**How to build a calculator - part 2**

**Published:** Mar 28, 2018

This is the second part of a three-part lesson about building a calculator. By the end of these three lessons, you should get a calculator that functions exactly like an iPhone calculator (without the +/- and percentage functionalities).

Note: please make sure you finish the [first part](https://zellwk.com/blog/calculator-part-1) before starting this article.

You’re going to learn to code for edge cases to make your calculator resilient to weird input patterns in this lesson.

To do so, you have to imagine a troublemaker who tries to break your calculator by hitting keys in the wrong order. Let’s call this troublemaker Tim.

Tim can hit these keys in any order:

1. A number key (0-9)
2. An operator key (+, -, ×, ÷)
3. The decimal key
4. The equal key
5. The clear key

What happens if Tim hits the decimal key

If Tim hits a decimal key when the display already shows a decimal point, nothing should happen.

Nothing happens when a user hits the decimal key when the display already shows a decimal pointNothing should happen even if the previous key isn't the decimal key

Here, we can check the displayed number contains a . with the includes method.

includes checks strings for a given match. If a string is found, it returns true; if not, it returns false. Note: includes is case sensitive

// Example of how includes work.

const string = 'The hamburgers taste pretty good!'

const hasExclaimation = string.includes('!')

console.log(hasExclaimation) // true

// Do nothing if string has a dot

if (!displayedNum.includes('.')) {

display.textContent = displayedNum + '.'

}

Next, if Tim hits the decimal key after hitting an operator key, the display should show 0..

Display should show `0.` if a user hits a decimal key after an operator key

Here we need to know if the previous key is an operator. We can tell by checking the the custom attribute, data-previous-key-type, we set in the previous lesson.

data-previous-key-type is not complete yet. To correctly identify if previousKeyType is an operator, we need to update previousKeyType for each clicked key.

if (!action) {

// ...

calculator.dataset.previousKeyType = 'number'

}

if (action === 'decimal') {

// ...

calculator.dataset.previousKeyType = 'decimal'

}

if (action === 'clear') {

// ...

calculator.dataset.previousKeyType = 'clear'

}

if (action === 'calculate') {

// ...

calculator.dataset.previousKeyType = 'calculate'

}

Once we have the correct previousKeyType, we can use it to check if the previous key is an operator.

if (action === 'decimal') {

if (!displayedNum.includes('.')) {

display.textContent = displayedNum + '.'

} else if (previousKeyType === 'operator') {

display.textContent = '0.'

}

calculator.dataset.previousKeyType = 'decimal'

}

What happens if Tim hits an operator key

First, if Tim hits an operator key first, the operator key should light up. (We’ve already covered for this edge case, but how? See if you can identify what we did).

Operator key should light up if it's the first key.

Second, nothing should happen if Tim hits the same operator key multiple times. (We’ve already covered for this edge case as well).

Note: if you want to provide better UX, you can show the operator getting clicked on again and again with some CSS changes. We didn’t do it here because I took recorded all the GIFs before I could fix that.

Operator key remains depressed if clicked on multiple times

Third, if Tim hits another operator key after hitting the first operator key, the first operator key should be released; the second operator key should be depressed. (We covered for this edge case too; but how?).

The new operator key should be depressed

Fourth, if Tim hits a number, an operator, a number and another operator, in that order, the display should be updated to a calculated value.

Clicking on the operator when numbers are stored in the calculator results in a calculation

This means we need to use the calculate function when firstValue, operator and secondValue exists.

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

// Note: It's sufficient to check for firstValue and operator because secondValue always exists

if (firstValue && operator) {

display.textContent = calculate(firstValue, operator, secondValue)

}

key.classList.add('is-depressed')

calculator.dataset.previousKeyType = 'operator'

calculator.dataset.firstValue = displayedNum

calculator.dataset.operator = action

}

Although we can calculate a value when the operator key is clicked for a second time, we have also introduced a bug at this point—additional clicks on the operator key calculates a value when it shouldn’t.

Bug: subsequent clicks on the operator performs a calculation when it shouldn't

To prevent the calculator from performing calculation on subsequent clicks on the operator key, we need to check if the previousKeyType is an operator; if it is, we don’t perform a calculation.

if (firstValue && operator && previousKeyType !== 'operator') {

display.textContent = calculate(firstValue, operator, secondValue)

}

Fifth, after the operator key calculates a number, if Tim hits on a number, followed by another operator, the operator should continue with the calculation, like this: 8 - 1 = 7, 7 - 2 = 5, 5 - 3 = 2.

Calculator should be able to continue calculation when a user clicks on numbers, followed by operators, followed by numbers, followed by operators, and so on.

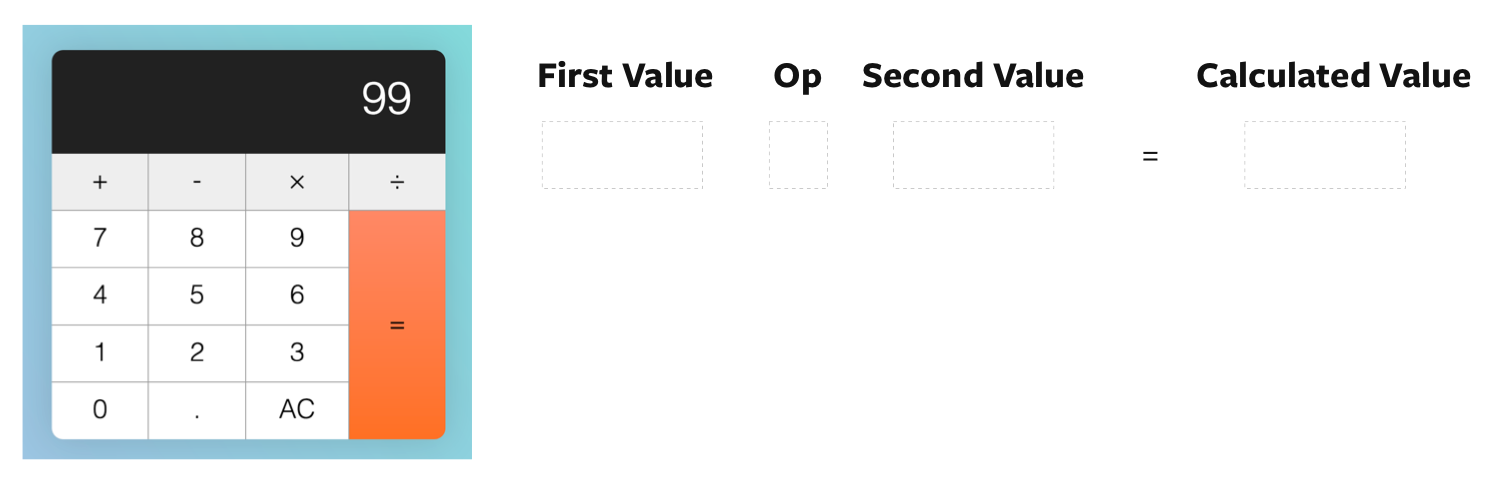
Right now, our calculator cannot make consecutive calculations. The second calculated value is wrong. Here’s what we have: 99 - 1 = 98, 98 - 1 = 0.

Calculated values are wrong. Second calculated value should be 97 instead of 0

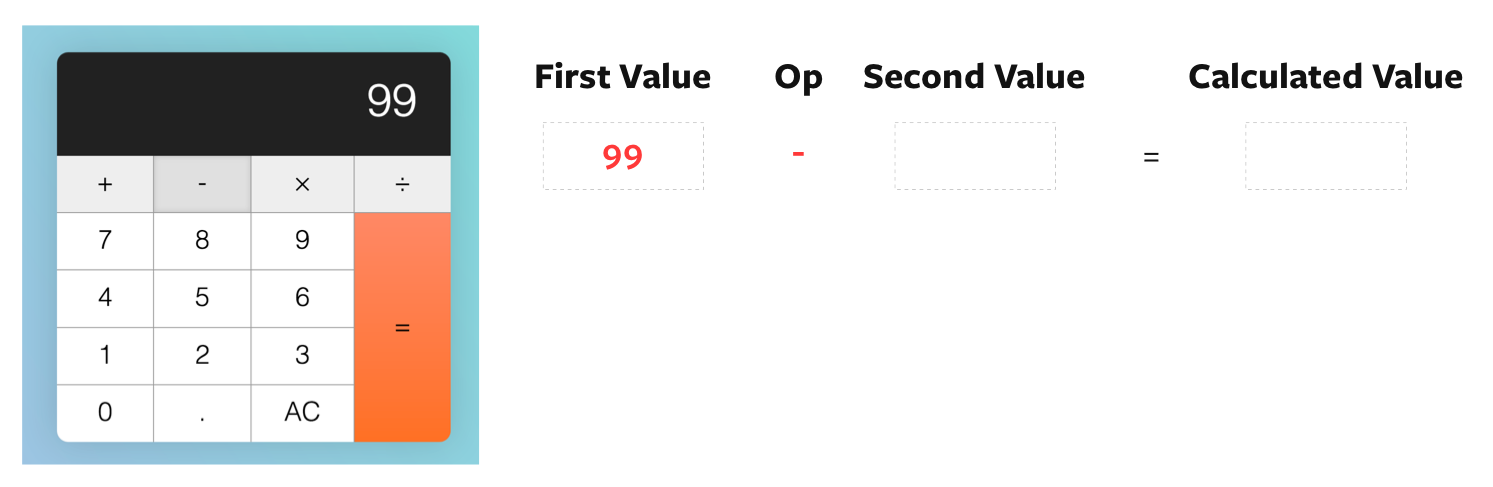
The second value is calculated wrongly because we fed the wrong values into the calculate function. Let’s go through a few pictures to understand what our code does.

**Understanding our calculate function**

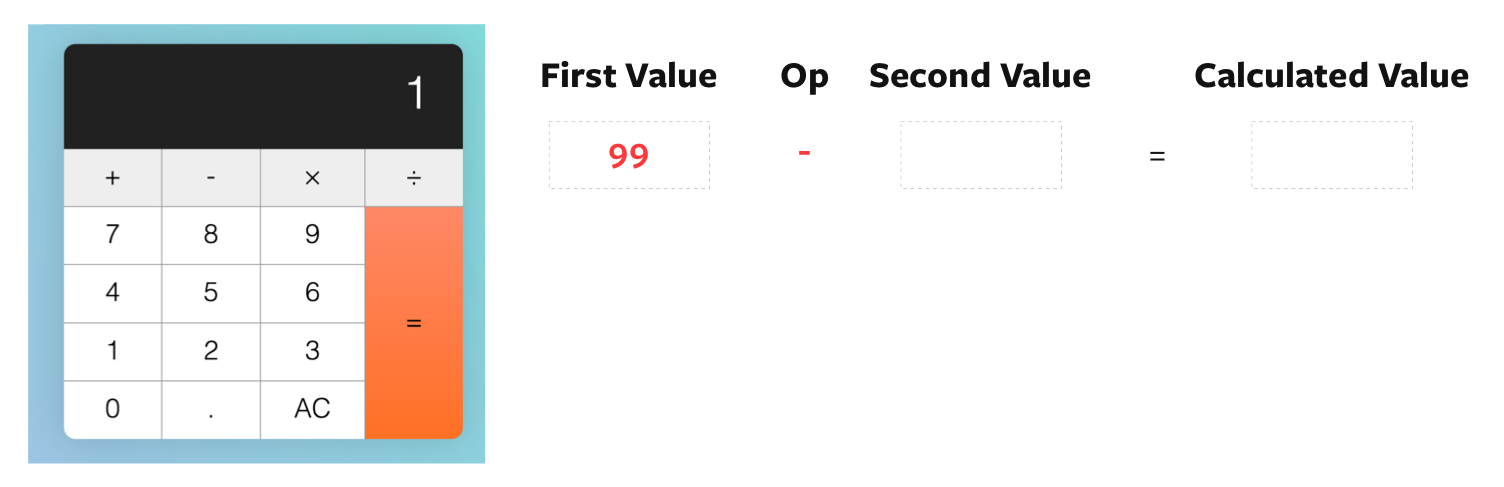
First, let’s say a user clicks on a number, 99. At this point, nothing is registered in the calculator yet.

When a user hits numbers, the calculator doesn't register `firstValue` or `operator`

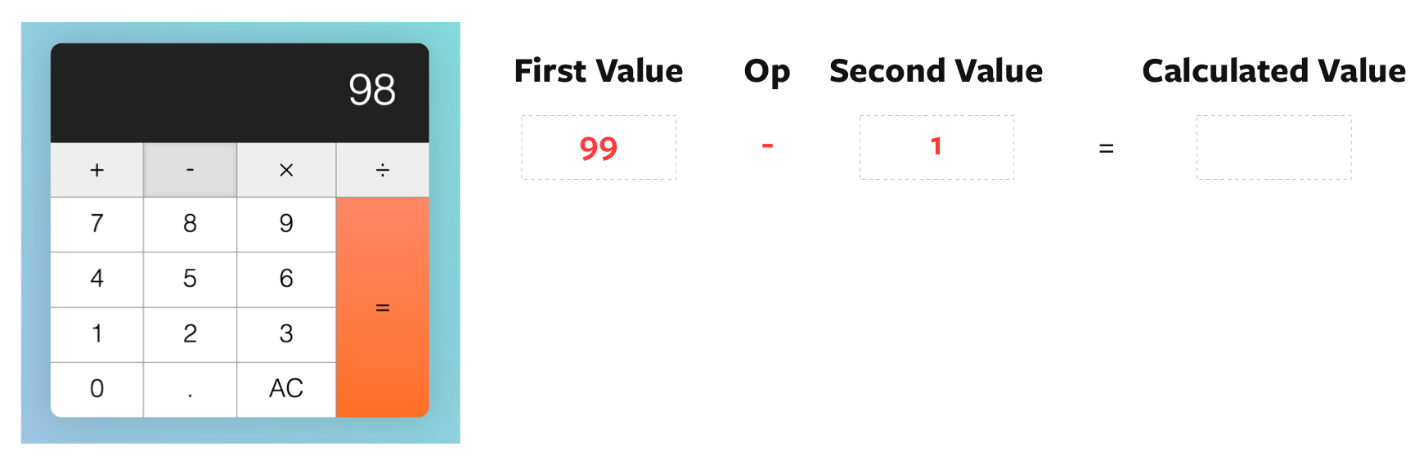
Second, let’s say the user clicks the subtract operator. After they click the subtract operator, we set firstValue to 99. We set also operator to subtract.

`firstValue` and `operator` are set after the operator button is clicked

Third, let’s say the user clicks on a second value; this time, it’s 1. At this point, the displayed number gets updated to 1, but our firstValue, operator and secondValue remains unchanged.

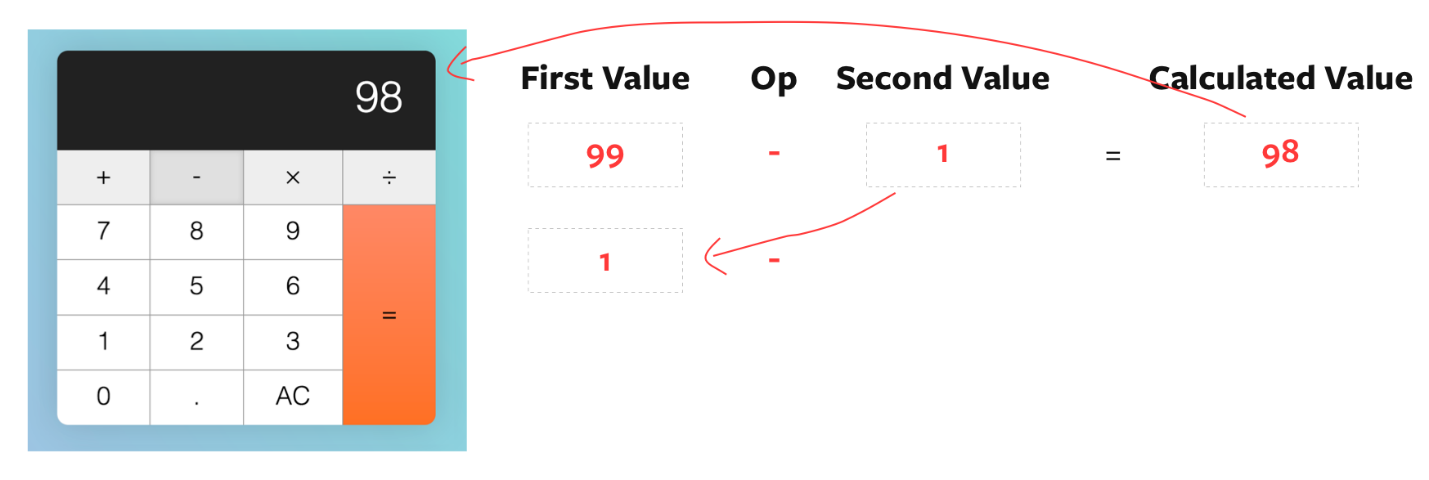
Display updates to 1, but `firstValue` and `operator` remains at `99` and `subtract`

Fourth, the user clicks on subtract again. Right after they click subtract, before we calculate the result, we set secondValue as the displayed number.

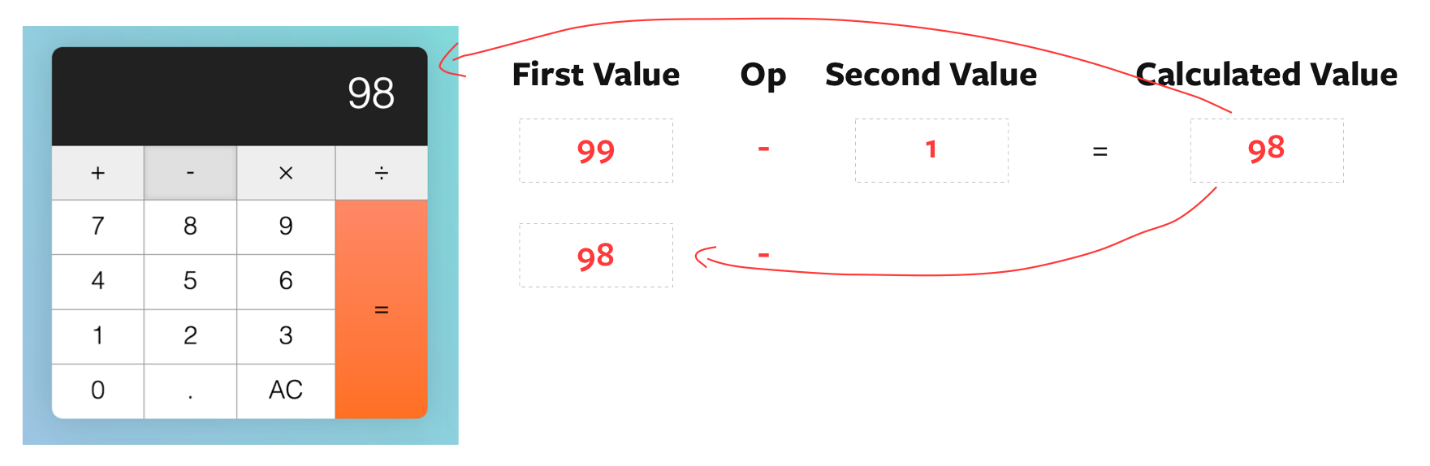
We set `secondValue` to 1

Fifth, we perform the calculation with firstValue 99, operator subtract, and secondValue 1. The result is 98.

Once the result is calculated, we set the display to the result. Then, we set operator to subtract, and firstValue to the previous displayed number.

After calculation, firstValue is set to `displayedNum`

Well, that’s terribly wrong! If we want to continue with the calculation, we need to update firstValue with the calculated value.

updates calculated value as `firstValue`

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

if (firstValue && operator && previousKeyType !== 'operator') {

const calcValue = calculate(firstValue, operator, secondValue)

display.textContent = calcValue

// Update calculated value as firstValue

calculator.dataset.firstValue = calcValue

} else {

// If there are no calculations, set displayedNum as the firstValue

calculator.dataset.firstValue = displayedNum

}

key.classList.add('is-depressed')

calculator.dataset.previousKeyType = 'operator'

calculator.dataset.operator = action

With this fix, consecutive calculations done by operator keys should now be correct.

Consecutive calculations done with the operator key is now correct

What happens if Tim hits the equal key?

First, nothing should happen if Tim hits the equal key before any operator keys,

Calculator should show zero if equal key is hit firstWhen no calculation is required, display remains the same

We know that operator keys have not been clicked yet if firstValue is not set to a number. We can use this knowledge to prevent the equal from calculating.

if (action === 'calculate') {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

if (firstValue) {

display.textContent = calculate(firstValue, operator, secondValue)

}

calculator.dataset.previousKeyType = 'calculate'

}

Second, if Tim hits a number, followed by an operator, followed by a equal, the calculator should calculate the result such that:

1. 2 + = —> 2 + 2 = 4
2. 2 - = —> 2 - 2 = 0
3. 2 × = —> 2 × 2 = 4
4. 2 ÷ = —> 2 ÷ 2 = 1

The calculator should treat first and second values as the same numbers if it's missing a value

We have already taken this weird input into account. Can you understand why? :)

Third, if Tim hits the equal key after a calculation is completed, another calculation should be performed again. Here’s how the calculation should read:

1. Tim hits key 5 - 1
2. Tim hits equal. Calculated value is 5 - 1 = 4
3. Tim hits equal. Calculated value is 4 - 1 = 3
4. Tim hits equal. Calculated value is 3 - 1 = 2
5. Tim hits equal. Calculated value is 2 - 1 = 1
6. Tim hits equal. Calculated value is 1 - 1 = 0

When a user hits the equal key multiple times, the calculator should continue to calculate

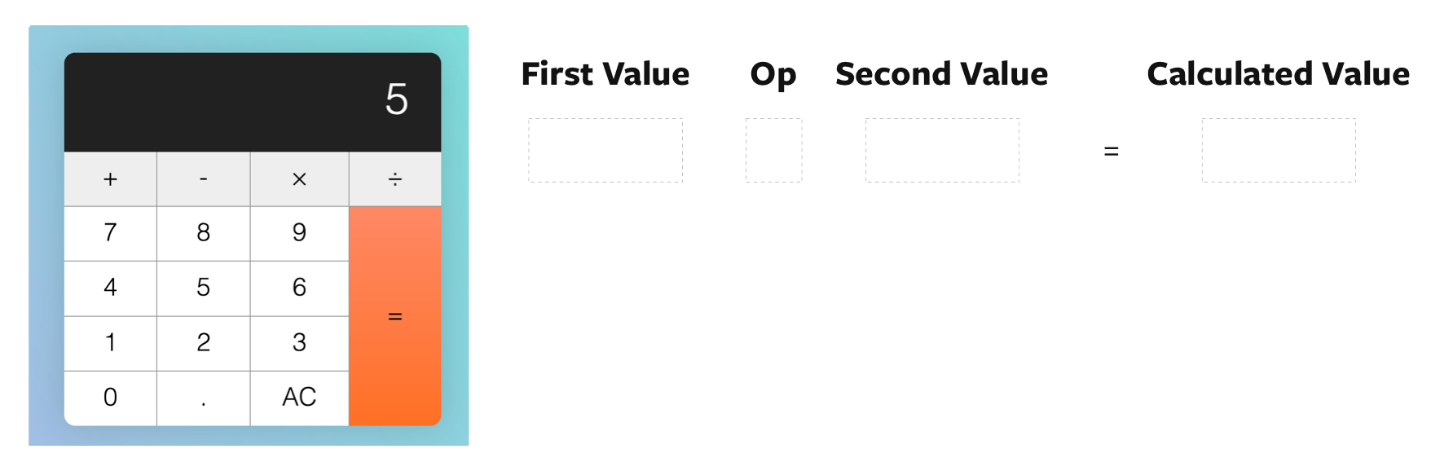
Unfortunately, our calculator messes this calculation up. Here’s what our calculator shows:

1. Tim hits key 5 - 1
2. Tim hits equal. Calculated value is 4
3. Tim hits equal. Calculated value is 1

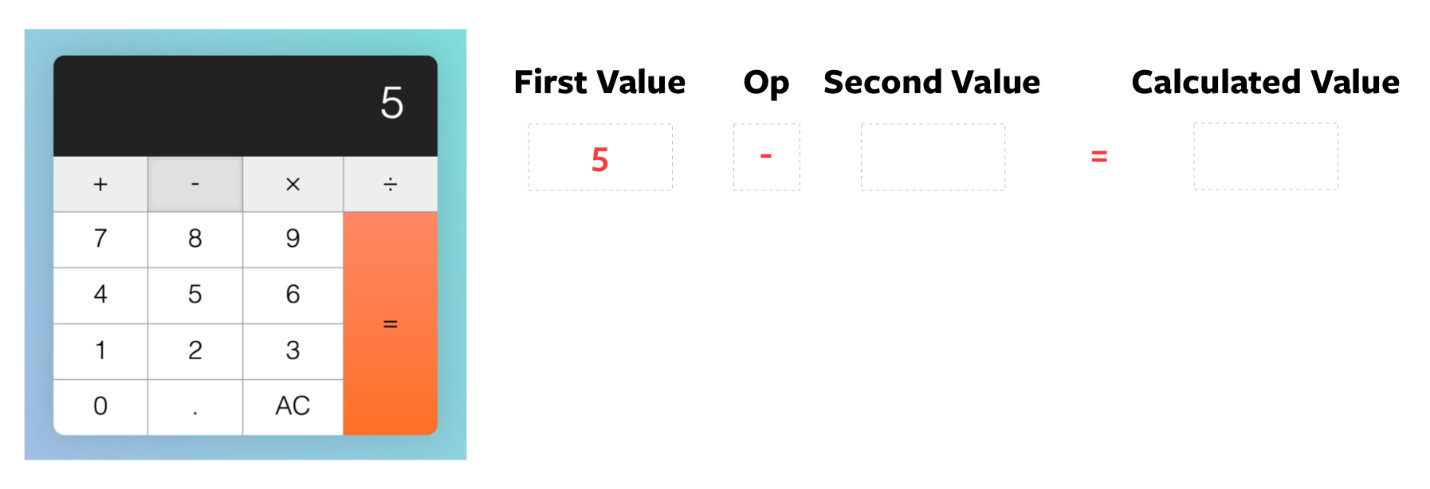
Equal key consecutive calculation gives a wrong result

**Correcting the calculation**

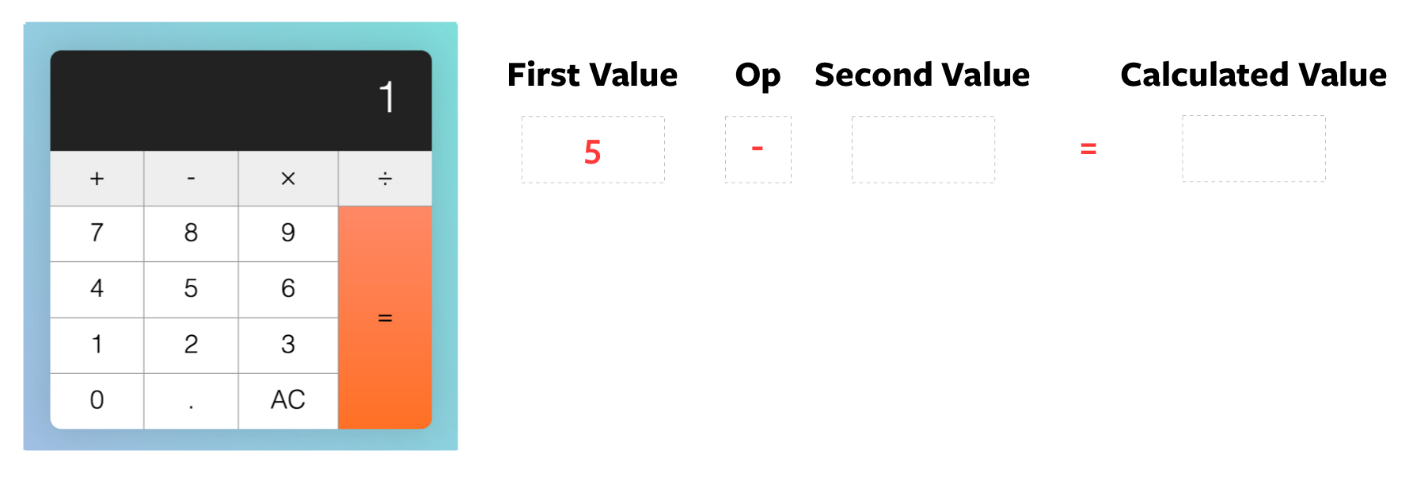
First, let’s say our user we clicks 5. At this point, nothing is registered in the calculator yet.

When a user clicked on the first number the calculator doesn't register `firstValue` or `operator`

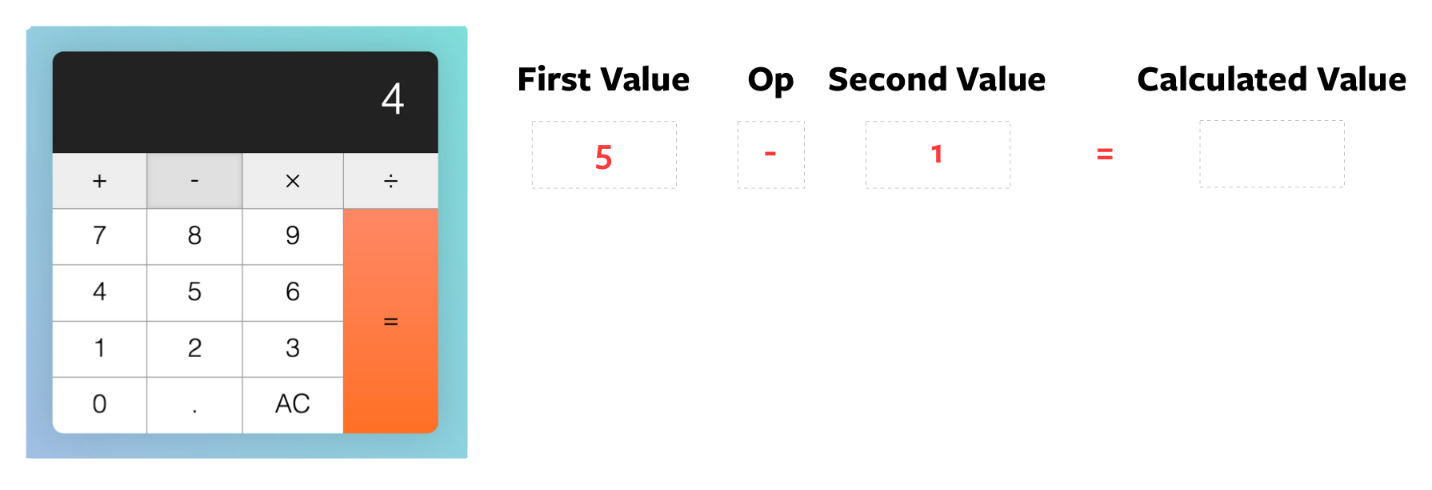
Second, let’s say the user clicks the subtract operator. After they click the subtract operator, we set firstValue to 5. We set also operator to subtract.

`firstValue` and `operator` are set after the operator button is clicked

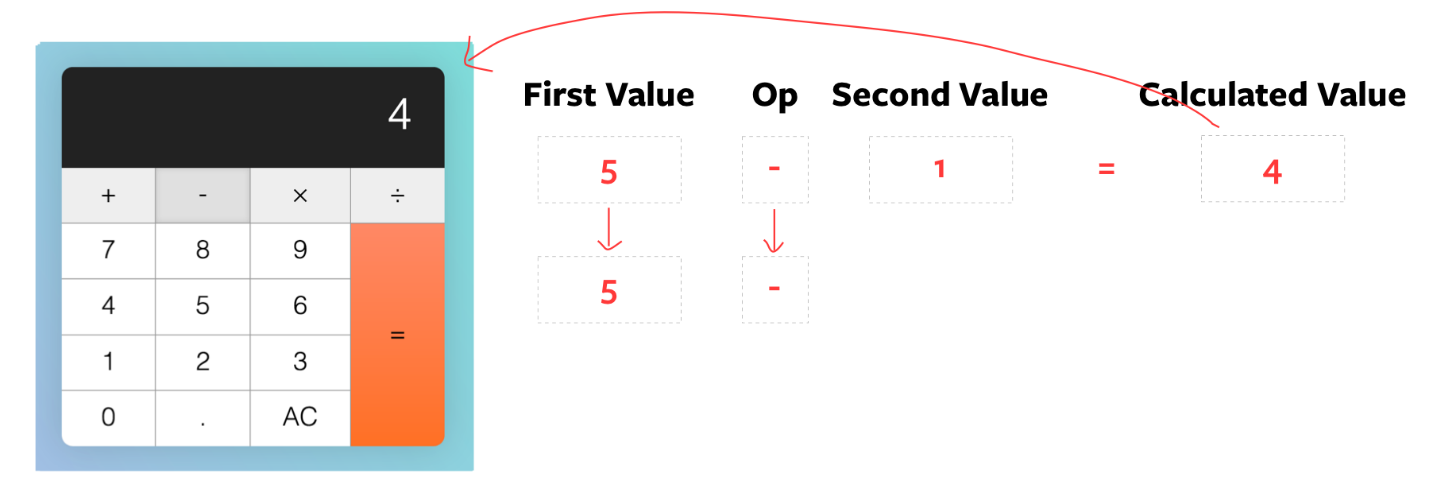
Third, the user clicks on a second value. Let’s say it’s 1. At this point, the displayed number gets updated to 1, but our firstValue, operator and secondValue remains unchanged.

Display updates to 1, but `firstValue` and `operator` remains at `5` and `subtract`

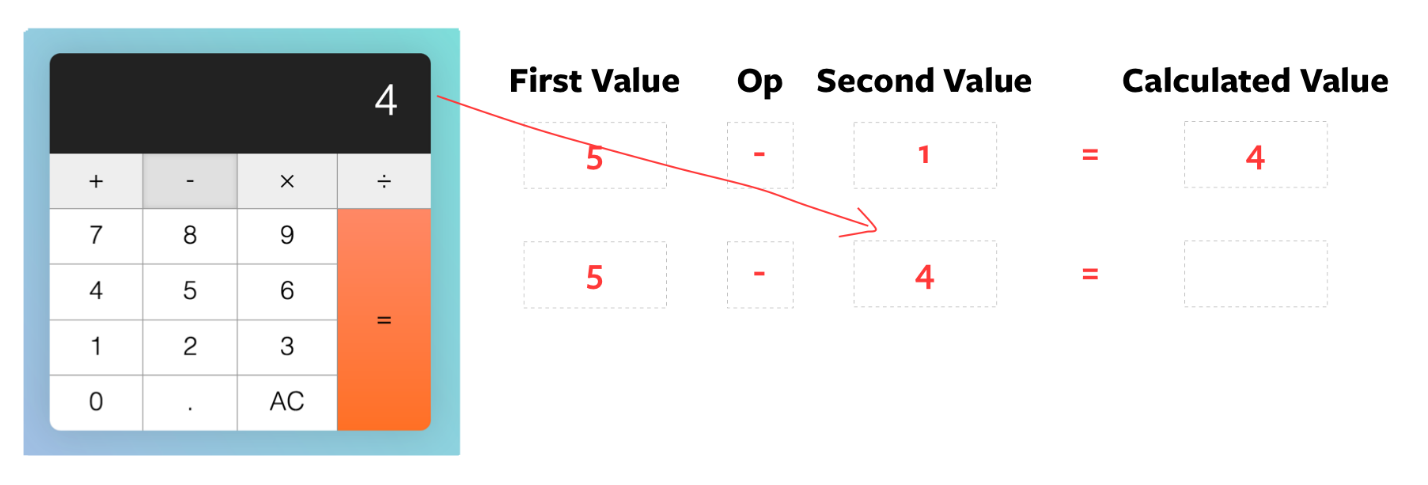
Fourth, the user clicks the equal key. Right after they click equal, but before the calculation, we set secondValue as displayedNum

We set `secondValue` as `displayedNum`

Fifth, the calculator calculates the result of 5 - 1 and gives 4. The result gets updated to the display. firstValue and operator gets carried forward to the next calculation since we did not update them.

`firstValue` and `operator` are used for the next operation

Sixth, when the user hits equal again, we set secondValue to displayedNum before the calculation.

Once again, displayed num is set as the `secondValue` before the calculation

You can tell what’s wrong here.

Instead of secondValue, we want the set firstValue to the displayed number.

if (action === 'calculate') {

let firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

}

display.textContent = calculate(firstValue, operator, secondValue)

}

calculator.dataset.previousKeyType = 'calculate'

}

We also want to carry forward the previous secondValue into the new calculation. For secondValue to persist to the next calculation, we need to store it in another custom attribute. Let’s call this custom attribute modValue (stands for modifier value).

if (action === 'calculate') {

let firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

}

display.textContent = calculate(firstValue, operator, secondValue)

}

// Set modValue attribute

calculator.dataset.modValue = secondValue

calculator.dataset.previousKeyType = 'calculate'

}

If the previousKeyType is calculate, we know we can use calculator.dataset.modValue as secondValue. Once we know this, we can perform the calculation.

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

secondValue = calculator.dataset.modValue

}

display.textContent = calculate(firstValue, operator, secondValue)

}

With that, we have the correct calculation when the equal key is clicked consecutively.

Consecutive calculations made by the equal key is now fixed

**Back to the equal key**

Fourth, if Tim hits a decimal key or a number key after the calculator key, the display should be replaced with 0. or the new number respectively.

Here, instead of just checking if the previousKeyType is operator, we also need to check if it’s calculate.

if (!action) {

if (

displayedNum === '0' ||

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

) {

display.textContent = keyContent

} else {

display.textContent = displayedNum + keyContent

}

calculator.dataset.previousKeyType = 'number'

}

if (action === 'decimal') {

if (!displayedNum.includes('.')) {

display.textContent = displayedNum + '.'

} else if (

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

) {

display.textContent = '0.'

}

calculator.dataset.previousKeyType = 'decimal'

}

Fifth, if Tim hits an operator key right after the equal key, calculator should NOT calculate.

Operator keys should not perform calculations if they're clicked after the equal key

To do this, we check if the previousKeyType is calculate before performing calculations with operator keys.

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

// ...

if (

firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

) {

const calcValue = calculate(firstValue, operator, secondValue)

display.textContent = calcValue

calculator.dataset.firstValue = calcValue

} else {

calculator.dataset.firstValue = displayedNum

}

// ...

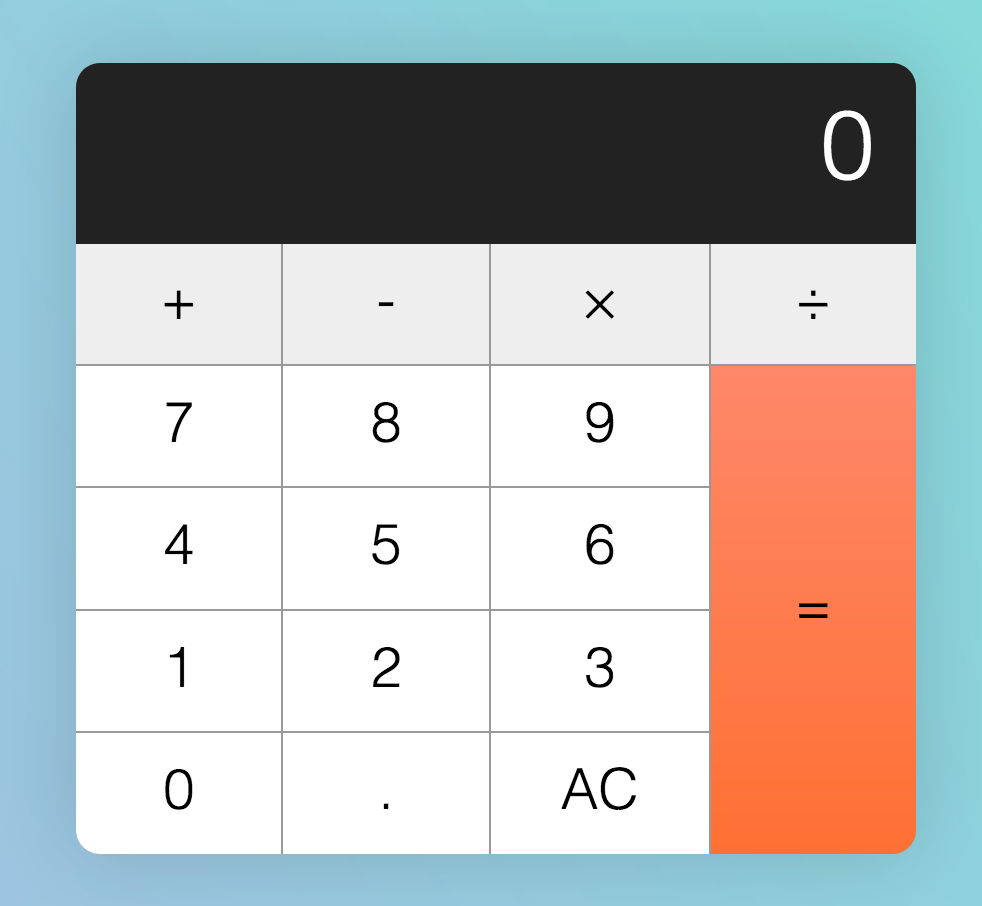
}

What happens if Tim hits the clear key?

The clear key has two uses:

1. All Clear (denoted by AC) clears everything and resets the calculator to its initial state.
2. Clear entry (denoted by CE) clears the current entry. It keeps previous numbers in memory.

When the calculator is in its default state, AC should be shown.

AC should be shown in the initial state

First, if Tim hits a key (any key except clear), AC should be changed to CE.

AC changes to CE when a key (except clear) gets hit

We do this by checking if the data-action is clear. If it’s not clear, we look for the clear button and change its textContent.

if (action !== 'clear') {

const clearButton = calculator.querySelector('[data-action=clear]')

clearButton.textContent = 'CE'

}

Second, if Tim hits CE, the display should read 0. At the same time, CE should be reverted to AC so Tim can reset the calculator to its initial state.\*\*

If CE is clicked, AC should show

if (action === 'clear') {

display.textContent = 0

key.textContent = 'AC'

calculator.dataset.previousKeyType = 'clear'

}

Third, if Tim hits AC, reset the calculator to its initial state.

To reset the calculator to its initial state, we need to clear all custom attributes we’ve set.

if (action === 'clear') {

if (key.textContent === 'AC') {

calculator.dataset.firstValue = ''

calculator.dataset.modValue = ''

calculator.dataset.operator = ''

calculator.dataset.previousKeyType = ''

} else {

key.textContent = 'AC'

}

display.textContent = 0

calculator.dataset.previousKeyType = 'clear'

}

Wrapping up

That’s it! Building a calculator is hard, don’t berate yourself if you cannot build a calculator without making mistakes.

For homework, write down all the edge cases mentioned above on a piece of paper, then proceed to build the calculator again from scratch. See if you can get the calculator up. Take your time, clear away your bugs one by one and you’ll get your calculator up eventually.

I hope you enjoyed this article. If you did, you’ll want to check out [Learn JavaScript](https://learnjavascript.today/)—a course to help you learn JavaScript once and for all.

In the [next lesson](https://zellwk.com/blog/calculator-part-3), you’ll learn to refactor the calculator with best practices.