**How to build a calculator - part 3**

**Published:** Apr 4, 2018

You learned how to build a robust calculator in the previous two lessons. Unfortunately, the code we created together confusing. If you tried to read the code again, you’ll definitely get lost.

That’s why we need to refactor. In this lesson, you’ll learn how to refactor the calculator with some JavaScript best practices.

Prerequisites

Before you start this lesson, please make sure you have completed the first two lessons. Their links are as follows:

1. [Part 1—happy path](https://zellwk.com/blog/calculator-part-1)
2. [Part 2—edge cases](https://zellwk.com/blog/calculator-part-2)

You also need to know some semi-advanced Javascript practices:

1. [Early returns](http://blog.timoxley.com/post/47041269194/avoid-else-return-early)
2. [Ternary operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)
3. [Pure functions](https://medium.com/@jamesjefferyuk/javascript-what-are-pure-functions-4d4d5392d49c)
4. [ES6 Destructuring](https://zellwk.com/blog/es6#destructuring)

With that, let’s begin!

Refactoring the calculator

When you refactor, you’ll often start with the most obvious improvements. In this case, let’s start with the calculate.

Refactoring the calculate function

Here’s what we have so far.

const calculate = (*n1*, *operator*, *n2*) => {

let result = ''

if (operator === 'add') {

result = parseFloat(n1) + parseFloat(n2)

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

result = parseFloat(n1) - parseFloat(n2)

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

result = parseFloat(n1) \* parseFloat(n2)

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

result = parseFloat(n1) / parseFloat(n2)

}

return result

}

You learned that we should reduce reassignments as much as possible. Here, we can remove assignments if we return the result of the calculation within the if and else if statements:

const calculate = (*n1*, *operator*, *n2*) => {

if (operator === 'add') {

return firstNum + parseFloat(n2)

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

return parseFloat(n1) - parseFloat(n2)

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

return parseFloat(n1) \* parseFloat(n2)

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

return parseFloat(n1) / parseFloat(n2)

}

}

Since we return all values, we can use **early returns**. If we do so, there’s no need for any else if conditions.

const calculate = (*n1*, *operator*, *n2*) => {

if (operator === 'add') {

return firstNum + parseFloat(n2)

}

if (operator === 'subtract') {

return parseFloat(n1) - parseFloat(n2)

}

if (operator === 'multiply') {

return parseFloat(n1) \* parseFloat(n2)

}

if (operator === 'divide') {

return parseFloat(n1) / parseFloat(n2)

}

}

And since we have one statement per if condition, we can remove the brackets. (Note: some developers swear by curly brackets though). Here’s what the code would look like:

const calculate = (*n1*, *operator*, *n2*) => {

if (operator === 'add') return parseFloat(n1) + parseFloat(n2)

if (operator === 'subtract') return parseFloat(n1) - parseFloat(n2)

if (operator === 'multiply') return parseFloat(n1) \* parseFloat(n2)

if (operator === 'divide') return parseFloat(n1) / parseFloat(n2)

}

Finally, we called parseFloat eight times in the function. We can simplify it by creating two variables to contain float values:

const calculate = (*n1*, *operator*, *n2*) => {

const firstNum = parseFloat(n1)

const secondNum = parseFloat(n2)

if (operator === 'add') return firstNum + secondNum

if (operator === 'subtract') return firstNum - secondNum

if (operator === 'multiply') return firstNum \* secondNum

if (operator === 'divide') return firstNum / secondNum

}

We’re done with calculate now. Don’t you think it’s easier to read compared to before?

Refactoring the event listener

The code we created for the event listener is huge. Here’s what we have at the moment:

keys.addEventListener('click', *e* => {

if (e.target.matches('button')) {

if (!action) {

/\* ... \*/

}

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

/\* ... \*/

}

if (action === 'clear') {

/\* ... \*/

}

if (action !== 'clear') {

/\* ... \*/

}

if (action === 'calculate') {

/\* ... \*/

}

}

})

How do you begin refactoring this piece of code? If you don’t know any programming best practices, you may be tempted to refactor by splitting up each kind of action into a smaller function:

// Don't do this!

const handleNumberKeys = (/\* ... \*/) => {

/\* ... \*/

}

const handleOperatorKeys = (/\* ... \*/) => {

/\* ... \*/

}

const handleDecimalKey = (/\* ... \*/) => {

/\* ... \*/

}

const handleClearKey = (/\* ... \*/) => {

/\* ... \*/

}

const handleCalculateKey = (/\* ... \*/) => {

/\* ... \*/

}

Don’t do this. It doesn’t help because you’re merely splitting up blocks of code. When you do so, the function gets harder to read.

A better way is to split the code into pure and impure functions. If you do so, you’ll get code that look like this:

keys.addEventListener('click', *e* => {

// Pure function

const resultString = createResultString(/\* ... \*/)

// Impure stuff

display.textContent = resultString

updateCalculatorState(/\* ... \*/)

})

Here, createResultString is a pure function that returns what needs to be displayed on the calculator. updateCalculatorState is an impure function that changes the calculator’s visual appearance and custom attributes.

Making createResultString

As mentioned before, createResultString should return the value that needs to be displayed on the calculator. You can get these values through parts of the code that says display.textContent = 'some value.

display.textContent = 'some value'

Instead of display.textContent = 'some value', we want to return each value so we can use it later.

// replace the above with this

return 'some value'

Let’s go through this together, step by step, starting with number keys.

**Making result string for number keys**

Here’s the code we have for number keys:

if (!action) {

if (

displayedNum === '0' ||

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

) {

display.textContent = keyContent

} else {

display.textContent = displayedNum + keyContent

}

calculator.dataset.previousKeyType = 'number'

}

The first step is to copy parts that say display.textContent = 'some value' into createResultString. When you do this, make sure you change display.textContent = into return.

const createResultString = () => {

if (!action) {

if (

displayedNum === '0' ||

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

) {

return keyContent

} else {

return displayedNum + keyContent

}

}

}

Next, we can convert the if/else statement to a ternary operator:

const createResultString = () => {

if (!action) {

return displayedNum === '0' ||

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

? keyContent

: displayedNum + keyContent

}

}

When you refactor, remember to keep note down a list of variables you need. We’ll come back to the list later.

const createResultString = () => {

// Variables required are:

// 1. keyContent

// 2. displayedNum

// 3. previousKeyType

// 4. action

if (!action) {

return displayedNum === '0' ||

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

? keyContent

: displayedNum + keyContent

}

}

**Making result string for the decimal key**

Here’s the code we have for the decimal key:

if (action === 'decimal') {

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

display.textContent = displayedNum + '.'

} else if (

previousKeyType === 'operator' ||

previousKeyType === 'calculate'

) {

display.textContent = '0.'

}

calculator.dataset.previousKeyType = 'decimal'

}

As before, we want to move anything that changes display.textContent into createResultString.

const createResultString = () => {

// ...

if (action === 'decimal') {

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

return = displayedNum + '.'

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

return = '0.'

}

}

}

Since we want to return all values, we can convert else if statements into early returns.

const createResultString = () => {

// ...

if (action === 'decimal') {

if (!displayedNum.includes('.')) return displayedNum + '.'

if (previousKeyType === 'operator' || previousKeyType === 'calculate')

return '0.'

}

}

A common mistake here is to forget to return the currently displayed number when neither conditions are matched. We need this because we will replace the display.textContent with the value returned from createResultString. If we missed it, createResultString will return undefined, which is not what we desire.

const createResultString = () => {

// ...

if (action === 'decimal') {

if (!displayedNum.includes('.')) return displayedNum + '.'

if (previousKeyType === 'operator' || previousKeyType === 'calculate')

return '0.'

return displayedNum

}

}

As always, take note of the variables that are required. At this point, the required variables remain the same as before:

const createResultString = () => {

// Variables required are:

// 1. keyContent

// 2. displayedNum

// 3. previousKeyType

// 4. action

}

**Making result string for operator keys**

Here’s the code we wrote for operator keys.

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

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

}

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

calculator.dataset.previousKeyType = 'operator'

calculator.dataset.operator = action

}

You know the drill by now; we want to move everything that changes display.textContent into createResultString. Here’s what needs to be moved:

const createResultString = () => {

// ...

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

if (

firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

) {

return calculate(firstValue, operator, secondValue)

}

}

}

Remember, createResultString needs to return the value to be displayed on the calculator. If the if condition did not match, we still want to return the displayed number.

const createResultString = () => {

// ...

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

if (

firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

) {

return calculate(firstValue, operator, secondValue)

} else {

return displayedNum

}

}

}

We can then refactor the if/else statement into a ternary operator:

const createResultString = () => {

// ...

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const secondValue = displayedNum

return firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

? calculate(firstValue, operator, secondValue)

: displayedNum

}

}

If you look closely, you’ll realize that there’s no need to store a secondValue variable. We can use displayedNum directly in the calculate function.

const createResultString = () => {

// ...

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

return firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

? calculate(firstValue, operator, displayedNum)

: displayedNum

}

}

Finally, take note of the variables and properties required. This time, we need calculator.dataset.firstValue and calculator.dataset.operator.

const createResultString = () => {

// Variables & properties required are:

// 1. keyContent

// 2. displayedNum

// 3. previousKeyType

// 4. action

// 5. calculator.dataset.firstValue

// 6. calculator.dataset.operator

}

**Making result string for the clear key**

We wrote the following code to handle the clear key.

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'

}

As above, want to move everything that changes display.textContent into createResultString.

const createResultString = () => {

// ...

if (action === 'clear') return 0

}

**Making result string for the equal key**

Here’s the code we wrote for the equal key:

if (action === 'calculate') {

let firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

let secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

secondValue = calculator.dataset.modValue

}

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

}

calculator.dataset.modValue = secondValue

calculator.dataset.previousKeyType = 'calculate'

}

As above, we want to copy everything that changes display.textContent into createResultString. Here’s what needs to be copied:

if (action === 'calculate') {

let firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

let secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

secondValue = calculator.dataset.modValue

}

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

}

}

When copying the code into createResultString, make sure you return values for every possible scenario:

const createResultString = () => {

// ...

if (action === 'calculate') {

let firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

let secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

secondValue = calculator.dataset.modValue

}

return calculate(firstValue, operator, secondValue)

} else {

return displayedNum

}

}

}

Next, we want to reduce reassignments. We can do so by passing in the correct values into calculate through a ternary operator.

const createResultString = () => {

// ...

if (action === 'calculate') {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const modValue = calculator.dataset.modValue

if (firstValue) {

return previousKeyType === 'calculate'

? calculate(displayedNum, operator, modValue)

: calculate(firstValue, operator, displayedNum)

} else {

return displayedNum

}

}

}

You can further simplify the above code with another ternary operator if you feel comfortable with it:

const createResultString = () => {

// ...

if (action === 'calculate') {

const firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

const modValue = calculator.dataset.modValue

return firstValue

? previousKeyType === 'calculate'

? calculate(displayedNum, operator, modValue)

: calculate(firstValue, operator, displayedNum)

: displayedNum

}

}

At this point, we want to take note of the properties and variables required again:

const createResultString = () => {

// Variables & properties required are:

// 1. keyContent

// 2. displayedNum

// 3. previousKeyType

// 4. action

// 5. calculator.dataset.firstValue

// 6. calculator.dataset.operator

// 7. calculator.dataset.modValue

}

**Passing in necessary variables**

We need seven properties/variables in createResultString:

1. keyContent
2. displayedNum
3. previousKeyType
4. action
5. firstValue
6. modValue
7. operator

We can get keyContent and action from key. We can also get firstValue, modValue, operator and previousKeyType from calculator.dataset.

That means the createResultString function needs three variables—key, displayedNum and calculator.dataset. Since calculator.dataset represents the state of the calculator, let’s use a variable called state instead.

const createResultString = (*key*, *displayedNum*, *state*) => {

const keyContent = key.textContent

const action = key.dataset.action

const firstValue = state.firstValue

const modValue = state.modValue

const operator = state.operator

const previousKeyType = state.previousKeyType

// ... Refactor as necessary

}

// Using createResultString

keys.addEventListener('click', *e* => {

if (e.target.matches('button')) return

const displayedNum = display.textContent

const resultString = createResultString(

e.target,

displayedNum,

calculator.dataset

)

// ...

})

Feel free to destructure variables if you desire:

const createResultString = (*key*, *displayedNum*, *state*) => {

const keyContent = key.textContent

const { action } = key.dataset

const { firstValue, modValue, operator, previousKeyType } = state

// ...

}

**Consistency within if statements**

In createResultString, we used the following conditions to test for the type of keys that was clicked:

// If key is number

if (!action) {

/\* ... \*/

}

// If key is decimal

if (action === 'decimal') {

/\* ... \*/

}

// If key is operator

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

/\* ... \*/

}

// If key is clear

if (action === 'clear') {

/\* ... \*/

}

// If key is calculate

if (action === 'calculate') {

/\* ... \*/

}

They’re not consistent, so they’re hard to read. If possible, we want to make them consistent so we can write something like this:

if (keyType === 'number') {

/\* ... \*/

}

if (keyType === 'decimal') {

/\* ... \*/

}

if (keyType === 'operator') {

/\* ... \*/

}

if (keyType === 'clear') {

/\* ... \*/

}

if (keyType === 'calculate') {

/\* ... \*/

}

To do so, we can create a function called getKeyType. This function should return the type of key that was clicked.

const getKeyType = *key* => {

const { action } = key.dataset

if (!action) return 'number'

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

)

return 'operator'

// For everything else, return the action

return action

}

Here’s how you’d use the function:

const createResultString = (*key*, *displayedNum*, *state*) => {

const keyType = getKeyType(key)

if (keyType === 'number') {

/\* ... \*/

}

if (keyType === 'decimal') {

/\* ... \*/

}

if (keyType === 'operator') {

/\* ... \*/

}

if (keyType === 'clear') {

/\* ... \*/

}

if (keyType === 'calculate') {

/\* ... \*/

}

}

We’re done with createResultString. Let’s move on to updateCalculatorState.

Making updateCalculatorState

updateCalculatorState is a function that changes the calculator’s visual appearance and custom attributes.

As with createResultString, we need to check the type of key that was clicked. Here, we can reuse getKeyType.

const updateCalculatorState = *key* => {

const keyType = getKeyType(key)

if (keyType === 'number') {

/\* ... \*/

}

if (keyType === 'decimal') {

/\* ... \*/

}

if (keyType === 'operator') {

/\* ... \*/

}

if (keyType === 'clear') {

/\* ... \*/

}

if (keyType === 'calculate') {

/\* ... \*/

}

}

If you look at the leftover code, you may notice we change data-previous-key-type for every type of key. Here’s what the code looks like:

const updateCalculatorState = (*key*, *calculator*) => {

const keyType = getKeyType(key)

if (!action) {

// ...

calculator.dataset.previousKeyType = 'number'

}

if (action === 'decimal') {

// ...

calculator.dataset.previousKeyType = 'decimal'

}

if (

action === 'add' ||

action === 'subtract' ||

action === 'multiply' ||

action === 'divide'

) {

// ...

calculator.dataset.previousKeyType = 'operator'

}

if (action === 'clear') {

// ...

calculator.dataset.previousKeyType = 'clear'

}

if (action === 'calculate') {

calculator.dataset.previousKeyType = 'calculate'

}

}

This is redundant because we already know the key type with getKeyType. We can refactor the above to:

const updateCalculatorState = (*key*, *calculator*) => {

const keyType = getKeyType(key)

calculator.dataset.previousKeyType = keyType

if (keyType === 'number') {

/\* ... \*/

}

if (keyType === 'decimal') {

/\* ... \*/

}

if (keyType === 'operator') {

/\* ... \*/

}

if (keyType === 'clear') {

/\* ... \*/

}

if (keyType === 'calculate') {

/\* ... \*/

}

}

**Making**updateCalculatorState**for operator keys**

Visually, we need to make sure all keys release their depressed state. Here, we can copy and paste the code we had before:

const updateCalculatorState = (*key*, *calculator*) => {

const keyType = getKeyType(key)

calculator.dataset.previousKeyType = keyType

Array.from(key.parentNode.children).forEach(*k* =>

k.classList.remove('is-depressed')

)

}

Here’s what’s left from what we’ve written for operator keys, after moving pieces related to display.textContent into createResultString.

if (keyType === 'operator') {

if (

firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

) {

calculator.dataset.firstValue = calculatedValue

} else {

calculator.dataset.firstValue = displayedNum

}

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

calculator.dataset.operator = key.dataset.action

}

You may notice that we can shorten the code with a ternary operator:

if (keyType === 'operator') {

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

calculator.dataset.operator = key.dataset.action

calculator.dataset.firstValue =

firstValue &&

operator &&

previousKeyType !== 'operator' &&

previousKeyType !== 'calculate'

? calculatedValue

: displayedNum

}

As before, take note of the variables and properties you need. Here, we need calculatedValue and displayedNum.

const updateCalculatorState = (*key*, *calculator*) => {

// Variables and properties needed

// 1. key

// 2. calculator

// 3. calculatedValue

// 4. displayedNum

}

**Making**updateCalculatorState**for the clear key**

Here’s the leftover code for the clear key:

if (action === 'clear') {

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

calculator.dataset.firstValue = ''

calculator.dataset.modValue = ''

calculator.dataset.operator = ''

calculator.dataset.previousKeyType = ''

} else {

key.textContent = 'AC'

}

}

if (action !== 'clear') {

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

clearButton.textContent = 'CE'

}

There’s nothing much we can refactor here; feel free to copy/paste everything into updateCalculatorState.

**Making**updateCalculatorState**for the equal key**

Here’s the code we wrote for the equal key:

if (action === 'calculate') {

let firstValue = calculator.dataset.firstValue

const operator = calculator.dataset.operator

let secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

firstValue = displayedNum

secondValue = calculator.dataset.modValue

}

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

}

calculator.dataset.modValue = secondValue

calculator.dataset.previousKeyType = 'calculate'

}

Here’s what we’re left with if we remove everything that concerns display.textContent.

if (action === 'calculate') {

let secondValue = displayedNum

if (firstValue) {

if (previousKeyType === 'calculate') {

secondValue = calculator.dataset.modValue

}

}

calculator.dataset.modValue = secondValue

}

We can refactor this into the following:

if (keyType === 'calculate') {

calculator.dataset.modValue =

firstValue && previousKeyType === 'calculate' ? modValue : displayedNum

}

As always, take note of the properties and variables used:

const updateCalculatorState = (*key*, *calculator*) => {

// Variables and properties needed

// 1. key

// 2. calculator

// 3. calculatedValue

// 4. displayedNum

// 5. modValue

}

**Passing in necessary variables**

We know we need five variables/properties for updateCalculatorState:

1. key
2. calculator
3. calculatedValue
4. displayedNum
5. modValue

Since modValue can be retrieved from calculator.dataset, we only need to pass in four values:

const updateCalculatorState = (

*key*,

*calculator*,

*calculatedValue*,

*displayedNum*

) => {

// ...

}

keys.addEventListener('click', *e* => {

if (e.target.matches('button')) return

const key = e.target

const displayedNum = display.textContent

const resultString = createResultString(key, displayedNum, calculator.dataset)

display.textContent = resultString

// Pass in necessary values

updateCalculatorState(key, calculator, resultString, displayedNum)

})

Refactoring updateCalculatorState again

We changed three kinds of values in updateCalculatorState:

1. calculator.dataset
2. The class for pressing/depressing operators
3. AC vs CE text

If you want to make it cleaner, you can split (2) and (3) into another function—updateVisualState. Here’s what updateVisualState can look like:

const updateVisualState = (*key*, *calculator*) => {

const keyType = getKeyType(key)

Array.from(key.parentNode.children).forEach(*k* =>

k.classList.remove('is-depressed')

)

if (keyType === 'operator') key.classList.add('is-depressed')

if (keyType === 'clear' && key.textContent !== 'AC') {

key.textContent = 'AC'

}

if (keyType !== 'clear') {

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

clearButton.textContent = 'CE'

}

}

Wrapping up

The code become much cleaner after the refactor. If you look into the event listener you’ll know what each function does.

Here’s what the event listener looks like at the end:

keys.addEventListener('click', *e* => {

if (!e.target.matches('button')) return

const key = e.target

const displayedNum = display.textContent

// Pure functions

const resultString = createResultString(key, displayedNum, calculator.dataset)

// Update states

display.textContent = resultString

updateCalculatorState(key, calculator, resultString, displayedNum)

updateVisualState(key, calculator)

})

For your homework, go through the refactoring exercise on your own and see if you can get it to work without viewing the lesson.