

Crie um aplicativo iOS com SwiftUI

O código-fonte deste guia pode ser encontrado [no GitHub](#)

Neste tutorial, você usará Swift e SwiftUI para criar um pequeno aplicativo para recomendar novas atividades divertidas aos usuários. Ao longo do caminho, você conhecerá vários componentes básicos de um aplicativo SwiftUI, incluindo texto, imagens, botões, formas, pilhas e estado do programa.

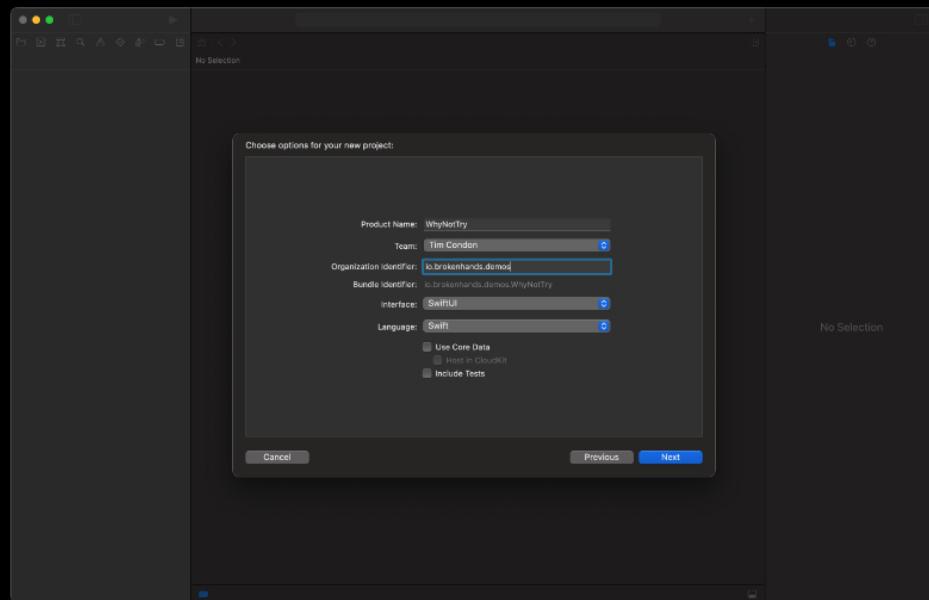
Para começar, você precisará [baixar o Xcode na Mac App Store](#). É gratuito e vem com Swift e todas as outras ferramentas que você precisa para seguir este tutorial.

Vá em frente e inicie o Xcode assim que estiver instalado e selecione Criar um novo projeto Xcode. Selecione a guia iOS na parte superior, selecione o modelo de aplicativo e pressione Avançar.

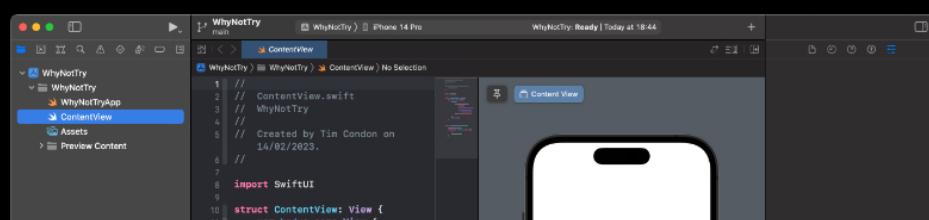
Dica: embora tenhamos como alvo o iOS 16, nosso código também funcionará muito bem no macOS Ventura e além.

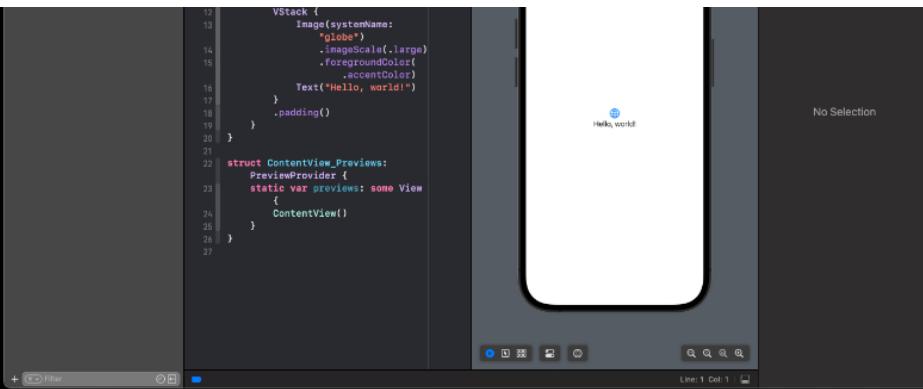
Ao fazer um novo projeto, o Xcode solicitará algumas informações:

- Para Nome do produto, digite "WhyNotTry".
- Para Identificador da organização, você pode inserir com.example. Em aplicativos reais, normalmente inseriríamos nosso próprio nome de domínio aqui, por exemplo, org.swift.
- Para Interface, certifique-se de que SwiftUI esteja selecionado.
- Você pode desmarcar as caixas Core Data e Incluir testes; não os usaremos aqui.



Ao pressionar Next, o Xcode perguntará onde você deseja salvar o projeto. Você pode escolher o que mais lhe convier, mas talvez você ache que sua área de trabalho é mais fácil. Feito isso, o Xcode criará o novo projeto para você e abrirá ContentView.swift para edição. É aqui que escreveremos todo o nosso código, e você verá algum código SwiftUI padrão para nós.

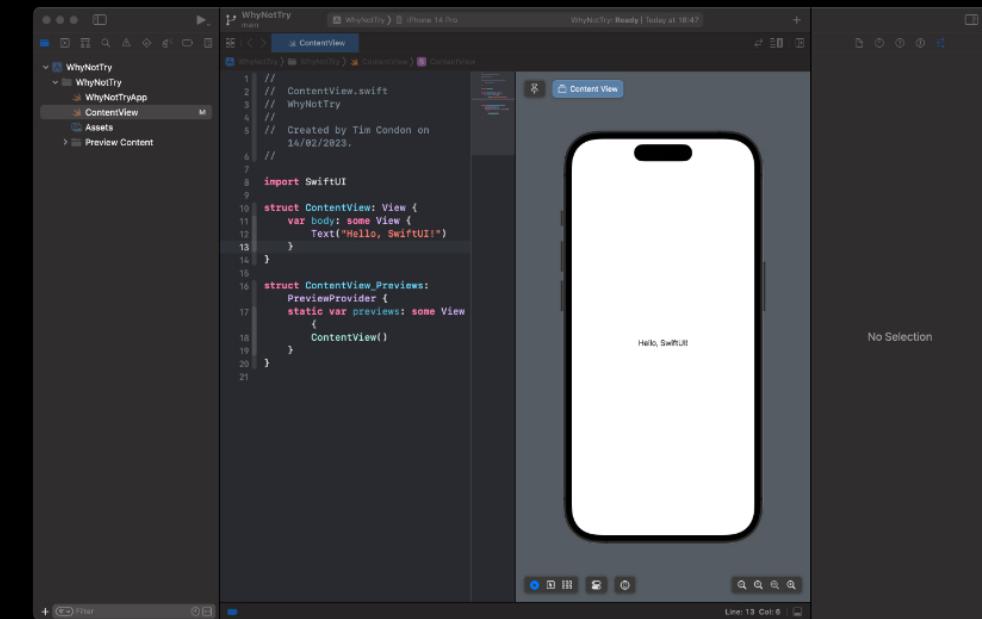
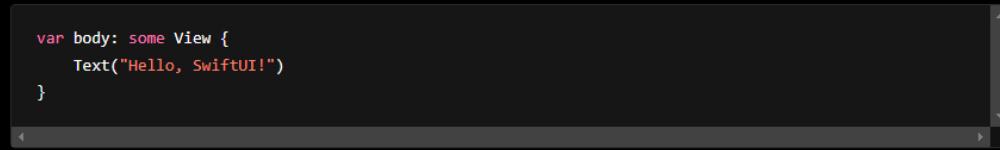




O código de exemplo que o Xcode criou para nós cria uma nova visualização chamada `ContentView`. As visualizações são como o SwiftUI representa a interface do usuário do nosso aplicativo na tela, e podemos adicionar layout e lógica personalizados a ela.

No lado direito do Xcode, você verá uma visualização ao vivo desse código em execução – se você fizer uma alteração no código à esquerda, ele aparecerá imediatamente na visualização. Se você não conseguir ver a visualização, siga [estas instruções](#) para ativá-la.

Por exemplo, tente substituir o `body` código padrão por este:

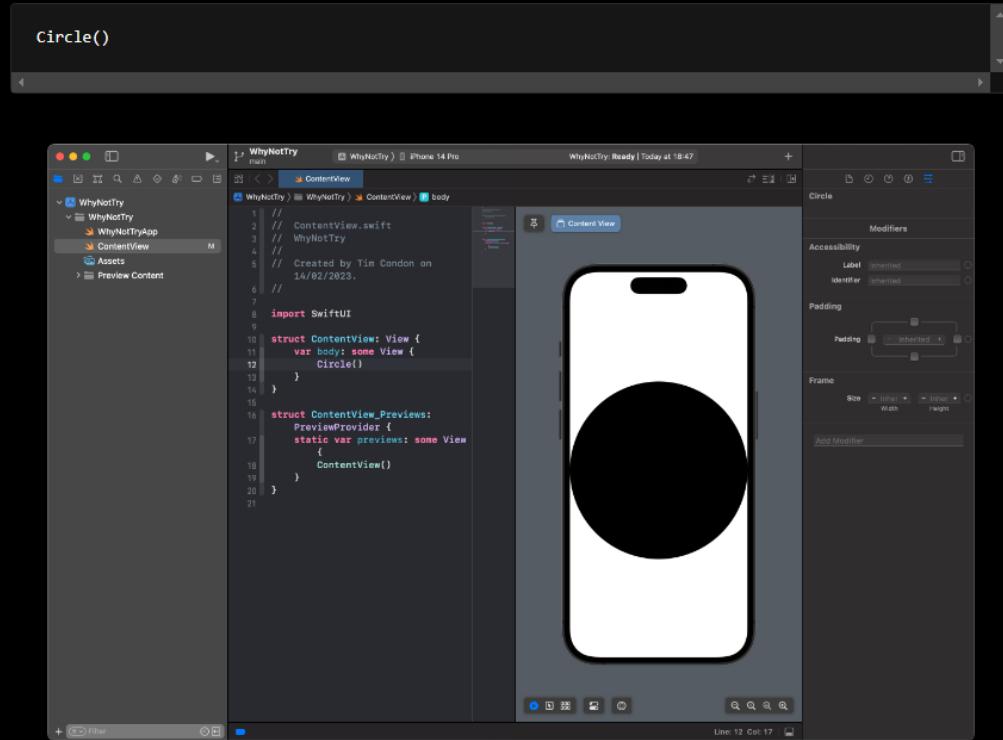


Você deverá ver a atualização da visualização imediatamente, o que torna a prototipagem muito rápida enquanto você trabalha. Esta é uma propriedade computada chamada `body`, e o SwiftUI irá chamá-la sempre que quiser exibir nossa interface de usuário.

Construindo uma UI estática

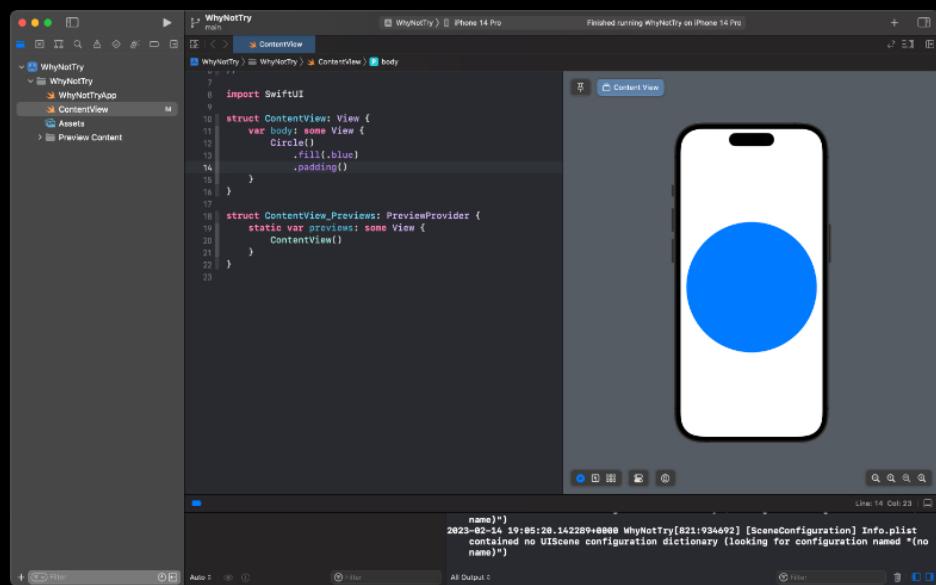
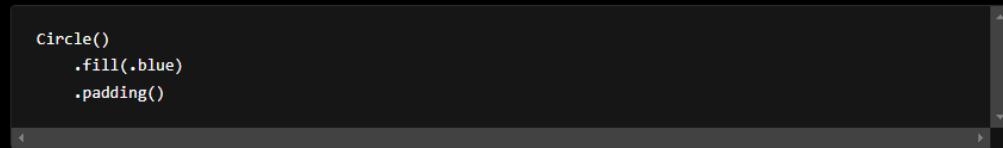
Neste app vamos mostrar ao usuário uma nova atividade que ele pode tentar para manter a forma, como basquete, golfe e caminhadas. Para torná-lo um pouco mais atraente, exibiremos cada atividade usando seu nome e também um ícone representando a atividade e, em seguida, adicionaremos um toque de cor atrás dela.

A parte principal da nossa interface de usuário será um círculo mostrando a atividade atualmente recomendada. Podemos desenhar círculos apenas escrevendo `Circle`, então substitua `Text("Hello, SwiftUI!")` visualização por isto:



Na sua visualização, você verá um grande círculo preto preenchendo a largura disponível da tela. Isso é um começo, mas não está certo – queremos um pouco de cor ali e, idealmente, adicionar um pouco de espaço em cada lado para que não pareça tão apertado.

Ambos podem ser realizados chamando métodos na `Circle` visualização. Chamamos esses *modificadores de visualização* no SwiftUI porque eles modificam a aparência ou funcionamento do círculo e, neste caso, precisamos usar o `fill()` modificador para colorir o círculo e, em seguida, o `padding()` modificador para adicionar algum espaço ao redor dele, assim:

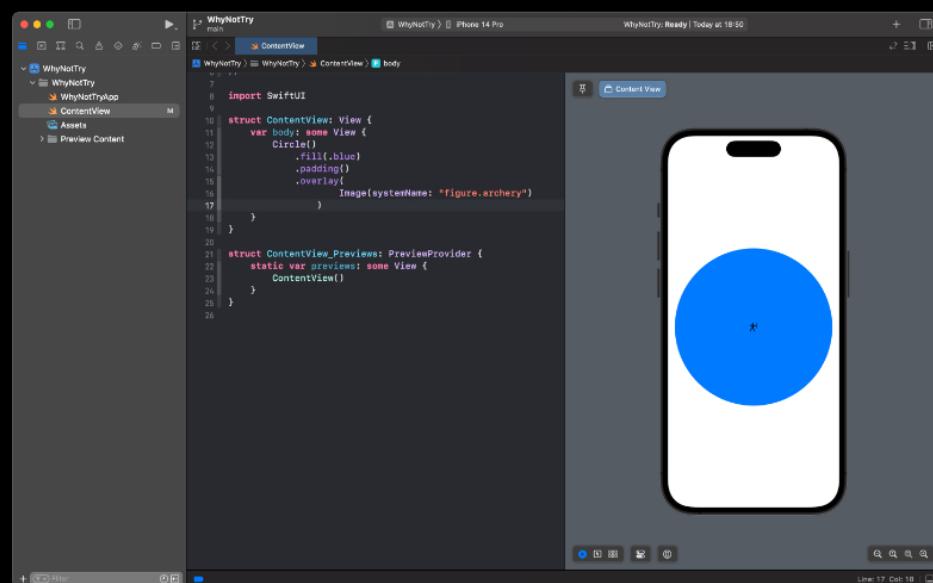


A `.blue` cor é uma das várias opções integradas, como `.red`, `.white`, e `.green`. Todos eles reconhecem a aparência, o que significa que parecem sutilemente diferentes dependendo se o dispositivo está no modo escuro ou no modo claro.

Sobre esse círculo azul colocaremos um ícone mostrando a atividade que recomendamos. O iOS vem com vários milhares de ícones gratuitos chamados *SF Symbols*, e há um aplicativo gratuito que você pode baixar que mostra todas as opções. Cada um desses ícones está disponível em vários pesos, pode ser aumentado ou diminuído suavemente e muitos também podem ser coloridos.

Aqui, porém, queremos algo simples e agradável: queremos apenas um ícone colocado sobre nosso círculo. Isso significa usar outro modificador chamado `overlay()`, que coloca uma visualização sobre a outra. Modifique seu código para isto:

```
Circle()
    .fill(.blue)
    .padding()
    .overlay(
        Image(systemName: "figure.archery")
    )
```

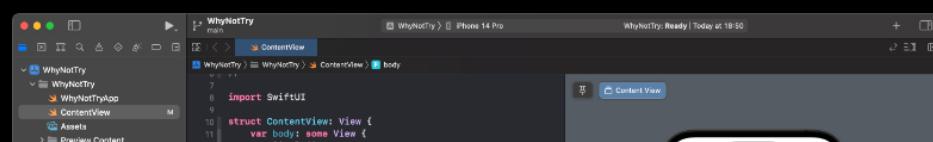


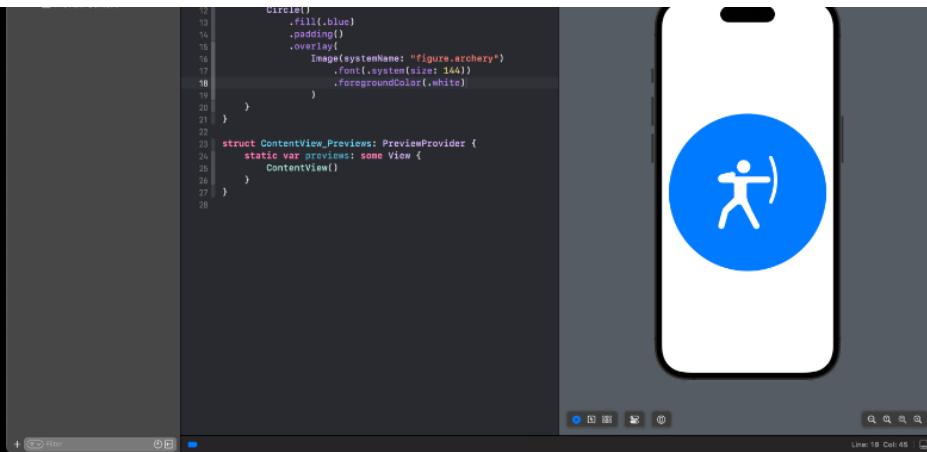
Você deverá ver um pequeno ícone preto de tiro com arco sobre nosso grande círculo azul – é a ideia certa, mas não parece ótima.

O que realmente queremos é que o ícone do tiro com arco seja muito maior e também muito mais visível nesse fundo. Para isso precisamos de mais dois modificadores: `font()` controlar o tamanho do ícone e `foregroundColor()` alterar sua cor. Sim, usamos um modificador de fonte para controlar o tamanho do ícone – Símbolos SF como este são dimensionados automaticamente com o resto do nosso texto, o que os torna realmente flexíveis.

Ajuste seu `Image` código para isto:

```
Image(systemName: "figure.archery")
    .font(.system(size: 144))
    .foregroundColor(.white)
```

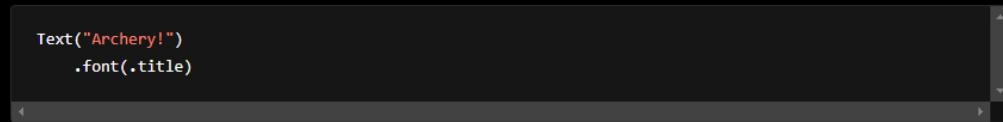




Dica: esse `font()` modificador pede uma fonte de sistema de 144 pontos, que é boa e grande em todos os dispositivos.

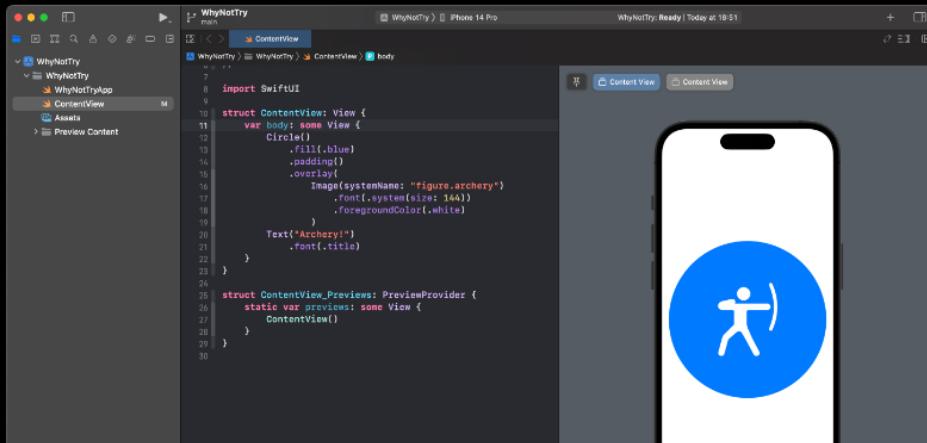
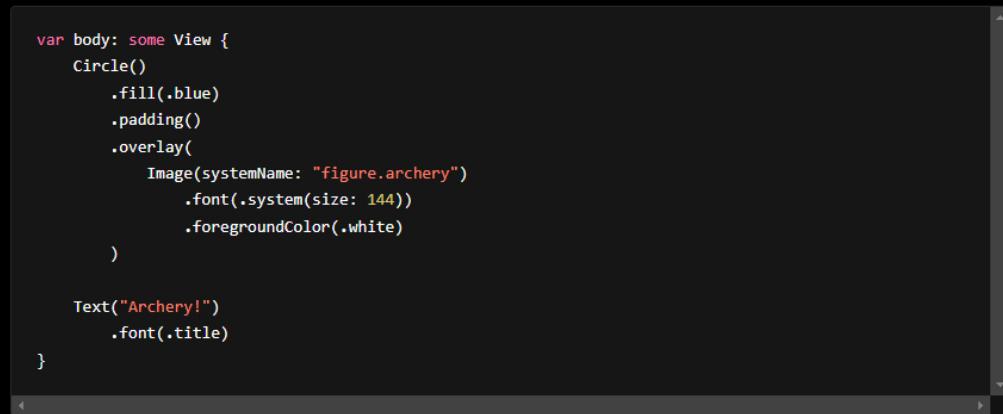
Isso agora deve parecer muito melhor.

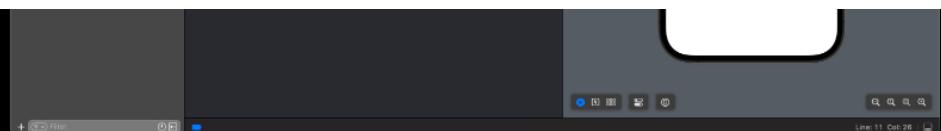
A seguir, vamos adicionar algum texto abaixo da imagem para que fique claro para o usuário qual é a sugestão. Você já conheceu a `Text` View e o `font()` modificador, então pode adicionar esse código abaixo do `Circle` código:



Em vez de usar um tamanho de fonte fixo, ele usa um dos tamanhos de tipo dinâmico integrados do SwiftUI chamado `.title`. Isso significa que a fonte aumentará ou diminuirá dependendo das configurações do usuário, o que geralmente é uma boa ideia.

Se tudo correu conforme o planejado, seu código deverá ficar assim:





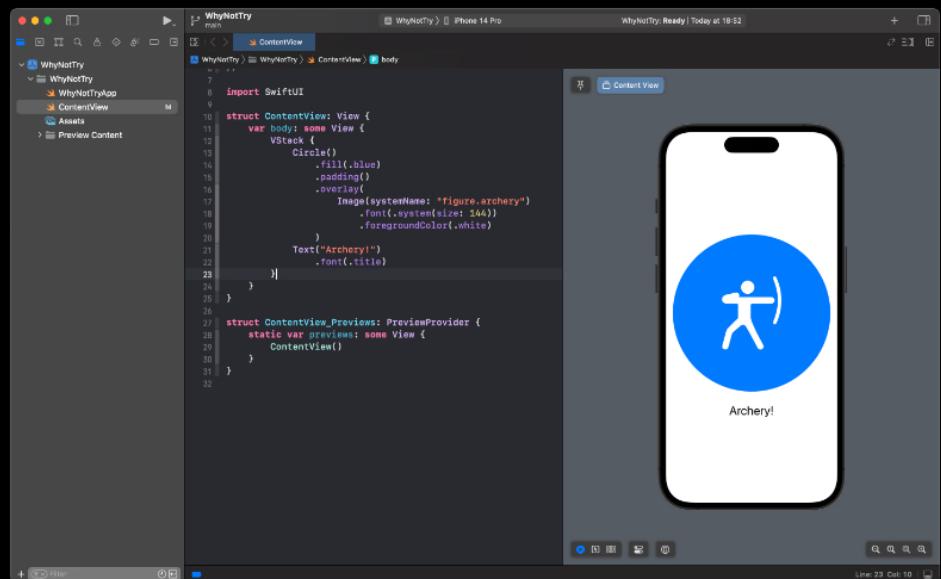
No entanto, o que você vê na visualização do Xcode provavelmente não corresponderá ao que você esperava: você verá o mesmo ícone de antes, mas nenhum texto. O que da?

O problema aqui é que dissemos ao SwiftUI que nossa interface de usuário terá duas visualizações internas – o círculo e algum texto – mas não dissemos como organizá-las. Queremos eles lado a lado? Um acima do outro? Ou em algum outro tipo de layout?

Podemos escolher, mas acho que aqui um layout vertical ficará melhor. No SwiftUI conseguimos isso com um novo tipo de visualização chamado `VStack`, que é colocado *em torno* do nosso código atual, assim:

```
 VStack {  
     Circle()  
         .fill(.blue)  
         .padding()  
         .overlay(  
             Image(systemName: "figure.archery")  
                 .font(.system(size: 144))  
                 .foregroundColor(.white)  
         )  
  
     Text("Archery!")  
         .font(.title)  
 }
```

E agora você deverá ver o layout que esperava anteriormente: nosso ícone de tiro com arco acima do texto “Tiro com arco!”.



Isso é muito melhor!

Para finalizar nossa primeira passagem nesta interface de usuário, podemos adicionar um título no topo. Já temos um `VStack` que nos permite posicionar as visualizações uma acima da outra, mas não quero o título lá também porque mais tarde adicionaremos alguma animação para essa parte da tela.

Felizmente, o SwiftUI nos permite aninhar pilhas livremente, o que significa que podemos colocar uma `VStack` dentro de outra `VStack` para obter o comportamento exato que desejamos. Então, altere seu código para este:

```

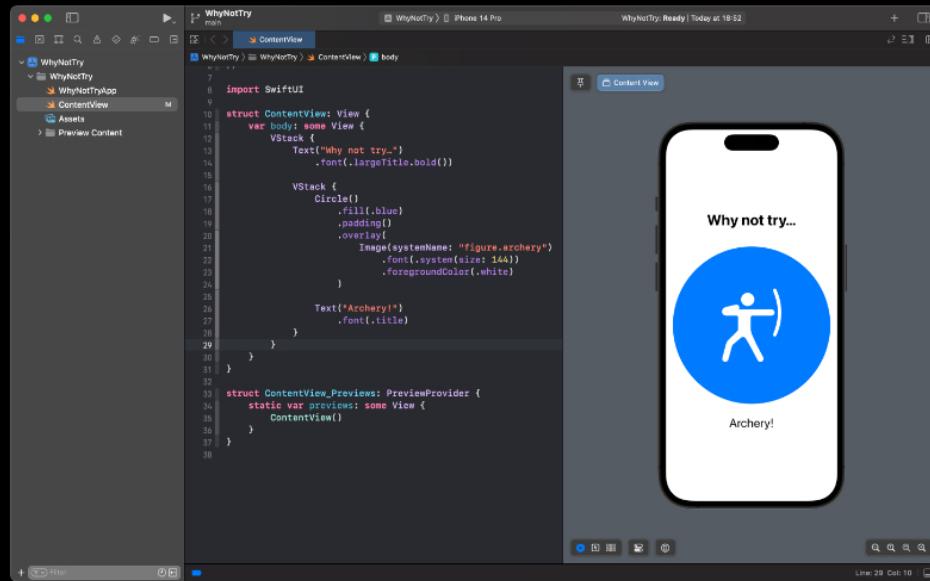
VStack {
    Text("Why not try...")
        .font(.largeTitle.bold())

    VStack {
        Circle()
            .fill(.blue)
            .padding()
            .overlay(
                Image(systemName: "figure.archery")
                    .font(.system(size: 144))
                    .foregroundColor(.white)
            )
    }

    Text("Archery!")
        .font(.title)
    }
}

```

Isso faz com que o novo texto tenha uma fonte de título grande e também em negrito para que se destaque melhor como um título real para nossa tela.



Agora temos duas `VStack` visualizações: uma interna que segura o círculo e “Tiro com arco!” texto e um externo que adiciona um título ao redor do arquivo interno `VStack`. Isso será muito útil mais tarde, quando adicionarmos animação!

Trazendo isso à vida

Por mais divertido que seja o tiro com arco, este aplicativo realmente precisa sugerir uma atividade aleatória aos usuários, em vez de mostrar sempre a mesma coisa. Isso significa adicionar duas novas propriedades à nossa visão: uma para armazenar o conjunto de atividades possíveis e outra para mostrar aquela que está sendo recomendada no momento.

SF Symbols tem muitas atividades interessantes para escolher, então escolhi algumas que funcionam bem aqui. Nossa `ContentView` struct já possui uma `body` propriedade contendo nosso código SwiftUI, mas queremos adicionar novas propriedades fora dela. Então, altere seu código para este:

```

struct ContentView: View {
    var activities = ["Archery", "Baseball", "Basketball", "Bowling", "Boxing", "Cricket", "Curling", "F
    var selected = "Archery"
}

```

```
var body: some View {  
    // ...  
}
```

Importante: Observe como as propriedades `activities` and estão *dentro* da estrutura – isso significa que elas pertencem a , em vez de serem apenas variáveis flutuantes em nosso programa. `selected` `ContentView`

Isso cria uma série de vários nomes de atividades e seleciona o tiro com arco como padrão. Agora podemos usar a atividade selecionada em nossa UI usando interpolação de strings – podemos colocar a `selected` variável diretamente dentro de strings.

Para o nome da atividade isso é simples:

```
Text("\(selected)!")  
.font(.title)
```

Para a imagem isso é um pouco mais complicado, porque precisamos prefixá-la com `figure.` o nome da atividade em letras minúsculas – queremos `figure.archery` em vez de `figure.Archery`, caso contrário o símbolo SF não será carregado.

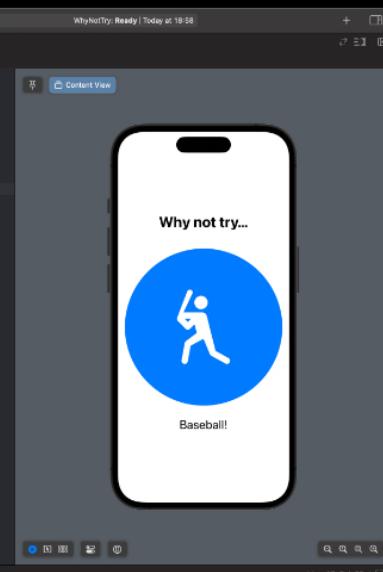
Então, altere seu `Image` código para isto:

```
Image(systemName: "figure.\(selected.lowercased())")
```

Essas mudanças significam que nossa UI exibirá qualquer que seja a `selected` propriedade definida, então podemos ver tudo mudar se você colocar uma nova string nessa propriedade:

```
var selected = "Baseball"
```

```
var selected = "Baseball"  
  
var body: some View {  
    VStack {  
        Text("Why not try...")  
            .font(.largeTitle.bold())  
  
        VStack {  
            Circle()  
                .fill(.blue)  
                .padding()  
                .overlay(  
                    Image(systemName:  
                        "figure.\(selected  
                            .lowercased())")  
                        .font(.system(size: 144))  
                        .foregroundColor(.white)  
                )  
  
            Text("\(selected)")  
                .font(.title)  
        }  
    }  
}  
  
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
    }  
}
```



Claro, queremos que isso mude *dinamicamente*, em vez de ter que editar o código todas as vezes, então vamos adicionar um botão abaixo do nosso botão interno `VStack` que mudará a atividade selecionada toda vez que for pressionado. Porém, ele ainda está dentro do outer `VStack`, o que significa que será organizado abaixo do título e do ícone de atividade.

Adicione este código agora:

```

Button("Try again") {
    // change activity
}
.buttonStyle(.borderedProminent)

```

```

1 WhyNotTry
2 main
3 ContentView
4 WhyNotTry
5 WhyNotTryApp
6 ContentView
7
8 import SwiftUI
9
10 struct ContentView: View {
11     var activities = ["Archery", "Baseball", "Basketball",
12         "Bowling", "Boxing", "Cricket", "Curling",
13         "Fencing", "Golf", "Hiking", "Lacrosse", "Rugby",
14         "Squash"]
15
16     var selected = "Baseball"
17
18     var body: some View {
19         VStack {
20             Text("Why not try...")
21                 .font(.largeTitle.bold())
22
23             VStack {
24                 Circle()
25                     .fill(.blue)
26                     .padding()
27                     .overlay(
28                         Image(systemName:
29                             "figure.\(selected)"
30                             .fontWeight(.bold))
31                         .font(.system(size: 164))
32                         .foregroundColor(.white)
33                     )
34
35             Button("Try again") {
36                 // change activity
37             }
38             .buttonStyle(.borderedProminent)
39         }
40     }
41 }

```

Então, sua estrutura deve ser esta:

```

VStack {
    // "Why not try..." text

    // Inner VStack with icon and activity name

    // New button code
}

```

O novo código do botão faz três coisas:

1. Criamos o `Button` passando um título para mostrar como o rótulo do botão.
2. O `// change activity` comentário é o código que será executado quando o botão for pressionado.
3. O `buttonStyle()` modificador diz ao SwiftUI que queremos que este botão se destaque, então você o verá aparecer em um retângulo azul com texto branco.

Apenas fazer um comentário sobre a ação do botão não é muito interessante – na verdade, queremos configurá-lo `selected` para um elemento aleatório do `activities` array. Podemos escolher um elemento aleatório do array chamando o `randomElement()` método nomeado de forma útil nele, então substitua o comentário por isto:

```

selected = activities.randomElement()

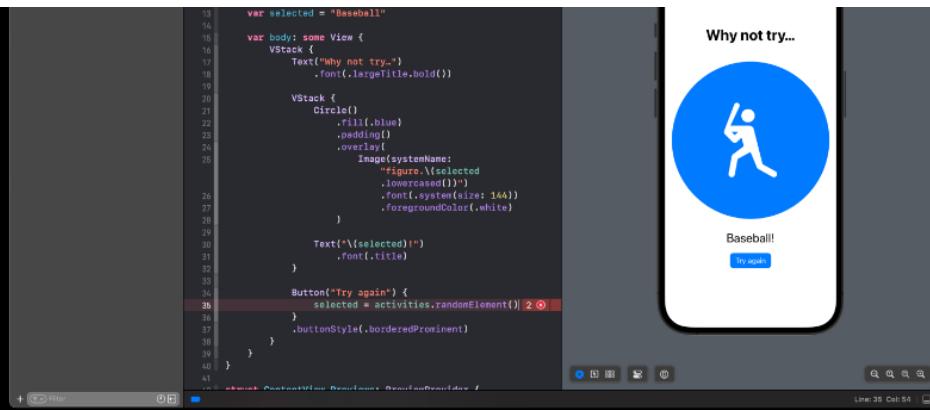
```

Esse código *parece* correto, mas na verdade causará erros no compilador. Estamos dizendo ao Swift para escolher um elemento aleatório do array e colocá-lo na `selected` propriedade, mas não há como o Swift ter certeza de que há algo naquele array – ele pode estar vazio e, nesse caso, não há nenhum elemento aleatório para retornar.

```

1 WhyNotTry
2 main
3 ContentView
4 WhyNotTry
5 WhyNotTryApp
6 ContentView
7
8 import SwiftUI
9
10 struct ContentView: View {
11     var activities = ["Archery", "Baseball", "Basketball",
12         "Bowling", "Boxing", "Cricket", "Curling",
13         "Fencing", "Golf", "Hiking", "Lacrosse", "Rugby",
14         "Squash"]
15
16     var selected = "Baseball"
17
18     var body: some View {
19         VStack {
20             Text("Why not try...")
21                 .font(.largeTitle.bold())
22
23             VStack {
24                 Circle()
25                     .fill(.blue)
26                     .padding()
27                     .overlay(
28                         Image(systemName:
29                             "figure.\(selected)"
30                             .fontWeight(.bold))
31                         .font(.system(size: 164))
32                         .foregroundColor(.white)
33                     )
34
35             Button("Try again") {
36                 // change activity
37             }
38             .buttonStyle(.borderedProminent)
39         }
40     }
41 }

```



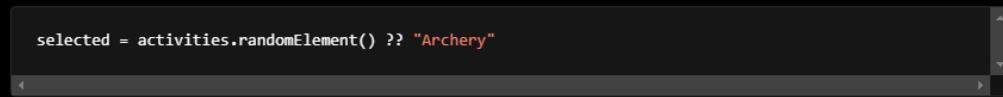
```

13     var selected = "Baseball"
14
15     var body: some View {
16         VStack {
17             Text("Why not try...")
18                 .font(.largeTitle.bold())
19
20             VStack {
21                 Circle()
22                     .fill(.blue)
23                     .padding()
24                     .overlay(
25                         Image(systemName:
26                             "figure.\(selected)
27                             .lowercased())")
28                             .font(.system(size: 144))
29                             .foregroundColor(.white)
30
31             Text("\(selected)!")
32                 .font(.title)
33         }
34
35         Button("Try again") {
36             selected = activities.randomElement() ?? "Archery"
37         }
38         .buttonStyle(.borderedProminent)
39     }
40 }

```

Swift chama esses *opcionais*: `randomElement()` não retornará uma string regular, retornará uma string *opcional*. Isso significa que a string pode não estar lá, portanto não é seguro atribuí-la à `selected` propriedade.

Mesmo sabendo que o array nunca estará vazio – ele *sempre* terá atividades nele – podemos dar ao Swift um valor padrão sensato para usar caso o array fique vazio no futuro, assim:

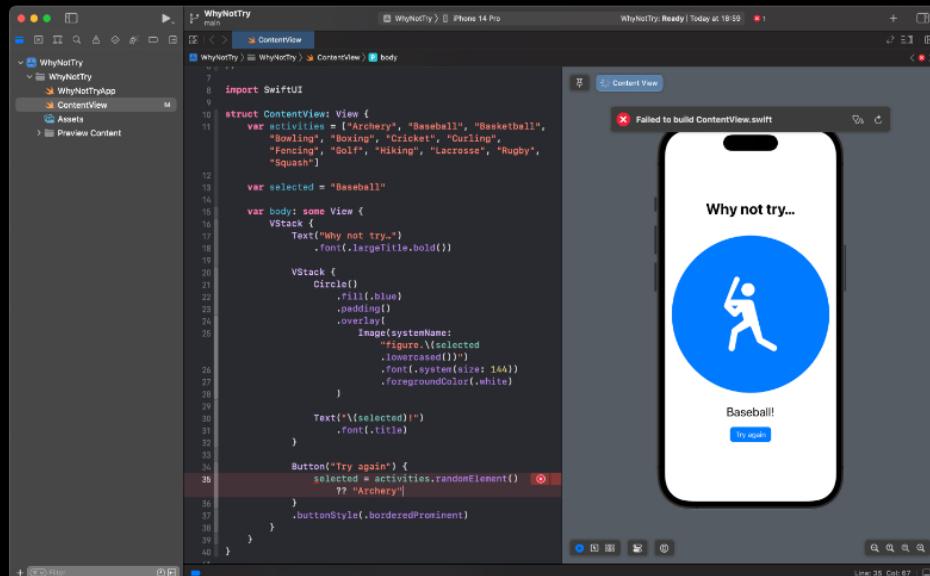


```

selected = activities.randomElement() ?? "Archery"

```

Isso corrige parcialmente nosso código, mas o Xcode ainda mostrará um erro. O problema agora é que o SwiftUI não gosta que alteremos o estado do nosso programa dentro de nossas estruturas de visualização sem aviso prévio – ele quer que marquemos todo o estado mutável com antecedência, para que ele saiba que deve observar as mudanças.



```

13     var selected = "Baseball"
14
15     var body: some View {
16         VStack {
17             Text("Why not try...")
18                 .font(.largeTitle.bold())
19
20             VStack {
21                 Circle()
22                     .fill(.blue)
23                     .padding()
24                     .overlay(
25                         Image(systemName:
26                             "figure.\(selected)
27                             .lowercased())")
28                             .font(.system(size: 144))
29                             .foregroundColor(.white)
30
31             Text("\(selected)!")
32                 .font(.title)
33         }
34
35         Button("Try again") {
36             selected = activities.randomElement() ?? "Archery"
37         }
38         .buttonStyle(.borderedProminent)
39     }
40 }

```

Isso é feito escrevendo `@State` antes de qualquer propriedade da view que será alterada, assim:



```

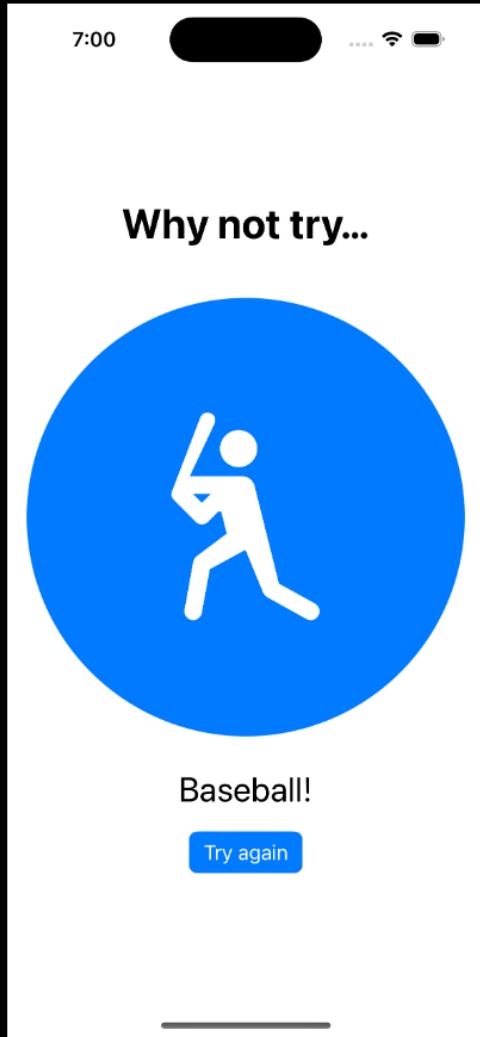
@State var selected = "Baseball"

```

Isso é chamado de *property wrapper*, o que significa que envolve nossa `selected` propriedade com alguma lógica extra. O `@state` wrapper de propriedade nos permite alterar o estado de visualização livremente, mas também monitora automaticamente suas propriedades em busca de alterações, para garantir que a interface do usuário permaneça atualizada com os valores mais recentes.

Isso corrige os dois erros em nosso código, então agora você pode pressionar Cmd+R para criar e executar seu aplicativo no simulador iOS. Ele irá sugerir beisebol por padrão, mas toda vez que você

pressionar “Tentar novamente”, você verá que isso muda.



Adicionando um pouco de polimento

Antes de terminarmos este projeto, vamos adicionar mais alguns ajustes para torná-lo melhor.

Primeiro, uma pergunta fácil: a Apple recomenda que o estado de visualização local seja sempre marcado com `private` controle de acesso. Em projetos maiores, isso significa que você não pode escrever acidentalmente um código que leia o estado local de uma visualização de outra, o que ajuda a manter seu código mais fácil de entender.

Isso significa modificar a `selected` propriedade assim:

```
@State private var selected = "Baseball"
```

Em segundo lugar, em vez de mostrar sempre um fundo azul, podemos escolher uma cor aleatória de cada vez. Isso leva duas etapas, começando com uma nova propriedade de todas as cores que queremos selecionar – coloque-a ao lado da `activities` propriedade:

```
var colors: [Color] = [.blue, .cyan, .gray, .green, .indigo, .mint, .orange, .pink, .purple, .red]
```

Agora podemos alterar o `fill()` modificador do nosso círculo para usar `randomElement()` naquele array, ou `.blue` se de alguma forma o array ficar vazio:

```
Circle()
    .fill(colors.randomElement() ?? .blue)
```

```
.fill(colors.randomElement() ?? .blue)
```

Terceiro, podemos separar a atividade `VStack` e o botão “Tentar novamente” adicionando uma nova visualização SwiftUI entre eles, chamada `Spacer`. Este é um espaço flexível que se expande automaticamente, o que significa que empurrará nosso ícone de atividade para o topo da tela e o botão para baixo.

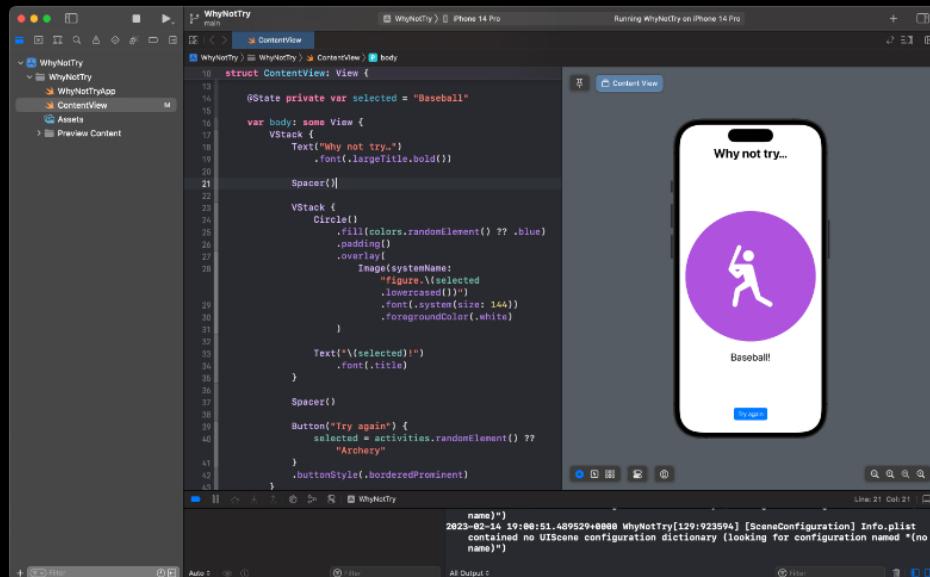
Insira-o entre os dois, assim:

```
 VStack {
    // current Circle/Text code
}

Spacer()

Button("Try again") {
    // ...
}
```

Se você adicionar vários espaçadores, eles dividirão o espaço igualmente entre eles. Se você tentar colocar um segundo espaçador antes do texto “Por que não tentar…”, você verá o que quero dizer – o SwiftUI criará uma quantidade igual de espaço acima do texto e abaixo do nome da atividade.



E quarto, seria bom se a mudança entre as atividades fosse mais suave, o que podemos fazer animando a mudança. No SwiftUI, isso é feito agrupando as alterações que queremos animar com uma chamada para a `withAnimation()` função, assim:

```
 Button("Try again") {
    withAnimation {
        selected = activities.randomElement() ?? "Archery"
    }
}
.buttonStyle(.borderedProminent)
```

Isso fará com que o pressionamento do botão se move entre as atividades com um desvanecimento suave. Se quiser, você pode personalizar essa animação passando a animação desejada para a `withAnimation()` chamada, assim:

```
withAnimation(.easeInOut(duration: 1)) {
    // ...
}
```

Isso é uma melhoria, mas podemos fazer melhor!

O desbotamento acontece porque o SwiftUI vê a mudança da cor de fundo, do ícone e do texto, portanto, remove as visualizações antigas e as substitui por novas visualizações. Anteriormente, fiz você criar um inner `vStack` para abrigar essas três visualizações, e agora você pode ver o porquê: vamos dizer ao SwiftUI que essas visualizações podem ser identificadas como um único grupo e que o identificador do grupo pode mudar com o tempo.

Para que isso aconteça, precisamos começar definindo mais alguns estados do programa dentro da nossa visão. Este será o identificador do nosso inner `vStack` e, como ele mudará à medida que nosso programa for executado, usaremos `@State`. Adicione esta propriedade ao lado de `selected`:

```
@State private var id = 1
```

Dica: esse é um estado de visualização mais local, por isso é uma boa prática marcá-lo com `private`.

A seguir, podemos dizer ao SwiftUI para alterar esse identificador toda vez que nosso botão for pressionado, assim:

```
Button("Try again") {
    withAnimation(.easeInOut(duration: 1)) {
        selected = activities.randomElement() ?? "Archery"
        id += 1
    }
}.buttonStyle(.borderedProminent)
```

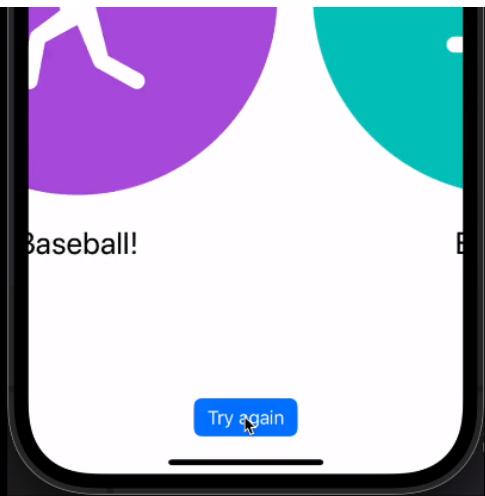
Finalmente, podemos usar o `id()` modificador do SwiftUI para anexar esse identificador ao todo inner `vStack`, o que significa que quando o identificador mudar, o SwiftUI deve considerar o todo `vStack` como novo. Isso fará com que o antigo `vStack` seja removido e um novo `vStack` seja adicionado, em vez de apenas as visualizações individuais dentro dele. Melhor ainda, podemos controlar como essa transição de adição e remoção acontece usando um `transition()` modificador, que possui várias transições integradas que podemos usar.

Então, adicione esses dois modificadores ao inner `vStack`, dizendo ao SwiftUI para identificar todo o grupo usando nossa `id` propriedade e anime suas transições de adição e remoção com um slide:

```
.transition(.slide)
.id(id)
```

Pressione Cmd + R para executar seu aplicativo uma última vez e você verá que pressionar “Tentar novamente” agora anima suavemente a atividade antiga fora da tela e a substitui por uma nova. Ele até sobrepõe animações se você pressionar “Tentar novamente” repetidamente!





Onde agora?

Abordamos muitos fundamentos do SwiftUI neste tutorial, incluindo texto, imagens, botões, pilhas, animação e até mesmo uso `@State` para marcar valores que mudam com o tempo. O SwiftUI é capaz de muito mais e pode ser usado para criar aplicativos complexos de plataforma cruzada, se necessário.

Se você quiser continuar aprendendo SwiftUI, há muitos recursos gratuitos disponíveis. Por exemplo, a Apple publica uma ampla variedade de tutoriais que cobrem tópicos essenciais, desenho e animação, design de aplicativos e muito mais. Também postaremos links aqui no Swift.org para alguns outros tutoriais populares – somos uma comunidade grande e acolhedora e estamos felizes por você participar!

O código-fonte deste guia pode ser encontrado [no GitHub](#)