GABRIEL DOS SANTOS LIMA | IAGO BARRETTO SOARES
RONE CLAY OLIVEIRA ANDRADE | JOSÉ APRÍGIO CARNEIRO NETO

APP INVENTOR

Lógica de Programação e Design

FORMA EDUCACIONAL

GABRIEL DOS SANTOS LIMA | IAGO BARRETTO SOARES
RONE CLAY OLIVEIRA ANDRADE | JOSÉ APRÍGIO CARNEIRO NETO

APP INVENTOR

Lógica de Programação e Design

FORMA
EDUCACIONAL

© 2023 – Forma Educacional Editora

www.formaeducacional.com.br
formaeducacional@gmail.com

Organizadores

Gabriel dos Santos Lima

Iago Barretto Soares

Rone Clay Oliveira Andrade

José Aprígio Carneiro Neto

Editor Chefe: Jader Luís da Silveira

Editoração e Arte: Resiane Paula da Silveira

Capa: Autores/Organizadores/Forma Educacional

Revisão: Respectivos autores do texto

Conselho Editorial

Ma. Heloisa Alves Braga, Secretaria de Estado de Educação de Minas Gerais, SEE-MG

Me. Ricardo Ferreira de Sousa, Universidade Federal do Tocantins, UFT

Me. Guilherme de Andrade Ruela, Universidade Federal de Juiz de Fora, UFJF

Esp. Ricael Spirandeli Rocha, Instituto Federal Minas Gerais, IFMG

Ma. Luana Ferreira dos Santos, Universidade Estadual de Santa Cruz, UESC

Ma. Ana Paula Cota Moreira, Fundação Comunitária Educacional e Cultural de João Monlevade, FUNCEC

Me. Camilla Mariane Menezes Souza, Universidade Federal do Paraná, UFPR

Ma. Jocilene dos Santos Pereira, Universidade Estadual de Santa Cruz, UESC

Ma. Tatiany Michelle Gonçalves da Silva, Secretaria de Estado do Distrito Federal, SEE-DF

Dra. Haiany Aparecida Ferreira, Universidade Federal de Lavras, UFLA

Me. Arthur Lima de Oliveira, Fundação Centro de Ciências e Educação Superior à Distância do Estado do RJ, CECIERJ

Dados Internacionais de Catalogação na Publicação (CIP)

APP INVENTOR: Lógica de Programação e Design

L732a / Gabriel dos Santos Lima, Iago Barretto Soares, Rone Clay Oliveira Andrade, et al (organizadores). – Formiga (MG): Forma Educacional Editora, 2023. 63 p. : il.

Outro organizador:
José Aprígio Carneiro Neto

Formato: PDF
Requisitos de sistema: Adobe Acrobat Reader
Modo de acesso: World Wide Web
Inclui bibliografia
ISBN 978-65-85175-08-1
DOI: 10.5281/zenodo.8023421

1. Lógica. 2. Programação. 3. Design. 4. Sistemas. I. Lima, Gabriel dos Santos. II. Soares, Iago Barretto. III. Andrade, Rone Clay Oliveira. IV. Título.

CDD: 005
CDU: 681.3

Os artigos, seus conteúdos, textos e contextos que participam da presente obra apresentam responsabilidade de seus autores.

Downloads podem ser feitos com créditos aos autores. São proibidas as modificações e os fins comerciais.
Proibido plágio e todas as formas de cópias.

Forma Educacional Editora
CNPJ: 35.335.163/0001-00
Telefone: +55 (37) 99855-6001
www.formaeducacional.com.br
formaeducacional@gmail.com
Formiga - MG
Catálogo Geral: https://editoras.grupomultiatual.com.br/

Acesse a obra originalmente publicada em:
https://www.formaeducacional.com.br/2023/06/app-inventor.html

APP INVENTOR

Lógica de Programação e Design

AUTORES:
GABRIEL DOS SANTOS LIMA, IAGO BARRETTO SOARES,
RONE CLAY OLIVEIRA ANDRADE, JOSÉ APRÍGIO CARNEIRO NETO

2023

Lógica de Programação e Design **7**
Introdução **8**
Lógica de programação, eu preciso mesmo ? 8
Mas serve só para TI ? 9
App Inventor **10**
Como funciona ? 10
Como usar ? 14
Lógica de Programação e Práticas **21**
Programação: O essencial para iniciar no *App Inventor* 21
Condições 21
Repetições 22
Eventos 24
Vamos praticar 24
Procedimentos 29
Vamos praticar 29
Tipos de Dados 32
Vamos praticar 33
Estruturas de Dados 35
Vamos praticar 37
Bibliotecas 40
Vantagens 40
Desvantagens 41
Vamos praticar 41
Mais exemplos 44
Projetos 48
Vamos praticar 48
Para Finalizar **62**
Recapitulando 62
Próximos passos 62
Palavras finais 62

LÓGICA DE PROGRAMAÇÃO E DESIGN

Repositório das aplicações

Decisão: github.com/TheGB0077/Intro-AppInventor-2022/blob/main/Aula-1/

Fibonacci: github.com/TheGB0077/Intro-AppInventor-2022/blob/main/Aula-1/

Eval App: github.com/TheGB0077/Intro-AppInventor-2022/tree/main/Aula-2/

TODO App: github.com/TheGB0077/Intro-AppInventor-2022/tree/main/Aula-3/

FotoPaint: github.com/TheGB0077/Intro-AppInventor-2022/tree/main/Aula-4/

2048: github.com/TheGB0077/Intro-AppInventor-2022/tree/main/Aula-4/

Palavreado: github.com/TheGB0077/Intro-AppInventor-2022/tree/main/Aula-5/

A fim de importar os projetos para a sua área de trabalho do App Inventor, basta utilizar os arquivos ***.aia*** *incluído nas pastas do repositório. Se não sabe como ainda, leia adiante!*

INTRODUÇÃO

Ei, você, querido leitor, este livro surgiu de um projeto de extensão acadêmico, com a missão de fornecer uma segunda visão no processo da aprendizagem de lógica de programação, de forma a compreender os elementos básicos de aplicações móveis e como estas são projetadas, isto com o auxílio da programação em blocos, ferramenta essencial para melhor visualização da lógica de programação enquanto ela é elaborada.

Fonte: IA *Expert Academy*[1]

Adiante, neste material, trataremos das estruturas principais que compõem o tópico da lógica de programação, mas, primeiramente, vamos discutir sobre certos porquês:

LÓGICA DE PROGRAMAÇÃO, EU PRECISO MESMO ?

A grande maioria dos campos de conhecimento possuem alguma base fundamental na qual devemos construir nosso saber, notamos então alguns exemplos: Na matemática essa base é a aritmética, para aprender uma língua diferente a base do aprendizado se encontra na gramática da mesma e para o desenvolvimento de *software* essa base é a lógica de programação.

[1]Fonte:<https://iaexpert.academy/wp-content/uploads/2021/08/Logo-Lo%CC%81gica-de-Programac%CC%A7a%CC%83o.png>

Se você, nosso leitor, almeja uma carreira dentro do campo da Tecnologia da Informação ou da Engenharia de Software, então é necessário dominar o essencial, isto é, deve-se haver o claro entendimento e capacidade de execução das práticas e conhecimentos que compõem a lógica de programação. Felizmente, os meios para esse estudo são diversos e de fácil acesso, agora mesmo estamos fazendo progresso nessa missão através da leitura deste livro.

Logo, percebemos que o porquê de precisarmos compreender a lógica de programação, se encontra justamente no caminho que buscamos trilhar, seja na área da computação ou simplesmente ao procurar aprimorar nossas capacidades de pensamento algorítmico computacional. Ainda neste tópico, outra pergunta pode surgir:

MAS SERVE SÓ PARA TI ?

A didática do livro é voltada para o tópico de desenvolvimento de aplicativos, que é, naturalmente, um tema recorrente da Tecnologia da Informação, no entanto, a base teórica é abrangente para todos os interessados no essencial da programação, tal que é útil para diversos campos da ciência, pois a programação pode ser aplicada para o desenvolvimento de *software* tão bem quanto pode atender às necessidades específicas de outros campos do conhecimento.

Vemos exemplos de problemas solucionáveis pela programação por toda parte, alguns destes são: simulações digitais em laboratórios, processamento de recursos financeiros dentro da bolsa de valores e, também, em análises ambientais que focam na obtenção de estatísticas sobre a saúde do planeta. Visto que, essas tarefas são, de forma geral, impossíveis de serem realizadas sem o auxílio tecnológico, fica claro que existe valor em compreender os fundamentos da TI independente de campo de atuação, portanto, não tema em prosseguir a leitura, já que certamente algo de valor será aprendido ao final.

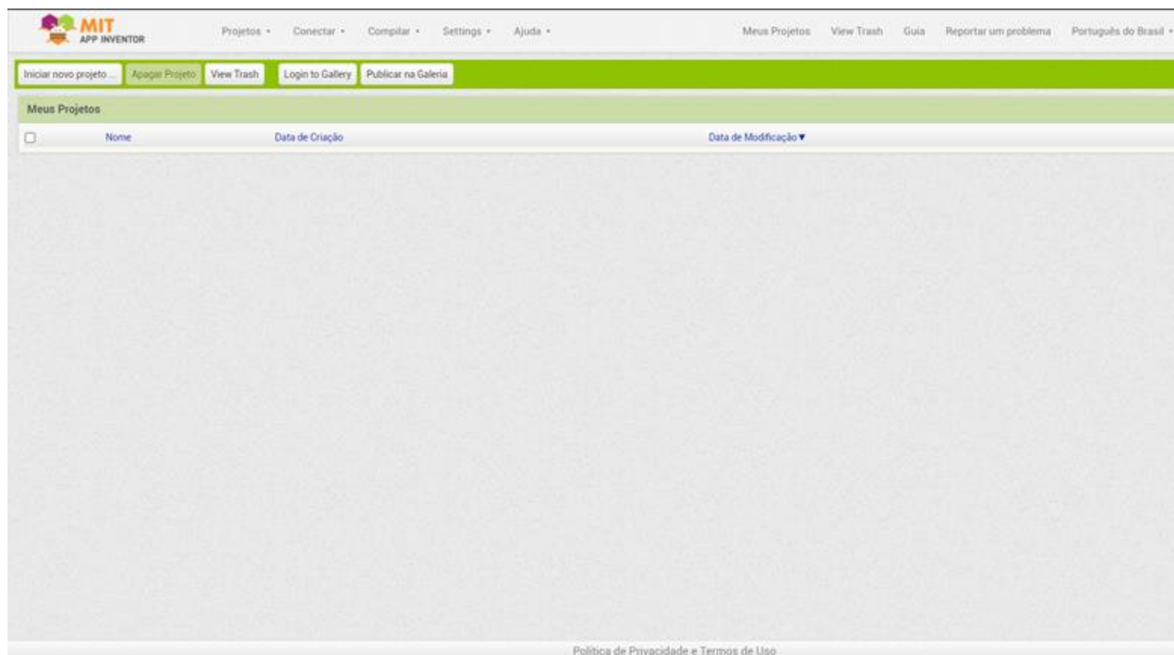
Sem mais delongas, vamos explorar a plataforma que nos ajudará a aprender e praticar os conceitos da lógica de programação, o *App Inventor*.

APP INVENTOR

COMO FUNCIONA ?

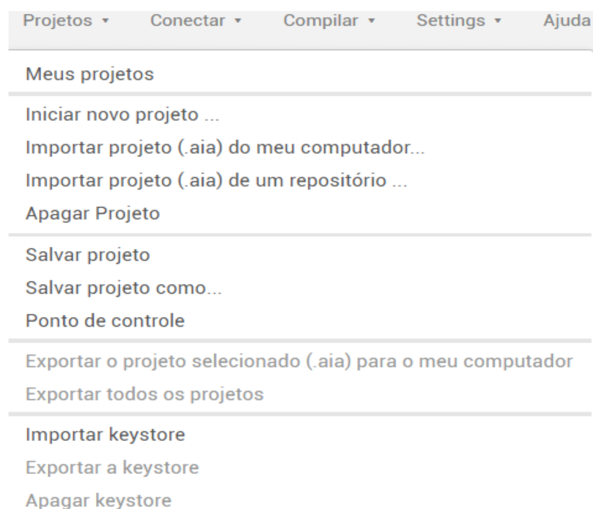
Antes de tudo, vamos explorar as nossas opções dentro da plataforma do *App Inventor*! E para isso vamos por partes, da criação da conta até como elaborar nossas aplicações.

Essa é a página inicial do *MIT App inventor*, nela temos acesso às demais páginas do *site*, essas que nos dão acesso a informações, notícias, recursos educacionais, além de expor o alcance que a plataforma teve ao longo dos anos. Mas o importante nesse primeiro momento é aquele ícone laranja localizado no canto superior esquerdo, o "***Create Apps***". Assim que clicado, uma nova guia será aberta para autenticação no serviço via Contas da *Google*. Depois de efetuar seu *login*, essa nova guia será exibida:

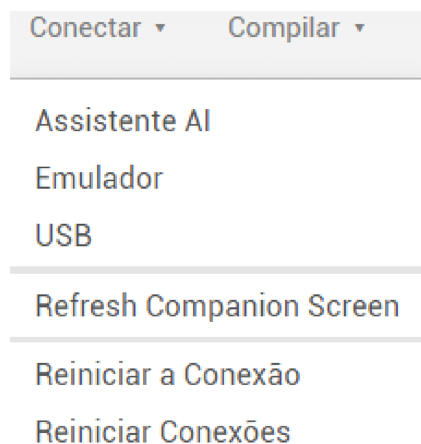


Essa tela é o gerenciador de projetos do *App Inventor*, aqui temos uma visão geral de todos os aplicativos que criamos dentro da plataforma, bem como podemos acessar os controles para criação e publicação de projetos.

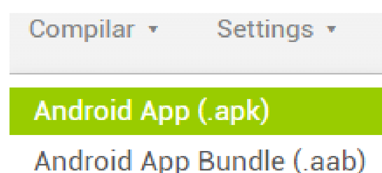
Acima nessa guia alguns menus estão disponíveis. Vamos começar pelo de "**Projetos**" que, ao clicar com o *mouse*, exibe uma lista com as funcionalidades de criação de um novo projeto, importação de um projeto do computador ou repositório, exportação de projetos, funcionalidades de salvamento e, até mesmo, permite importar uma *keystore* personalizada.



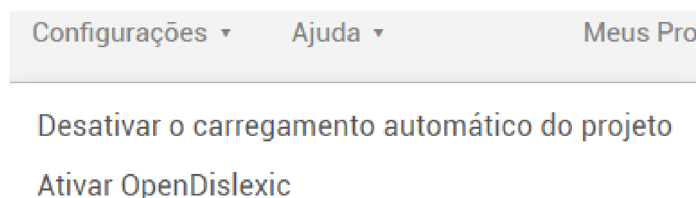
À direita de projetos, temos o menu de conexões. O *App inventor* permite que suas aplicações sejam executadas no seu próprio celular pelo AI2 *Companion App*[2], e essa funcionalidade pode ser acessada ao clicar no botão de "**Assistente AI**". Apontamos também que, além dessa opção, podemos testá-las em um emulador, bem como, existe a possibilidade de conectar um telefone via *USB* ao computador caso o aplicativo não funcione na sua rede. As opções de reiniciar as conexões também são disponibilizadas nesta lista suspensa.



Os projetos criados na ferramenta *MIT App inventor* podem ser compilados para dois tipos de extensões: A extensão **.apk**, que é o tipo de extensão padrão utilizada para aplicativos *mobile* na plataforma *Android*, e a extensão **.aab**, que é o tipo de extensão obrigatório para quem deseja submeter sua aplicação em lojas como a *Google Play*.

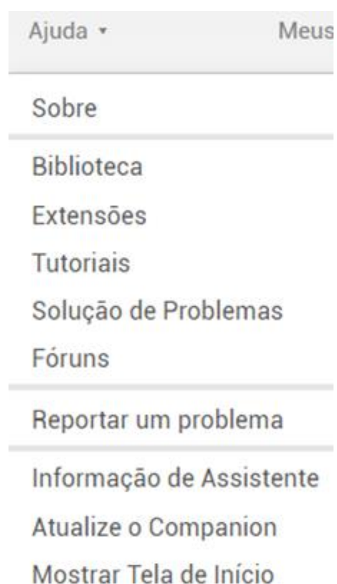


Em "**Configurações**", logo ao lado do menu de "**Compilar**", temos duas opções, a de ativar ou desativar o carregamento automático do projeto, e a opção que permite usar a fonte *OpenDislexic*, que possibilita que pessoas com dislexia usem a plataforma com maior facilidade.



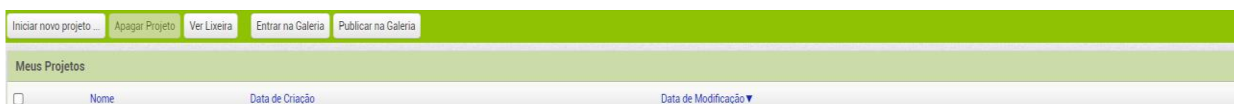
[2] **Baixe nas lojas**:
Para *Android* <https://play.google.com/store/apps/details?id=edu.mit.appinventor.aicompanion3>
Para *iOS* <https://apps.apple.com/br/app/mit-app-inventor/id1422709355>

Também existe um menu de “**Ajuda**”, onde pode-se obter informações sobre a ferramenta do *App Inventor* de forma geral, além de fornecer informações sobre bibliotecas, extensões, tutoriais, acesso a fóruns e, até mesmo, lista a página onde é possível reportar problemas encontrados.

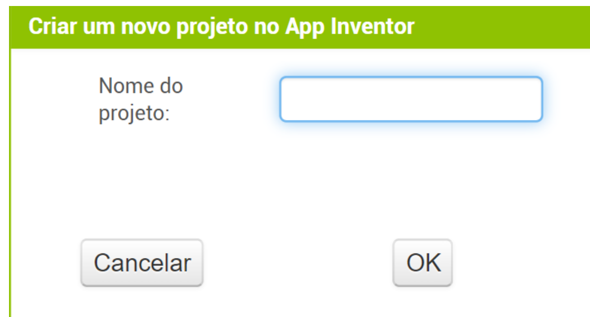


Além destes menus já citados, também temos fixados alguns atalhos úteis que direcionam para páginas de uso frequente:

Mais abaixo na página, é possível ter uma visão dos projetos do usuário e algumas informações destes, como o nome, a data de criação e a data da última modificação. Acima dessa área geral de exibição dos projetos, temos botões com funcionalidades diferentes, como o botão de “**Iniciar**” um novo projeto ou “**Apagar**” um projeto, além disso, existe a possibilidade de acessar diretamente a “**Lixeira**” através do botão presente nesse menu. Para entrar na galeria de projetos ou publicar na galeria de projetos, basta clicar nos botões também localizados nessa parte da tela.

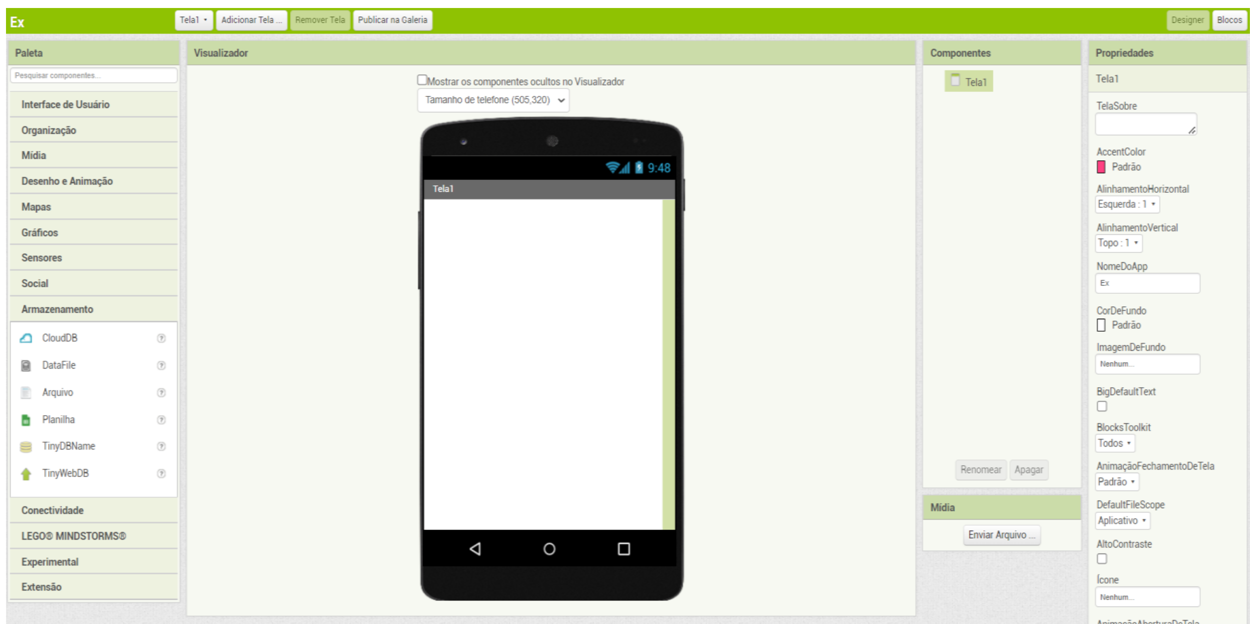


Ao clicar no botão de "**Criar**" um novo projeto é apresentada a caixa de diálogo abaixo, onde é colocado o "**Nome do projeto**" e logo em seguida a página é redirecionada para a tela de desenvolvimento.



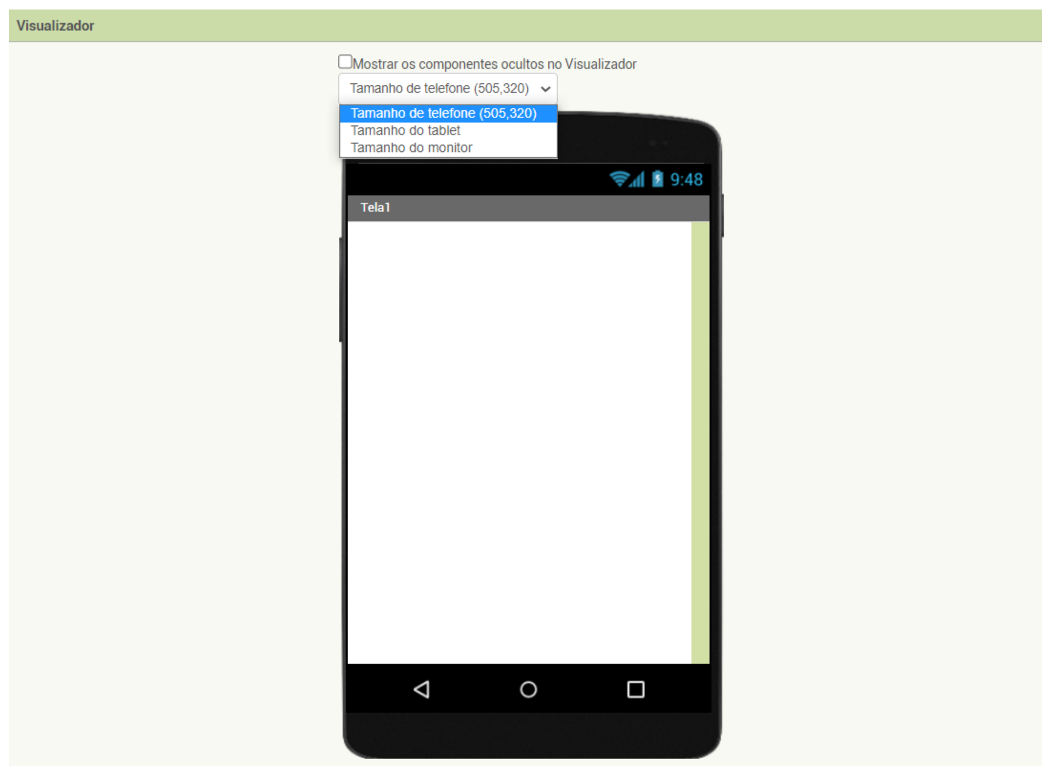
COMO USAR ?

Na criação de um projeto temos a tela de *designer* da aplicação, onde são adicionados todos os componentes da interface: as telas da aplicação, suas propriedades, como esses componentes são organizados, onde também são adicionadas mídias e extensões. Além da tela de *designer*, existe a tela de blocos, nessa é onde definimos toda lógica da aplicação, como cada componente se comporta, como os componentes reagem a outros componentes ou entradas do usuário. Vamos começar pela tela de *designer*:

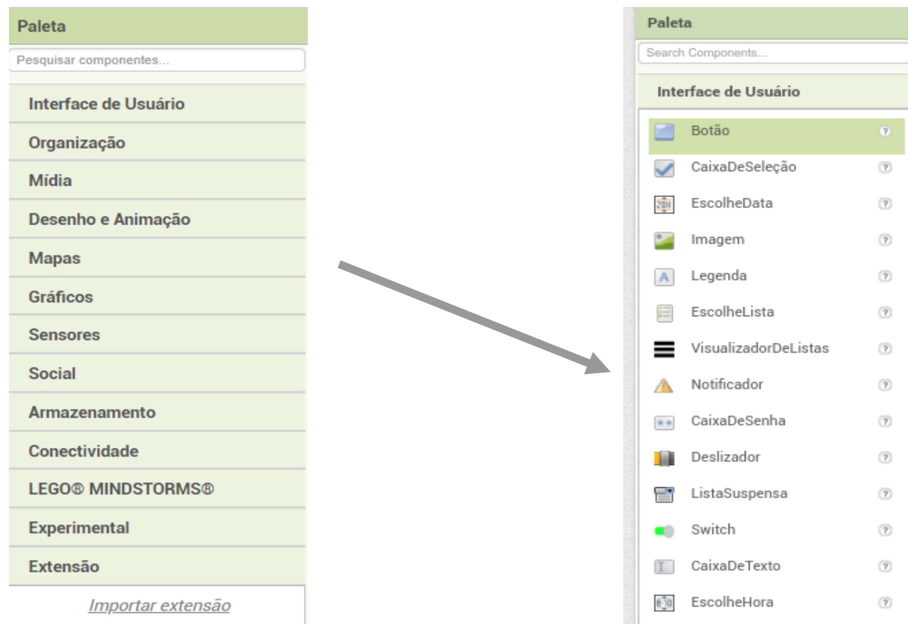


Essa tela é dividida em seis seções. A primeira é onde fica localizado o nome do projeto, neste exemplo o nome usado foi “**TODO_app**”. Aqui é onde pode ser feita a adição, remoção e a troca de telas da aplicação, caso necessário. Já no canto direito, existem os botões que trocam a tela de *designer* pela tela de blocos.

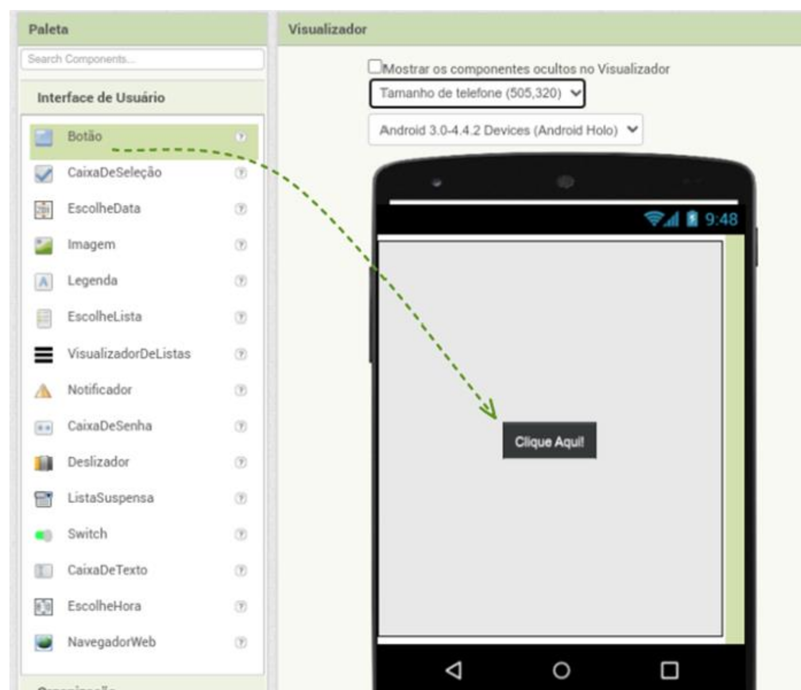
Na seção central, temos a visualização de como os componentes são organizados no *layout* da aplicação. A ferramenta dá 3 opções de tamanhos de telas diferentes, o que é útil para se ter uma visão de como a aplicação se comporta em diferentes dispositivos, no sentido de visualização.



Do lado esquerdo da tela é possível ver a seção da “**Paleta**”, onde selecionamos os componentes que pretendemos adicionar na aplicação. O *App Inventor* disponibiliza vários tipos de componentes que vão desde botões, caixa de textos, imagens, legendas, até coisas como sensores, gráficos, mapas, componentes ligados a armazenamento, e diversos outros.

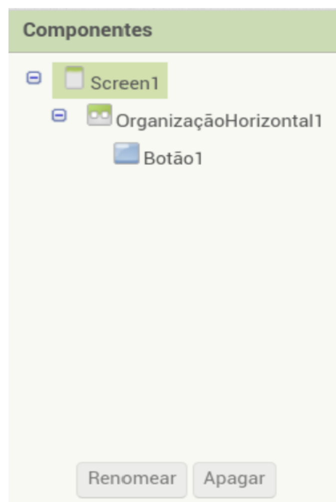


Para adicionar componentes no *App Inventor* basta arrastá-los para tela do celular no editor:

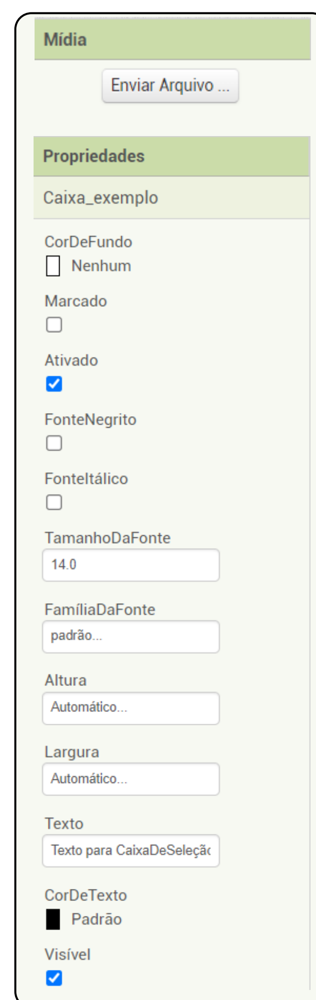


Do lado direito da aba de visualização, fica a aba dos componentes adicionados a aplicação, incluindo a própria tela e componentes não visíveis como sensores,

conectores de bancos de dados, dentre outros possíveis. Nessa aba também podemos renomear e excluir esses componentes.

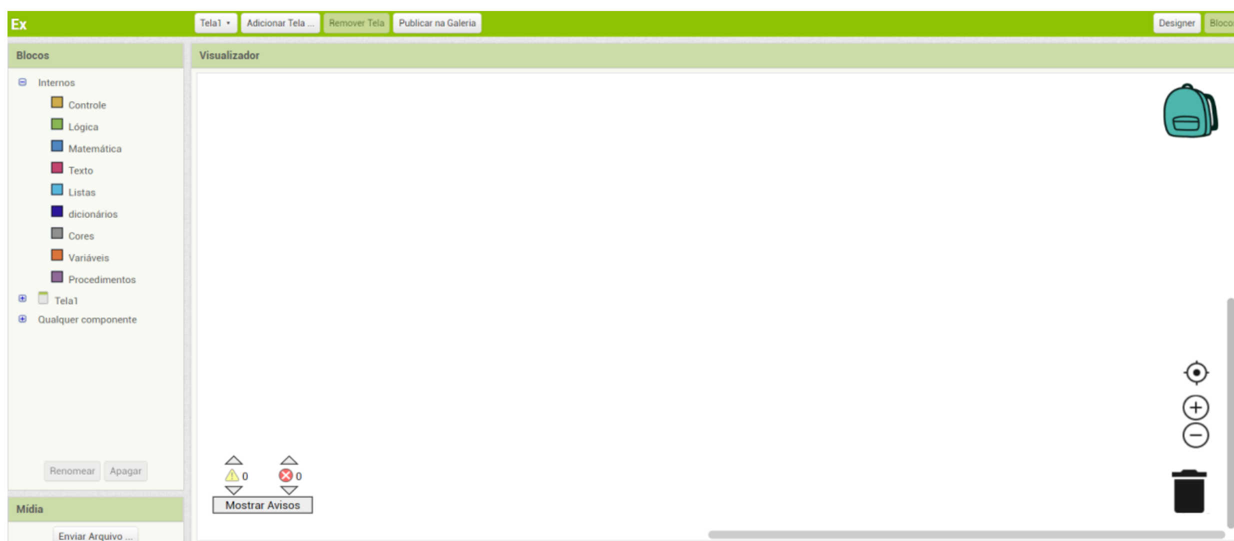


Logo abaixo dos “**Componentes**”, temos um pequeno botão responsável pelo *upload* de mídias para aplicação (fotos, vídeos, áudios, dentre outros).

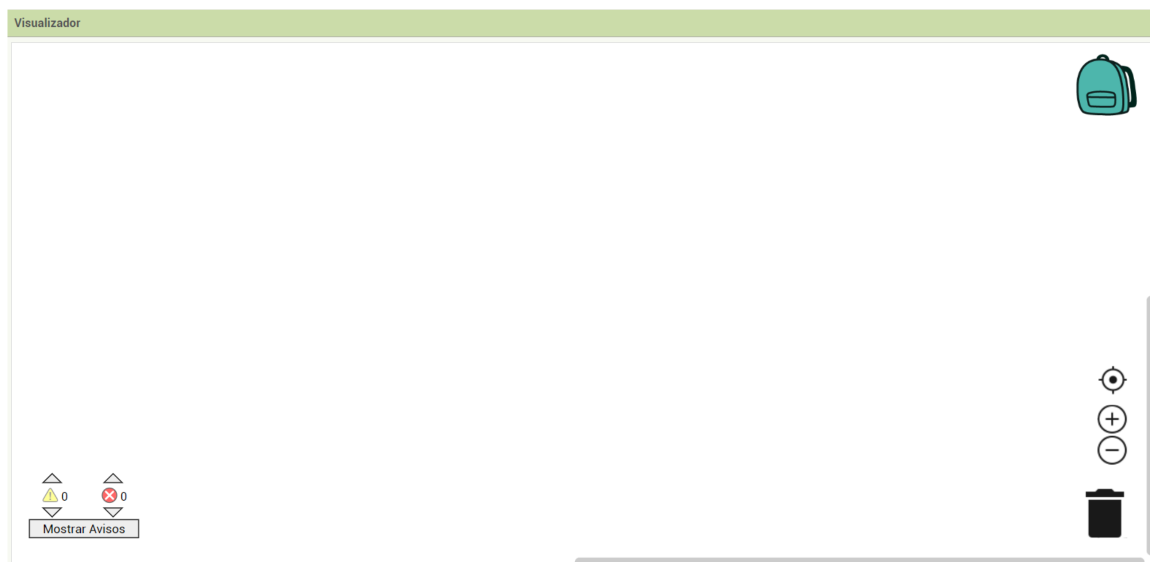


E a última seção da tela de *designer* é a aba de “**Propriedades**”. Ela é responsável pelas propriedades de cada componente da aplicação. Nesse exemplo está exibindo as informações de uma caixa de seleção, e é possível ver que a ferramenta dá a opção de alterar suas propriedades. Cada componente possui suas propriedades específicas, permitindo a personalização de diversos detalhes da sua aparência e comportamento. No exemplo ao lado, podemos notar algumas propriedades, como: cor de fundo da caixa de seleção, o tamanho da fonte do texto interno, o tipo da fonte, a visibilidade do componente e assim por diante.

A segunda metade do editor se trata da tela de blocos e, como foi dito anteriormente, é nessa tela que é feita toda lógica funcional dos componentes da aplicação. A visualização dessa tela é diferente da tela de *designer*, ela possui uma área livre mais ampla para facilitar a organização ao programar com os blocos.



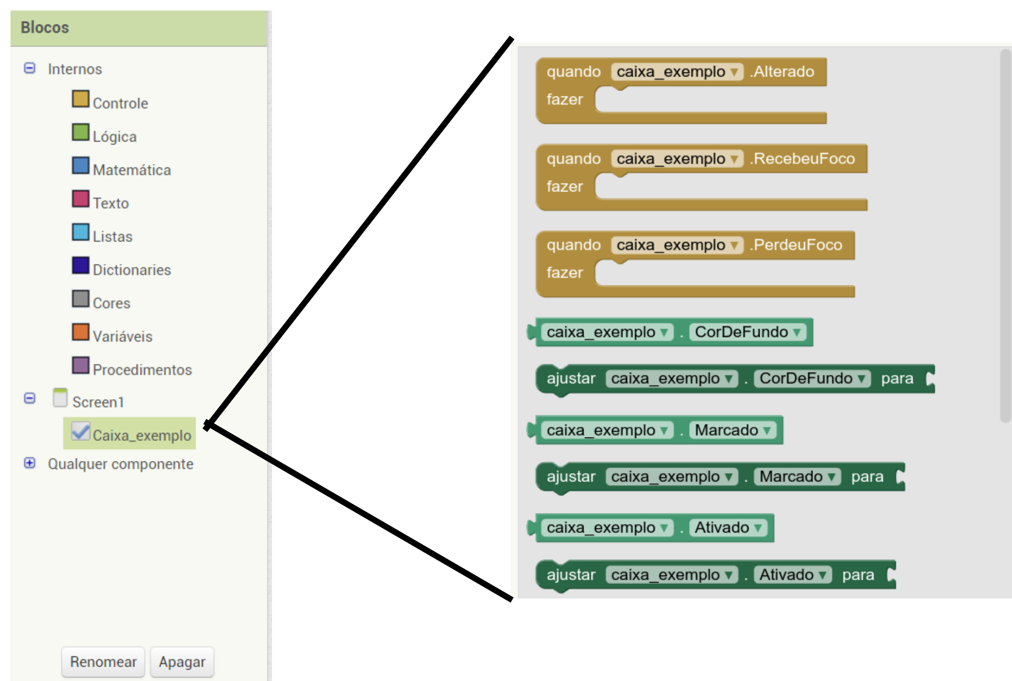
Essa área ampla é o "**Visualizador de Blocos**", além de ser o espaço que desenvolvemos o código do aplicativo, ela possui certos ícones, como a lixeira para descartar blocos de código e o da mochila, que armazena blocos de código. Encontramos aqui também botões para controlar o *zoom*, centralizar a tela, mostrar algum aviso de erro ou advertências de incoerências.



E por último, temos a aba mais importante do editor dos blocos, a seção onde estão dispostos os blocos de códigos. Os blocos no *App inventor* possuem cores diferentes para cada tipo de ação que eles realizam.

Os blocos laranjas condizem com ações de controle da lógica da aplicação, os blocos verdes possuem relação com expressões lógicas e os blocos azuis com expressões matemáticas, os blocos roxos correspondem a ações relacionadas a textos e blocos na cor ciano com ações de listas, já os blocos azul marinho são relacionados ao uso de dicionários, blocos cinza são relacionados a cores, enquanto blocos laranjas têm relação com variáveis usadas durante a aplicação e blocos lilás correspondem a procedimentos.

Além desses, cada tipo de componente também pode ter uma série de blocos que manipulam suas propriedades ou gerenciam eventos relacionados a eles, respeitando também o esquema de cores estabelecido.



Vale a pena ressaltar que os blocos de código também são adicionados no editor ao arrastar o bloco até a área de visualização, da mesma forma que no modo *Designer*.

E com isso, percorremos todas as partes necessárias para começar a usar a plataforma e praticar lógica de programação com exemplos apresentados mais à frente ou, até mesmo, com ideias que você mesmo gostaria de desenvolver.

LÓGICA DE PROGRAMAÇÃO E PRÁTICAS

PROGRAMAÇÃO: O ESSENCIAL PARA INICIAR NO APP INVENTOR

Seja no mercado ou na área acadêmica do desenvolvimento de *software*, existe uma ampla diversidade de linguagens de programação competindo pelo interesse de profissionais e, para um iniciante que deseja entrar na área de programação e desenvolvimento, essa grande quantidade de tecnologias pode gerar uma série de incertezas por onde se deve começar, no entanto, essa preocupação não é tão relevante em perspectiva, pois a experiência obtida ao programar pode ser reinterpretada em outra sintaxe[3] de programação.

Deve-se ter em mente que cada linguagem de programação tem suas próprias particularidades e paradigmas que evoluíram de acordo com o passar dos anos. Entretanto, todas essas linguagens, incluindo o *App Inventor*, possuem a mesma base de programação, baseados em estruturas de condições e repetições que serão explicadas abaixo, juntamente com o conceito de eventos, que é o paradigma muito presente no desenvolvimento *mobile*, este que é o modelo em que o *App Inventor* é baseado.

CONDIÇÕES

A estrutura condicional, ou também conhecida como estrutura de decisão, é basicamente a forma em que definimos o controle de fluxo do programa. Essa estrutura define os códigos que serão executados a partir do resultado de uma condição que foi imposta nesse bloco, podendo levar a diferentes resultados. Um exemplo de um problema humano que pode facilitar o entendimento sobre esse tipo de estrutura seria o seguinte:

Se amanhã fizer sol,

então irei à praia.

Vemos nesse exemplo que a condição imposta para o problema é o fato de no dia seguinte haver sol, e o resultado seria a ida à praia. Esse exemplo, explorando um pouco

---

[3] Sintaxe, na programação, se trata das regras que definem instruções e expressões válidas que podemos usar na língua.

mais a estrutura condicional, pode ter diferentes caminhos a depender do resultado da expressão, vejamos:

Se amanhã fizer sol,

então irei à praia,

senão,

ficarei em casa e verei um filme.

Como podemos observar, no caso da condição não ser atendida, ou seja, **não fazer sol**, seria feito outra coisa, podemos imaginar como um comando diferente a depender da resposta.

Outro ponto a ser abordado sobre estruturas condicionais são as estruturas de condição alinhadas, que são estruturas que possuem uma condição dentro de outra, veja no exemplo:

Se amanhã fizer sol,

e se eu acordar cedo,

então irei à praia,

senão,

ficarei em casa e verei um filme.

Nesse exemplo podemos ver que não existe somente uma condição para que se vá a praia, o fato de fazer sol deixa de ser o único motivo, e daí também entra a condição de acordar cedo, fazendo com que uma decisão seja alinhada a outra.

REPETIÇÕES

Outra das estruturas quando se trata de programação básica é a estrutura de repetição. O conceito dela é simples e se baseia no fato de uma ação, ou comando no âmbito da programação, ser repetida enquanto uma condição for verdadeira ou um determinado número de vezes.

As estruturas de repetição são importantes quando é necessário que ações repetidas sejam executadas mais de uma vez, como por exemplo verificar um elemento em uma lista de dados, caso fosse necessário escrever linhas de código para verificar cada item, um por um, dessa lista, deixaria o programa enorme. Existem 4 diferentes estruturas de repetição, porém todas com o mesmo conceito de executar ações de forma repetida. A primeira estrutura é fundamentada na ideia de os comandos serem executados até que uma condição seja falsa, exemplo:

Enquanto a música estiver tocando,

nós iremos dançar.

Essa estrutura na maioria das linguagens é marcada pela palavra-chave ***while***, que tem como tradução "**enquanto**", denotando que a ação deve ser executada enquanto a variável ou expressão for verdadeira. No exemplo, notamos que a condição seria a música tocando e a ação é dançar.

Outro tipo de estrutura é utilizada quando já se sabe quantas vezes quer que a ação seja repetida, tomando como exemplo o percurso de uma lista, vejamos:

Para todos os alunos dessa sala,

dê um chocolate no Dia do Estudante.

Assim como a estrutura de repetição anterior tinha a palavra-chave ***while***, essa estrutura tem sua palavra-chave, na maioria das linguagens é a palavra ***for***, que traduzindo do inglês seria "**para**", como no exemplo. Nesse exemplo é notável que a ação foi executada para um número determinado de vezes, no caso o número de alunos que a sala possui. Também é possível limitar o alcance da repetição usando constantes numéricas, veja o exemplo:

Para os cinco primeiros alunos desta sala,

dê um chocolate no Dia do Estudante.

Ainda existem outros dois tipos de estrutura de repetição, a ***do..while*** (***faça..enquanto***) que é semelhante a estrutura ***while***, porém garante a execução do código seja executada pelo menos uma vez e o último tipo que vamos abordar é a

estrutura conhecida por ***foreach*** (**para cada**), que seria uma simplificação do ***for*** quando usada especificamente para percorrer listas aplicando cada elemento a um procedimento definido no código.

EVENTOS

Diante de vários paradigmas de linguagem de programação, um dos mais prevalentes no desenvolvimento *mobile* e *web*, incluindo a ferramenta *App Inventor*, é o paradigma orientado a eventos. Esse paradigma se baseia na ideia de que uma parte do programa se dedica a criar e monitorar ouvintes de eventos, estes são objetos que ficam à espera de interações por parte do usuário da aplicação, para assim ativar o que são chamadas[4] de eventos, que são procedimentos que realizam a função associada ao evento criado, esses ouvintes podem ser botões, caixas de texto, caixas de seleção, dentre outros.

Um programa que é desenvolvido nesse paradigma não possui um ponto de parada definido, já que ele é programado para reagir de acordo com as ações do usuário, por exemplo, para fechar a aplicação haveria um evento programado no botão de "**Fechar**", que quando clicado chamaria o procedimento responsável por fechar a aplicação. Os eventos também podem ser chamados de gatilhos, já que são ativados com alguma interação do usuário. Procurando por exemplos no cotidiano teríamos:

Quando a campainha tocar,

atenda a porta.

VAMOS PRATICAR

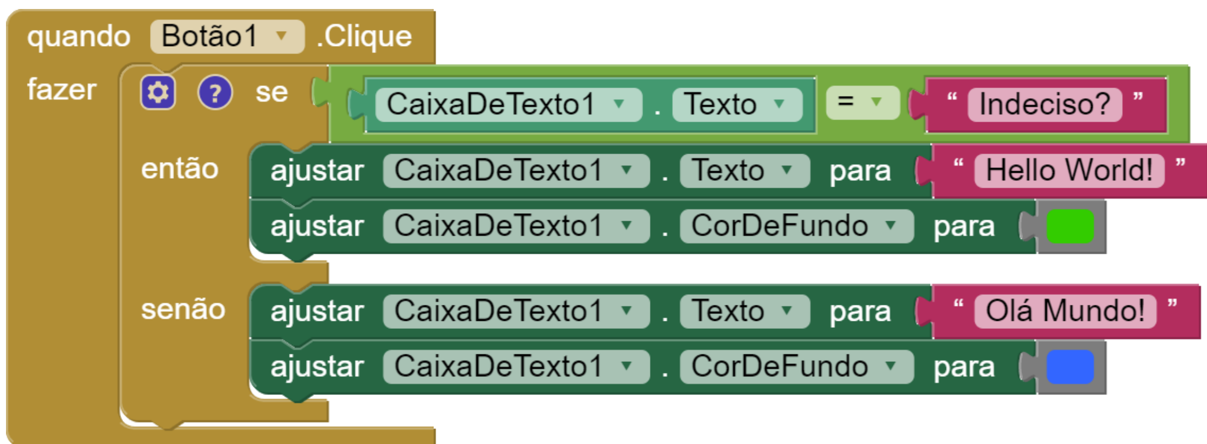
A fim de exemplificar o uso e funcionamento das estruturas condicionais foi desenvolvido uma aplicação simples, separada em duas telas diferentes com um exemplo em cada uma. De início veremos o exemplo onde botões foram utilizados para auxiliar o processo de decisão:

[4] Chamadas consistem em utilizar um procedimento quando este é referenciado pelo seu nome no código, a fim de executar sua rotina de código.

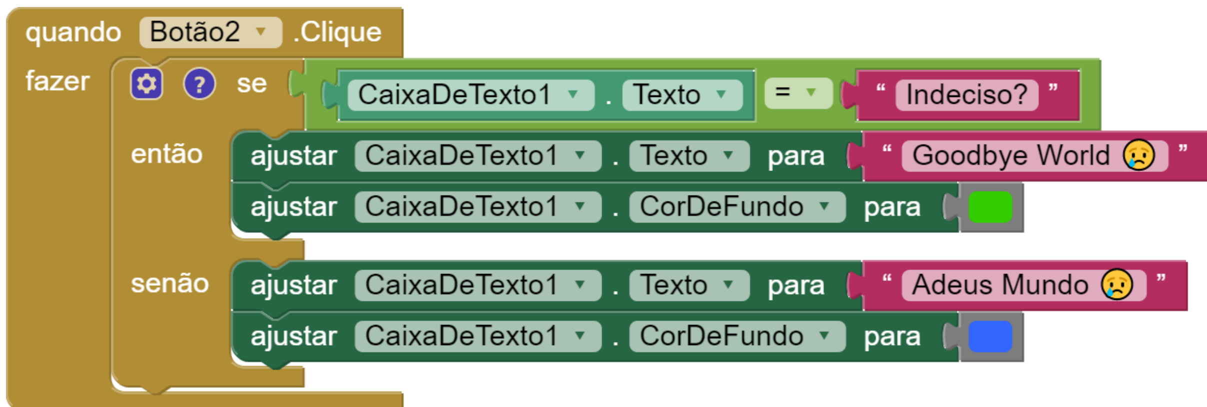
Nesse exemplo, utilizaremos os botões e o tempo de clique de cada um como condição, resultando em uma resposta exibida na caixa de texto localizada acima dos botões.

Para julgar o tempo de clique de cada botão é necessário o uso de um bloco auxiliar, nele saberemos quando houver um clique prolongado e mudaremos o estado da caixa de texto.

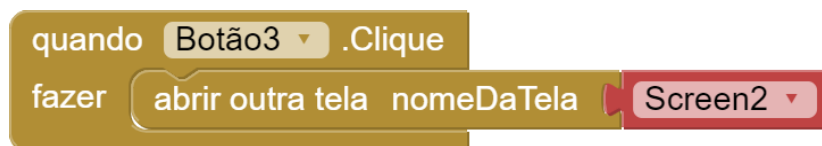
Com a mudança da caixa de texto acionada pelo evento do bloco anterior, podemos estruturar o nosso bloco condicional para o primeiro botão, onde teremos duas respostas diferentes apresentadas para o usuário a depender do resultado da condição **CaixaDeTexto = "Indeciso?"**.



No segundo bloco a mesma lógica foi aplicada, sendo que a única entre eles está na resposta exibida ao usuário.

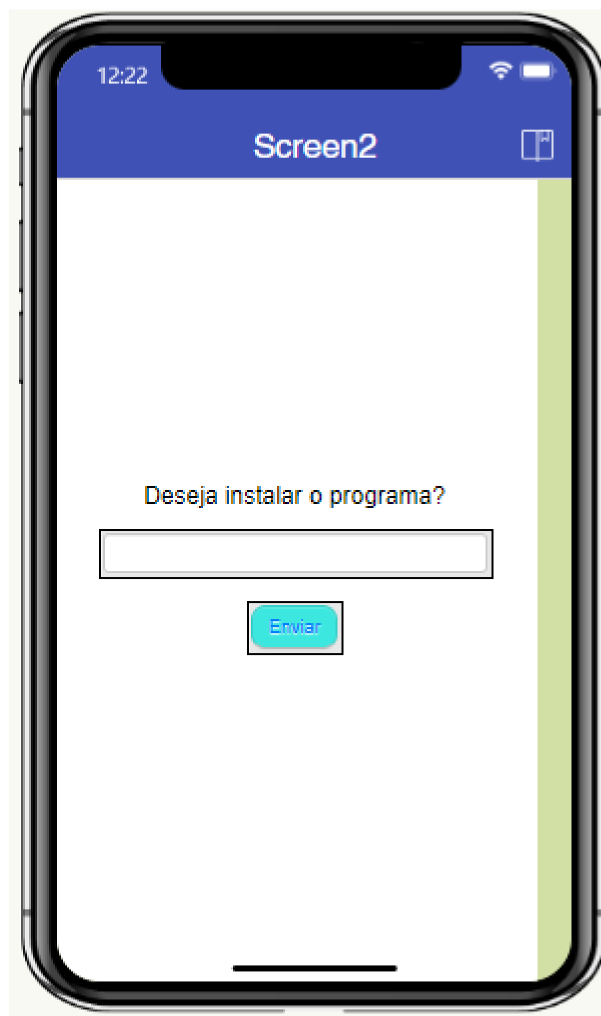


Já o terceiro botão, rotulado de "**Próximo Exemplo**" possui como evento a mudança de telas, trazendo assim, um novo conjunto de blocos e uma nova tela de *design*.



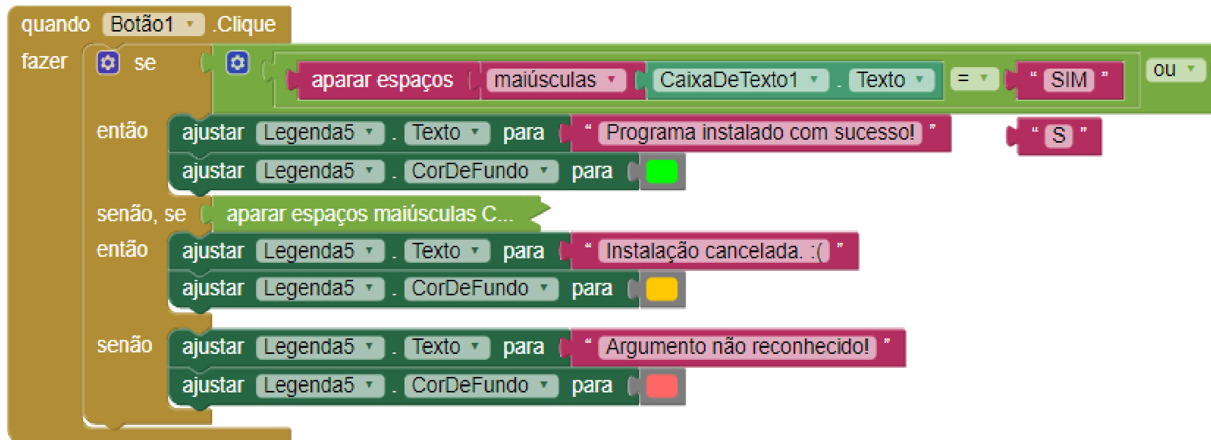
Agora na segunda tela é apresentado o exemplo 2 das estruturas condicionais, nele pode-se observar uma caixa de entrada que permite ao usuário enviar uma resposta para a pergunta anunciada, com a adição de um botão para envio da resposta.

Assim, a partir dessa entrada do usuário, nosso sistema lógico condicional poderá avaliar o que deve fazer a seguir.

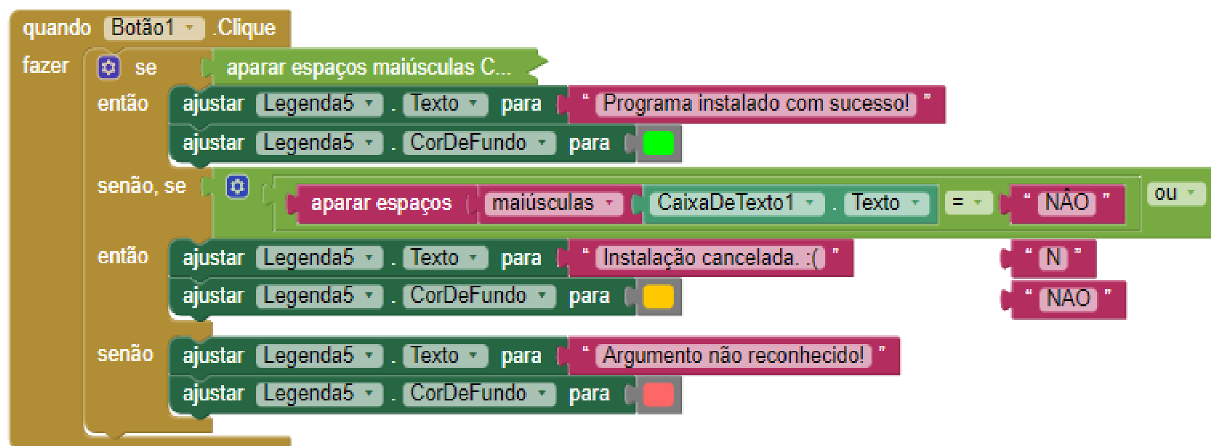


Voltando a atenção para os blocos veremos que o sistema compreende apenas duas entradas como respostas aceitáveis, essas respostas ou entradas dos usuários serão então tratadas e utilizadas como argumento condicional em nosso bloco. Esse tratamento ou limpeza da informação passada pelo usuário, neste caso, consiste em limpar espaços em branco e reformatar toda a resposta para letras maiúsculas, a fim de facilitar a comparação com as respostas esperadas pelo código.

A primeira resposta, essa que pode ser observada na imagem abaixo, é o “**Sim**”, que após o tratamento se torna “**SIM**”. Poderíamos já parar por aí, porém como existem várias formas, nem sempre corretas, de escrevermos uma palavra, devemos garantir outras alternativas para a condicional. No caso do “**SIM**”, essa forma alternativa seria o “**S**”, logo a condição foi acomodada a aceitar “**SIM**” ou “**S**”.



Na segunda condição de nosso bloco, denominada como “**senão, se**” no *App Inventor*, é tratada a resposta “**Não**” do usuário. Nesse cenário tudo transcorre da mesma forma que no anterior em relação ao tratamento da entrada, logo lidamos com um “**NÃO**” na nossa comparação, porém assim como o “**Sim**” que pode ser compreendido pelo sistema como “**S**”, a condição do “**NÃO**” também foi acomodada para aceitar “**NÃO**”, “**NAO**” (sem o til), ou “**N**”.



PROCEDIMENTOS

Estudaremos agora algo instrumental para a programação: Os procedimentos. Estes permitem que o seu código seja repartido em blocos funcionais que executam uma tarefa específica, sendo assim, essa segmentação de código proporciona certas vantagens para aplicações que fazem um bom uso desse princípio, tal como:

- A divisão do código em partes menores torna-o mais gerenciável;
- Procedimentos podem ser facilmente reutilizados em contextos diferentes, o que possibilita reuso de código;
- Quando os procedimentos são utilizados de forma intuitiva, o código se torna mais fácil de entender, preservar e modificar.

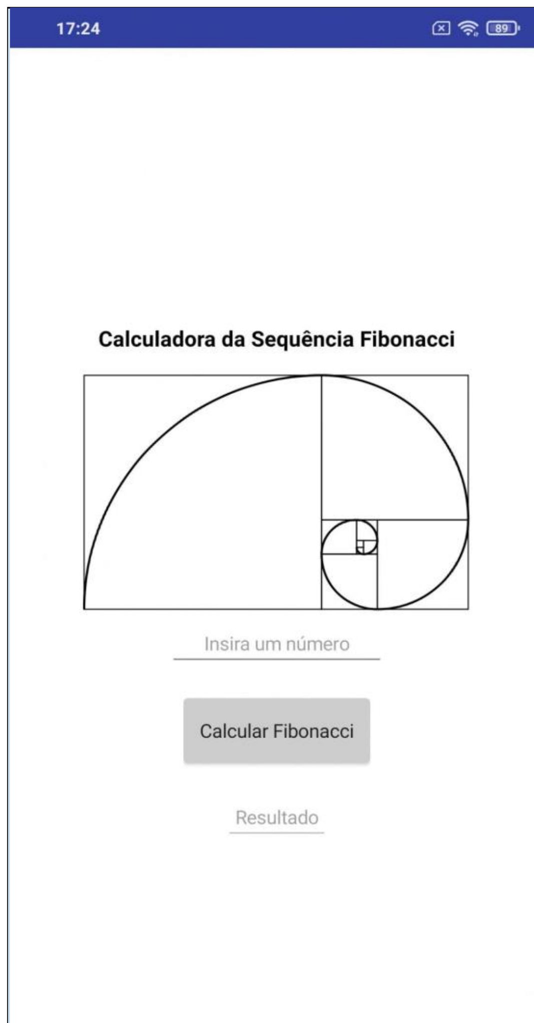
Os procedimentos são definidos pelo programador, cuja declaração consiste na definição de um nome para ele, seguido de sua implementação. Vale ressaltar que procedimentos podem ser criados com receber parâmetros de entrada, sendo estes valores que o procedimento recebe quando chamado dentro da aplicação e utiliza internamente para realizar sua tarefa, bem como, é possível estabelecer variáveis locais nesses procedimentos, essas que podem ser utilizadas apenas dentro dos procedimentos onde são definidas. Além disso, eles podem retornar valores de saída, que são resultados que são passados de volta para a parte do programa que chamou o procedimento.

A sintaxe do uso de procedimentos muda para cada linguagem de programação, algumas podem pedir por especificações de tipo de retorno para identificar o seu início, enquanto outras requerem uma palavra-chave como inicializador do procedimento, por exemplo: *function, fn, func e def*.

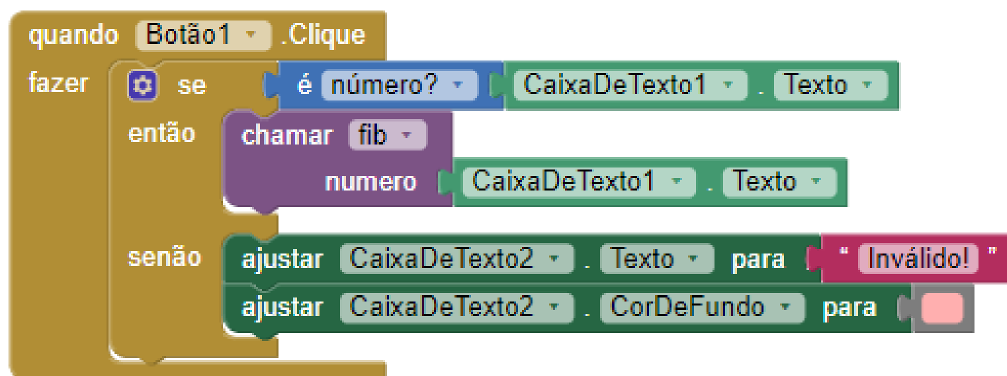
Com essa ideia geral dos procedimentos, vamos explorar essas ideias no *App Inventor*!

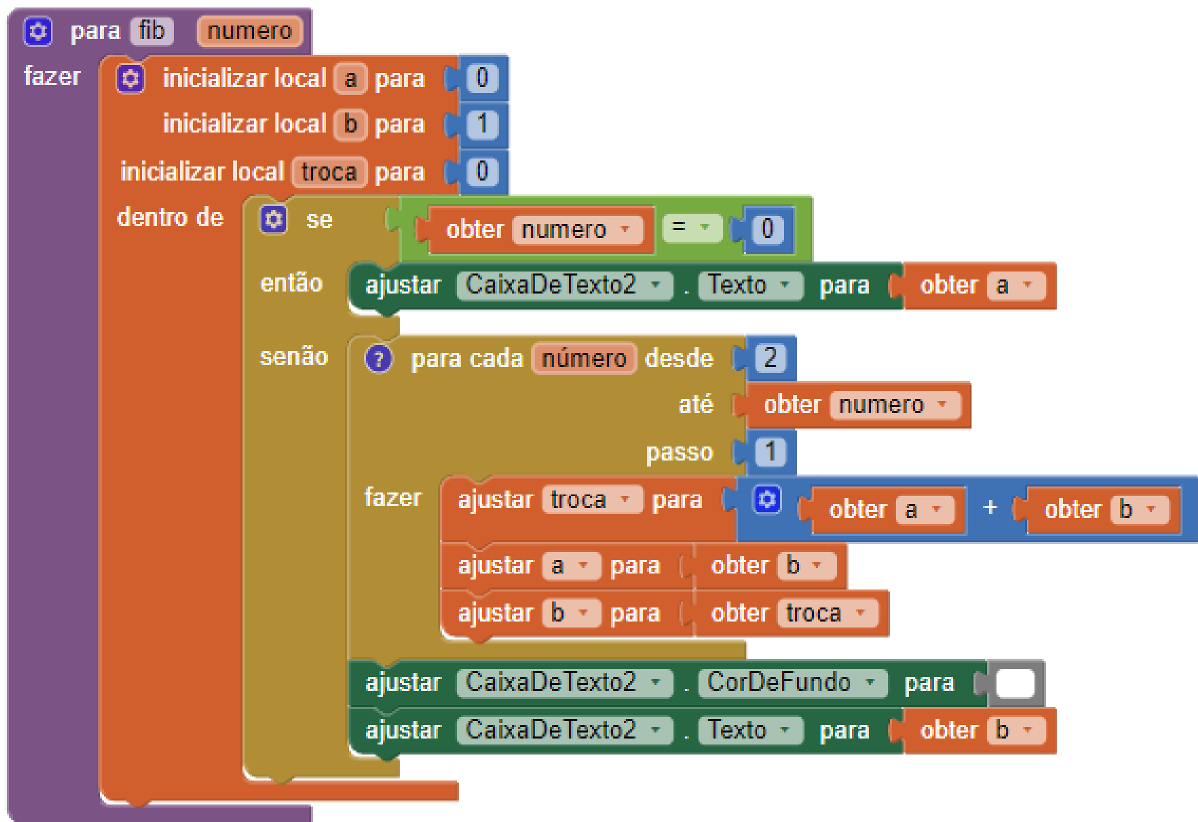
VAMOS PRATICAR

Para aplicar essa teoria propomos a elaboração de uma aplicação clássica no aprendizado de lógica de programação, uma calculadora da sequência de *Fibonacci*.



Para alcançar a funcionalidade desta aplicação utilizaremos os conceitos de procedimentos para organizar nosso código, veja a seguir os blocos elaborados:





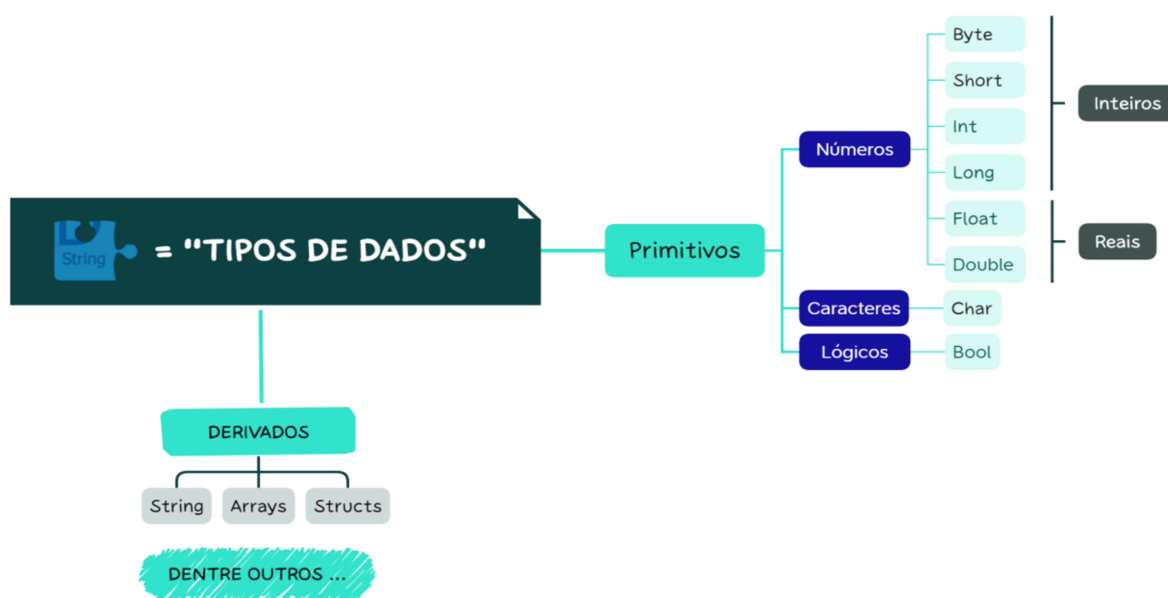
Compreenda agora como funciona. No bloco inicial, o qual gerencia o evento de, clique no botão de “**Calcular Fibonacci**”, teremos uma verificação essencial: se a entrada de usuário não for um número, então o cálculo não pode ocorrer, pois a função “**fib**” só aceita um número, caso seja uma letra ou outro símbolo não numérico, não é possível utilizar o valor como referência para o número de repetições do cálculo necessário, ou seja, no momento em que o usuário inserir um número válido na caixa de texto, então o botão vai funcionar como esperado.

Já no segundo bloco, a lógica pertinente ocorre de fato, primeiramente são inicializadas as variáveis locais do procedimento (**a**, **b** e **troca**), estas que servirão para calcular o valor final que será exibido. Logo em seguida, para casos em que o número inserido não é **0** ou **1**, o laço de repetição executa, tal que realiza a soma dos dois últimos números da sequência até alcançar a posição na sequência *Fibonacci* desejada, com o resultado sempre sendo armazenado em **b**, o valor mais atual do cálculo. Ao final da

execução a variável **b** é utilizada para atualizar a caixa de texto que exibe o resultado final, informando ao usuário o valor calculado da posição informada ao procedimento.

TIPOS DE DADOS

Dentro da programação, um tópico de alta relevância são os diferentes tipos de dados, tais que representam dados no computador, porém com classificações que distinguem como eles são interpretados, como são utilizados dentro de aplicações e como estes podem interagir ou como podem ser convertidos entre si. Cada um desses tipos de dados também reflete em como estes serão armazenados e processados no nível de máquina, pois cada um possui características específicas, como tamanho total de *bits*, diferentes representações através de sinalizadores, dentre outros fatores. Os tipos mais comuns de dados são os representados a seguir:



Explorando estes tipos primitivos, temos:

- Números inteiros (*Byte*, *Short*, *Int*, *Long*): São valores numéricos sem casas decimais, como 1, 2, 3, -4, -5, ...
- Números reais (*Float*, *Double*): São valores numéricos que permitem casas decimais, como 2.5, 3.14, -1.75, ...

- Booleanos (*Bool*): São valores interpretados como verdadeiros (*true*) ou falsos (*false*), que são usados em operações de lógica (*and*, *or*) e controle de fluxo (*if*, *else if*).
- Caracteres (*Char*): São valores que representam caracteres únicos, como 'a', 'b', 'y', 'z', ...

Vale ressaltar que, no caso das nomenclaturas dos tipos de dados relacionados aos números inteiros, a principal diferença é no tamanho alocado para cada variável criada, eles apresentam tamanhos de 1 *Byte* (*Byte*), 2 *Bytes* (*Short*), 4 *Bytes* (*Int*) e 8 *Bytes* (*Long*). Esses nomes podem variar a depender da sintaxe de linguagens de programação, algumas optam por omitir essas diferenças e tornar o *Int* o declarador de tipo padrão, já outras podem optar por uma forma padronizada para representá-los, por exemplo, i8 para 1 *Byte* de tamanho, i16 para 2 *Bytes*, i32 para 4 *Bytes* e assim por diante.

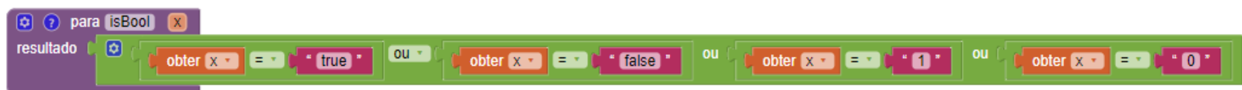
Além disso, também temos os tipos de dados derivados, estes são compostos de um ou mais dos elementos primitivos em certa organização estrutural. Por exemplo, *strings* são cadeias de caracteres que, ao serem processadas, resultam em linhas de texto, como "**Olá mundo!!!**". Outros tipos bem conhecidos de tipos de dados incluem Arranjos (ou *Arrays*), que são coleções de valores do mesmo tipo de dados organizados dentro de sua ordenação, e Estruturas (*Structs*), que são objetos estruturados que agrupam diferentes tipos de dados em uma única entidade definida pelo programador, estas permitem que dados associados sejam armazenados e acessados com facilidade.

Existem muitos outros tipos de dados, mas, sem mais delongas, exploraremos como que diferentes tipos de dados podem ser interpretados em uma aplicação:

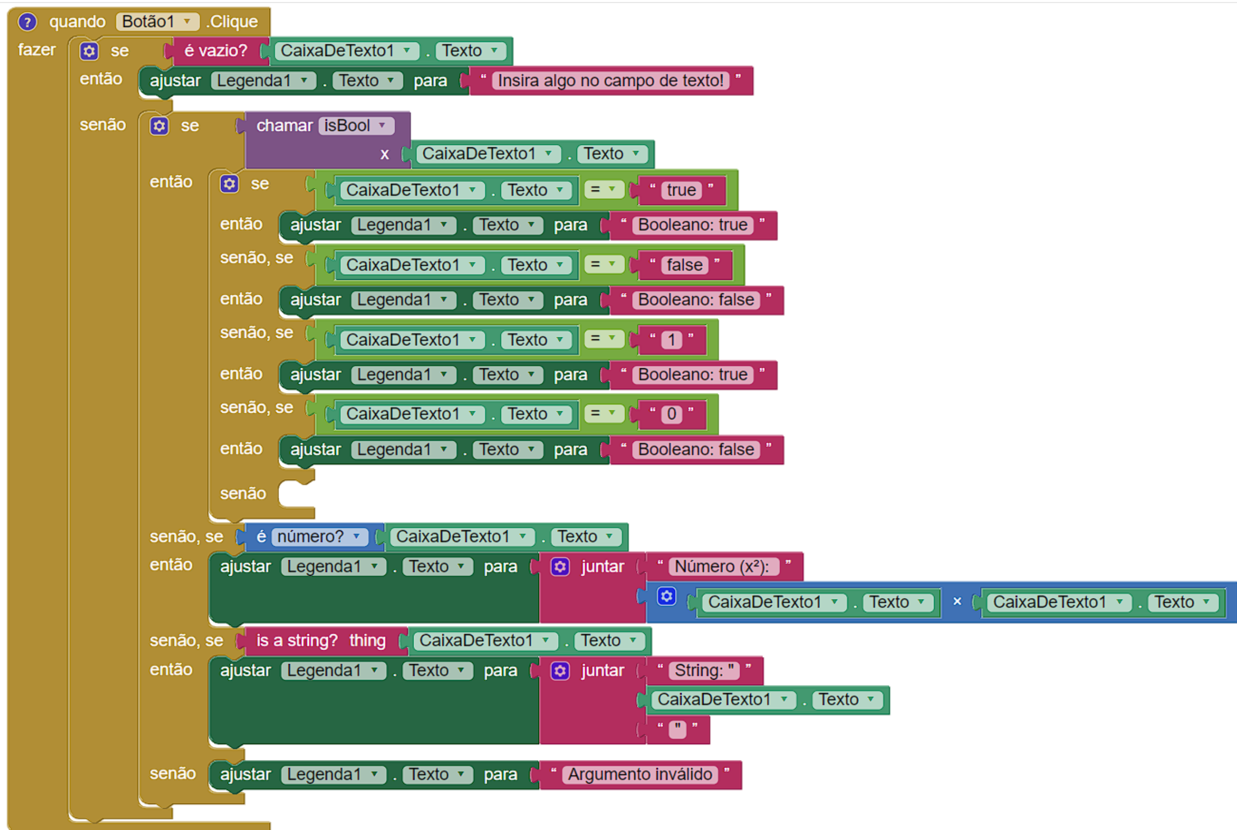
VAMOS PRATICAR

Quando tratamos dos tipos de dados em aplicações, uma das maiores preocupações que podem surgir é como gerar funcionalidades diferentes de acordo com os tipos de dados fornecidos à aplicação. Para explorar esse tópico vamos elaborar um aplicativo que avalia os tipos de dados que está recebendo.

Para esse aplicativo a ideia é a seguinte: será permitido que seja inserido no campo de texto da aplicação qualquer combinação de números ou letras. Caso o valor inserido seja **0** ou **1** e "***false***" ou "***true***", então interpretaremos esses dados como informações de valor lógico booleano, pois a representação binária de ***false*** é **0** enquanto a de ***true*** é **1**. Para os demais valores numéricos acima, vamos realizar uma operação matemática, vamos optar por elevar o número fornecido ao quadrado. Ademais, para combinações envolvendo letras e números, o aplicativo reconhecerá a entrada como uma *string*. Para reconhecer os valores booleanos vamos estabelecer um procedimento que retorna o resultado da expressão lógica que procura os valores especificados.



Esse procedimento é chamado dentro do bloco de lógica principal da aplicação, e permite uma avaliação para cada caso possível dos valores aceitos como booleano:



Para os demais casos, a sequência de "**senão, se**" garante que outras situações sejam analisadas. Como já sabemos que o texto inserido não é **0**, **1**, "***false***" ou "***true***", que são números e *strings* que estamos considerando como booleanos, então basta verificarmos se temos um número ou *string* no campo de texto. Para os números, o valor inserido será multiplicado por ele mesmo e, logo em seguida, o valor resultante será exibido como "**Número (x²): 49**" (Exemplo para entrada do programa igual a **7**), já no caso de *strings* o texto será exibido conforme informado com o seu tipo declarado como *String*. Se a aplicação receber algum símbolo que não siga nenhum dos padrões acima, então exibe ao final "**Argumento inválido**".

ESTRUTURAS DE DADOS

Na programação temos o conceito de estruturas de dados, estas que são tipos de dados compostos criados para melhor organizar e processar dados de uma forma prática

e otimizada nas nossas aplicações, sendo importante destacar que cada uma têm características bem definidas, operações com dados que armazenam e níveis de complexidade diferentes na sua implementação.

Algumas estruturas de dados são disponibilizadas por padrão em muitas linguagens de programação, você pode já até ter ouvido falar de algumas. Podemos trazer à tona Listas, Dicionários, Árvores, Grafos, Filas e Pilhas como exemplos, bem como, também há como termos estruturas de dados criadas pelo programador para representar uma necessidade específica.

Exemplificando: Vamos supor que uma aplicação precisa marcar a chegada e saída de empregados de uma empresa multinacional, neste caso, o código foi feito de forma que todos os dados serão armazenados em uma única alocação de variável, digamos que foi escolhido utilizar um arranjo. Teremos então um cenário onde haveria um esforço muito grande para otimizar a procura e inserção de informações dentro do arranjo em questão, além disso, seria mais complexo realizar qualquer tipo de operação com os dados associados ao monitoramento de chegada ou saída dos funcionários.

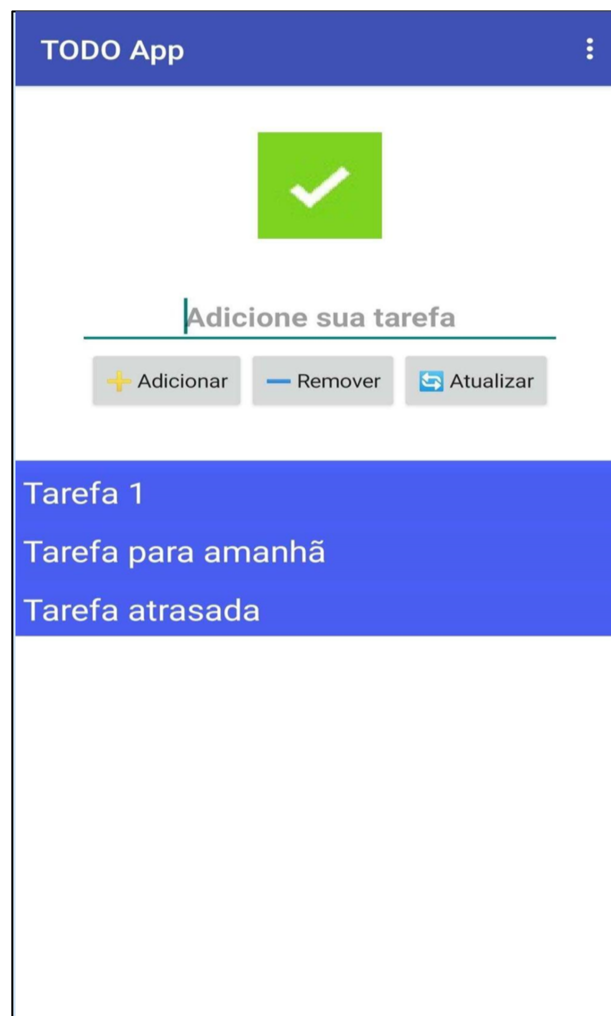
No entanto, se tivesse sido aplicada uma estrutura de dados apropriada, como uma Lista ou, até mesmo, um Dicionário, então se tornaria muito mais fácil e rápido buscar por um funcionário específico ou, quem sabe, calcular se cumpriu o expediente. Ou seja, o uso próprio dessas estruturas tende a resultar em um programa que executa mais rápido e que pode lidar com grandes quantidades de dados de forma mais fácil e, também, de forma mais organizada.

Com esse contexto em mente, compreendemos então que as estruturas de dados são essenciais no desenvolvimento de aplicações que têm como requisito eficiência, organização sistemática e facilidade de manutenção, pois estas são feitas com o intuito de servir como soluções práticas para diversos problemas comuns enfrentados na programação.

Esse é um tema com grande profundidade, mas, no intuito atual, essa introdução já é suficiente. Iremos então aplicar esse conhecimento:

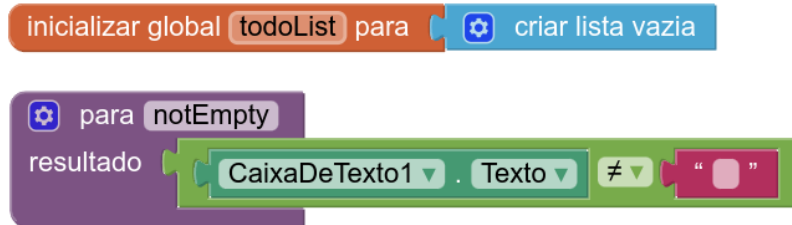
VAMOS PRATICAR

Nesta prática, vamos explorar uma aplicação que faz uso ideal de uma estrutura de dados mencionada acima, a Lista. O aplicativo proposto é uma simples agenda de tarefas que, intuitivamente, vai listar as tarefas que forem cadastradas pelo usuário, sendo assim, a correlação fica clara. Buscaremos utilizar a estrutura de dados da Lista para armazenar *strings* das nossas tarefas, bem como, podemos utilizá-la para remover e atualizar tarefas sem complexidade adicional.

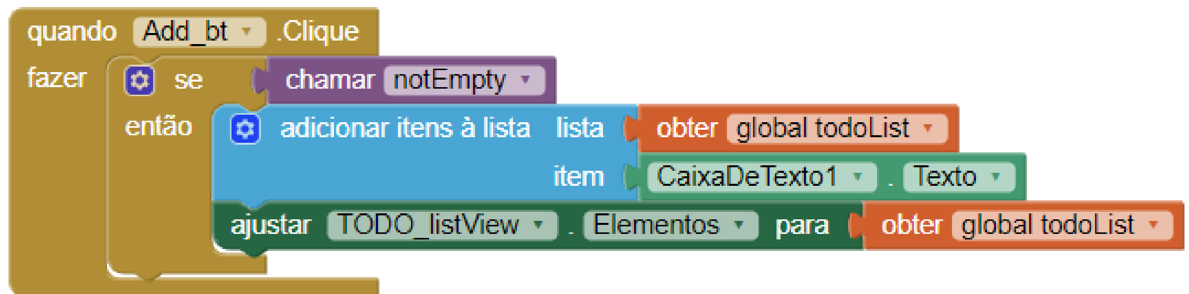


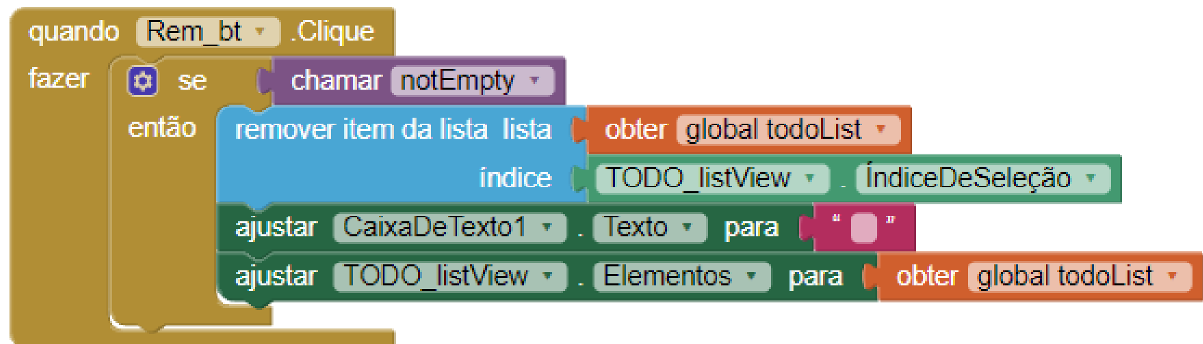
Primeiramente, é importante destacar que há dois elementos distintos que podem causar certa confusão em uso na aplicação: A Lista (a estrutura de dados em si) e o componente *ListView* que é o elemento gráfico que lê a lista armazenada e a exibe.

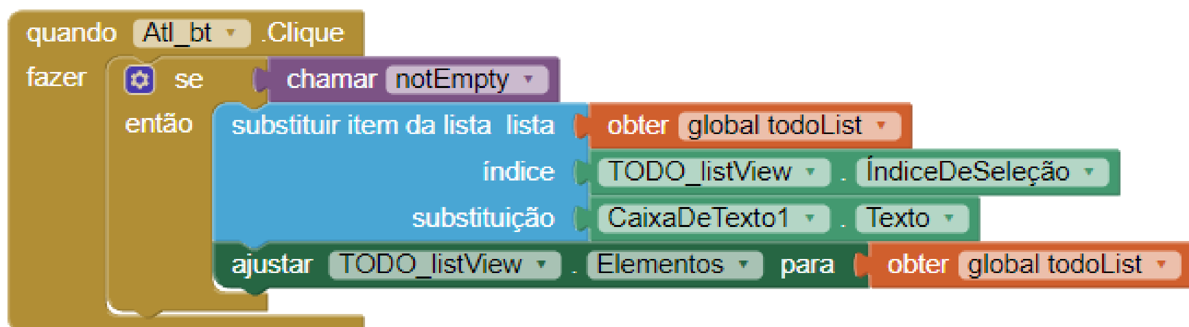
Nossos primeiros blocos são voltados justamente para a criação da variável que conterá nossa lista e do procedimento que checa se a caixa de texto não está vazia:



Em seguida os eventos dos botões devem ser configurados com a lógica das suas ações respectivas:

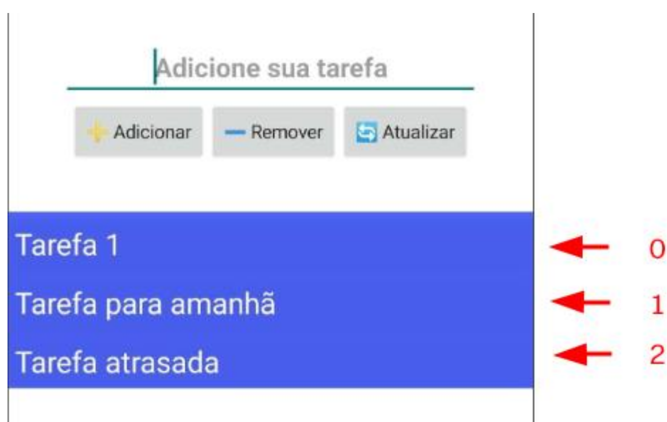






Embora estruturalmente semelhantes, esses eventos têm objetivos distintos. Os fatores comuns são a confirmação que a caixa de texto não está vazia pelo procedimento *notEmpty* e o ajuste da *ListView* (nosso elemento visual) para os dados mais atuais da *global todoList* (variável que contém a estrutura de dados), mas fora isso, utilizaremos métodos específicos de interação com Listas para alcançar os objetivos do aplicativo.

Relembrando então, temos um botão de "**Adicionar**" (***Add_bt***), um botão de "**Remover**" (***Rem_bt***) e um outro para "**Atualizar**" tarefas (***Atl_bt***) e, respectivamente, as funções utilizadas são adicionar item, remover item e substituir item. O método de adição é simples, basta ler o conteúdo da caixa de texto e adicionar à lista esse conteúdo, porém remover e substituir envolve o *index*, tal ilustrado na imagem a seguir:



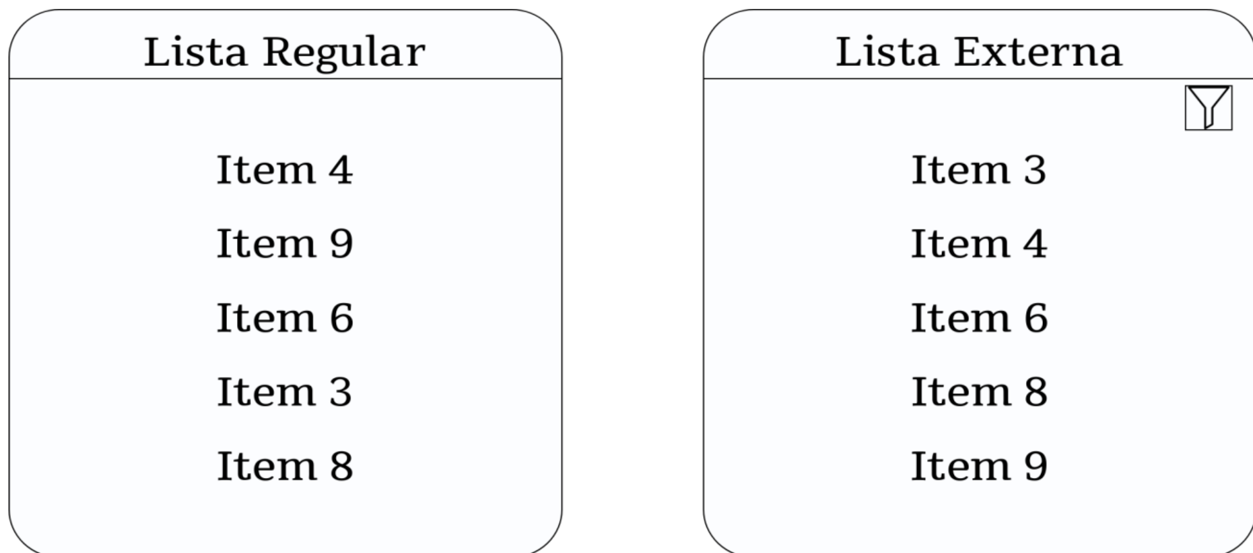
O *index* será somente o número correspondente a posição da Lista em que certa tarefa se encontra e a *ListView* oferece esse *index* como uma das suas propriedades quando uma das tarefas é clicada, **servindo como ponteiro** para onde as funções de remover e substituir devem agir. Com isso em mente, a lógica fica clara.

Enquanto o botão de "**Remover**" vai apagar o registro da tarefa na variável *todoList* no *index* informado, a função de substituir irá atualizar a tarefa com o que está atualmente no campo de texto da aplicação.

Para um toque extra de interatividade no aplicativo, criaremos outro evento que copia uma tarefa para a caixa de texto. Isso para demonstrar que a ação do usuário surtiu efeito.

BIBLIOTECAS

Bibliotecas, na programação, são conjuntos de recursos que podem ser utilizados em uma aplicação para facilitar o processo da programação, disponibilizando funcionalidades prontas para uso no contexto de um programa. Existem bibliotecas padrões que são disponibilizadas junto à tecnologia escolhida para desenvolver, bem como, existem bibliotecas criadas por outros programadores que disponibilizam certas utilidades mais específicas, geralmente com um grau de complexidade maior do que o disponível por padrão. Levantando um exemplo hipotético: por padrão é possível utilizar uma biblioteca que gerencie e exiba uma lista de dados, porém uma biblioteca externa pode oferecer uma lista com filtros e auto-organização. Veja a representação abaixo:



As bibliotecas promovem certas vantagens e desvantagens, vamos explorá-las!

VANTAGENS

- Tempo de desenvolvimento poupado;
- Bibliotecas padrão sob constante revisão e manutenção;
- Maior eficiência do código;
- Feitas com reutilização em mente;
- Abstração de partes complexas de forma simples.

DESVANTAGENS

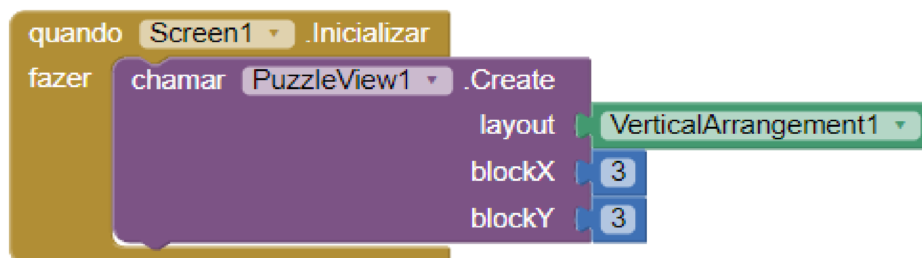
- Excesso de dependências carregadas em um programa;
- Potenciais conflitos entre bibliotecas;
- A alteração do comportamento e performance das bibliotecas é limitado;
- Bibliotecas externas podem ser descontinuadas e quebrar no futuro.

De forma abrangente, incluir bibliotecas em um projeto não é estritamente necessário, é possível codificar as mesmas funções por conta própria e, então, possuir mais controle sobre o que foi criado, mas ao mesmo tempo isso requer mais esforço e tempo, além de aumentar o código que precisa ser gerenciado, coisa que a programação em blocos não facilita.

VAMOS PRATICAR

Neste exemplo inicial vamos utilizar uma biblioteca simples que permite recriar a lógica do jogo *mobile* “**2048**”, isso com o intuito de ilustrar a facilidade de uso das bibliotecas no desenvolvimento de aplicativos. Para isso, a biblioteca *PuzzleView*[5] foi utilizada a fim de obter total funcionalidade do aplicativo com um amplo suporte que abrange desde o *layout* da tela até a movimentação dos blocos no jogo.

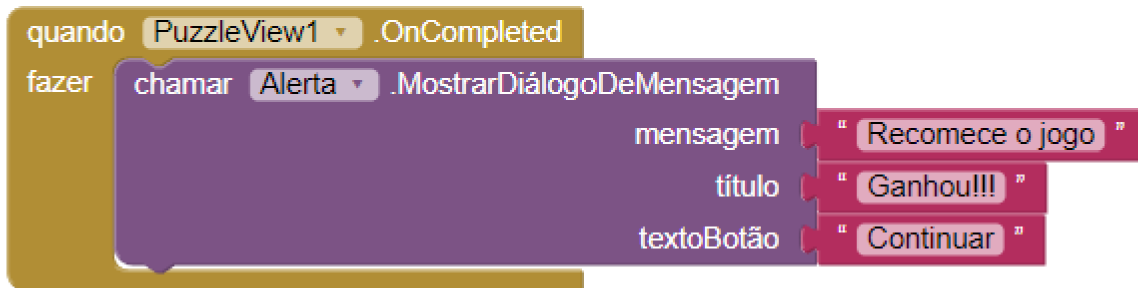
A princípio usamos um dos métodos pré-desenvolvidos da biblioteca para configurar o *layout* do nosso quebra-cabeça quando a tela inicializa, dada uma altura, largura e orientação de nossa aplicação.



Com isso configurado, podemos passar para as notificações mostradas a cada evento realizado. Na imagem abaixo podemos observar a notificação de vitória acionada

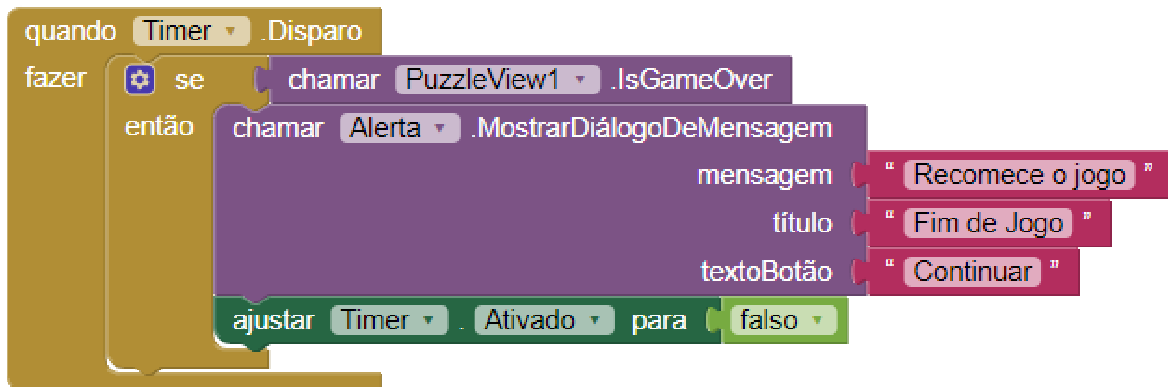
[5] Disponível em:<https://getaix.com/extension/PG>

ao concluir o jogo. Lembre-se de que o estado do jogo está sendo constantemente monitorado e verificado pelos métodos disponibilizados pela biblioteca, minimizando o esforço necessário no desenvolvimento do "**código**".



Da mesma forma que no evento de vitória, o evento *GameOver* também precisa ser monitorado, nesse caso, o "**Fim de Jogo**" é tratado com o auxílio do método *.isGameOver* da biblioteca, que retorna um booleano que informa se não há movimentos possíveis. Ao utilizar essa função em um bloco condicional é possível informar ao jogador que ele perdeu e que deve recomeçar.

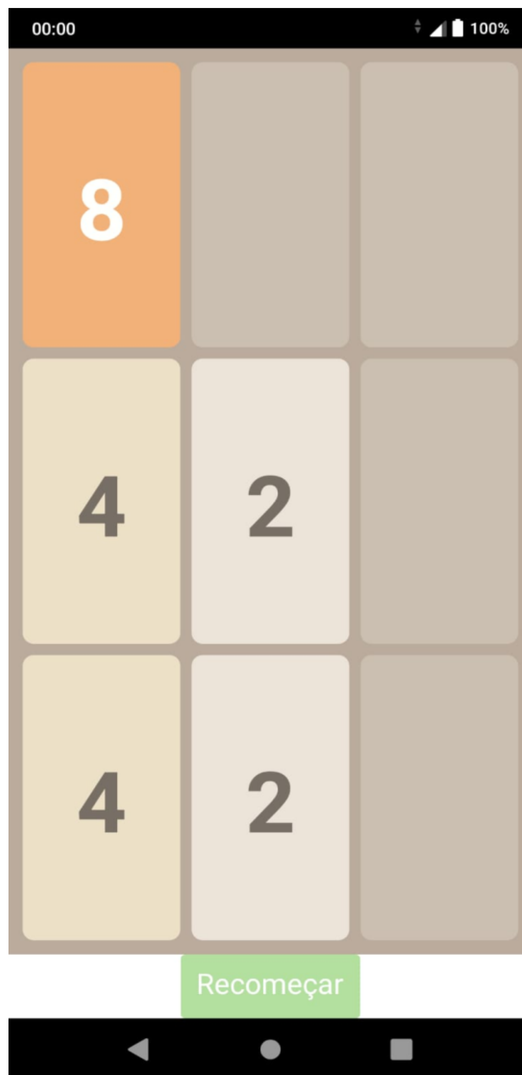
Também é importante notar que estamos usando o relógio interno do celular para acompanhar o "**Fim do Jogo**". É por meio dele que poderemos verificar o estado do jogo após cada jogada.



Para finalizar, um simples botão "**Reiniciar**" é usado com o intuito de redefinir o estado de todo o jogo quando o usuário desejar, esse que é implementado pelo método *.RestartGame*, também próprio da biblioteca *PuzzleView*.

```
quando Recomeça .Clique
fazer chamar PuzzleView1 .RestartGame
      ajustar Timer . Ativado para verdadeiro
```

Com esses poucos blocos de lógica, teremos um jogo totalmente funcional como podemos ver na imagem abaixo. Incentivamos você a replicar a aplicação e ajustá-la como achar melhor, adicionando funcionalidades, dentro é claro, das limitações da biblioteca.



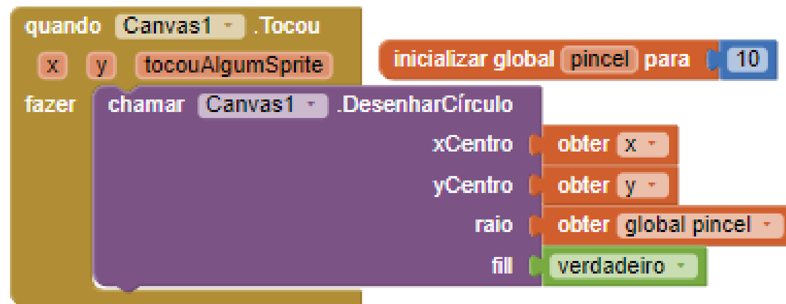
MAIS EXEMPLOS

Explorando um pouco mais o assunto, nesse novo exemplo, utilizamos duas bibliotecas nativas do *App Inventor*, a **Câmera** e o ***Canvas***, ambas entrelaçadas a fim de concluir o aplicativo, com o objetivo de possibilitar que o usuário tire fotos e desenhe nelas e no final que salve essa imagem editada em seu aparelho.

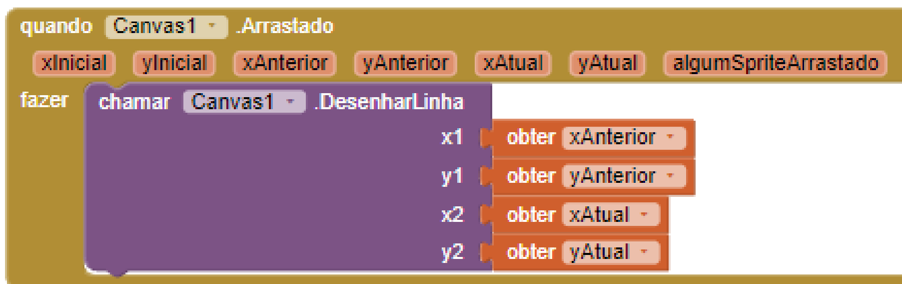
Dessa vez, no entanto, a lógica dos blocos pode ficar um pouco mais complexa, ao menos quando comparada com a biblioteca *PuzzleView*, pois estamos lidando com duas bibliotecas sendo utilizadas ao mesmo tempo, mesmo assim os métodos pré-definidos farão a maior parte do trabalho como veremos a seguir.

Começando na aba de *design* do *App Inventor*, podemos ver como essa aplicação é predisposta com a área ocupada pelo *Canvas*, seguida dos controles deslizantes de cores e dos botões para tirar fotos, salvá-las, bem como o que permite limpar a tela.

Pensando nisso, ao dar uma olhada nos blocos, podemos separar as bibliotecas e o que elas fazem. Então, primeiro vamos configurar o *Canvas* para podermos desenhar nele, especificando as ações de toque e deslize da tela para desenhar pontos ou linhas respectivamente.

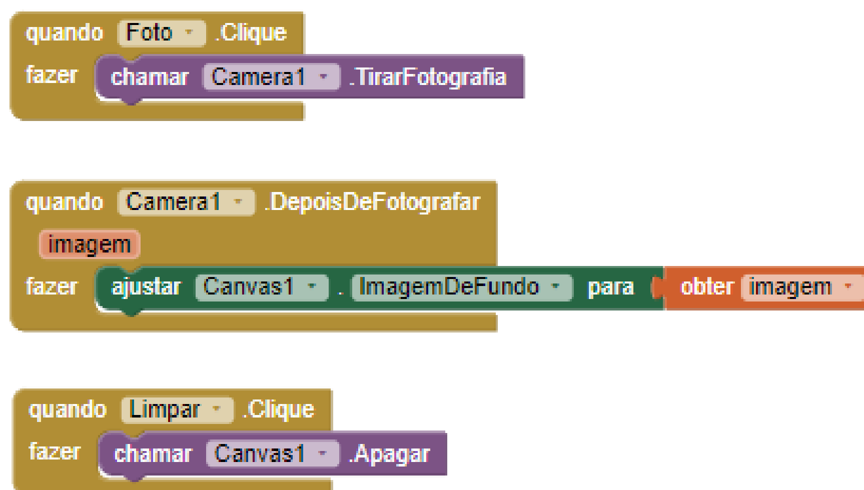


Com o suporte da biblioteca, podemos ter acesso aos pontos nos eixos x e y da nossa tela dentro dos blocos de eventos, simplificando a forma como lidamos com o desenvolvimento do *Canvas*.

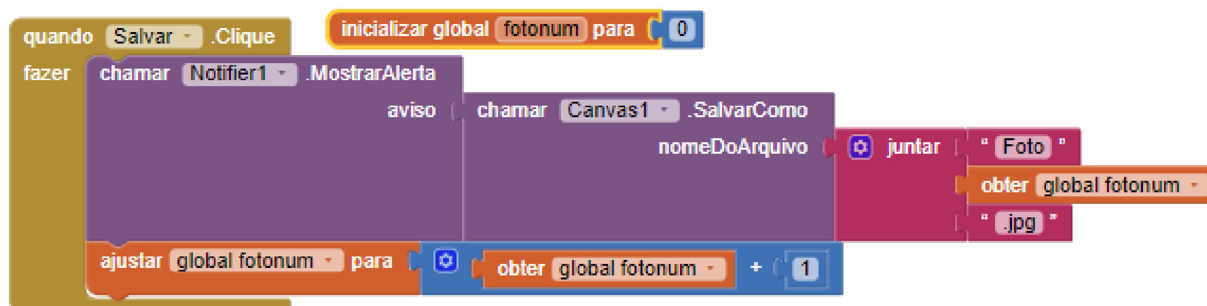


O seletor de cores também não deve ser ignorado, pois também está relacionado a tela, apesar de não ser nativo a biblioteca. Portanto, para que funcione como pretendido, configuramos 3 controles deslizantes com intervalo de **0** a **255** para simular uma paleta **RGB**.

E agora, para a segunda biblioteca, nossos blocos são bastante claros e diretos. A Câmera basicamente permite que o aplicativo tenha acesso à câmera do telefone para tirar uma foto e, a partir daí, podemos usar essa imagem capturada da maneira que quisermos. No nosso caso, definimos como imagem de fundo do *Canvas*, essa que pode ser descartada ao apagar o *Canvas* atual e, para isso mesmo, definimos o nosso último bloco que damos a funcionalidade de "**Limpar**", associada ao botão com esse nome.

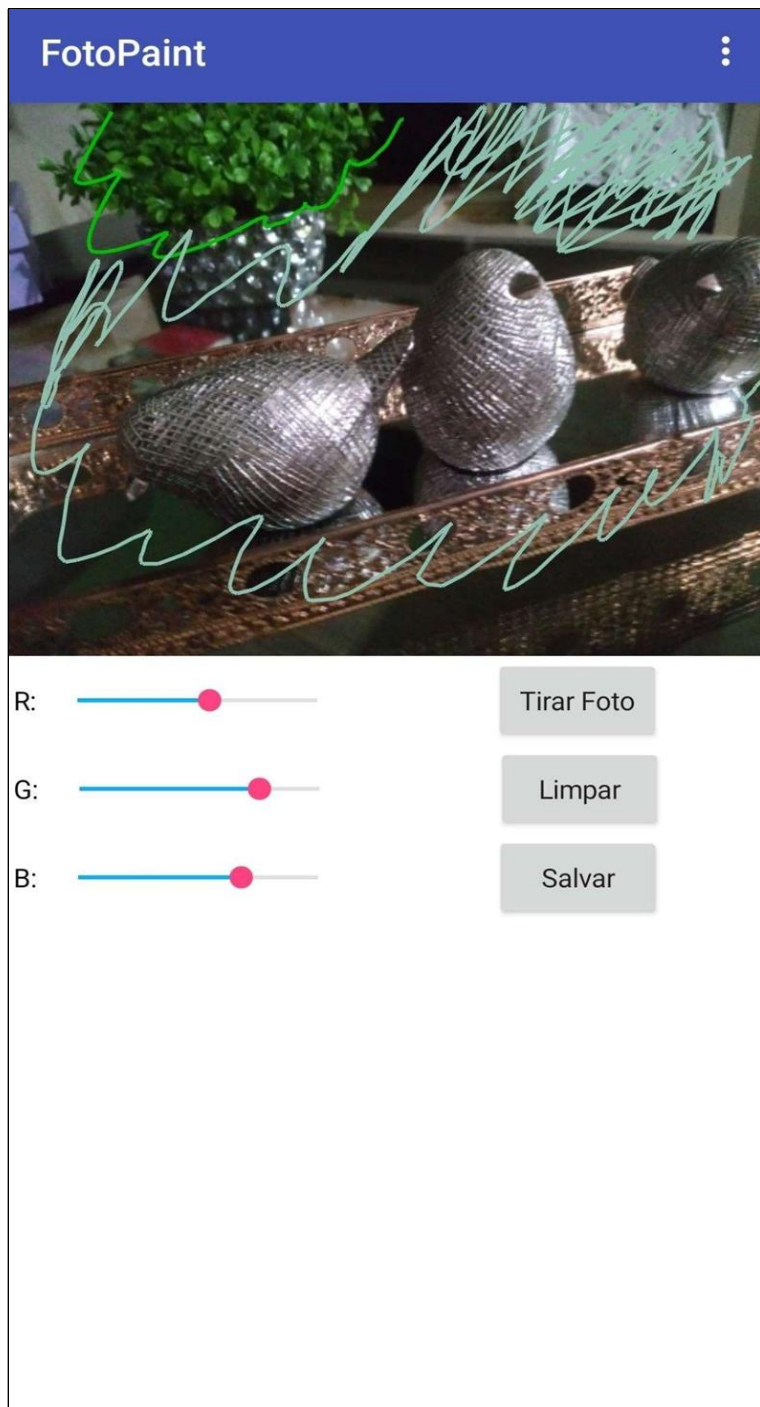


Para finalizar, precisamos poder salvar nossa obra de arte em nosso aparelho, sendo assim, usando o método **.SalvarComo** do *Canvas*, podemos fazer isso. Quanto ao nome do arquivo, podemos usar uma simples variável numérica crescente acrescentada no final do nome do arquivo, com a intenção de salvar cada arquivo com nome único, evitando arquivos conflitantes.



Com a aplicação em mãos temos algo como a imagem abaixo, uma tela onde fotos são inseridas após a captura e que permite que desenhos sejam feitos sob a tela.

Possuindo também o controle de cor do pincel feito pelos controles deslizantes, assim como os botões que permitem limpar os desenhos e salvar o conteúdo do *Canvas*.



PROJETOS

Ao desenvolver aplicações complexas, é essencial que o time ou indivíduo por trás de um projeto tenha pleno entendimento do que se procura alcançar com tal empreendimento, ou seja, a visão do projeto necessita estar bem delineada e comunicada entre todos os participantes, pois isso possibilita a escolha das opções que melhor se encaixam dentro do que precisa ser feito. Nesse processo é necessário abordar uma variedade dos tópicos já discutidos neste livro, por exemplo, a decisão de usar ou não bibliotecas, a identificação de quais estruturas de dados podem beneficiar a arquitetura da aplicação, quais melhores práticas serão utilizadas para reuso de código, dentre outras questões.

Para projetos de grande porte, a porção de planejamento tende a ser minuciosa em sua organização, ao menos no que diz respeito a projetos comerciais é evidente que considerações como orçamento, prazo e riscos devem ser considerados com cautela, porém pensando em menor escala, projetos pessoais também são importantes, estes permitem experimentação e autodesenvolvimento do desenvolvedor na sua capacidade de resolução de problemas, além de favorecer um ciclo de desenvolvimento menos regrado, livre do risco corporativo e impacto econômico de um projeto falho. Vejamos a seguir um exemplo de um projeto de cunho pessoal prototipado dentro do *App Inventor*.

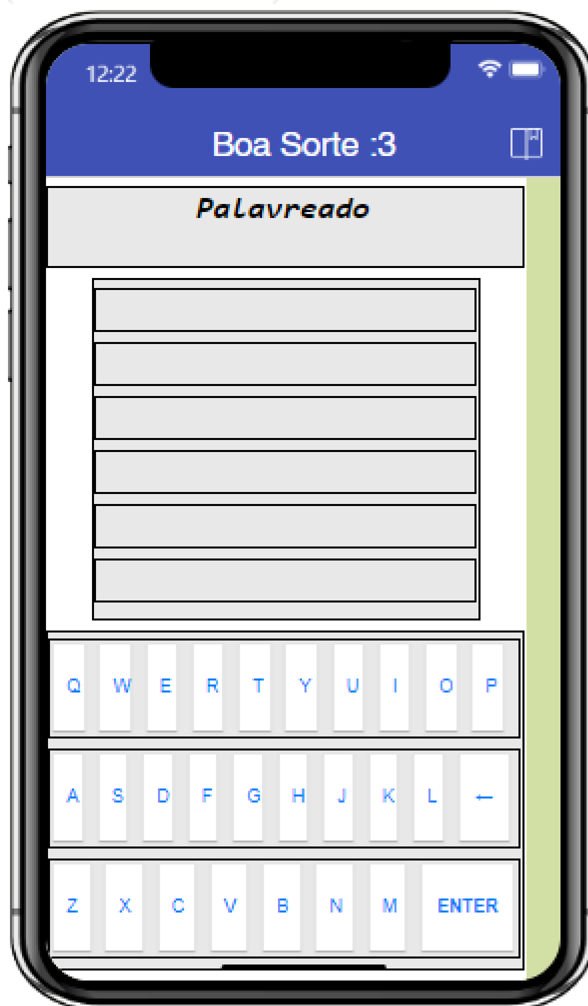
VAMOS PRATICAR

Com o intuito de reforçar os conceitos apresentados até o momento, para o exemplo final, os tópicos abordados foram combinados em uma só aplicação de maior complexidade. Nela, utilizamos listas, procedimentos, blocos de decisão e de repetição, variáveis com escopos, dicionários e eventos.

Visando elaborar um jogo de palavras semelhante ao popular *Wordle*[6], desenvolvemos a nossa versão, **o Palavreado**! Considerando a maior complexidade envolvida, para destrinchar todos os blocos, vamos quebrar o aplicativo em partes.

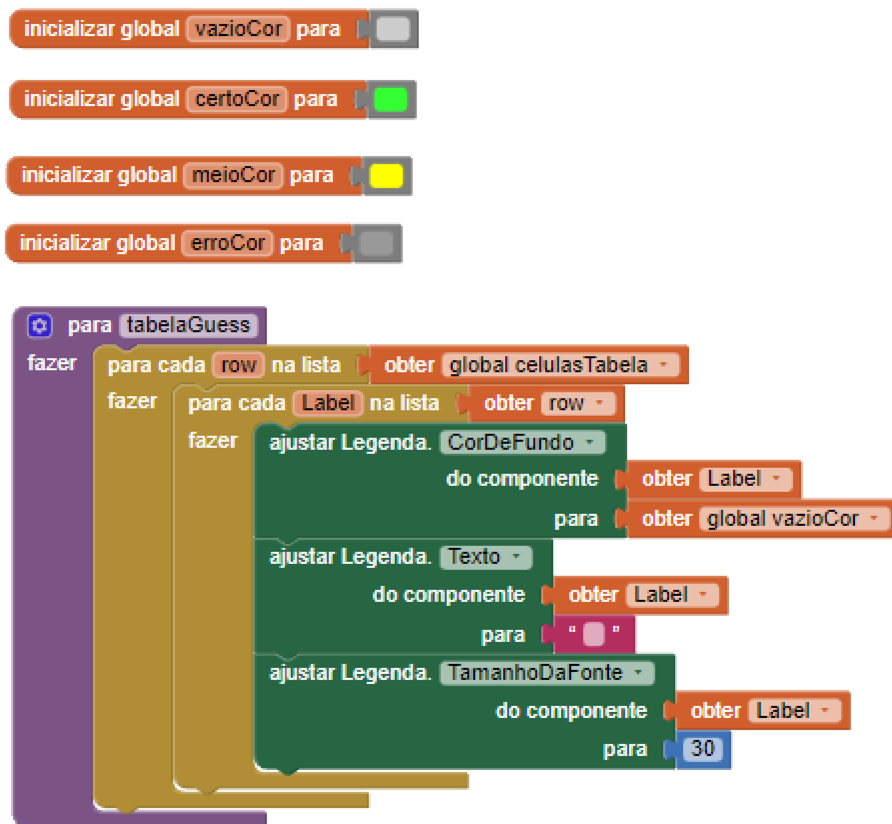
[6] Jogo original disponível em:<https://www.nytimes.com/games/wordle/index.html>

Observando a forma em que o *design* está disposto abaixo, nota-se que o *layout* possui duas funcionalidades importantes, sendo elas a tabela de palavras adivinhadas e o teclado embutido na aplicação.

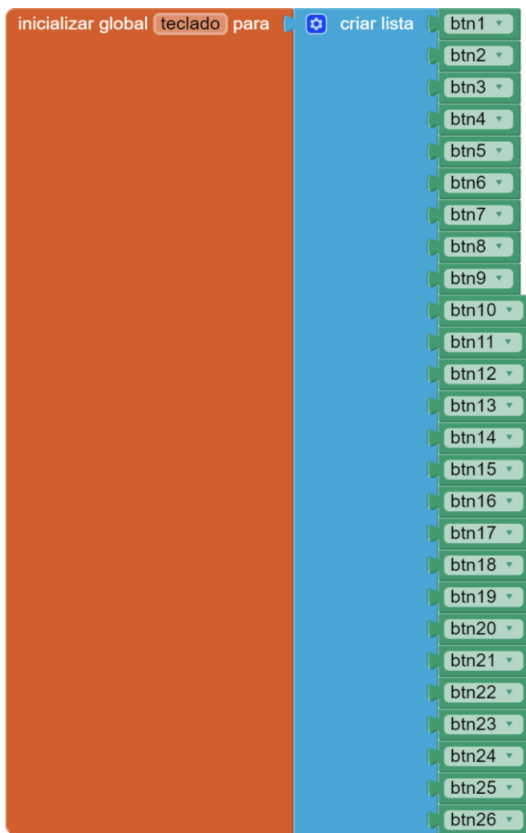


Como de costume nos jogos nesse estilo, conforme você envia seus palpites cada letra recebida recebe uma cor diferente dependendo da sua presença ou omissão na palavra a ser adivinhada, sendo verde a cor designada para uma letra no lugar correto, amarelo sendo a cor de uma letra que pertence a palavra que, contudo, não está no lugar correto e o cinza escuro representa uma letra não contida na palavra. Logo temos variáveis globais que representam os estados dessas possíveis letras e que serão utilizados durante o desenvolvimento.

Ainda no tópico das palavras adivinhadas, temos a configuração de toda tabela armazenada no procedimento *tabelaGuess* que tem o objetivo de retornar todas as células existentes para seu estado padrão, restabelecendo a cor, fonte e esvaziando-as de qualquer texto.



Na imagem abaixo podemos visualizar melhor a forma com que a tabela de palpites está disposta, na qual, cada palpite é composto por uma lista de *Labels* que serão preenchidas pelo jogador e, igualmente, vale observar que todos os palpites fazem parte uma lista única a fim de facilitar operações com ela.



Já para o teclado, uma simples lista de 26 botões representando as letras do alfabeto garante o necessário para sua configuração. Note que os botões como “***Enter***” e “***Delete***” não são inseridos nessa lista, já que eles possuem comportamentos próprios e serão tratados separadamente.

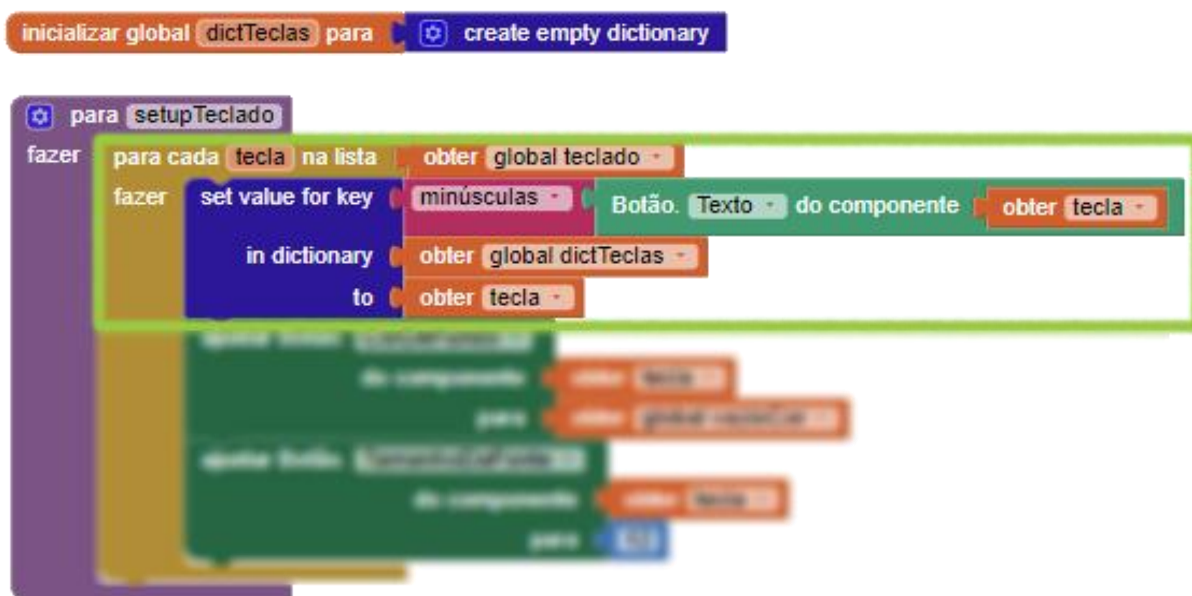
Para a configuração do teclado é utilizado um procedimento nomeado *setupTeclado*, esse que deve ser chamado toda vez que necessário reiniciar o teclado, já que ele retorna as teclas para seu estado padrão, restabelecendo a cor de fundo e tamanho da fonte padrão das teclas e, além disso, ele também cria um novo dicionário limpo para o teclado.

Mas por que utilizar um dicionário? E como ele é configurado? Veremos abaixo:

Voltando nosso foco para o dicionário das teclas, podemos perceber que os blocos que executam sua lógica estão em inglês e isso se deve ao fato de que no momento da elaboração desse conteúdo nem toda ferramenta do *App Inventor* foi traduzida para o português brasileiro.

Mas enfim, a resposta para a pergunta do porquê utilizar um dicionário é simples, o dicionário serve para atribuir um valor para cada botão/tecla do nosso teclado embutido, o que facilitará muito na recuperação das informações de cada letra futuramente. Assim não precisaremos buscar nas propriedades de cada botão o que ele representa, basta esclarecer que o botão de texto (propriedade do botão) “**a**” representa a letra “**a**” e assim por diante.

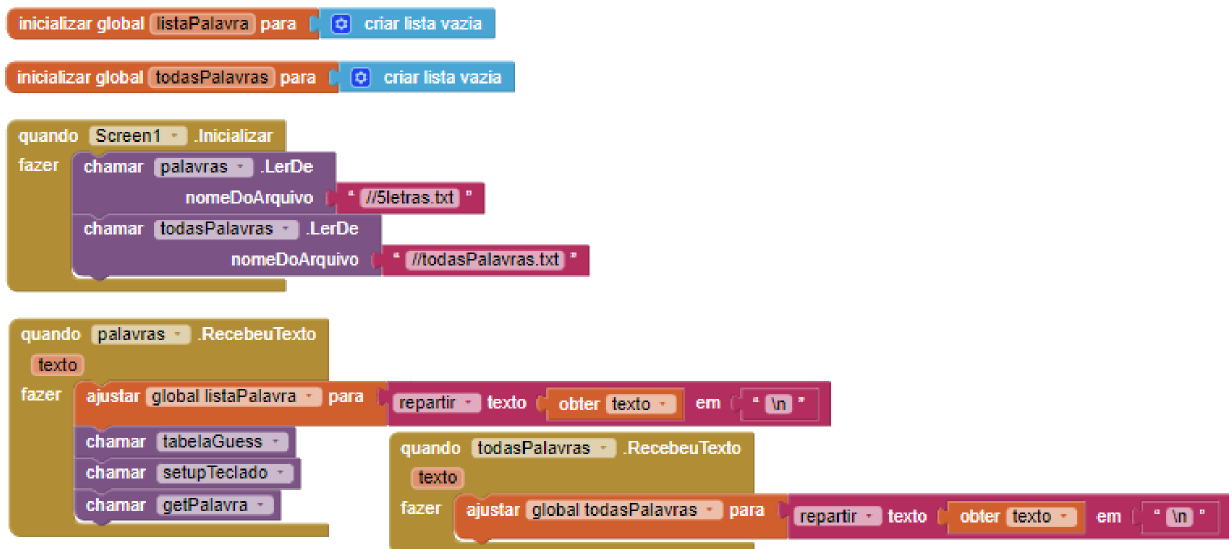
Sabendo disso fica mais fácil visualizar a composição do nosso dicionário, nele definimos nossa chave como o texto do botão, então declaramos o dicionário de referência e então definimos nosso valor como a letra que tal botão representa.



Agora vamos configurar a forma com que nossas palavras serão selecionadas e onde elas serão armazenadas. Para isso, foi utilizado um banco de palavras pré-selecionadas de 5 letras como “**vocabulário**” da aplicação, estas que serão lidas de um arquivo .txt e armazenadas no leitor de arquivos nativo da ferramenta que nomeamos de palavras e, também, há um segundo arquivo .txt com possíveis palavras aceitáveis,

porém que não serão selecionadas como a palavra da vez, mas serão comparadas com as respostas do usuário para garantir a validade da resposta enviada.

Dessa forma, quando o leitor de palavras receber o conteúdo do arquivo com sucesso, o texto contido será tratado, sendo repartido em toda quebra de linha, e armazenado em uma lista onde, futuramente, uma palavra será selecionada pelo método *getPalavra*. É válido ressaltar que sempre que o leitor processar texto o teclado será reiniciado ao seu estado padrão já que uma nova palavra será escolhida.



Antes de chegarmos no desenvolvimento dos eventos de botões, é necessário amarrar algumas pontas e preparar algumas variáveis.

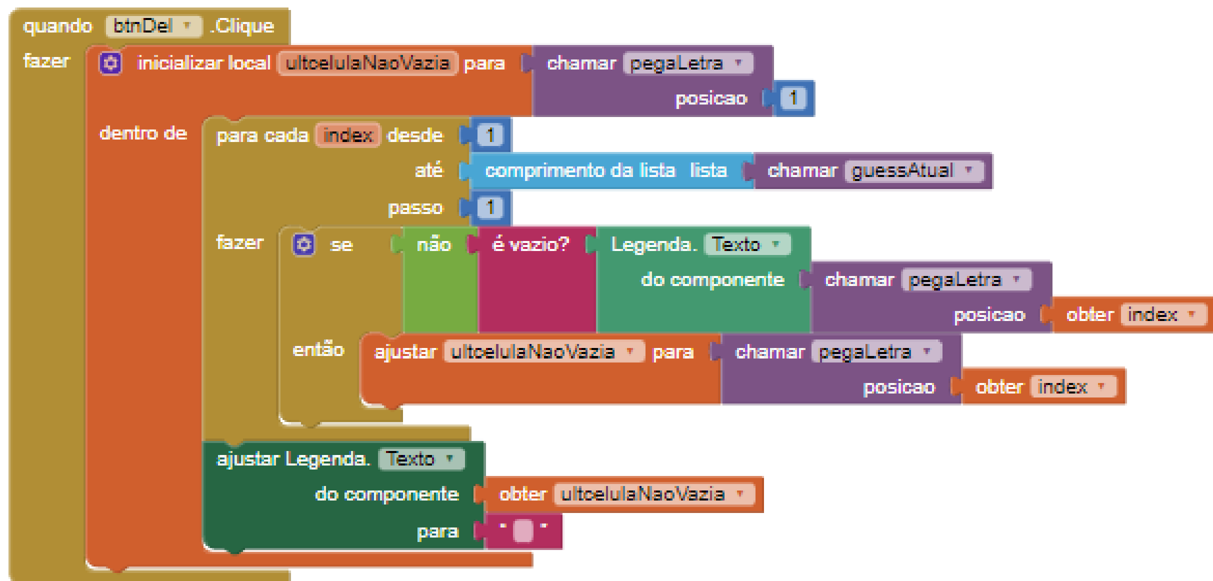
Primeiro, o método *getPalavra*, que foi previamente mencionado, é declarado e nele executaremos uma escolha aleatória de uma das palavras da lista global, para assim conseguirmos a palavra que será usada no jogo, e então armazenaremos ela na variável *palavraAtual*.

Acaba sendo necessário manter a contagem do atual palpite e a progressão dos testes nas letras que compõem um palpite, por isso foram declarados a variável *celulaAtual*, tal que aponta, em primeiro momento, para o primeiro palpite e o procedimento *guessAtual* que, a partir dessa primeira letra, seleciona a primeira linha da

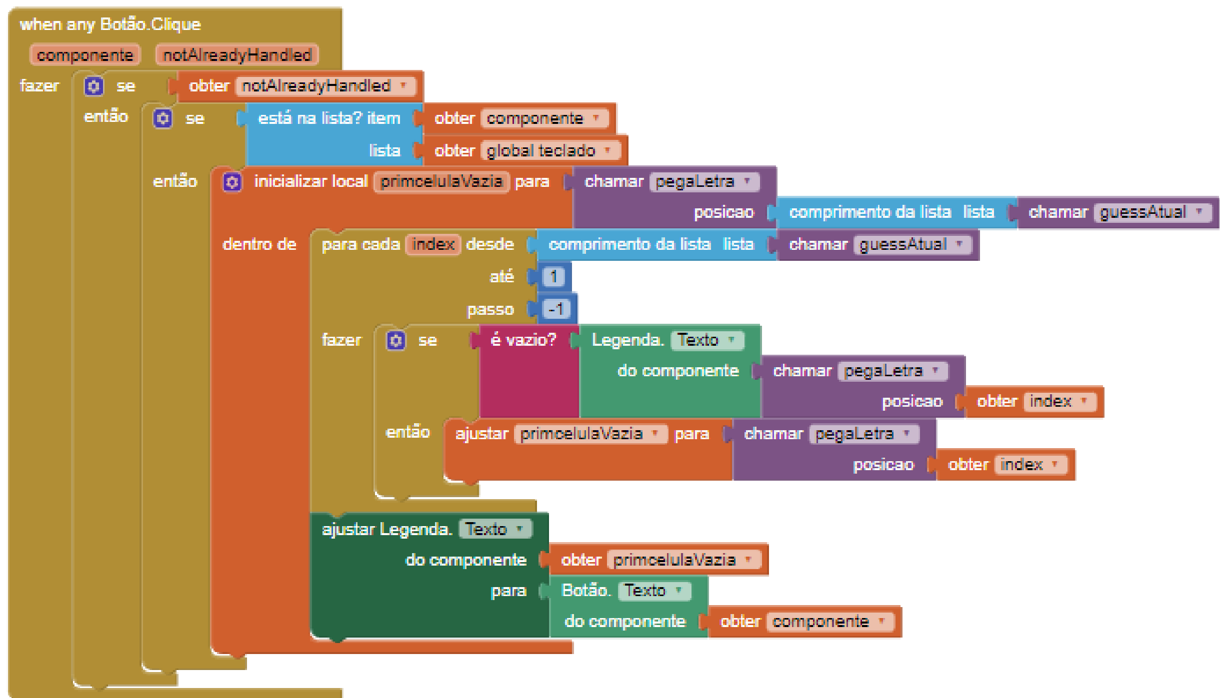
tabela de palpites. Já o procedimento *pegaLetra*, busca, a partir de um índice recebido como parâmetro, ou seja, a letra referente a tal posição dentro da *guessAtual*.

Dando início ao desenvolvimento dos botões, destrinchamos a princípio o botão delete, por ser os mais simples dos três. Neste caso, o principal ponto da lógica gira em torno da última célula não vazia, ou seja, qual a última letra que foi inserida no palpite, pois com ela saberemos qual célula deverá ser apagada.

Para esse objetivo declaramos uma variável com nome *ultcelulaNaoVazia* que é atualizada dentro de um bloco de repetição que parte da primeira até a última das células existentes em um palpite, isso enquanto checa em cada parada se a célula está preenchida utilizando os blocos "**não**" e "**é vazio?**" em conjunto. Só então, ao final do nosso *loop*, que podemos deletar a última célula não vazia encontrada.

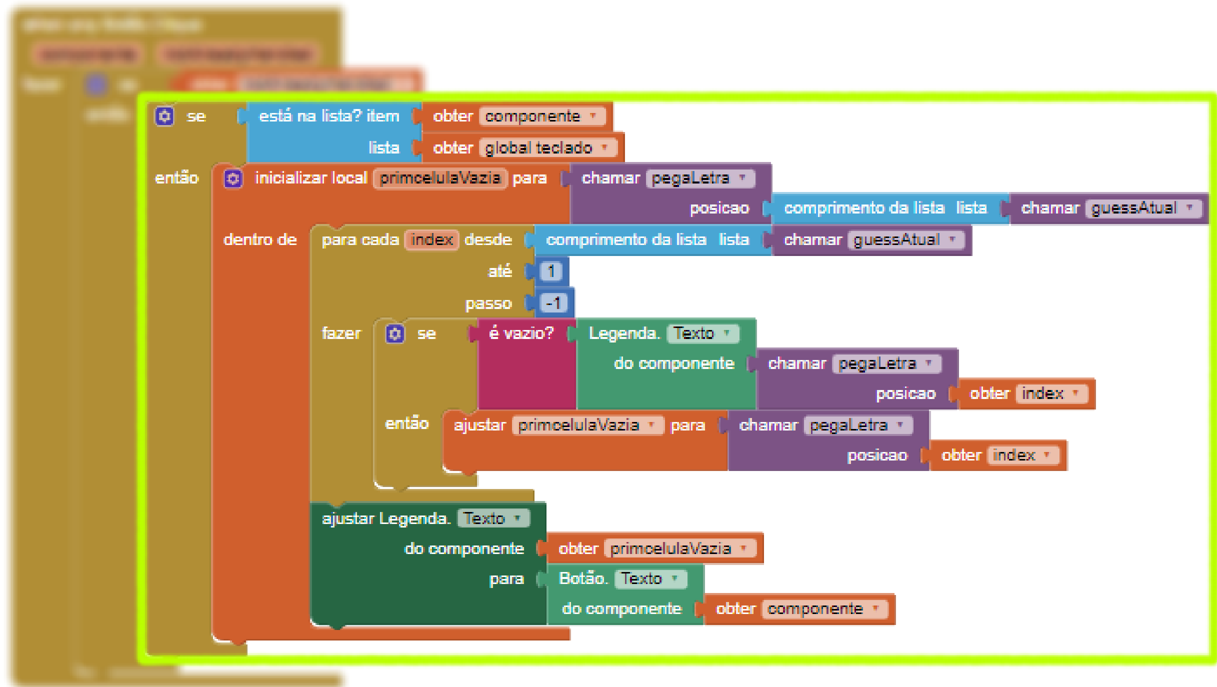


Já para os botões do teclado, utilizamos o bloco de evento que é ativado quando qualquer botão é clicado e, por esse motivo, temos que tomar precauções para evitar um mal funcionamento do teclado, logo a lógica a ser feita deve protegida por duas condições, sendo elas, em ordem, o *notAlreadyHandled* nativo do evento de clique, que será explicado mais a frente, e a condicional que envolve a existência do componente que foi clicado na lista de teclas do teclado, que basicamente checa se o botão clicado foi realmente um botão de letra.



Enquanto isso, o *notAlreadyHandled*, que pode ser traduzido para "**aindaNãoTratado**", funciona como uma barreira que impede tanto os botões específicos "***Enter***" e "***Delete***" de passarem, já que seus eventos devem ser tratados pelos seus blocos próprios, como também evita qualquer possível problema onde teclas ativam mais de uma vez com apenas um clique.

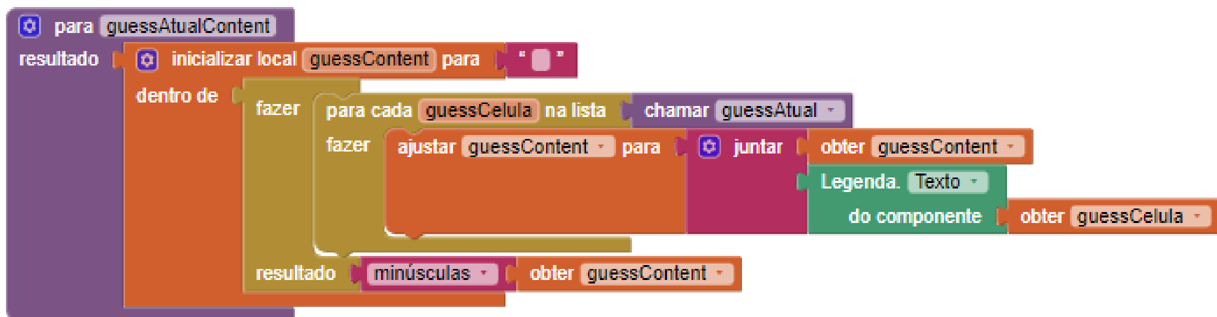
Ao passar por ambas as condicionais descritas anteriormente, chegamos na lógica dos botões do teclado. Em sua implementação buscaremos a primeira célula vazia, assim saberemos onde inserir a letra.



Observando os blocos, vemos que ao inicializar a variável local *primcelulaVazia* atribuímos a ela a última célula do palpite atual e então verificamos o resto das células através de um bloco de repetição que inicia no final do palpite atual e vai até a primeira letra em passo -1, ou seja, em ordem reversa. Dentro do *loop* é checado, com o auxílio de um bloco condicional, se a célula naquela posição está vazia e, se for esse o caso, então o valor da variável *primcelulaVazia* é atualizado. Ao final do *loop* podemos utilizar a posição vazia encontrada para inserir a letra recebida que foi digitada no teclado do aplicativo.

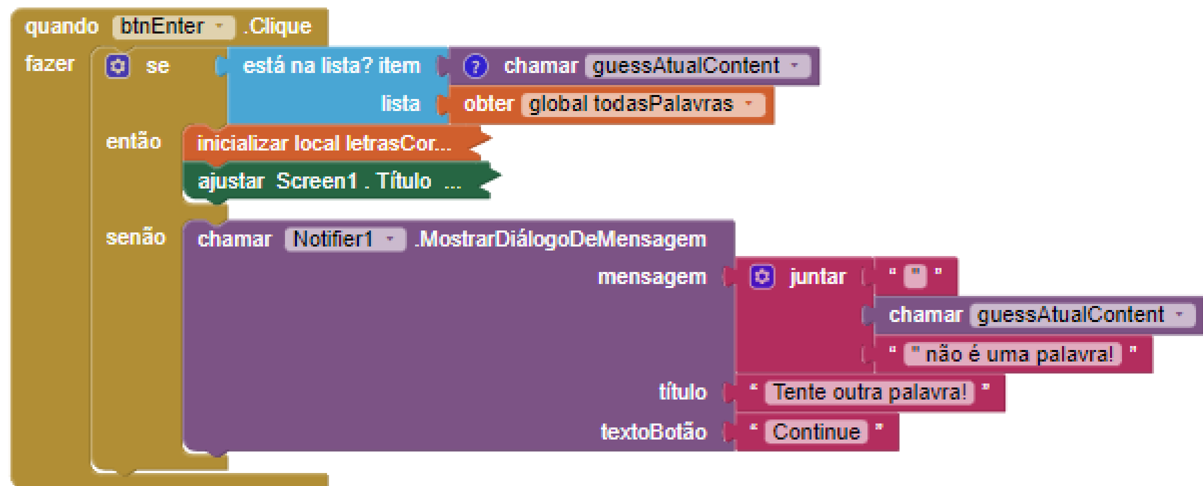
Partindo para outra parte da lógica, o procedimento *guessAtualContent* (ilustrado na imagem abaixo) auxilia no tratamento da palavra que o jogador confirmou como seu palpite e, logo mais, esse procedimento será utilizado no bloco de evento do botão "***Enter***" para comparar a palavra atual com a palavra que deve ser descoberta nesta rodada.

Para tal intuito, o bloco lógico busca cada letra que foi inserida na célula do palpite atual e, logo em seguida, junta essas letras em uma só palavra armazenada na variável *guessContent.*



Já que mencionamos o botão "***Enter***", vamos explorar o seu papel. Seu evento é responsável por toda verificação de acertos e erros, além de manter a contagem das tentativas e garantir que o jogo transcorra como esperado. Em função disso, a complexidade do bloco se torna maior que a de todos os outros e por esse motivo ele será quebrado em algumas partes para melhor entendimento.

Para começo de conversa, precisamos entender a condicional que controla a validade da resposta enviada pelo usuário! Essencialmente, o bloco checa a existência da palavra que o jogador tentou confirmar (via procedimento *guessAtualContent*) dentro da lista contendo todas as possíveis palavras previstas como respostas válidas e, ao passar por essa condicional, então a palavra é tratada como uma entrada válida, caso contrário, uma notificação será exibida ao jogador informando que o palpite enviado não é uma palavra.

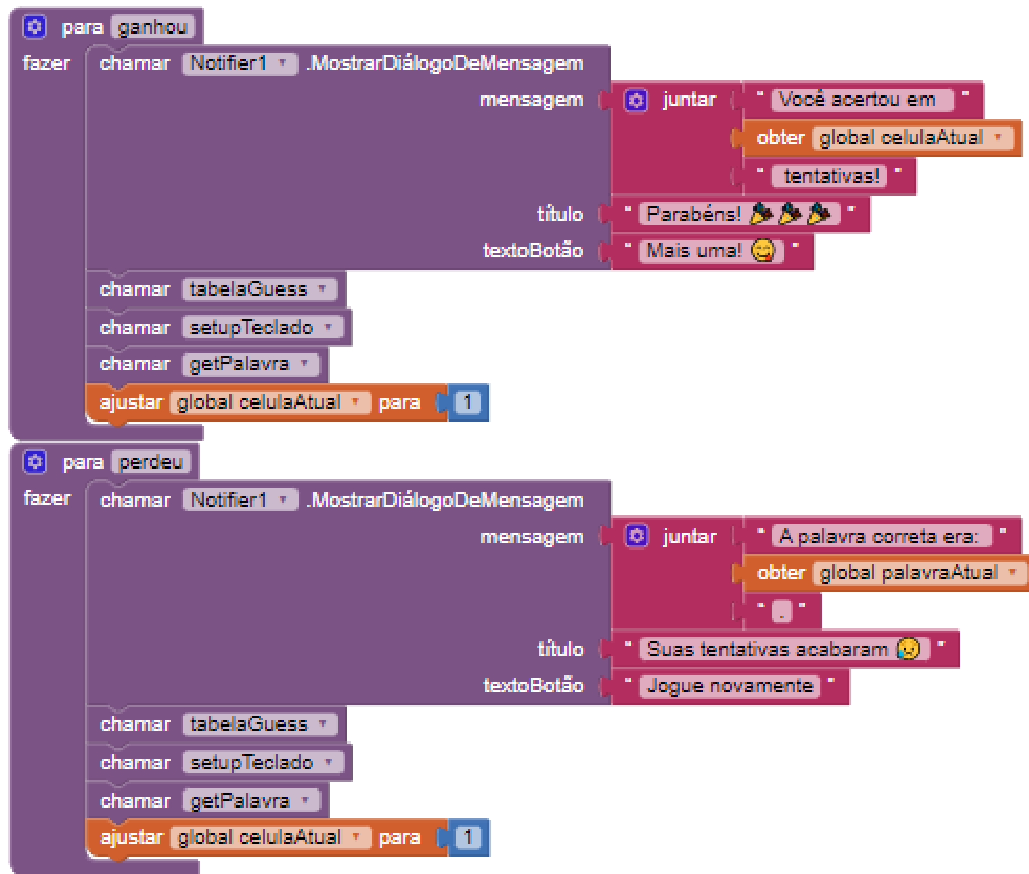


Partindo para a lógica de verificação do botão "***Enter***", iremos, a princípio, avançar pelas células do palpite atual comparando cada letra presente com as letras existentes na palavra sorteada, sempre considerando sua posição.

Durante o processo de comparação, os dados das letras da célula atual e da cor que modificamos essa célula, que por padrão será a cor cinza de erro, são processados em variáveis locais e ajustados na interface da aplicação, isso por meio de uma condicional que verifica se a letra em questão existe na palavra na mesma posição "**n**" (correspondente ao número do *loop*) e, quando a condicional se prova verdadeira, modificamos a cor daquela célula para verde e acrescentarmos na contagem de *letrasCorretas*. Caso contrário, faremos a confirmação da existência da letra naquela palavra e então modificamos sua cor para amarelo e, por fim, se a letra não estiver contida na palavra sua cor será alterada para o cinza padrão presente na variável local cor.

Após o desdobramento do *loop* é necessário checar o estado do jogo a fim de prosseguir com a partida, seja passando para o próximo palpite ao incrementar a <u>*celulaAtual*</u> ou, se for o caso, informar ao usuário a notificação de sua vitória ou derrota.

O pedaço final do quebra-cabeça são as notificações em si, que não só indicam ao usuário o resultado do jogo, como também chamam os procedimentos necessários para reiniciar o estado geral do **Palavreado**, possibilitando começar uma nova partida.



PARA FINALIZAR

RECAPITULANDO

Ao longo da leitura cobrirmos diversos tópicos introdutórios dentro do contexto da programação, de assuntos básicos, como métodos de controle de fluxo, até certos assuntos intermediários, bem como, quando exploramos as estruturas de dados e bibliotecas, isso com o intuito de introduzir esses conceitos da forma mais acessível via construção em blocos facilitada pela plataforma do *App Inventor*. Além da base teórica, as práticas ao longo deste livro buscaram promover um entendimento dos elementos que compõem aplicações do dia a dia, tais como caixas de texto, visualizadores de listas, imagens, botões, dentre outros exemplos. Com isso, o maior ensinamento que esperamos que tenha sido transmitido é o pensar sistemático por trás do desenvolvimento de aplicações, incluindo certas melhores práticas e o "*know-how"* necessário para alcançar objetivos propostos.

PRÓXIMOS PASSOS

Se você chegou até aqui, decerto que seu interesse pela programação não acaba junto com este material, portanto o que fazer a seguir é manter esse entusiasmo, continuar praticando a programação em blocos e, quando você estiver pronto/a/e, tomar o próximo passo ao escolher uma linguagem de programação que te agrade, o *App Inventor* é uma ótimo opção para ganhar dimensão de como aplicações são construídos hoje em dia, mas fora do escopo de prototipagem existem limitações que dificultam o gerenciamento de aplicações de média ou larga escala, por isso mesmo, indicamos o estudo além do que foi ensinado aqui.

PALAVRAS FINAIS

Agradecemos por ler nosso livro. Acreditamos que a forma mais recompensadora de aprender é quando podemos ver o resultado do que empenhamos em prática, por isso mesmo desejamos que o material lhe tenha ajudado na sua jornada para o domínio da lógica de programação.

Sobre os Autores

Gabriel dos Santos Lima

Estudante do Bacharelado em Ciência da Computação no Instituto Federal de Sergipe Campus Itabaiana e entusiasta da área da engenharia de software!

gabriel.academico@gmx.com.br

Iago Barretto Soares

Estudante do Bacharelado em Ciência da Computação no Instituto Federal de Sergipe Campus Itabaiana, com foco no campo da inteligência artificial.

iago.barretto1@hotmail.com

Rone Clay Oliveira Andrade

Estudante do Bacharelado em Ciência da Computação no Instituto Federal de Sergipe Campus Itabaiana, com interesse na área de desenvolvimento de software.

ronec1r550@gmail.com

José Aprígio Carneiro Neto

Pós-doutor em Ciência da Computação pela Universidade Federal de Sergipe (UFS). Doutor em Ciência da Propriedade Intelectual pela Universidade Federal de Sergipe (UFS) e Professor efetivo do Instituto Federal de Sergipe (IFS) na área de Ciência da Computação.

jose.neto@ifs.edu.br

APP INVENTOR

Lógica de Programação e Design