



# Projeto calculadora

Ewerton J. Silva

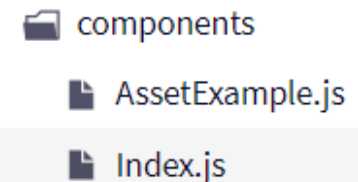
[ewerton.silva41@etec.sp.gov.br](mailto:ewerton.silva41@etec.sp.gov.br)

# Criando componente principal

```
1  import React, { useState } from 'react';
2  import { Text, View, StyleSheet } from 'react-native';
3
4  export default function Index() {
5    return (
6      <View style={styles.container}>
7        <Text style={styles.paragraph}> Exemplo 7 </Text>
8      </View>
9    );
10 }
11
12 const styles = StyleSheet.create({
13   container: {
14     flex: 1,
15     justifyContent: 'center',
16     backgroundColor: '#EEE',
17     padding: 8,
18   },
19   paragraph: {
20     margin: 6,
21     fontSize: 18,
22     fontWeight: 'bold',
23     textAlign: 'center',
24     color: '#AAA',
25   },
26 });
```

Em um novo projeto dentro da pasta “components” crie um novo arquivo com o nome “Index.js” e insira o código apresentado ao lado.

Este componente vai retornar uma View com um Text dentro.



```
components
├── AssetExample.js
└── Index.js
```

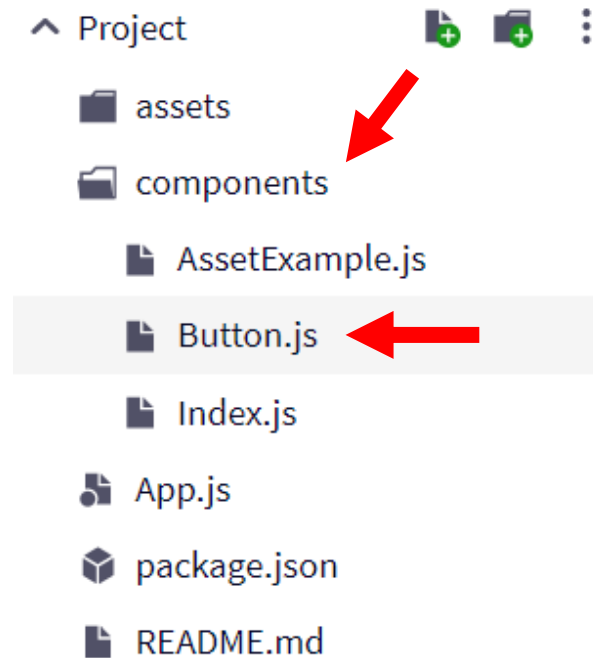
Em “App.js”, vamos importar o componente “Index.js” e utilizá-lo dentro do “App.js”. Deixe o código de “App.js” conforme o exemplo ao lado, aqui iremos retornar uma “View” com o componente criado anteriormente.

```
1  import * as React from 'react';
2  import { View, StyleSheet } from 'react-native';
3  import Constants from 'expo-constants';
4
5  import Index from './components/Index';
6
7  export default function App() {
8    return (
9      <View style={styles.container}>
10        <Index />
11      </View>
12    );
13  }
14
15  const styles = StyleSheet.create({
16    container: {
17      flex: 1,
18      justifyContent: 'center',
19      paddingTop: Constants.statusBarHeight,
20      backgroundColor: '#AAA',
21      padding: 8,
22    },
23  });
```

Exemplo 7

# Especializando os componentes

Dentro de components crie um arquivo com o nome “Button.js”



Realize os “imports” necessários, observe que neste exemplo ao tentar importar o objeto “RFPercentage” da biblioteca “react ... -fontsize” um erro é apresentado, pois esta biblioteca não faz parte das bibliotecas padronizadas do Expo...

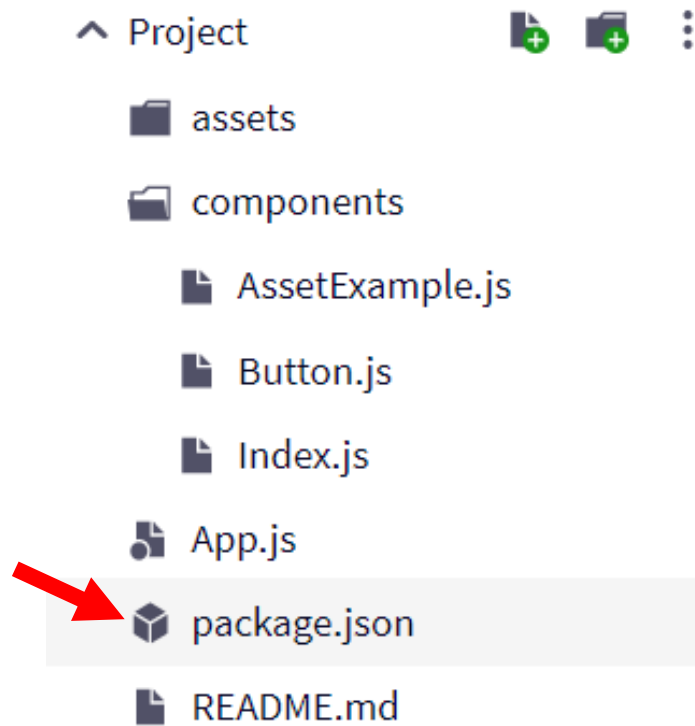
```
1  import * as React from 'react';  
2  import { Text, TouchableHighlight, StyleSheet, Dimensions } from 'react-native';  
3  import { RFPercentage } from "react-native-responsive-fontsize"; ←
```

... Para resolver isto basta clicar sobre o texto “Add dependency” conforme apontado abaixo...

The screenshot shows the Expo Snack editor interface. The top bar includes the project name "PAM - Ex 007 - Calculadora", a status "Device connected!", and buttons for "Save", "My Device", "iOS", "Android", "Web", and a search icon. The left sidebar shows the file explorer with "App.js", "AssetExample.js", "Button.js", and "Index.js" under the "Project" section. The main editor displays the code for "Button.js", which imports "react", "react-native", and "react-native-responsive-fontsize". The code defines a "Button" component using "TouchableHighlight" and "Text" from "react-native". A red arrow points to the "Add dependency" link in the error message at the bottom. The error message is displayed in a red box on the right side of the editor, stating: "Unable to resolve module 'react-native-responsive-fontsize.js' Evaluating react-native-responsive-fontsize.js Evaluating components/Button.js Evaluating components/Index.js Evaluating App.js Loading App.js Error: Unable to resolve module 'react-native-responsive-fontsize.js' at Object.eval (react-native-responsive-fontsize.js:1:1 at eval (react-native-responsive-". The bottom status bar shows the error message: "components/Button.js (3:30) 'react-native-responsive-fontsize' is not defined in dependencies. Add dependency".

components/Button.js (3:30) 'react-native-responsive-fontsize' is not defined in dependencies. **Add dependency**

... Após o procedimento citado anteriormente a referencia a esta biblioteca será adicionada automaticamente no arquivo “package.json” do nosso projeto conforme apontado abaixo:



```
1  {
2    "dependencies": {
3      "react-native-paper": "3.6.0",
4      "expo-constants": "~9.3.3",
5      "react-native-responsive-fontsize": "*"
6    }
7  }
```

Volte ao arquivo “Button.js” iremos definir o componente que será exportado.

Em “TouchableHighlight” (área de toque) iremos receber um valor no evento “onPress” que será passado através de uma função do botão “onClick” que foi clicado.

O texto do botão vai exibir uma propriedade recebida por este componente em sua chamada.

```
1  import * as React from 'react';
2  import { Text, TouchableHighlight, StyleSheet, Dimensions } from 'react-native';
3  import { RFPercantage } from "react-native-responsive-fontsize";
4  export default function Button(props) {
5    return(
6      <TouchableHighlight onPress={props.onClick}>
7        <Text style={styles.button}>{props.label}</Text>
8      </TouchableHighlight>
9    );
10 }
```



Agora insira o código para definir a formatação do botão.

Nas dimensões passamos o valor da largura da tela dividido por 4, pois assim nossa tela terá espaço para 4 botões não importando o tamanho da mesma, como os botões serão quadrados, passamos o mesmo valor para altura. Além disso utilizamos para o tamanho da fonte o objeto “RFPercantage” que permite deixar o tamanho do texto responsivo.

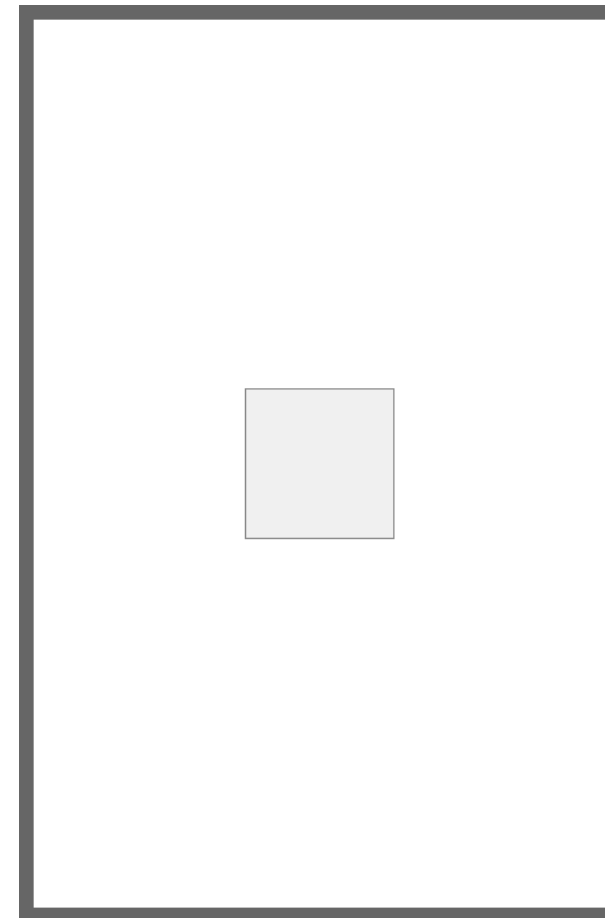
```
13  const styles = StyleSheet.create({
14    button: {
15      fontSize: RFPercantage(5),
16      height: (Dimensions.get('window').width / 4) - 4,
17      width: (Dimensions.get('window').width / 4) - 4,
18      padding: 20,
19      backgroundColor: '#F0F0F0',
20      textAlign: 'center',
21      borderWidth: 1,
22      borderColor: '#888',
23    },
24  });
```

**Obs:** “-4” no final faz referência a 8 de “padding” no app(4 de cada lado)

Agora que temos o botão implementado, volte ao “Index.js” e deixe o código como no exemplo:

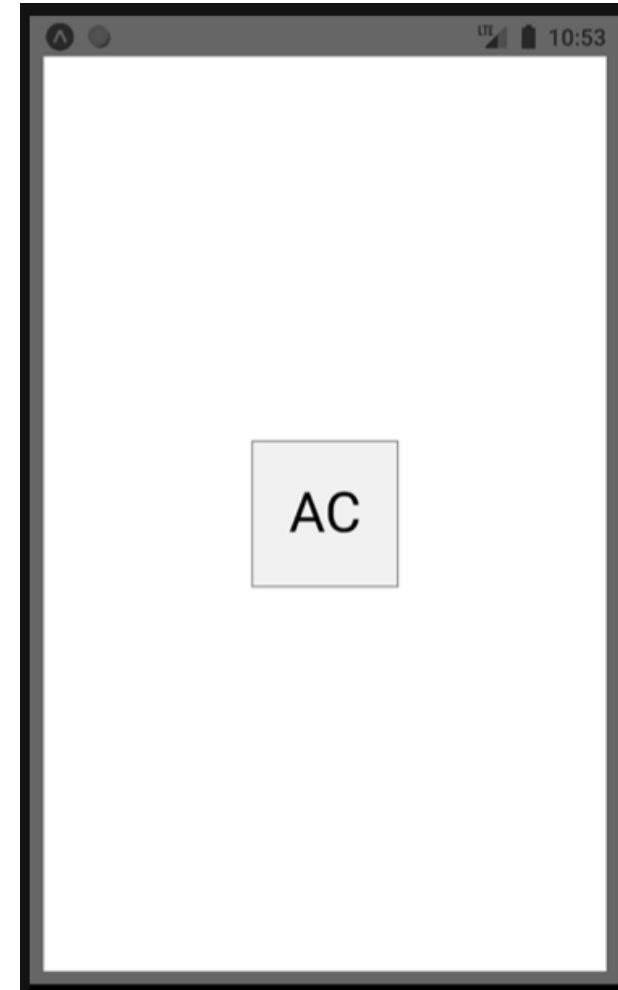
**Obs:** o estilo chamado “buttons” será criado posteriormente.

```
1  import React, { useState } from 'react';
2  import { View, StyleSheet } from 'react-native';
3
4  import Button from './Button';
5
6  export default function Index() {
7    return (
8      <View style={styles.container}>
9        <View style={styles.buttons}>
10          <Button />
11        </View>
12      </View>
13    );
14  }
15
16  const styles = StyleSheet.create({
17    container: {
18      flex: 1,
19      backgroundColor: '#FFF',
20      alignItems: 'center',
21      justifyContent: 'center',
22    },
23  });
```



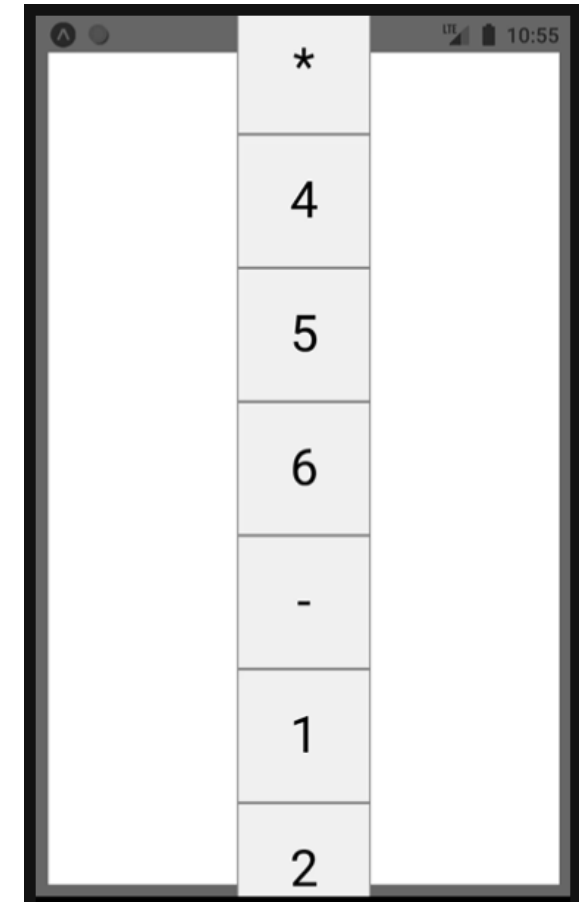
Para o botão inserido vamos passar uma “prop” referente ao texto que será apresentado no mesmo.

```
<View style={styles.container}>  
  <View style={styles.buttons}>  
    <Button label='AC' />  
  </View>  
</View>
```



A seguir insira os outros botões que teremos em nossa calculadora na ordem apresentada no código ao lado, eles serão inseridos um abaixo do outro devido as configurações do flexBox.

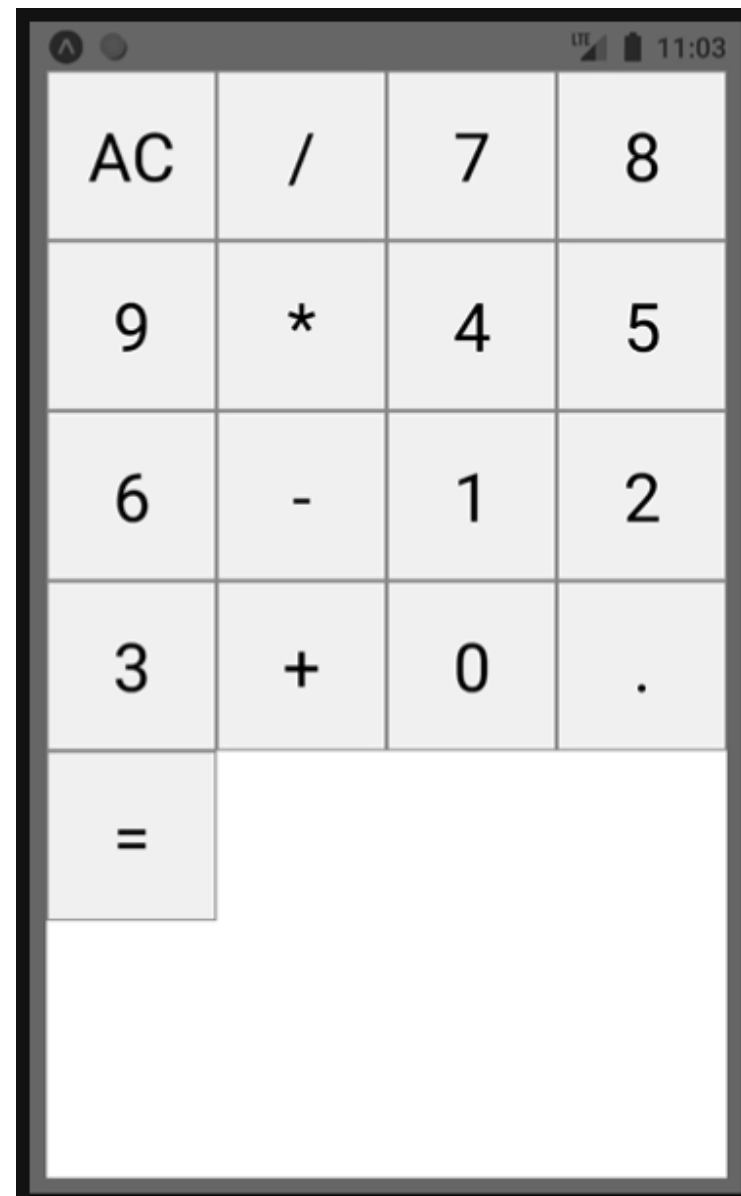
```
<View style={styles.buttons}>  
  <Button label='AC' />  
  <Button label='/' />  
  <Button label='7' />  
  <Button label='8' />  
  <Button label='9' />  
  <Button label='*' />  
  <Button label='4' />  
  <Button label='5' />  
  <Button label='6' />  
  <Button label='-' />  
  <Button label='1' />  
  <Button label='2' />  
  <Button label='3' />  
  <Button label='+' />  
  <Button label='0' />  
  <Button label='.' />  
  <Button label='=' />  
</View>
```



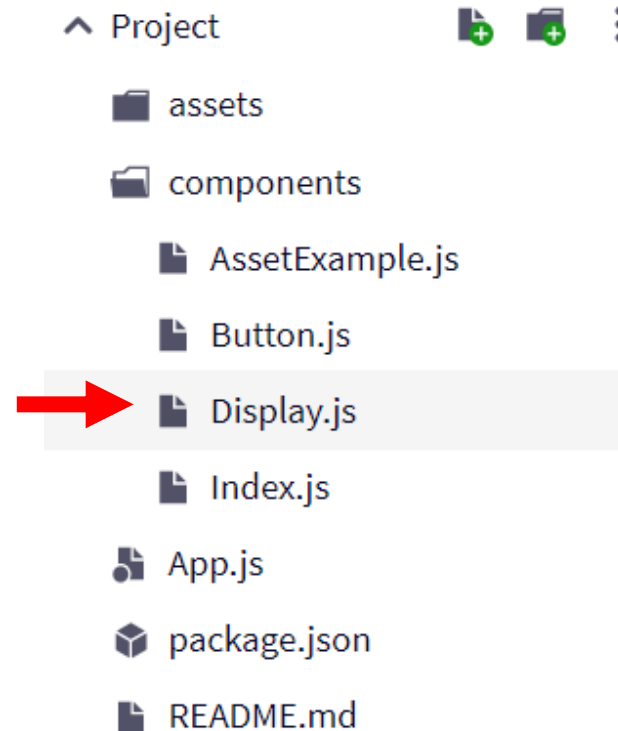
Agora redefina os novos estilos conforme o código abaixo:

Em `flexDirection` além de definir que o alinhamento será horizontal, também teremos que definir que teremos quebra nas linhas com `flexWrap`. Cabem 4 botões por linha conforme nossa configuração de tamanho para os mesmos.

```
32  const styles = StyleSheet.create({
33    container: {
34      flex: 1,
35      backgroundColor: '#FFF',
36    },
37    buttons: {
38      flexDirection: 'row',
39      flexWrap: 'wrap'
40    }
41  });
```



Na pasta “components” crie um novo arquivo com o nome “Display.js”, faça os imports necessários:



```
1  import * as React from 'react';
2  import { Text, View, StyleSheet, Dimensions } from 'react-native';
3  import { RFPpercentage } from "react-native-responsive-fontsize";
```

Aqui iremos criar um componente funcional da forma mais simples possível retornando uma “View”, dentro dela será apresentado um “Text” com o valor passado pela “prop” de nome “value” assim apresentaremos display da calculadora.

```
5  export default props =>
6    <View style={styles.display}>
7      <Text style={styles.displayValue} numberOfLines={1}>{props.value}</Text>
8    </View>
```

Em seguida defina os estilos relacionados ao Display.

```
10  const styles = StyleSheet.create({
11    display: {
12      flex: 1,
13      padding: 20,
14      justifyContent: 'center',
15      backgroundColor: 'rgba(0, 0, 0, 0.6)',
16      alignItems: 'flex-end',
17    },
18    displayValue: {
19      fontSize: RFPercentage(8),
20      color: '#FFF',
21    },
22  });
```



Volte ao componente “Index.js” para importar o ultimo componente criado e definir um state que vai armazenar os valores que serão apresentados no display.

```
1  import React, { useState } from 'react';
2  import { View, StyleSheet } from 'react-native';
3
4  import Button from './Button';
5  → import Display from './Display';
6
7  export default function Index() {
8
9  →  const [displayValue, setDisplayValue] = useState('0');
10
11  return (
```

O display vai ficar dentro de container, mas fora da área dos botões.

```
return (  
  <View style={styles.container}>  
    <Display value={displayValue} />  
    <View style={styles.buttons}>  
      <Button label='AC' />  
    </View>  
  </View>  
)
```

0			
AC	/	7	8
9	*	4	5
6	-	1	2
3	+	0	.
=			

# Adicionando estilos no botão

Deixaremos as operações com cores diferentes e alguns botões terão o espaço de ocupação maior.

Vá em “Button.js” e adicione mais três objetos de estilos conforme a imagem.

```
27     },
28     operationButton: {
29       color: '#FFF',
30       backgroundColor: '#FA8231',
31     },
32     buttonDouble: {
33       width: ((Dimensions.get('window').width / 4) - 4) * 2,
34     },
35     buttonTriple: {
36       width: ((Dimensions.get('window').width / 4) - 4) * 3,
37     },
38   });
```

Ainda no componente “Button.js” iremos definir um estilo padrão para todos os botões como uma constante, assim outros estilos podem ser aplicados condicionalmente nos botões que necessitarem de um estilo diferente. Os estilos condicionais serão aplicados apenas se uma determinada propriedade for passada.

```
export default function Button(props) {  
  const stylesButton = [styles.button];  
  return(  
    <TouchableHighlight onPress={props.onClick}>  
      <Text style={styles.button}>{props.label}</Text>  
    </TouchableHighlight>  
  );  
}
```



Em seguida verificamos se a propriedade “double” foi passada, se sim, iremos passar para dentro do array o estilo “buttonDouble”, o mesmo vale para “triple” e “operation”.

No text, substitua o valor Styles.button pelo array “stylesButton”.

```
export default function Button(props) {  
  const stylesButton = [styles.button];  
  if (props.double) stylesButton.push(styles.buttonDouble);  
  if (props.triple) stylesButton.push(styles.buttonTriple);  
  if (props.operation) stylesButton.push(styles.operationButton);  
  return(  
    <TouchableHighlight onPress={props.onClick}>  
      <Text style={stylesButton}>{props.label}</Text>  
    </TouchableHighlight>  
  );  
}
```

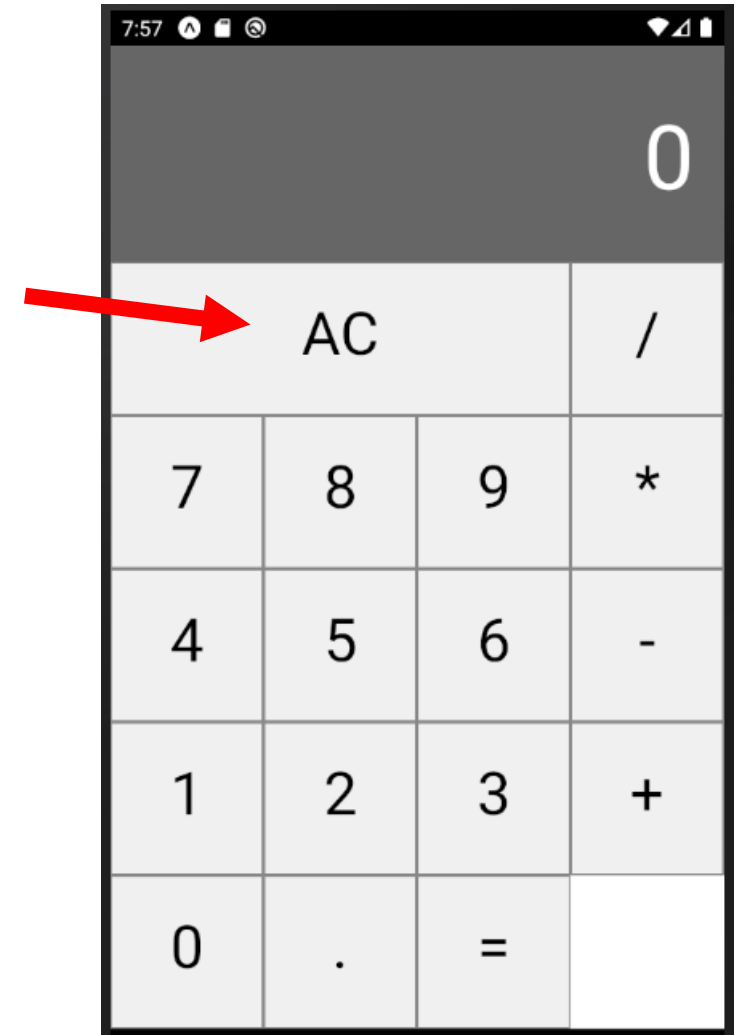
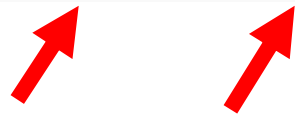


Volte ao componente “Index.js” e crie três funções. Sendo uma para adicionar um dígito no display. Uma para limpar a memória da calculadora. E uma para receber como parâmetro a operação que será realizada.

```
7   export default function Index() {  
8  
9     const [displayValue, setDisplayValue] = useState('0');  
10  
11     function addDigit(n) {  
12       setDisplayValue(n);  
13     }  
14  
15     function clearMemory() {  
16       setDisplayValue('0');  
17     }  
18  
19     function defineOperacao(operacao) {  
20       |  
21     }  
22  
23     return (  
24       <View style={styles.container}>
```

Agora vamos passar as formatações que determinados botões irão receber, além das funções respectivas.

```
<View style={styles.container}>  
  <Display value={displayValue} />  
  <View style={styles.buttons}>  
    <Button label='AC' triple onClick={() => clearMemory()} />
```



Aplique as configurações dos botões conforme apontado abaixo:

Para as operações e o números será necessário passar um parâmetro na chamada da função no evento “onClick”.

```
<View style={styles.container}>
  <Display value={displayValue} />
  <View style={styles.buttons}>
    <Button label='AC' triple onClick={() => clearMemory()} />
    <Button label='/' operation onClick={() => definieOperacao('/') } />
    <Button label='7' onClick={() => addDigit(7)} />
  </View>
</View>
```



Após configurar os botões AC, / e 7, já é possível testar e visualizar algumas alterações. Como a passagem do valor, o limpar e a cor da “/”.

7			
AC			/
7	8	9	*
4	5	6	-
1	2	3	+
0	.	=	

0			
AC			/
7	8	9	*
4	5	6	-
1	2	3	+
0	.	=	


Agora passe as configurações para os botões restantes:

```
<Button label='AC' triple onClick={() => clearMemory()} />
<Button label='/' operation onClick={() => definieOperacao('/')} />
<Button label='7' onClick={() => addDigit(7)} />
<Button label='8' onClick={() => addDigit(8)} />
<Button label='9' onClick={() => addDigit(9)} />
<Button label='*' operation onClick={() => definieOperacao('*')} />
<Button label='4' onClick={() => addDigit(4)} />
<Button label='5' onClick={() => addDigit(5)} />
<Button label='6' onClick={() => addDigit(6)} />
<Button label='-' operation onClick={() => definieOperacao('-')} />
<Button label='1' onClick={() => addDigit(1)} />
<Button label='2' onClick={() => addDigit(2)} />
<Button label='3' onClick={() => addDigit(3)} />
<Button label='+' operation onClick={() => definieOperacao('+')} />
<Button label='0' double onClick={() => addDigit(0)} />
<Button label='.' onClick={() => addDigit('.')} />
<Button label='=' operation onClick={() => definieOperacao('=')} />
```

0			
AC			/
7	8	9	*
4	5	6	-
1	2	3	+
0		.	=

Observe que tanto nas operações quanto nos números, passamos o mesmo valor do “label” como parâmetro. Dessa forma, poderíamos fazer uma alteração para passar o valor automaticamente.

Vá em “Button.js” e altere a linha apontada conforme o exemplo abaixo:



```
<TouchableHighlight onPress={() => props.onClick(props.label)}>  
  <Text style={styles.Button}>{props.label}</Text>  
</TouchableHighlight>
```

Volte ao componente “Index.js” e em todos os números, altere o código conforme apontado abaixo:

```
<Button label='7' onClick={addDigit} />
```

A red bracket is drawn above the code, starting from the opening curly brace of the onClick prop and ending at the closing curly brace, pointing to the text addDigit.

Aplique este novo recurso em todos os números e nas operações (exceto o botão “AC”).

```
<Button label='AC' triple onClick={() => clearMemory()} />
<Button label='/' operation onClick={definieOperacao} />
<Button label='7' onClick={addDigit} />
<Button label='8' onClick={addDigit} />
<Button label='9' onClick={addDigit} />
<Button label='*' operation onClick={definieOperacao} />
<Button label='4' onClick={addDigit} />
<Button label='5' onClick={addDigit} />
<Button label='6' onClick={addDigit} />
<Button label='-' operation onClick={definieOperacao} />
<Button label='1' onClick={addDigit} />
<Button label='2' onClick={addDigit} />
<Button label='3' onClick={addDigit} />
<Button label='+' operation onClick={definieOperacao} />
<Button label='0' double onClick={addDigit} />
<Button label='.' onClick={addDigit} />
<Button label='=' operation onClick={definieOperacao} />
```

# Lógica da calculadora

Ao utilizar uma calculadora, quando digitamos um número ele vai sendo incrementado conforme digitamos, ao selecionar uma operação, este número fica armazenado na memória, junto com a operação que será realizada e o usuário tem que digitar o próximo número que fará parte da operação. Se inserirmos o número 10, clicar no sinal de “+” e depois o número 5, e na sequência clicar no sinal de “-”, o resultado será armazenado em memória e apresentado, e o próximo número digitado será subtraído. Ao pressionar o “=” a operação é encerrada e o resultado é apresentado, quando digitamos outro número uma nova operação será iniciada.

# Lógica da calculadora

No componente “Index.js” será necessário armazenar alguns estados (state), um será o valor que irá aparecer no display onde teremos também um array que em seu primeiro elemento terá o valor digitado, no segundo elemento do array será armazenado o novo número que faz parte da operação e uma variável para armazenar a operação. Quando clicamos em um sinal de outra operação, o cálculo é realizado, o array é resetado e o resultado é passado para a primeira posição do array e o novo sinal de operação é armazenado. Quando clicamos no “=” a operação tem o resultado mostrado no display na primeira posição do array.

Em “Index.js” adicione os “states” apontados abaixo onde será criado um estado que servira para iniciar a calculadora e zerar os valores quando necessário com o “clearDisplay” que identifica se o display precisa ser limpo ao digitar um novo dígito ou não. Outro para armazenar a operação. Um array de valores com o nome que vai armazenar até dois valores para a realização de uma operação entre eles. O último que irá apontar qual dos índices do array está sendo utilizado.

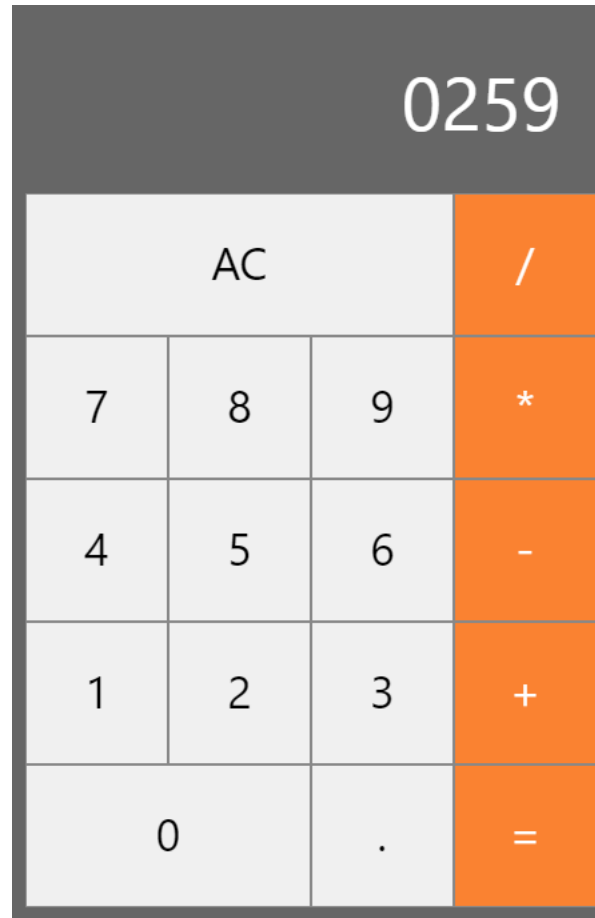
```
7   export default function Index() {  
8  
9     const [displayValue, setDisplayValue] = useState('0');  
10    {  
11      const [clearDisplay, setClearDisplay] = useState(false);  
12      const [operation, setOperation] = useState(null);  
13      const [values, setValues] = useState([0, 0]);  
      const [current, setCurrent] = useState(0);  
    }
```



Voltando a função “addDigit” iremos tratar para que mais de um dígito seja apresentado no display por vez. Começamos fazendo uma verificação no valor de “clearDisplay”, caso seja verdadeiro significa que iremos limpar a tela e uma String vazia será passada para a constante criada, senão o valor do display será passado. Em seguida atualizamos o display concatenando o valor atual com o do botão clicado, na terceira linha atualizamos o state “clearDisplay” para “false”, garantindo que ao passar um número para o display, este pronto para receber o próximo.

```
function addDigit(n) {  
  const currentValue = clearDisplay ? '' : displayValue;  
  setDisplayValue(currentValue + n);  
  setClearDisplay(false);  
}
```

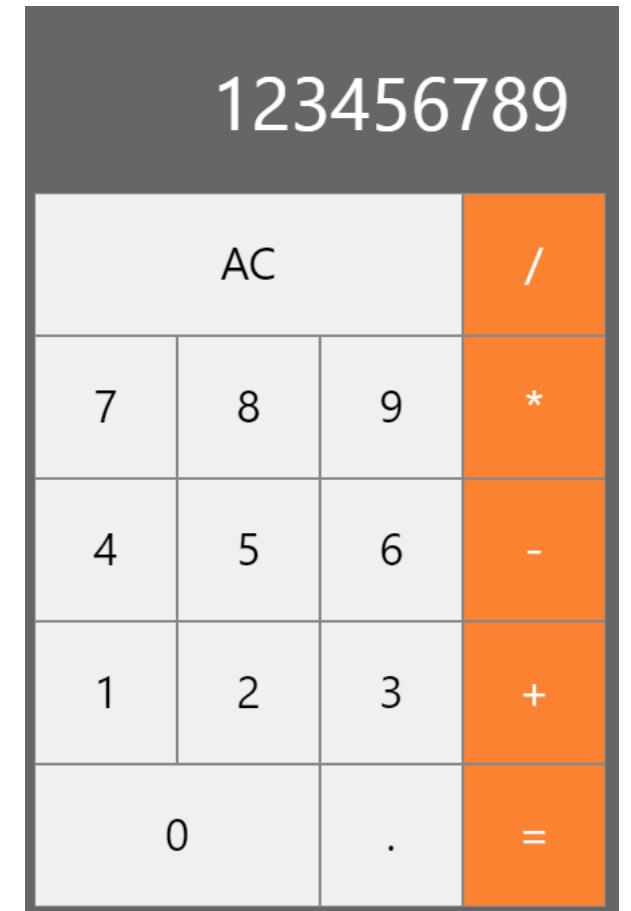
Ao testar o programa veremos que é possível inserir mais de um número no display, entretanto algo que não é comum acontecer em uma calculadora, é o zero ser mantido.



Adicione a linha apontada, a qual vai verificar se o valor atual do “displayValue” é zero ou se “clearDisplay” é verdadeiro, caso uma dessas opções seja verdadeira o valor de “clearDisplay” será “true”.

Quando o valor de “clearDisplay” é verdadeiro o valor do display será apagado, dando lugar ao novo valor a ser digitado.

```
function addDigit(n) {  
  → const limpaTela = displayValue === '0' || clearDisplay;  
  
  const currentValue = limpaTela ? '' : displayValue;  
  setDisplayValue(currentValue + n);  
  setClearDisplay(false);  
}
```



Ainda testando podemos identificar que a calculadora permite a inserção de vários pontos e não podemos deixar que este sinal seja inserido mais de uma vez na calculadora, realize as alterações apontadas a seguir para corrigir esta falha.

2...3.			
AC			/
7	8	9	*
4	5	6	-
1	2	3	+
0		.	=

Com essa instrução verificamos se o caractere a ser inserido é um ponto, se o valor de “clearDisplay” é falso e se já existe algum ponto entre os caracteres existentes em “clearDisplar”, caso as três expressões sejam verdadeiras, acionamos o comando “return” da função, para que esta seja interrompida neste momento e não seja possível inserir um ponto repetido no display.

```
function addDigit(n) {  
    const limpaTela = displayValue === '0' || clearDisplay;  
  
    {  
        if (n === '.' && !limpaTela && displayValue.includes('.')) {  
            return;  
        }  
    }  
  
    const currentValue = limpaTela ? '' : displayValue;  
    setDisplayValue(currentValue + n);  
    setClearDisplay(false);  
}
```

Depois temos que verificar se inserimos um dígito diferente de “.” , quando isso ocorrer armazenamos o valor do display convertido em número de ponto flutuante em uma variável, depois clonamos o array que temos em memória para um novo array que pode ser modificado,

Em seguida neste novo array passamos o novo valor na posição corrente.

E no final atualizamos o valor do array do state.

```
function addDigit(n) {  
  
    const limpaTela = displayValue === '0' || clearDisplay;  
  
    if (n === '.' && !limpaTela && displayValue.includes('.')) {  
        return;  
    }  
  
    const currentValue = limpaTela ? '' : displayValue;  
    setDisplayValue(currentValue + n);  
    setClearDisplay(false);  
  
    if (n !== '.') {  
        const novoValor = parseFloat(displayValue);  
        const atualizaValor = values;  
        atualizaValor[current] = novoValor;  
        setValues(atualizaValor);  
    }  
}
```

# Corrigindo a calculadora

O display é a ultima coisa que deve ser atualizada, caso contrário poderíamos ter um atraso nos valores apresentados, pois a ultima alteração de state não interfere na renderização.

Insira os códigos apontados ao lado para evitar este problema.

```
function addDigit(n) {  
  
    const limpaTela = displayValue === '0' || clearDisplay;  
  
    if (n === '.' && !limpaTela && displayValue.includes('.')) {  
        return;  
    }  
  
    const currentValue = limpaTela ? '' : displayValue;  
    → const valorAtualizar = currentValue + n;  
  
    if (n !== '.') {  
        const novoValor = parseFloat(valorAtualizar);  
        const atualizaValor = values;  
        atualizaValor[current] = novoValor;  
        setValues(atualizaValor);  
    }  
  
    {  
        setDisplayValue(valorAtualizar);  
        setClearDisplay(false);  
    }  
}
```

Na função “clearMemory” iremos restaurar o estado inicial da calculadora.

```
function clearMemory() {  
  setDisplayValue('0');  
  setClearDisplay(false);  
  setOperation(null);  
  setValues([0, 0]);  
  setCurrent(0);  
}
```

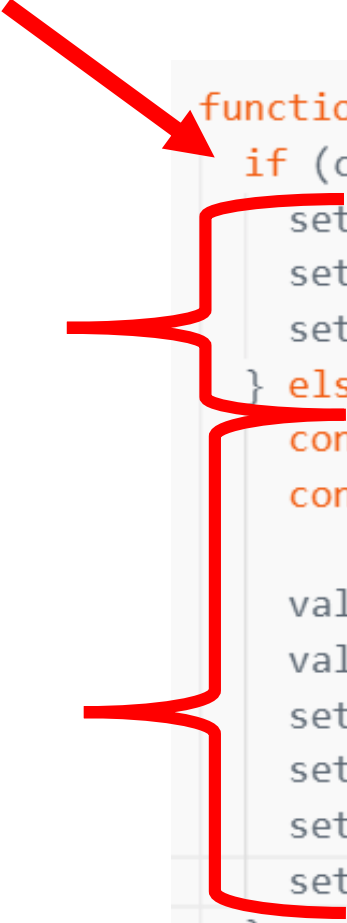


Quando apontamos uma operação mudamos o valor de current e o elemento do array que irá receber o valor.

Se o valor de “current” é igual a 0, ou seja quando selecionamos a operação mudamos o current, além de resetar o display. Assim o próximo numero digitado será armazenado na segunda posição do array.

Se não for o primeiro valor é necessário tratar a operação que será realizada, além de armazenar o valor na segunda posição do array.

## Tratando as operações



```
function defineOperacao(operacao) {  
  if (current === 0) {  
    setOperation(operacao);  
    setCurrent(1);  
    setClearDisplay(true);  
  } else {  
    const sinalIgual = operacao === '=';  
    const valores = values;  
  
    valores[0] = eval(valores[0] + operation + valores[1]);  
    valores[1] = 0;  
    setDisplayValue(values[0].toString());  
    setCurrent(sinalIgual ? 0 : 1);  
    setClearDisplay(true);  
    setValues(valores);  
  }  
}
```

Se a operação for de “=”  
teremos um comportamento  
tratado de forma diferente,  
por isso é importante saber  
quando o “=” foi clicado, se a  
operação selecionada for esta  
a variável “sinalIgual” recebe  
o valor verdadeiro.

Na sequência passamos o  
valor do “state” para uma  
constante, pois nela  
podemos manipular os  
valores de cada posição do  
array.

## Tratando as operações


```
function defineOperacao(operacao) {  
  if (current === 0) {  
    setOperation(operacao);  
    setCurrent(1);  
    setClearDisplay(true);  
  } else {  
    const sinalIgual = operacao === '=';  
    const valores = values;  
  
    valores[0] = eval(valores[0] + operation + valores[1]);  
    valores[1] = 0;  
    setDisplayValue(values[0].toString());  
    setCurrent(sinalIgual ? 0 : 1);  
    setClearDisplay(true);  
    setValues(valores);  
  }  
}
```

Com o valor da posição 1 do array, a operação e a 2ª posição do array, utilizamos o comando “eval” que serve para avaliar a sentença (expressão) e tratar a operação, assim temos um comando que funciona com qualquer operação.

O resultado será armazenado na posição 0 do array de valores.

## Tratando as operações


```
function defineOperacao(operacao) {  
  if (current === 0) {  
    setOperation(operacao);  
    setCurrent(1);  
    setClearDisplay(true);  
  } else {  
    const sinalIgual = operacao === '=';  
    const valores = values;  
  
    valores[0] = eval(valores[0] + operation + valores[1]);  
    valores[1] = 0;  
    setDisplayValue(values[0].toString());  
    setCurrent(sinalIgual ? 0 : 1);  
    setClearDisplay(true);  
    setValues(valores);  
  }  
}
```



É necessário zerar o valor da posição dois do array, pois se continuarmos a inserir números, estes serão armazenados na segunda posição do array, permitindo que uma nova instrução de operação seja passada.

## Tratando as operações


```
function defineOperacao(operacao) {  
  if (current === 0) {  
    setOperation(operacao);  
    setCurrent(1);  
    setClearDisplay(true);  
  } else {  
    const sinalIgual = operacao === '=';  
    const valores = values;  
  
    valores[0] = eval(valores[0] + operation + valores[1]);  
    valores[1] = 0;  
    setDisplayValue(values[0].toString());  
    setCurrent(sinalIgual ? 0 : 1);  
    setClearDisplay(true);  
    setValues(valores);  
  }  
}
```



No state referente a operação se foi clicado no igual o valor da operação será nulo e se não o valor será o da operação que está armazenada no parâmetro desta função onde apontamos qual será a próxima operação que será executada.

## Tratando as operações

```
function defineOperacao(operacao) {  
  if (current === 0) {  
    setOperation(operacao);  
    setCurrent(1);  
    setClearDisplay(true);  
  } else {  
    const sinalIgual = operacao === '=';  
    const valores = values;  
  
    valores[0] = eval(valores[0] + operation + valores[1]);  
    valores[1] = 0;  
    setDisplayValue(values[0].toString());  
    setCurrent(sinalIgual ? 0 : 1);  
    setClearDisplay(true);  
    setValues(valores);  
  }  
}
```



Se clicou no botão “igual” o valor corrente irá receber o índice 0 do array, se não apontará no índice 1.

Sempre que selecionamos uma operação iremos limpar o display, o resultado é apresentado e o próximo valor inserido fará parte de outra operação.

Por fim atualizamos o array de valores.

## Tratando as operações

```
function defineOperacao(operacao) {  
  if (current === 0) {  
    setOperation(operacao);  
    setCurrent(1);  
    setClearDisplay(true);  
  } else {  
    const sinalIgual = operacao === '=';  
    const valores = values;  
  
    valores[0] = eval(valores[0] + operation + valores[1]);  
    valores[1] = 0;  
    setDisplayValue(values[0].toString());  
    setCurrent(sinalIgual ? 0 : 1);  
    setClearDisplay(true);  
    setValues(valores);  
  }  
}
```

# Atividade

Agora que a calculadora está Ok arrume o seguinte problema: Ao iniciar um valor com “.” faça com que apareça um zero antes do mesmo automaticamente. Teste as operações para garantir que continuam funcionando após a implementação.

			.1
AC			/
7	8	9	*
4	5	6	-
1	2	3	+
0	.	=	

			0.1
AC			/
7	8	9	*
4	5	6	-
1	2	3	+
0	.	=	