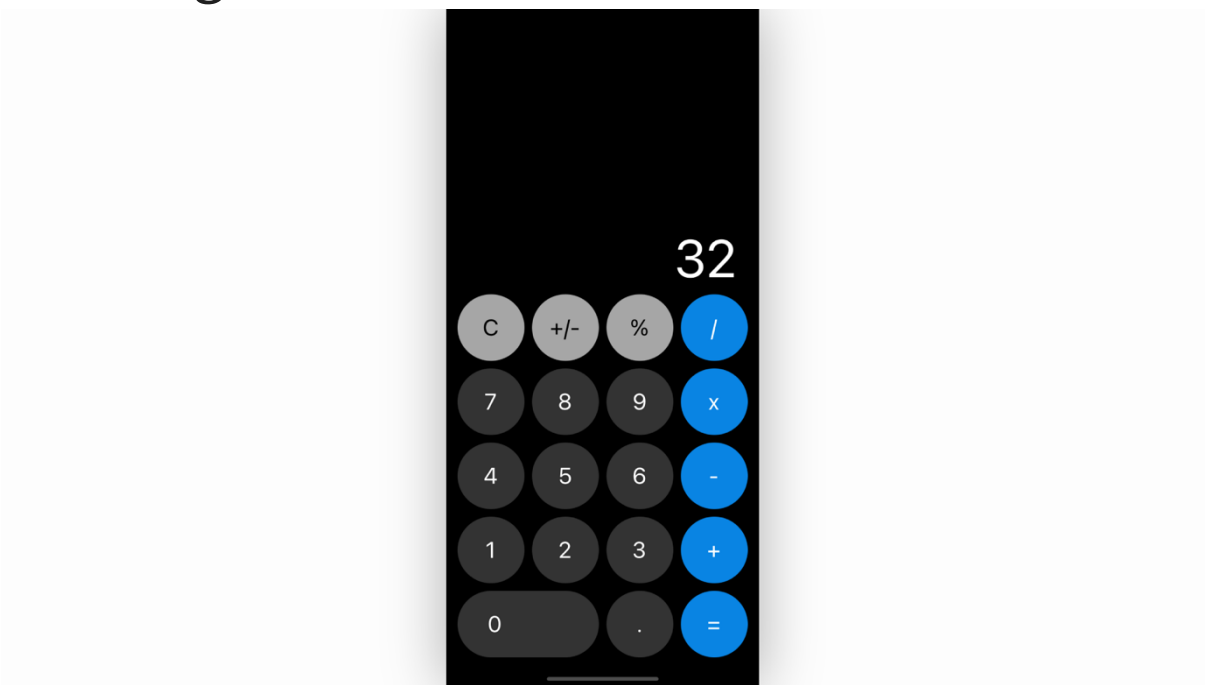
Creating a calculator



In this assignment, we are going to make a fully functioning calculator which mimics the style of iOS's default calculator, which is actually styled on the Braun ET66 pocket calculator (1987).

For this graded assessment, I'm giving you a guide through the development process, but leaving the specifics to you. We're covering everything that we have looked at so far, so if you need a refresher, don't hesitate to go back and re-watch a few videos.

How do I submit this?

This graded assessment is one of five graded assessments which will make up your midterm assignment submitted in week 10. I strongly advise you to complete it now as it is crucial to expanding your understanding of the subsequent weeks. Once you have completed it store it somewhere safe and do not share it with others. Instead, you can upload it with the other assignments during week 10.

Debugging

Throughout the steps to complete are "***Check:***" moments. These describe what should be visible if you run the application at this step. Make sure to complete these, if they are not correct you should pause and debug before continuing.

Steps to complete:

Firstly, we will be using App as a functional component with this demo, what this means is you don't need to create a class extending App, instead we will just generate a blank project and start adding to it.

So first, set up a new React Native project using expo. Refer back to the setup instructions earlier in this module if you have forgotten how to do this.

(1) Import required libraries

- Import useState from react, as we'll need it to store our calculator's state
- Import Dimensions, SafeAreaView and TouchableOpacity from react-native

(2) Style the application:

- Set the background colour of the container as black
- Set the status bar to be 'light content' so that we can see the phone's status bar on the black background.
- Change the container styling to justify contents so that all child elements align to the bottom of the container vertically

(3) Device accommodations

- Wrap the contents of the outer container View in a SafeAreaView so that we can avoid drawing under any 'notch' in the display. You will want to apply the same container styles to the SafeAreaView
- Remove any text elements added from the template project. Keep the Status Bar though.

(4) Results field

- Create a Text element that is drawn **within** the 'main' View, at the beginning. For now, add a value of '0' to the Text element.
- Style the Text element so that the text element matches the example screenshot. You'll need to change the colour, font size, margins and text alignment.
- **Check:** When you run the application you should have a white zero at the bottom of the screen.

(5) Rows of buttons

- Create a new View below our Results field we'll call this a 'row'.
- Apply styling to this 'row' so that the child elements are in line with each other horizontally.
- Inside of this 'row' add a TouchableOpacity. Style this TouchableOpacity so that it has a light grey background and appropriate margins.
- Next, we need to set the width of the button style. To do this we need to calculate a value based on the width of the screen. Define a new const in the global space which defines the width of buttons based on the correct proportion of the screen width. Update the button styling to use this height.
- We also want the buttons to be circular, so set the corner radius to a value that makes the ends of the buttons perfectly circular.
- Inside of this TouchableOpacity, add a Text element, with the value of "C". Style this text element so that the font size is correct and the colour white.
- **Check:** You should now see a large rounded button with value C spanning the entire bottom portion of the screen, under the results field.

(6) Duplicate the buttons/rows

- Duplicate the touchable opacities so that you have multiple buttons in each 'row'. Alter the text to match the button text seen in the screenshot.
- Next, duplicate the rows so that there's five. Don't worry about the long button or colours, we'll sort all of these things next. Also, make sure to duplicate the buttons before the rows, it'll save you some work.
- Update the text elements so each button has the correct label.

(7) Alternate colour schemes + long button

- In the demo screenshot, the top row and side row has different colours. Create two new custom styles and apply them to the correct buttons so that it looks like the screenshot.

- For the long button, we want to apply a custom style to the first button to stretch it out. If you haven't already, delete one button so that the last row only has 3 buttons. Then apply a style to the first button which adjusts the width to be the correct value.

(8) Set up the state

- We need React to remember values so that we can make our calculator interactive. Firstly, create a new hook for *answerValue* and *setAnswerValue*, set the default state to 0.
- Set the results field to equal the *answerValue*
- **Check:** You should now see 0 in the results field.

(9) Press the buttons!

- We need to be able to press buttons, so set up a new function *buttonPressed* which takes one parameter *value*. For each of the TouchableOpacities trigger buttonPressed when the element is touched, pass its value as a parameter.
- Add a dummy alert to *buttonPressed* so that you can quickly check if everything is working as expected.
- **Check:** When you press any button you get the dummy alert.

(10) Let's make the calculators brain

- Inside button pressed, check if *value* is a number, if it is set the *answerValue* to the returns of a new function *handleNumber*. Remember to create that function.
- Next, create a new state element *readyToReplace* and *setReadyToReplace*, with the default as true. This dictates if we should replace what is on screen next time a number is pressed. So in *handleNumber*, check if *readyToReplace* is true, if so return the button value. This will cause the calculator to display the pressed button number. Also, add an else which simply appends the button value to the end of the *answerValue.*
- **Check:** When zero is present in the results and you press a number it should replace the value, then if you follow another number it should get appended to the end of the number.
- Create two new state elements, for *memoryValue* and *operatorValue*.
- Next, create the clear button. Create an if statement to check for the clear button being pressed and then set *answerValue*, *memoryValue* and *operatorValue* to 0. Additionally set *readyToReplace* to true. This resets everything.
- Create an if statement to detect if the button value is an operator. Within this statement set the *memoryValue* to the *answerValue.* Set *readyToReplace* to true and set the *operatorValue* to the button operator. All of this sets the calculator ready to perform the operation once we add another value and press equals.
- Create an if statement to detect if the button value is the equals sign. Inside that statement call a new function called *calculateEquals.* In this function define two new variables previous and current, containing the *memoryValue* and *answerValue* parsed as a float. Then use a switch operator to check if the *operatorValue* is of a specific type e.g. +, if so set the answer value to previous + current substituting the correct operator. Make sure to also return this value to stop the switch but also allow us to get the value when calling which will come in handy later.
- Under *calculateEquals()* inside your equals if statement set *memoryValue* to 0 and *readyToReplace* to true.
- Now we've got an issue. If you want to chain multiple calculations e.g. 5+5/2 this won't work. So let's fix this. Inside *buttonPressed*'s operator if statement, at the beginning check if the operator value is 0, if it is **not**, call *calculateEquals* and save its returns to a new local variable. Use this local variable to set the value of the *memoryValue* later in the statement. This chains the calculations!
- Last few buttons, let's add the functionality for +/-. For this, detect the button press, create an if statement and set the *answerValue* to be the positive/negative equivalent e.g. -5 becomes +5 and +5 becomes -5. Next, for the % functionality multiply the current value by 0.01.

(11) Now you have a functioning calculator, feel free to customise this calculator however you'd like. Perhaps you could also add more functionality, such as the following:

- Change the C button to AC whenever relevant
- Add indicators to the calculator to reflect the current operator stored
- Update the calculator to become a scientific calculator (e.g. it shows the mathematical equation when calculating - 5+5=10)

Mark scheme

Description	Marks
Styling	
No attempt made	0
Basic styling that loosely resembles the screenshots	1
Styling that is clearly influenced by the screenshot, but is missing elements	2
Styling that is identical to the screenshot	3
Unique additions to the styling	4
Advanced additions to the styling	5
UI	
No attempt made	0
The UI is partly constructed	1
The UI is mostly constructed	2
The UI is well constructed and represents the screenshot yet is approached incorrectly	3
The UI is well constructed and attempted correctly	4
The UI has been improved upon	5
Brain	
No attempt made	0
The brain is partly constructed, yet does not function	1
The brain partly functions	2
The brain is mostly constructed, yet doesn't function completely	3
The brain functions completely	4
The brain has been improved	5
Additions	
No attempt made	0
Some improvements have been made to the UI	1
Minor improvements to the UI and UX	2
Minor improvements to the UI, UX and brain function	3
Decent improvements to any of the above	4
Significant improvements to any of the above	5