute em qualquer lugar) e criar

uma plataforma única de desenvolvimento que executasse em diversos dispositivos, independentemente do fabricante. Os celulares Java eram embarcados com uma JVM (Java Virtual Machine) capaz de executar esses aplicativos.

A ideia foi fantástica e funcionou. Na época, já era possível criar um aplicativo que era instalado em um celular, independentemente do fabricante: podia ser Nokia, Motorola, LG etc. Isso era fantástico; rapidamente houve um rápido crescimento no mercado de aplicativos. Contudo, a fama e a adoção do J2ME foram aos poucos acabando diante do reinado exclusivo do BlackBerry e Pocket PC, deixando o mercado de dispositivos móveis cada vez mais aquecido.

Porém, o J2ME tinha um problema: era uma especificação, e não uma implementação. Para explicar, imagine que, na especificação (criada pela Sun Microsystems – criadora do Java e hoje comprada pela Oracle), apenas citava-se que, para criar a interface do aplicativo, era preciso existir um componente de Botão, outro para um Campo de Texto, Checkbox, Imagem etc., porém cada fabricante podia implementar esses componentes visuais como quisesse. Na prática, a interface de um botão e checkbox de um celular do fabricante X ficava diferente do celular que foi feito pelo fabricante Y, e isso era inaceitável por grandes empresas. Imagine que uma grande instituição financeira queira fazer um aplicativo. Naturalmente ela quer que a interface respeite todas as cores e todo o guia de interface da empresa. Esse foi o primeiro problema com o J2ME.

O segundo grande problema eram as diferenças de execução entre um celular e outro. Para exemplificar, digamos que seu aplicativo precisasse tocar um áudio. O que acontecia é que, no celular do fabricante X, funcionava e, no celular do fabricante Y, não.

E por que isso acontecia? Justamente porque a especificação da J2ME era aberta; alguns fabricantes a implementaram por completo, e outros acabavam fazendo apenas algumas partes. Também era comum aplicativos funcionarem melhor em alguns celulares do que em outros, porque tudo dependia de quão boa era a implementação do fabricante, assim como os recursos de hardware e de memória disponíveis naquele modelo de celular.

Pois muito bem. Chega de história! Vamos falar do Android?

Em meados de 2007, o Google se juntou a gigantes da tecnologia, com quem fez uma aliança, futuramente chamada de *OHA* (Open Handset Alliance). O objetivo da aliança era criar um sistema operacional móvel único para todos

os fabricantes. No site da OHA (http://www.openhandsetalliance.com) existe uma ótima descrição do que é essa aliança. O texto está em inglês e foi escrito por volta daquela época. Fique à vontade para ir ao site e ler o texto original. Eu vou apenas traduzir uma breve citação aqui.

> Hoje, existem 1,5 bilhão de aparelhos de televisão em uso em todo o mundo e 1 bilhão de pessoas têm acesso à internet. No entanto, quase 3 bilhões de pessoas têm um telefone celular, tornando o aparelho um dos produtos de consumo mais bem-sucedidos do mundo. Dessa forma, construir um aparelho celular superior melhoraria a vida de inúmeras pessoas em todo o mundo. A Open Handset Alliance é um grupo formado por empresas líderes em tecnologia móvel que compartilham essa visão para mudar a experiência móvel de todos os consumidores. (OHA, *on-line*, tradução nossa)

Para termos ideia da força desse acordo, na época existiam 88 empresas integrantes do grupo – para citar apenas algumas: Google, Samsung, Intel, HTC, LG, Motorola, Sony Ericsson, Toshiba, Huawei, Sprint Nextel, China Mobile, T-Mobile, ASUS, Acer, Dell, Garmin, dentre outras. Como podemos ver, foi um grande grupo criado por fabricantes de celulares, processadores e GPS, operadoras de celular, líderes em tecnologia etc., todos juntos, com um único propósito: criar o sistema operacional que conhecemos hoje como Android.

Para deixar bem claras as vantagens dessa união, vamos ilustrar um pequeno exemplo: como o código-fonte era aberto, todos podiam compartilhar e ajudar. Se a Samsung fizesse melhorias no software da câmera, isso ficaria disponível no código-fonte aberto do Android, e todos os fabricantes poderiam se beneficiar disso. Da mesma forma, se a Sony fizesse melhorias nos *wiadgets* que ficam na Home do Android, todos teriam acesso a isso. Podemos citar outros exemplos, mas acho que já foi suficiente para captar a ideia. Logo, com a força e a união de todos, liderados pelo Google, a primeira versão do Android foi lançada em 2008 e não parou de evoluir até os dias atuais.

Desde que foi criado, não demorou muito para o Android se tornar o sistema operacional móvel mais utilizado no mundo. Atualmente, está disponível para diversas plataformas, como smartphones, tablets, TV (Google TV), relógios (Android Wear), dispositivos inteligentes (Android Things) e carros (Android Auto).

Apenas para ter ideia de alguns números: no Google I/O 2017 foi anunciado que já existiam mais de 2 bilhões de dispositivos Android ativados no mundo; no Google I/O 2019, esse número passou a ser de 2,5 bilhões.

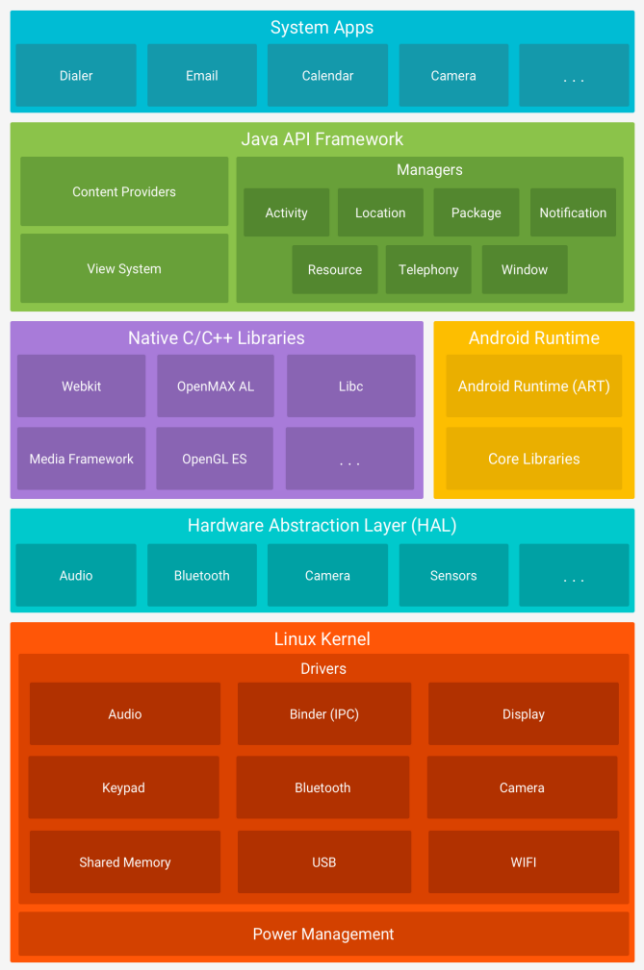
1.2 Conceitos fundamentais do Android

Antes de você começar a desenvolver, é bom se familiarizar com alguns conceitos importantes.

O sistema operacional do Android é um Linux. Isso significa que ele tem pastas e arquivos, exatamente como o sistema operacional para desktop, porém é otimizado para celulares.

Dentro do sistema operacional existe uma pilha de camadas de softwares, as quais podemos visualizar na figura abaixo, extraída da documentação oficial:

Figura 1 – Arquitetura do Android



Crédito: Developer.Android (on-line).

Devemos ler essa figura de baixo para cima. Em vermelho, na parte inferior da pilha, temos todo o sistema operacional do Linux, que fornece solidez e segurança necessárias para todo o sistema.

Em cima da camada do Linux, temos os componentes de hardware, como áudio, *bluetooth*, câmera e sensores, e que fornecem uma interface padrão de comunicação com as APIs do sistema.

No centro da imagem temos a camada que faz acesso às chamadas nativas com C/C++ para alta performance. Por exemplo, toda a camada que desenha a interface gráfica do sistema é feita em OpenGL. Ainda no centro da imagem temos demonstrado o Android Runtime, que é a máquina virtual capaz de executar os aplicativos Android, distribuídos por um arquivo com a extensão APK (Android Package). O Android Runtime existe desde o Android 5 (Lollipop). Antigamente, até o Android 4.4 (KitKat), a máquina virtual se chamava “Dalvik”.

Já perto do topo da figura temos a Java API Framework, que consiste em um conjunto de APIs que os desenvolvedores utilizam para acessar todos os recursos disponíveis no Android. Por exemplo, podemos utilizar a API da câmera e tirar uma foto com facilidade no código do aplicativo; internamente o sistema estará se comunicando com as camadas de níveis inferiores.

No topo da imagem temos os aplicativos de sistema e nossos próprios aplicativos. Não importa se o aplicativo é o discador do telefone, a agenda, o calendário ou o seu aplicativo, todos eles são construídos da mesma forma, com as linguagens Java ou Kotlin e utilizando o Android SDK.

A Figura 1 (arquitetura do Android) visa apenas que você entenda como é a base do sistema operacional, mas, na prática, os desenvolvedores não precisam se preocupar com isso, pois toda a complexidade fica abstraída.

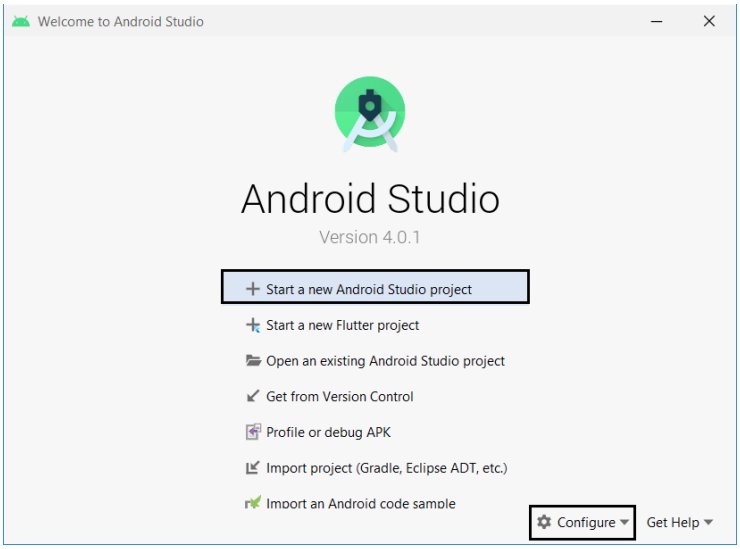
TEMA 2 – CRIANDO UM PROJETO NO ANDROID STUDIO

Chegou o grande momento! Vamos criar o primeiro projeto no Android Studio.

Para prosseguir, você precisa ter instalado o Android Studio, Android SDK, e já ter configurado algum emulador. Antes de prosseguir, verifique na **Aula Prática 1** como fazer toda a instalação do ambiente de desenvolvimento.

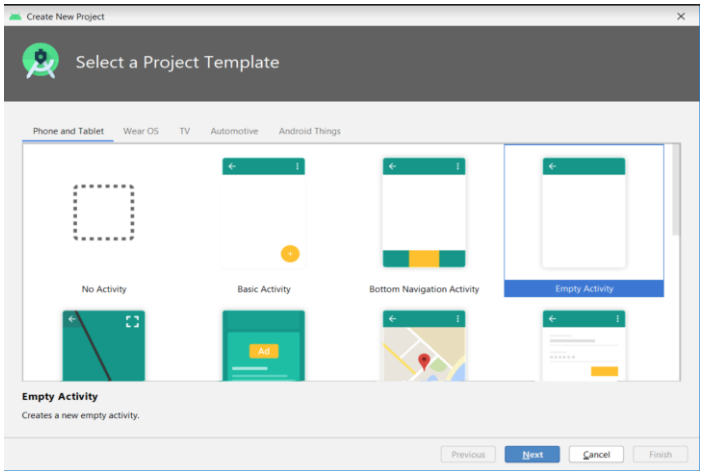
Para criar um novo projeto no Android Studio, abra a tela inicial e clique no botão **Start a new Android Studio Project**. Como demonstrado na Aula Prática 1, você pode clicar no botão **Configure**, se for necessário configurar o SDK ou criar algum emulador.

Figura 2 – Android Studio



O wizard de criação de projetos vai lhe perguntar qual template você deseja utilizar. Selecione a opção **Empty Activity**.

Figura 3 – Wizard do Android Studio

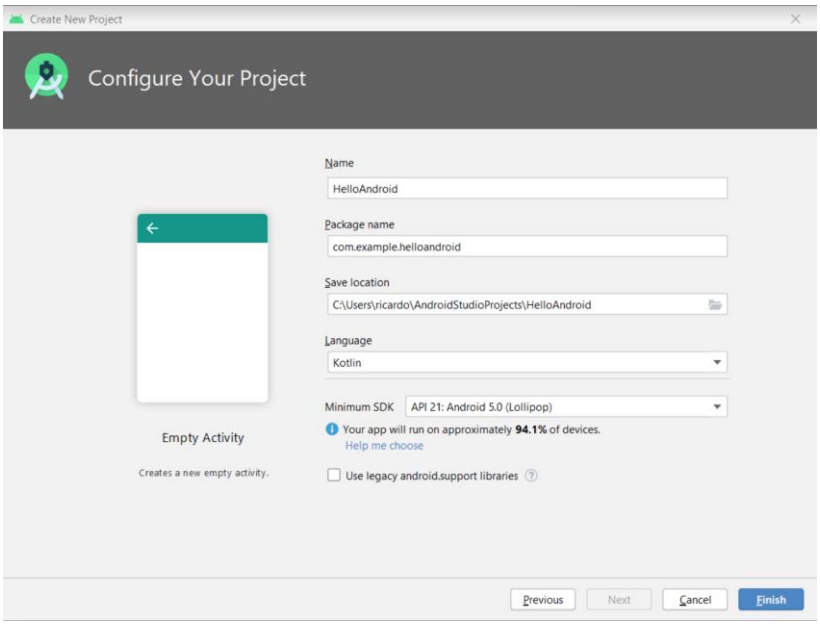


Obs.: o template "Empty Activity" indica que será criada uma tela vazia apenas com o template básico de uma tela. Uma *Activity* nada mais é do que a classe que vai conter o código e a lógica de uma tela. Logo vamos ver mais detalhes e tudo ficará mais claro. Com o tempo você, pode explorar os outros templates. Basta criar um projeto, selecionar o template e executar no emulador para ver o resultado e até conferir o código-fonte. Fica como ideia de exercício pra você!

Na próxima página, digite as informações do projeto, conforme explicado a seguir:

- **Name**: nome do projeto, digite *HelloAndroid*.
- **Package name**: nome do pacote que será utilizado como base dos *imports* para as classes Java ou Kotlin. No meu caso, gerei como **com**.**example**.**helloandroid**. O nome do pacote também é conhecido internamente como **application id** e representa o identificador do seu aplicativo no Google Play. Futuramente isso pode ser mudado, mas, por enquanto, aceite o padrão que o Android Studio gerar.
- **Save location**: local no qual o projeto será salvo no seu computador.
- **Language**: selecione **Kotlin**. Kotlin é a linguagem oficial do Android desde o anúncio feito no Google I/O 2017. Vamos utilizar essa linguagem no curso e vamos estudá-la em detalhes na próxima aula.
- **Minimum SDK**: selecione o API 21 (Android 5.0). Essa será a versão mínima suportada pelo seu aplicativo. Atualmente o Google recomenda criar aplicativos compatíveis com no mínimo essa versão. Se você trabalha para uma instituição financeira ou qualquer empresa que tenha altos padrões de segurança, deixe a API 23 (Android 6.0) como a mínima suportada, pois foram feitas significantes melhorias de segurança do sistema operacional nessa versão.

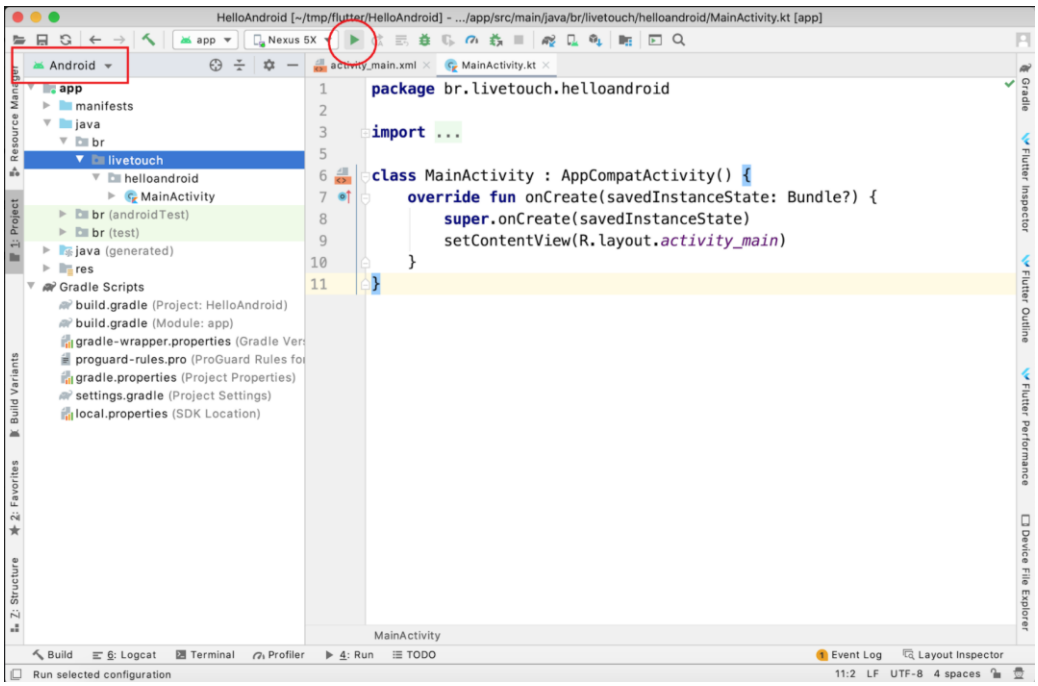
Figura 4 – Wizard do Android Studio



Com todas as informações devidamente preenchidas e revisadas, clique no botão **Finish** para criar o projeto.

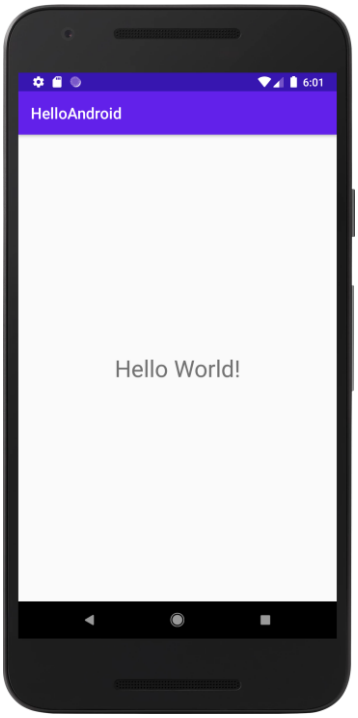
Depois de criar o projeto, é possível visualizar no Android Studio a estrutura de pastas e arquivos no lado esquerdo do projeto e o código-fonte no centro. Na barra de ferramentas você vai localizar o botão verde **Run**, conforme indicado na figura. Clique nesse botão para executar o projeto no emulador. Também está indicado na figura, no lado esquerdo, a palavra **Android**. Esse é o modo de visualização otimizado para Android dessa árvore de diretórios e arquivos; fica na esquerda. Se você clicar, há várias opções. Outra forma bem comum é selecionar a navegação **Project**, que mostra todos os arquivos, como se você estivesse no Windows Explorer. Geralmente deixamos isso com o modo **Android** de navegação selecionado.

Figura 5 – Projeto no Android Studio



Ao clicar no botão Run, o código-fonte do projeto foi compilado e um arquivo apk foi gerado e instalado no emulador automaticamente. O resultado, se tudo der certo, será a mensagem de “Hello World” no centro da tela do emulador.

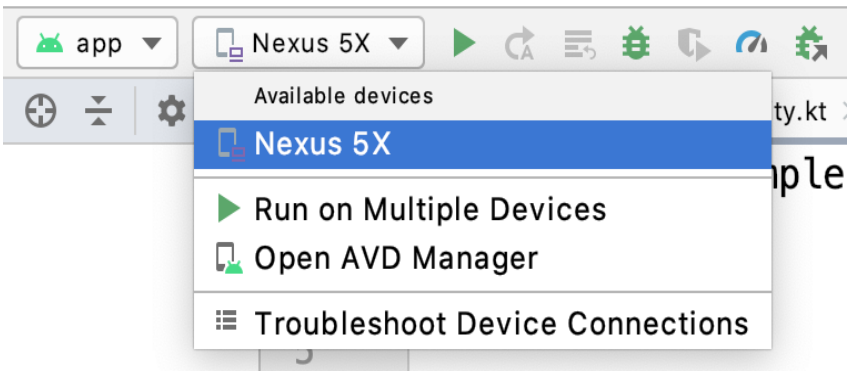
Figura 6 – Emulador do Android



Crédito: Ricardo Rodrigues Lecheta.

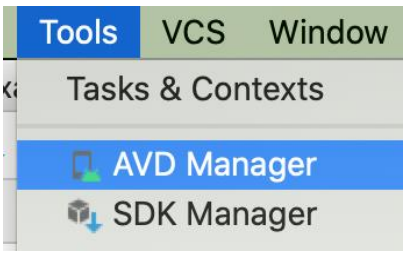
A partir do Android Studio 4.0, caso o emulador estiver fechado, você pode selecionar qual emulador deve abrir para executar, sempre nesse combo ao lado do botão **Run**. Note também que é possível abrir o **AVD Manager** para configurar a lista de emuladores nessa lista. Caso você ainda não tenha nenhum emulador, entre no **AVD Manager** e crie um.

Figura 7 – Executando o emulador



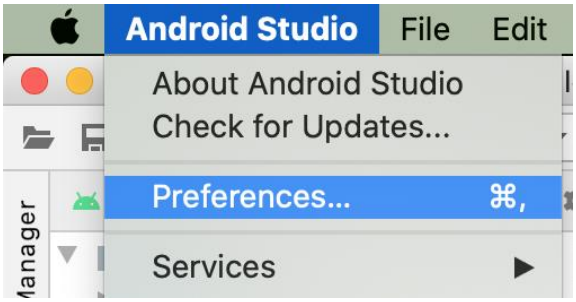
Outra opção para abrir o AVD Manager e SDK Manager quando o Android Studio já está aberto é pelo menu “Tools”, conforme a figura a seguir:

Figura 8 – Menu para acessar o AVD Manager e SDK Manager



Por fim, é muito importante aprender a abrir a tela de configurações do Android Studio, chamada popularmente de "Preferências". No Windows, essa tela pode ser acessada pelo menu **> Window > Preferences**. No Mac, ela fica em **> Android Studio > Preferences**.

Figura 9 – Menu para acessar as configurações do Android Studio



Note que a tela de preferências do Android Studio tem diversas configurações, inclusive o Android SDK, que já vimos que fica lá. Utilize o campo de busca no canto superior esquerdo para filtrar o que você deseja. Na aula prática, veremos mais detalhes sobre várias configurações.

*

Se você chegou até aqui e conseguiu rodar o emulador, dê uma pausa para comemorar, pois essa com certeza foi a etapa mais difícil! Meus parabéns! Caso não tenha conseguido, não desista, pois às vezes alguns problemas com a configuração do ambiente ou mesmo com o computador podem ter atrapalhado. Na nossa área de computação, é importante não desistir rápido; é preciso aprender a investigar e pesquisar sobre os erros no Google, até conseguir resolvê-los, independentemente de quais sejam. Quanto antes você se tornar autodidata e aprender a resolver os erros que aparecem, mais rápido vai acelerar sua carreira.

TEMA 3 – ENTENDENDO O CÓDIGO-FONTE QUE FOI GERADO

Uma vez que já criamos o projeto e o executamos com sucesso no emulador, vamos estudar um pouco do código-fonte que foi criado.

MainActivity.kt: a extensão .kt é da linguagem Kotlin, pois foi a linguagem que selecionamos no wizard de criação de projetos. Esse arquivo fica localizado na pasta **/src/main/java/** e embaixo do pacote que escolhemos para as classes **/com/example/helloandroid/**. A seguir temos a explicação de cada linha demarcada no código.

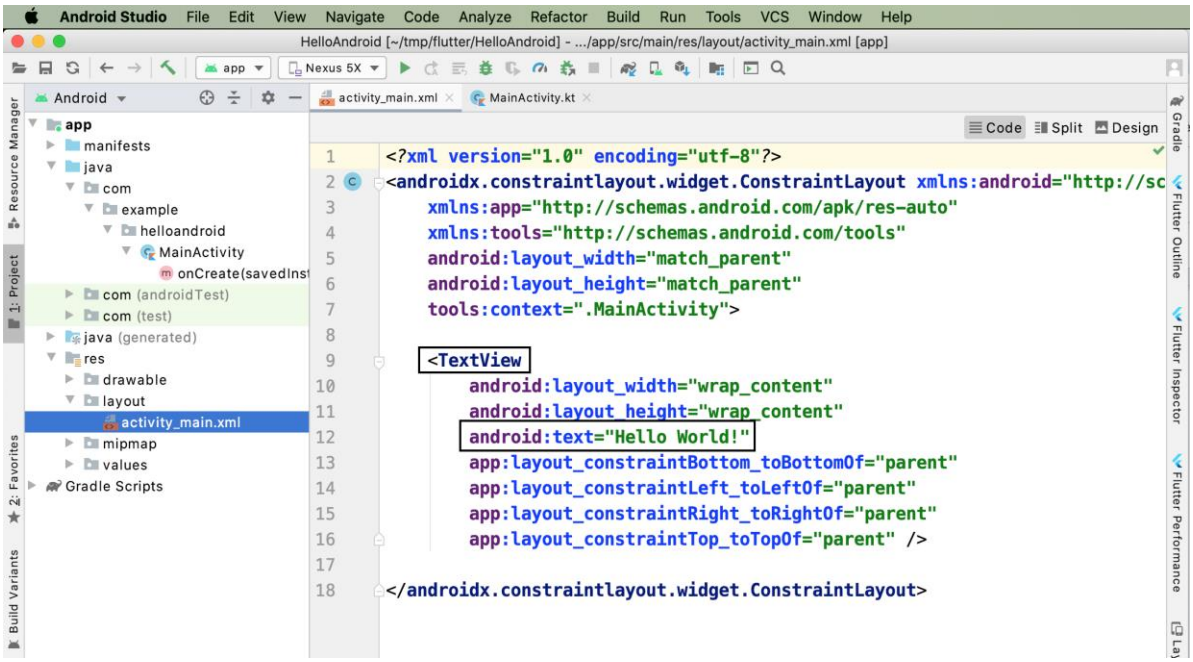
```
package com.example.helloandroid                    // 1

import androidx.appcompat.app.AppCompatActivity     // 2
import android.os.Bundle

class MainActivity : AppCompatActivity() {          // 3
    override fun onCreate(savedInstanceState: Bundle?) {
// 4
        super.onCreate(savedInstanceState)          // 5
        setContentView(R.layout.activity_main)      // 6
    }
}
```

1. Declaração do pacote: caso não esteja acostumado com Java e Kotlin, saiba que os pacotes se tornam pastas no computador, ajudando a separar e organizar os arquivos.
2. Imports: assim como no Java, temos que importar todas as classes que vamos utilizar. Nesse caso, estamos importando a classe **AppCompatActivity**, que será a classe-mãe da classe **MainActivity**, criada no projeto. A classe **Bundle**, também importada, é usada dentro do método **onCreate(bundle)**. É como se ela fosse uma HashTable que tem uma estrutura de dados de chave e valor, utilizada para passar parâmetros para a activity. Vamos estudá-la com mais detalhes depois.
3. Nessa linha é criada a classe **MainActivity**. Os dois-pontos configuram o sinal de herança do Kotlin. Se fosse Java, essa linha seria escrita assim: “class MainActivity extends AppCompatActivity”.

4. Nessa linha estamos declarando o método **onCreate(bundle)**. Veja que o método foi anotado como a palavra reservada **override**, pois estamos sobrescrevendo esse método da classe-mãe. Na verdade, somos obrigados a fazer isso. Saiba, desde já, que uma activity é uma tela do seu aplicativo, e o **onCreate(bundle)** é o método chamado para inicializar a tela com algum layout XML. Esse método será chamado apenas uma vez, mas vamos estudar esse e todos os métodos que fazem parte do ciclo de vida de uma *activity*.
5. A linha **setContentView(R.layout.activity_main)** faz a mágica de mostrar um layout na tela. O arquivo de layout pode ser encontrado na pasta **src/main/res/layout/activity_main.xml**, conforme figura a seguir.

Obs.: para construir uma tela no Android, sempre temos essa dupla formada pela classe da *Activity* (onde fica o código e a lógica) e o seu arquivo XML de layout (onde fica o design da tela).

Figura 10 – Layout em XML



No Android, o layout da tela é escrito em XML. Por padrão, o arquivo de layout começa com a palavra “activity” e termina com o nome da classe.

Seguem alguns exemplos para você entender a nomenclatura:

- LoginActivity > activity_login.xml
- HomeActivity > activity_home.xml

- CadastroActivity > activity_cadastro.xml

Veja que cada *activity* é formada pela dupla classe com código + layout XML.

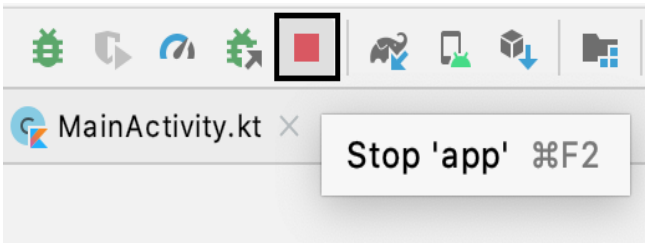
Vamos voltar ao arquivo de layout **/res/layout/activity_main.xml**.

Se você procurar, vai encontrar uma tag **<TextView>** nesse layout, a qual representa um texto na tela. Dentro dessa tag encontrará esta linha: **android:text="Hello World!"**.

É aqui que foi definido aquele texto “Hello World” que vimos ao executar o emulador. Experimente alterar esse texto e rodar o aplicativo novamente.

Dica: caso o aplicativo esteja executando, você pode clicar no botão vermelho **Stop** para pará-lo e, depois, clicar no botão verde **Run** novamente.

Figura 11– Botão Stop



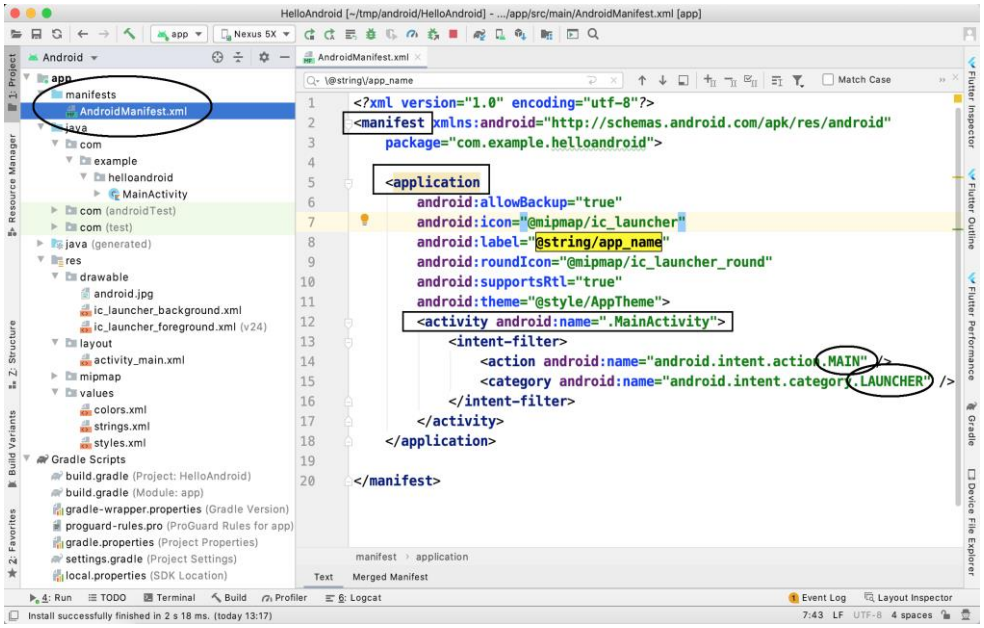
3.1 O arquivo androidmanifest.xml

Vamos estudar os principais arquivos que foram gerados no projeto. Um dos principais arquivos de configuração do projeto é o *AndroidManifest.xml*, também conhecido como “arquivo de manifesto”.

Nele é declarada a tag **<application>** com as configurações globais do aplicativo e também todas as *activities* com a tag **<activity>**. Como um aplicativo geralmente tem várias telas, na prática vamos criar duplas daqueles arquivos com a classe da Activity e o arquivo XML de layout. Sempre que criarmos essa classe da *activity*, precisamos vir aqui no arquivo de manifesto e configurá-la, para que o Android saiba que ela existe.

A figura a seguir mostra o arquivo de manifesto aberto.

Figura 12 – AndroidManifest.xml



Veja que dentro da tag ***<activity>*** existe a tag **<intent-filter>** com algumas configurações.

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

Vamos explicar cada uma delas agora:

- **<action android:name="android.intent.action.MAIN" />**: indica que a classe *MainActivity* será o ponto de entrada do aplicativo, ou seja, ela será executada quando o aplicativo for iniciado. Por convenção, essa classe se chama *MainActivity*, a fim de lembrar da famosa função ***main()*** presente em diversas linguagens de programação.
- **<category android:name="android.intent.category.LAUNCHER" />**: indica que o ícone dessa *activity* ficará visível na Home do Android para o usuário abrir o aplicativo.

O arquivo de manifesto também tem outras configurações importantes, como a declaração das permissões de sistema. O código a seguir mostra como configurar permissões para utilizar internet e GPS no aplicativo. Note que a tag **<uses-permission>** deve sempre ficar antes da tag **<application>**, em que ficam as *activities*. Mas fique tranquilo: vamos exercitar isso nas aulas práticas.

```
      <?xml version="1.0" encoding="utf-8"?>
      <manifest
xmlns:android="http://schemas.android.com/apk/res/android"
         package="com.example.helloandroid">

         <uses-permission
android:name="android.permission.INTERNET" />
         <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />

         <application ... />

      </manifest>
```

3.2 A pasta /Res/Mipmap

Essa pasta contém o ícone do aplicativo. Existem celulares com diversas resoluções de telas e, por isso, temos muitas variações dessa pasta (por exemplo, **mipmap-hdpi** e **mipmap-xhdpi**, cada uma com sua resolução). Existem geradores de ícones na internet que já criam os ícones no formato correto que o Android precisa: basta procurar no Google por “Android icon generator”. Costumo deixar isso para os designers, pois, se você trabalha com Android, poderá contar com um designer na empresa, e ele cuidará dessa parte.

3.3 A pasta /Res/Drawable: trabalhando com imagens

Nessa pasta podemos adicionar figuras para mostrar na tela do aplicativo. Logo vamos fazer um exercício e colocar o componente **ImageView** no layout XML para brincarmos com imagens.

3.4 Definindo ícone e nome do aplicativo

Ainda no arquivo *AndroidManifest.xml*, dentro da tag **<application>**, existem duas linhas interessantes:

```
android:icon="@mipmap/ic_launcher"
```

Essa tag configura o ícone do aplicativo. A notação **@mipmap** acessa a imagem **ic_launcher.png** que está dentro da pasta **/res/mipmap**. Veja que não é necessário colocar a extensão da imagem ao declará-la no XML. A figura a seguir mostra o ícone do aplicativo na tela Home do Android. Se quiser alterar esse ícone, basta substituir o arquivo **ic_launcher.png**.

Figura 13 – Ícone do aplicativo no Android

Crédito: Ricardo Rodrigues Lecheta.

```
android:label="@string/app_name"
```

Essa tag configura o nome do aplicativo que fica com o ícone na tela Home do Android. Observe que foi utilizada a notação **@string**, que acessa um texto cadastrado no arquivo */res/values/strings.xml*. Se você alterar o texto dessa chave **app_name**, vai alterar o nome do aplicativo.

3.5 O arquivo /res/values/strings.xml

O arquivo */res/values/strings.xml* tem vários textos separados por chave e valor. Veja que a chave **app_name** tem o valor **HelloAndroid**. Como vimos, essa chave é referenciada no arquivo *AndroidManifest.xml* para ser o nome do aplicativo.

```
<resources>
   <string name="app_name">HelloAndroid</string>
</resources>
```

A grande ideia desse arquivo é deixar todos os textos do aplicativo separados aqui, para que seja possível internacionalizá-los para vários idiomas, se necessário.

Para criar um aplicativo com suporte a vários idiomas, basta criar uma pasta */res/values/values-(código do idioma)* e traduzir o arquivo */res/values/strings.xml*. A seguir, temos um exemplo de uma pasta para mensagens em inglês e outra em português.

- /res/values-en-rUS/strings.xml
- /res/values-pt-rBR/strings.xml

Isso costuma dar certo trabalho, e acho que temos muito a aprender antes de brincar com essa ferramenta. Mas é bom você saber desde já que esta é uma boa prática: deixar todos os textos no arquivo /res/values/strings.xml.

3.6 O arquivo /res/values/colors.xml

Esse arquivo define as cores como constantes na notação hexadecimal, a fim de serem utilizadas no aplicativo. Essa é uma boa prática de configuração, pois podemos alterar as cores do aplicativo de forma global.

/res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="colorPrimary">#6200EE</color>
   <color name="colorPrimaryDark">#3700B3</color>
```

```
        <color name="colorAccent">#03DAC5</color>
    </resources>
```

- O atributo **colorPrimary** define a cor primária do tema do aplicativo. Essa é a cor que aparece na App Bar (barra de navegação).
- O atributo **colorPrimaryDark** define a cor da Status Bar que fica logo acima da App Bar.
- O atributo **colorAccent** define a cor de acentuação, que dá destaque em alguns componentes, como *radiobutton*, *checkbox* etc.

3.7 O arquivo /res/values/styles.xml

Esse arquivo define o tema do aplicativo, que é chamado de **AppTheme**. Ele herda do tema **AppCompat** do Android, que implementa o Material Design.

O tema é configurado com as cores que estão lá no arquivo **colors.xml**. Recomendo não mexer nesse arquivo enquanto estiver no início dos estudos. Deixe para brincar com temas quando já estiver voando baixo com o Android.

```
    <resources>
     <style name="AppTheme"
parent="Theme.AppCompat.Light.DarkActionBar">

      <item name="colorPrimary">@color/colorPrimary</item>
      <item
name="colorPrimaryDark">@color/colorPrimaryDark</item>
      <item name="colorAccent">@color/colorAccent</item>
     </style>

    </resources>
```

3.8 O arquivo build.gradle

Atenção: existem dois arquivos *build.gradle*. Um fica na pasta raiz do projeto e o outro fica na pasta app. O arquivo ***build.gradle*** "raiz" contém configurações globais de compilação do projeto. Ele declara alguns *plugins* e repositórios dos quais as bibliotecas serão baixadas. Raramente você mexerá nesse arquivo, mas, quando o fizer, tenha bastante atenção.

O arquivo ***app/build.gradle*** é o local das configurações de compilação do seu aplicativo, que fica localizado na pasta **app**.

Embora quase todos os projetos Android tenham sempre a pasta app, é importante entender que essa pasta é um módulo no qual ficam os códigos para *smartphone* e que, por convenção, chama-se “app”. Por exemplo, podemos ter módulos para relógios e carros, e o código-fonte desses módulos específicos para esses dispositivos ficaria em outra pasta. Cada um desses módulos teria o seu arquivo *build.gradle* com as configurações necessárias.

*

Agora que já entendemos o que significa o arquivo ***app/build.gradle***, vamos dar uma analisada no seu código-fonte:

```
apply plugin: 'com.android.application' // 1
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
   compileSdkVersion 29          // 2
   buildToolsVersion "30.0.1"    // 3

   defaultConfig {
       applicationId "com.example.helloandroid" // 4
       minSdkVersion 21          // 5
       targetSdkVersion 29       // 6
       versionCode 1             // 7
       versionName "1.0"         // 8
   }

   buildTypes { // 9
       release { . . . }
   }
}

dependencies { // 10
   . . .
   implementation 'androidx.appcompat:appcompat:1.2.0'
}
```

A seguir, temos a explicação de cada linha demarcada no código. Muitas delas são configurações-padrão que você dificilmente vai ter que mexer, mas algumas delas vamos alterar a todo momento. Portanto, é bom conhecê-las.

1. O **gradle** é um sistema de *build* genérico utilizado em diversos sistemas, não apenas no Android. No início do arquivo temos as configurações dos *plugins* para habilitar os módulos de compilação do Android e Kotlin no gradle.
2. O **compileSdkVersion** representa a API Level para a qual o projeto está sendo compilado. É sempre indicado compilar com as versões mais novas do Android. No caso, API Level 29 corresponde ao Android 10.
3. O **buildToolsVersion** corresponde à versão do SDK Build Tools instalado no Android SDK. Basicamente, é essa a ferramenta de compilação utilizada pelo projeto. Essa linha não é obrigatória; se você a remover, será utilizada a versão do *build tools* padrão do *plugin* do gradle que está instalado no projeto.
4. O **applicationId** é uma das configurações mais importantes e representa o id do seu aplicativo no Google Play, portanto ele deve ser único.
5. O **minSdkVersion** representa a API Level mínima suportada pelo seu aplicativo. Nesse caso, a API Level 21 corresponde ao Android 5.0.
6. O **targetSdkVersion** sempre precisa estar igual ao **compileSdkVersion** para garantir que seu aplicativo seja compilado com uma versão recente do Android.
7. O **versionCode** é um número inteiro que representa o número da versão de um *build* do aplicativo. Esse número precisa ser obrigatoriamente incrementado sempre que publicar o aplicativo no Google Play. Na última aula prática, veremos como publicar o aplicativo na loja.
8. O **versionName** também é utilizado pelo Google Play, mas o objetivo dele é mostrar um número de versão amigável para o usuário – por exemplo, 1.0.0 ou 1.0.1. Podemos aumentar a versão do aplicativo sempre que necessário.
9. Na seção **buildTypes** são configuradas as opções de *build*, como **debug** e **release**. O *build* de *release* é sempre utilizado para enviar o aplicativo para a loja. Também vermos detalhes dessas configurações na última aula prática sobre como publicar o aplicativo no Google Play.

10. Na seção **dependencies** são adicionadas as configurações das bibliotecas utilizadas no aplicativo. Essas bibliotecas nos ajudam no desenvolvimento, facilitando a programação. É comum os desenvolvedores chamarem essas bibliotecas de “dependências” – acostume-se com o termo.

TEMA 4 – TRABALHANDO COM TEXTOS E IMAGENS

Neste tópico vamos brincar um pouco com o arquivo de layout padrão que foi gerado pelo wizard do projeto.

Para começar, vamos alterar a mensagem de “Hello World” que apareceu na tela ao executar o emulador.

Abra novamente o arquivo de layout */res/layout/activity_main.xml* e substitua o texto a seguir:

```
android:text="Hello World!"
```

por:

```
android:text="@string/hello"
```

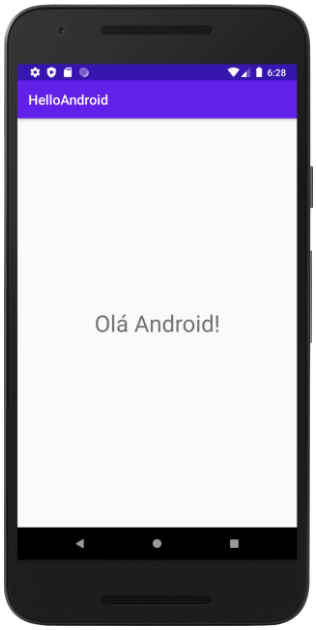
O código ficará vermelho, pois a chave ***hello*** ainda não existe no arquivo *strings.xml*. Vamos adicioná-la.

/res/values/strings.xml

```
<resources>
    <string name="app_name">HelloAndroid</string>
    <string name="hello">Olá, Android!</string>
</resources>
```

Pronto! Agora, ao executar o aplicativo novamente, você verá o texto *Olá, Android* no emulador. Com isso, o código ficou organizado e pronto para ter o arquivo *strings.xml* traduzido para vários idiomas, caso seja necessário.

Figura 14 – Aplicativo executando no emulador



Crédito: Ricardo Rodrigues Lecheta.

4.1 Alterando a cor do texto

Para alterar a cor do texto, podemos utilizar o atributo textColor no TextView e informar uma cor em hexadecimal. Embora possamos fazer isso diretamente no arquivo XML de layout, o recomendado, por questões de organização de código, é primeiro declarar a cor no arquivo *colors.xml* e depois referenciá-la no arquivo de layout. Para exemplificar, vamos alterar o texto Olá, Android! para a cor azul.

Para começar, temos que adicionar a cor azul no final do arquivo *colors.xml*:

/res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#03DAC5</color>
    <color name="azul">#0000ff</color>
</resources>
```

Por fim, vamos utilizar o atributo **textColor** referenciando a cor desejada, que nesse caso é a **@color/azul**.

res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<. . . >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        android:textColor="@color/azul"
        . . . />
</. . .>
```

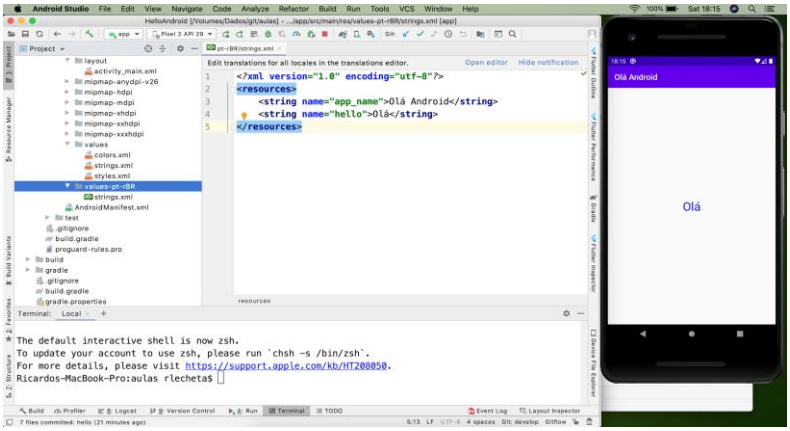
Pronto! Execute o emulador e você verá o texto Olá, Android! em azul.

4.2 Customizando o texto para português

Na maioria dos projetos, você colocará textos em português no arquivo *strings.xml*, mas, se for necessário criar um aplicativo com suporte a vários idiomas, o recomendado é deixar no arquivo *strings.xml* o texto todo em inglês como padrão e criar outros arquivos para os demais idiomas.

Para fazer um teste, crie a pasta **values-pt-rBR** e traduza todos os textos do arquivo *strings.xml* para português. Na aula prática será demonstrada uma maneira de criar esse arquivo utilizando o editor de idiomas do Android Studio, portanto, caso não consiga fazer, fique tranquilo.

Figura 15 – Exemplo de internacionalização



Para testar o suporte a vários idiomas, basta entrar nas configurações do sistema no emulador e configurar o idioma como Português – Brasil. A configuração é a mesma de um celular real, pois o emulador contém o mesmo sistema operacional do Android.

No emulador, geralmente as configurações de idioma ficam neste caminho:

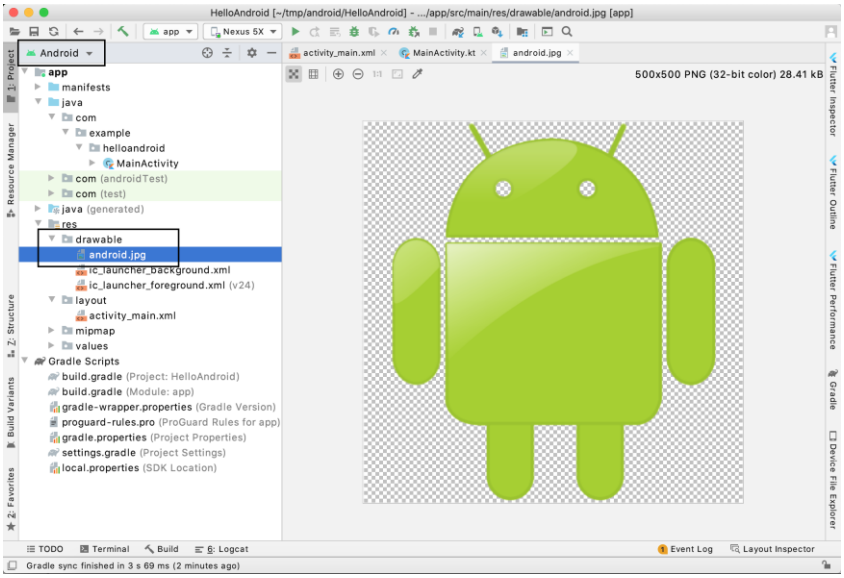
- Settings > System > Languages & input > Languages
- Configurações > Sistema > Idiomas e entrada > Idiomas

4.3 Trabalhando com figuras

Vamos brincar mais um pouquinho com o projeto para você ir se familiarizando com os conceitos. Procure alguma imagem no Google e faça o *download* dela para o seu computador (prefira imagens não muito grandes, de no máximo 1000x1000 px).

As imagens que vamos utilizar no projeto devem ser adicionadas à pasta **drawable**. Escolhi uma imagem do Android (android.jpg), como podemos ver:

Figura 16 – Trabalhando com figuras



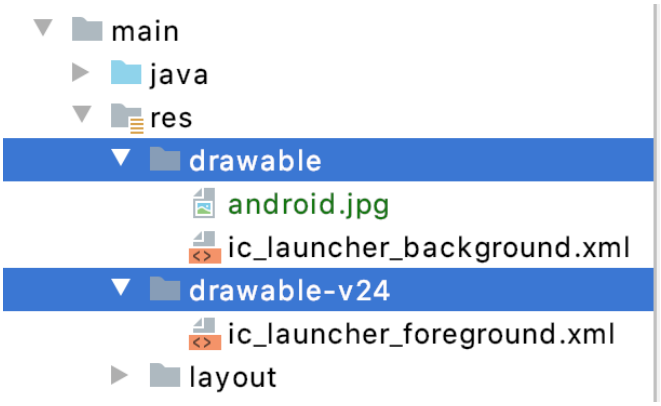
Talvez você tenha trabalho para copiar o arquivo para o projeto na primeira vez, mas logo se acostuma. Uma das maneiras é abrir o Windows Explorer na pasta na qual está o seu projeto e navegar até a pasta */app/src/main/res/drawable*. Ao colocar o arquivo no Windows Explorer e voltar ao Android Studio, o arquivo vai aparecer automaticamente. Se não aparecer,

clique com o botão direito na pasta e selecione a opção *Reload From Disk* para atualizar os arquivos. Outra opção para copiar o arquivo para o projeto é fazer o *download* no Windows Explorer e fazer um Ctrl+C (copiar) do arquivo; depois, selecione a pasta Drawable dentro do Android Studio e faça Ctrl+V (colar).

Algo extremamente importante que você deve aprender sobre o Android Studio é a diferença entre as navegações **Android** e **Project**, que podem ser selecionadas no topo à esquerda da árvore na qual ficam as pastas e os arquivos do projeto. Eu já tinha até comentado sobre isso, mas agora podemos ver um exemplo prático. Observe na figura que, no topo à esquerda, está marcado o modo **Project**, portanto conseguimos ver a estrutura real de diretórios, como se fosse o Windows Explorer. Note que pode existir mais de uma pasta **drawable** e mais de uma pasta **mipmap** (pasta de ícones/logo do aplicativo).

Adicione a figura que você escolheu à pasta **drawable** padrão, sem nenhum desses sufixos. A pasta **drawable-v24** que foi criada é uma configuração específica que funciona a partir do Android 7.0 (API Level 24). Está vendo agora como é importante conhecer os números de API Level? Nós, desenvolvedores, sempre trabalhamos com esses números nas configurações do projeto. Mas, por enquanto, ignore essas pastas, pois são mais avançadas. Sempre utilize a pasta **drawable** padrão.

Figura 17 – Pasta drawable com as figuras



Pronto! Tendo copiado a figura para o projeto, vamos alterar o código-fonte para mostrar essa figura no lugar do texto “Hello World”.

Abra o arquivo */res/layout/activity_main.xml*, e faça as seguintes alterações:

Troque a tag **TextView** por **ImageView** e altere o atributo:

```
android:text="@string/hello"
```

por:

```
android:src="@drawable/android"
```

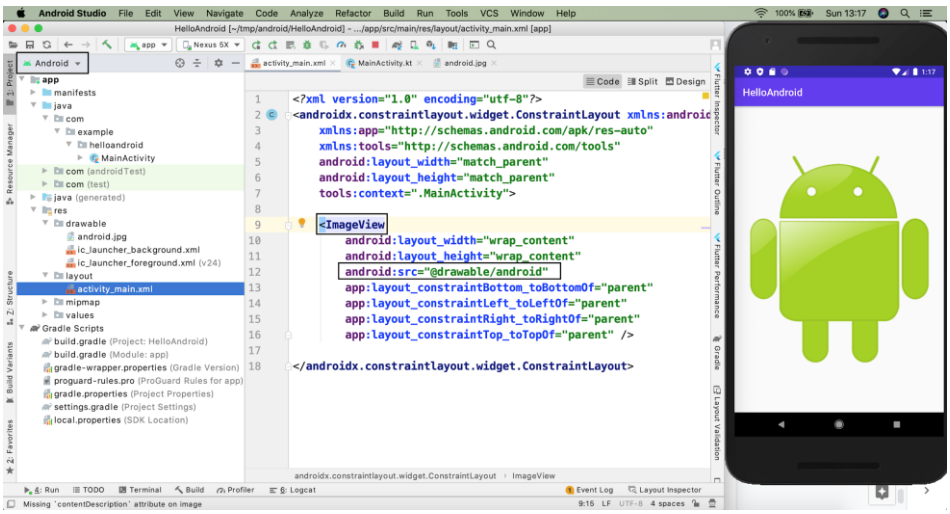
A seguir, podemos ver o código como ele deve ficar:

res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<. . . >
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/android"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</. . .>
```

Feito isso, execute o projeto novamente e configure o resultado no emulador, que deve ser como a próxima figura.

Figura 18 – Referenciando figuras no layout XML



TEMA 5 – ENTENDENDO O QUE É API LEVEL

Para finalizar nossa introdução ao Android, vamos entender o que é API Level e aproveitar para revisar alguns conceitos importantes.

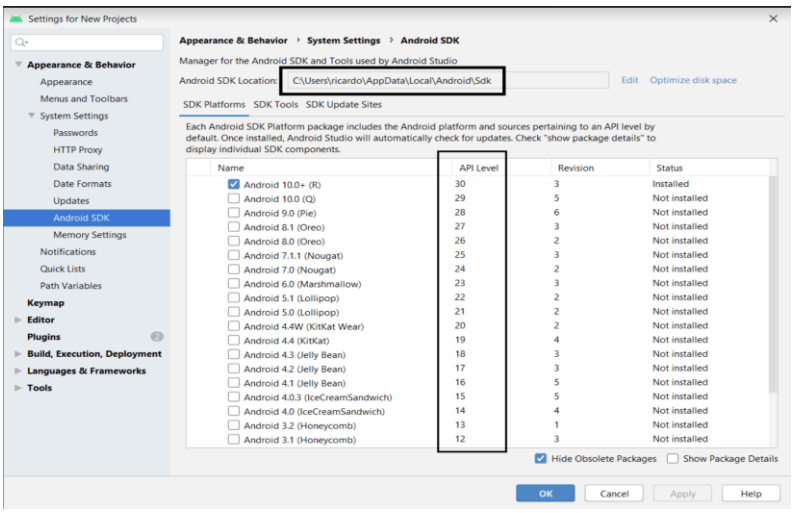
Lembra-se de quando criamos um projeto e você selecionou API 21 (Android 5) como a versão mínima?

Lembra-se de como, no arquivo build.gradle, essa versão 21 mínima foi referenciada na propriedade **minSdkVersion**? Da mesma forma, a propriedade **targetSdkVersion** ficou com a API 29 (Android 10.0). Esses números são identificadores de uma versão do SDK do Android, e nós os chamamos de “API Level”. Essa numeração começou com o número 1, lá no Android 1.0, e a cada nova versão do Android ela é incrementada.

Nós, desenvolvedores, sempre trabalhamos com o número da API, portanto é comum falarmos que os aplicativos são compatíveis com API 16, 21, 26 etc. Na prática, sabemos que cada número de API corresponde a uma versão do Android.

Uma boa maneira de ver essa lista sempre que precisar conferir qual versão corresponde a cada número é abrir o SDK Manager no Android Studio: entre no menu de preferências do Android Studio e filtre por **Android SDK**. Você verá a lista de API Levels, conforme a próxima figura. Na lista do centro, podemos ver as versões do Android que estão instaladas (no caso da figura, é o Android 10). Isso significa que podemos criar aplicativos e testá-los no emulador do Android 10 (API 30). Apenas para exemplificar: caso você queira testar como o seu aplicativo se comporta no Android 5, basta baixar o Android 5.0 (API 21).

Figura 19 – API LevFigura 20: SDK Tools



Dica: na tela do Android SDK, logo na parte superior, você poderá ver o local no qual o SDK está instalado. Isso pode ser útil caso você não saiba em que local ele está instalado.

Entender o que é API Level é fundamental para o desenvolvimento Android.

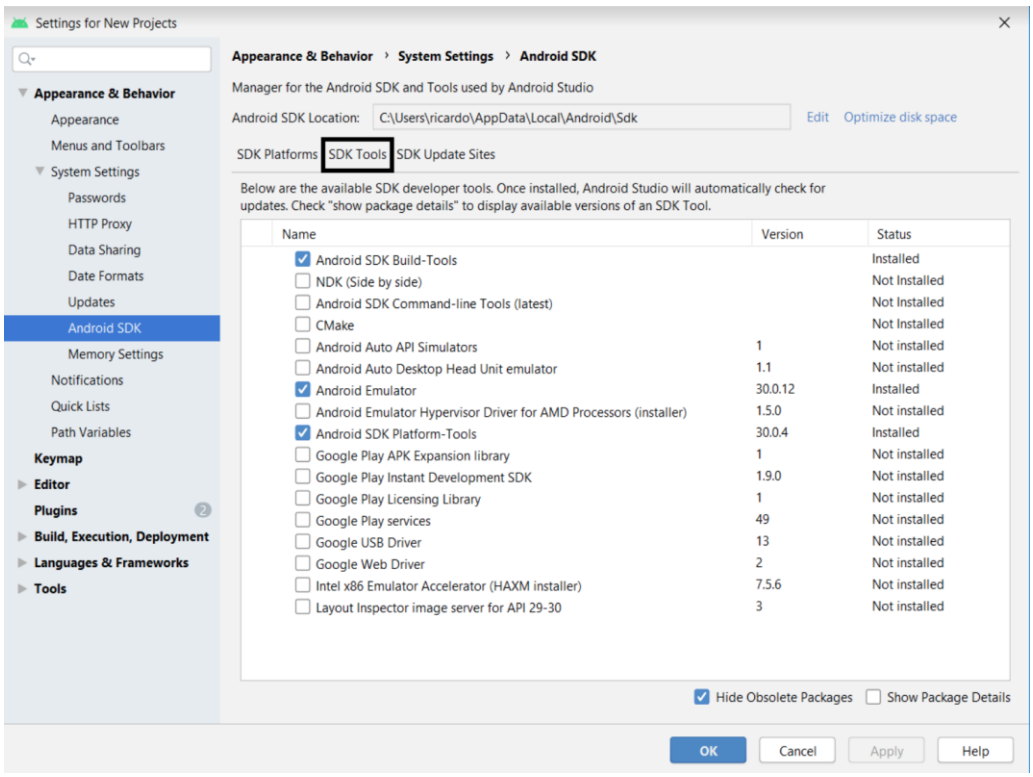
A partir do Android 6.0, por exemplo, o desenvolvedor precisa solicitar ao usuário que aceite as permissões antes de chamar alguma API segura. Para utilizar o GPS, precisamos que o usuário aceite as permissões – ou seja, no código, devemos validar se o celular que está executando o aplicativo tem API Level maior ou igual a 26 e mostrar aquele alerta de aceitar permissões para o usuário. Caso o celular tenha alguma API Level mais antiga, como a API 21 (Android 5.0), isso não é necessário.

Outro exemplo: algumas bibliotecas podem ter compatibilidade apenas com versões mais novas da API. É muito importante que, sempre que você utilizar uma biblioteca ou algum código do próprio Android SDK, seja validado se esse código é compatível com a versão do Android que está instalada no celular do usuário. Isso ficará mais claro no seu dia a dia como desenvolvedor. Mesmo que agora o conceito seja confuso, tente apenas se lembrar de que cada API Level corresponde ao número de determinada versão do Android. Cada versão do sistema operacional tem novos recursos e diversas tecnologias envolvidas.

Para finalizar, vamos falar da **SDK Tools**, outra aba importante no Android SDK. É nela que baixamos as ferramentas de compilação. É importante que você sempre mantenha atualizados os três itens marcados na próxima figura.

No início, parece ser uma enxurrada de conceitos que podem aparentar ser complicados, mas, com o dia a dia e a prática, logo você dominará todos esses itens com facilidade.

Figura 20 – Itens para manter atualizados



FINALIZANDO

Nesta aula, aprendemos um pouco sobre a história do Android e criamos nosso primeiro projeto.

A classe **MainActivity** é a porta de entrada para o aplicativo, e foi nela que internamente o Android chamou o método **onCreate(bundle)**, para que o aplicativo pudesse escolher qual arquivo de layout deve ser mostrado na tela. Aprendemos que os arquivos de layout ficam na pasta **/res/layout** e que as figuras podem ser adicionadas em **/res/drawable**.

Ainda temos muito o que estudar e aprender. Esta foi a primeira etapa. Mais detalhes serão passados na aula prática.

Até mais!

REFERÊNCIAS

DEVELOPER. ANDROID. Disponível em: https://developer.android.com/. Acesso em: 30 dez. 2020.

OHA. Open Handset Alliance. Disponível em: http://www.openhandsetalliance.com. Acesso em: 30 dez. 2020.