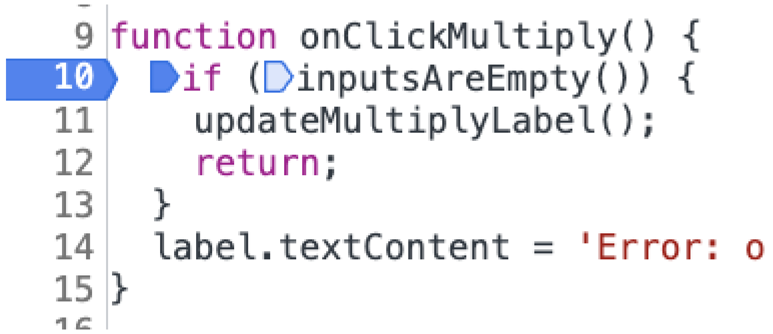
Multiply

Let's try and fix the multiplication functionality first. Enter 3 in the number 1 input, and 4 in the number 2 input. When we click multiply, we get an 'Error: one or both inputs are empty.' label.
Try other combinations of numbers - we can see that no matter what 2 numbers we enter, we run into the same problem.

Set a breakpoint on the first line, inside the `onClickMultiply` function, by clicking the number (in this case 10) next to the first line, as per :

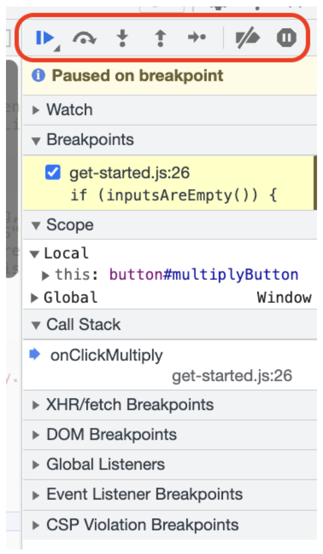


```
 9 function onClickMultiply() {
10   if (inputsAreEmpty()) {
11     updateMultiplyLabel();
12     return;
13   }
14   label.textContent = 'Error: o
15 }
```

Click the multiply button again. We should see execution paused at the breakpoint, with the above line highlighted in blue.

Let's click step inside to continue debugging inside the inputsAreEmpty() function.

NOTE: all step functions should be in the top of the debugging pane (pictured below), which can be found either to the right, or left of the file editor.



Paused on breakpoint

Watch

Breakpoints

get-started.js:26
`if (inputsAreEmpty()) {`

Scope

```
Local
  this: button#multiplyButton
Global                Window
```

Call Stack

onClickMultiply
get-started.js:26

XHR/fetch Breakpoints

DOM Breakpoints

Global Listeners

Event Listener Breakpoints

CSP Violation Breakpoints

Looking inside, the check that the function is doing seems fine.

Step over once, and we can see the function is returning false, as both numbers have values - which is correct! This means the bug must be something different. We can return back to the function that called this, by clicking step out ⭡.

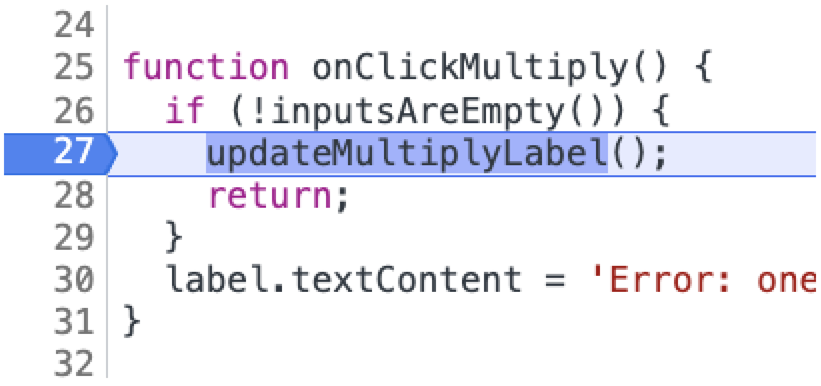
NOTE: Step out can be especially useful when debugging inside long functions, especially once we know the rest of the function should be correct.

We should now be paused on the last line inside the onClickMultiply function again. Here, we can see that the way the code is executing doesn't look right. inputsAreEmpty() is resolving to false, so we are skipping the if block that does the multiply calculation, and instead setting the label to the error message.

If we look a bit closer, we can see the if block check doesn't look right, and should be `!inputsAreEmpty()` (ie if the inputs are not empty, we want to update the label). Go ahead and fix it, then resume execution by pressing ▶.

Click the multiply button again. We can see that this time if we enter 2 numbers, we don't get the error label anymore! But things still don't look right. 3 x 4 is resolving to 81, rather than 12, like we would expect.

Let's set a breakpoint on the `updateMultiplyLabel()` function **call** this time.



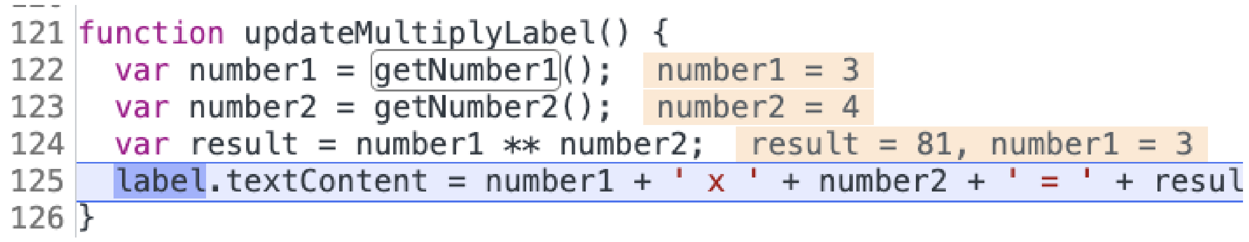
```

function onClickMultiply() {
  if (!inputsAreEmpty()) {
    updateMultiplyLabel();
    return;
  }
  label.textContent = 'Error: one
}

```

This time, let's step inside ⭣ to inspect what is happening inside the function.

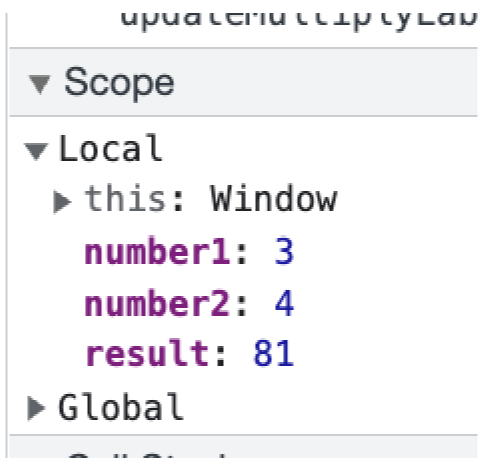
Note how as we do this, the call stack updates to show where in the code we are. We see `updateMultiplyLabel` at the top, as this is where we have paused execution, and beneath it `onClickMultiply` - the function that called the one we are currently in.

Stepping over the first couple of lines, look at the variables number1 and number2 next to the code (or alternatively in the scope pane), it looks like we are getting the numbers correctly this time.

Step over once more, we can see that the result has not been correctly calculated however.



```
function updateMultiplyLabel() {
  var number1 = getNumber1();   number1 = 3
  var number2 = getNumber2();   number2 = 4
  var result = number1 ** number2;   result = 81, number1 = 3
  label.textContent = number1 + ' x ' + number2 + ' = ' + resul
}
```

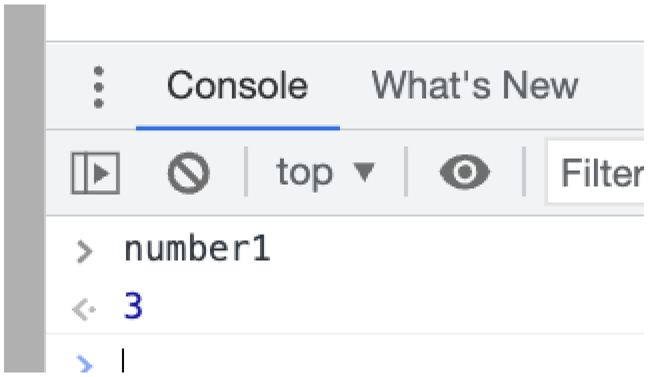


```
▼ Scope
▼ Local
  ▶ this: Window
    number1: 3
    number2: 4
    result: 81
▶ Global
```

Something must be going on in this line. Looks like we are using **, which is not how we multiply numbers in javascript.

<u>When we have paused execution at a breakpoint, the console will have access to all the variables in scope.</u>

For example, if we type number1 into the console, and hit enter, it will log the number the script is using (the same number we can see in the scope pane).



```
> number1
<· 3
>
```

Let’s use this to our advantage to test out how we could multiply 2 numbers.
If we enter `number1 * number2` into the console - we can see that this time we get the number we expect. Let’s update the code with the fix, and resume execution of the script by clicking ▶. Looks like the multiply functionality finally works as expected!

Alternatively, we could have entered `number1 * number2` into the watch pane to achieve a similar result.

Divide

The divide button doesn't seem to work either. When we enter 2 numbers, it seems to add them together - instead of dividing them?

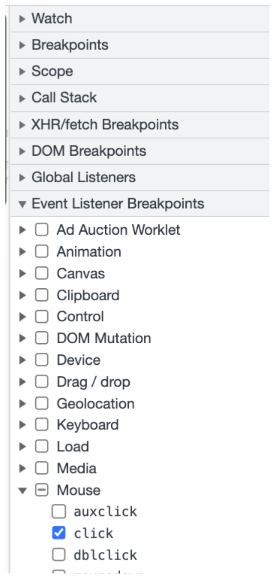
We can see an onClickDivide() function - let's set a breakpoint here and see what's going on.

Strangely, when we click the divide button, our breakpoint doesn't seem to be working! Something else must be happening. Let's try using a different type of breakpoint to see what's going on.

In the **Debugging** pane, click **Event Listener Breakpoints** to expand the section. DevTools reveals a list of expandable event categories, such as **Animation** and **Clipboard**.

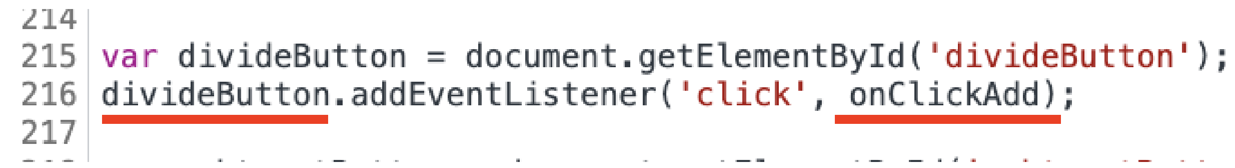
Next to the **Mouse** event category, click **Expand** ▶. DevTools reveals a list of mouse events, such as **click** and **mousedown**. Each event has a checkbox next to it.

Check the **click** checkbox. The debugger will now automatically pause when *any* `click` event listener executes. The debugging pane should look like this:



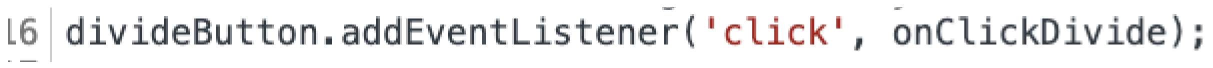
Click the Divide button again, and look where the execution has paused. That doesn't look right! If we look at where the code is paused, or look to the Call Stack pane on the right, we can see we are inside the onClickAdd function! It looks like when we click, the right function isn't being called (ie. the button isn't being mapped to the right function).

If we scroll to the bottom of the code, where the event listeners are set up, we can see we are using the wrong function for the divide button.



```
var divideButton = document.getElementById('divideButton');
divideButton.addEventListener('click', onClickAdd);
```

Let's map it to the right function, and rerun the code.
Note: you may need to refresh the entire page for the new event listener to work.



```
divideButton.addEventListener('click', onClickDivide);
```

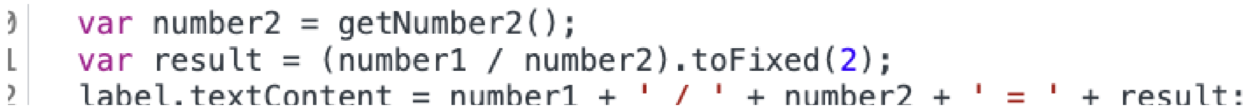
The divide functionality should work correctly now.

Note: You may want to uncheck the click event listener breakpoint to resume debugging as normal. However, you may also find it useful to reuse this functionality in future exercises. It will help prevent having to read through the code, and set breakpoints manually.

While we are here, let's use the divide button a bit more.

Enter 1000 in the number 1 input, and 7 in the number 2 input. The result doesn't look very clean.

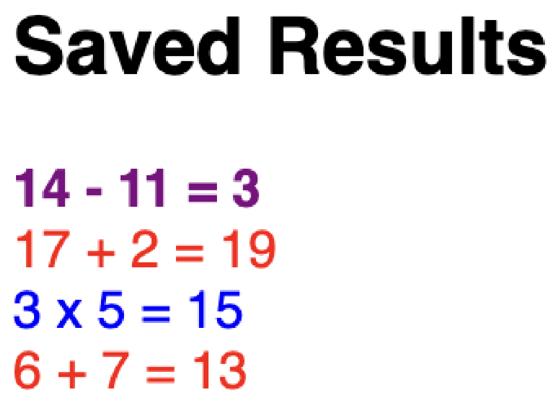
We can use the console (or watch expressions) to play around with how we could format this result, quickly and easily. Let's try functions like toPrecision or toFixed.
Create a variable in the console -
`var x = 1000 / 7`
And try out functions like
`x.toFixed(2)`
Or
`x.toPrecision(3)`
Find a format you like and update the code.



```
  var number2 = getNumber2();
  var result = (number1 / number2).toFixed(2);
  label.textContent = number1 + ' / ' + number2 + ' = ' + result;
```

Save

Let’s skip the subtract button for now, and instead have a look at the save functionality. This button helps us keep a history of results we’ve opted to save , coloured on the basis of the operation type, like this:



Saved Results

14 - 11 = 3
17 + 2 = 19
3 x 5 = 15
6 + 7 = 13

To test it out, enter 2 numbers into the inputs, and press add. We should see the result label updated as per the previous exercises.

However, when we click on the save button - nothing happens!

But wait - let’s look at the console. It looks like we are seeing an exception being thrown.

This could be a good chance to try out exception breakpoints. Exception breakpoints let you pause on any exception being thrown.
This means the debugger will pause **before** the red message appears in the Console, and give you a chance to inspect what may have gone wrong.

Add an exception breakpoint by clicking on the icon at the top of the debugging pane (next to the step buttons, at the end of the row).

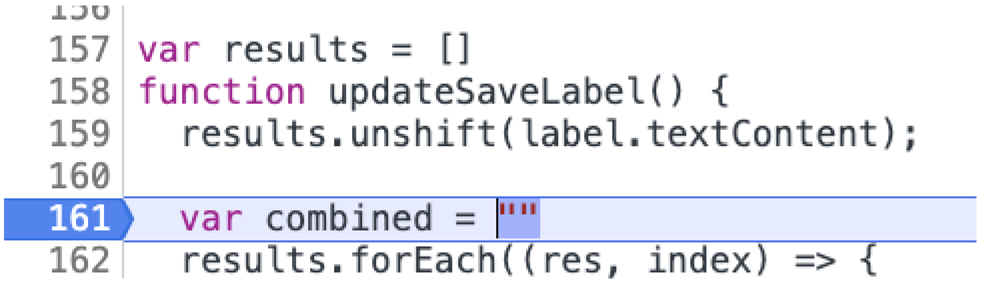
Next time we click the save button we can see the code pauses execution on the line that threw the exception.

Inspecting a bit further and hovering over the red cross icon, we see the function name is incorrect. Let’s update it.

The save button should be saving a history of results now!

Have a play around with this save functionality - it's a bit more complex than the previous buttons. Set a breakpoint on the onClickSave function, and try using the step buttons to trace the path of execution, and get more familiar with what the code is trying to do. In particular, look inside the `updateSaveLabel` function.

When you step through updateSaveLabel, you may notice that the results variable isn't visible inline, and not immediately obvious in the scope pane, as it's a global variable.

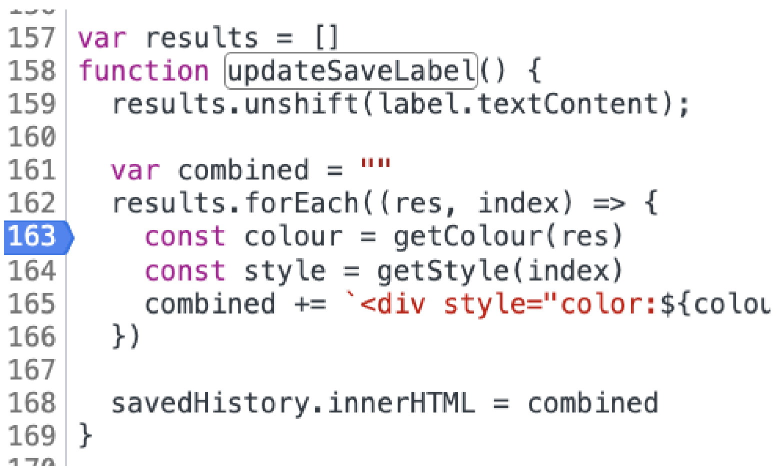


```
var results = []
function updateSaveLabel() {
  results.unshift(label.textContent);

  var combined = ""
  results.forEach((res, index) => {
```

To help inspect it, try the following when paused inside the function;

1. Add savedResults to the watch pane
2. Hover over the savedResults variable
3. Try entering console.table(savedResults) into the console window.

After experimenting with the save functionality, you may have noticed that when we divide 2 numbers, and try to save the result, it doesn't show up properly in the history. This seems to be an issue within the loop where we style the history of results. Let's set a breakpoint inside.



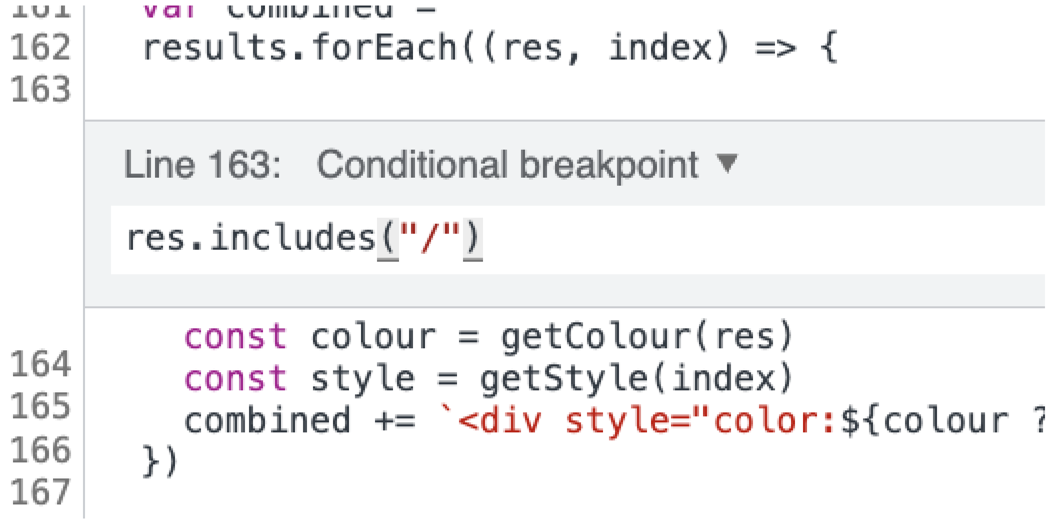
```
var results = []
function updateSaveLabel() {
  results.unshift(label.textContent);

  var combined = ""
  results.forEach((res, index) => {
    const colour = getColour(res)
    const style = getStyle(index)
    combined += `<div style="color:${colou
  })

  savedHistory.innerHTML = combined
}
```

However, after saving a few results, you may see it can be annoying to have to skip all the breakpoints for the results we know don't have a problem (ie adding or multiplying). This could be a good chance to try out conditional breakpoints.

Conditional breakpoints allow you to break inside a code block when a defined expression evaluates to true.

For us, let's set a conditional breakpoint on the first line of the loop in the `updateSaveLabel`.

Right click on the line number where we want to add it, and select Add Conditional Breakpoint, and enter an expression. Let's make the expression `res.includes("/")`, so it will only pause execution if it's a divide expression.

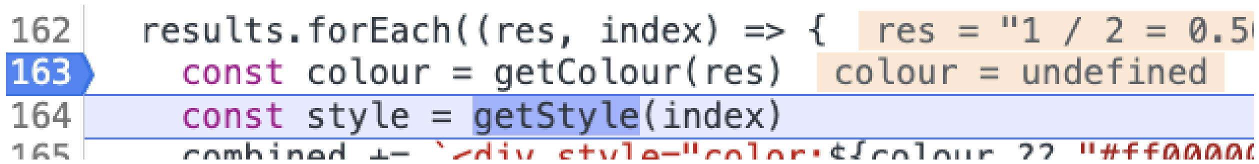


```
  var combined =
  results.forEach((res, index) => {

Line 163:   Conditional breakpoint ▼
res.includes("/")

    const colour = getColour(res)
    const style = getStyle(index)
    combined += `<div style="color:${colour ?
  })
```

We can confirm it has been created correctly, as conditional breakpoints appear as orange, instead of blue.

Divide 2 numbers. When the conditional breakpoint is hit, let's use step over to see the result of the `getColour` and `getStyle` function calls.

Looking at the scope pane, or the colour variable inline, it looks like the result of `getColour` is undefined. This is probably the issue.



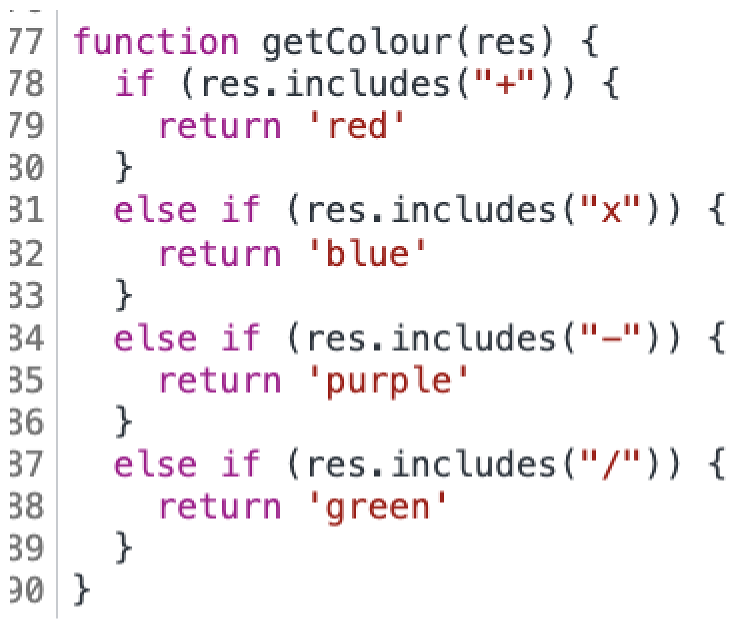
```
  results.forEach((res, index) => {   res = "1 / 2 = 0.5
    const colour = getColour(res)   colour = undefined
    const style = getStyle(index)
    combined += `<div style="color:${colour ?? "#ff0000
```

Let’s use the scope pane to change the execution of our program. In the scope pane, change the colour variable from undefined, to “green”.

Resume the script execution. We should now see the divide label be added to the Saved Results list in green! Looks like the problem is, as we suspected, that the getColour function isn’t returning a colour when we divide.

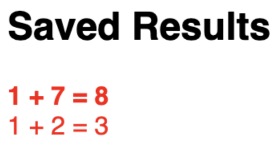
Let’s keep our conditional breakpoint, but this time, when we’re paused on the getColour call, let’s step inside to have a better look at what’s going on.

Stepping inside the function, we can see there’s no statement to handle the divide (“/”) case. Let’s add an additional case for res.includes(“/”), returning “green”.

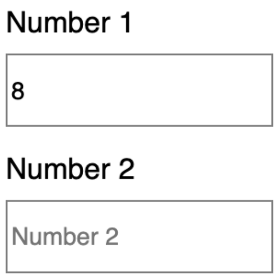


```
function getColour(res) {
  if (res.includes("+")) {
    return 'red'
  }
  else if (res.includes("x")) {
    return 'blue'
  }
  else if (res.includes("-")) {
    return 'purple'
  }
  else if (res.includes("/")) {
    return 'green'
  }
}
```

To complete the exercise, there are still a few bugs left to fix. Try and use the concepts covered thus far, to fix these on your own.

- The subtract button does not
    - calculate the correct value
    - Show the correct label
- The clear all functionality
    - Does not clear one of the inputs
    - Breaks the save button, after the first time it's used. (you will need to fix the previous bug to recreate this.)
- The ANS button should
    - Get the last saved result, set input number 1 to this value, and clear input number 2. That is, when the saved results look like this :
    - 

      Saved Results

      1 + 7 = 8
      1 + 2 = 3
    - If we then press ANS, the inputs SHOULD look like this:
    - 

      Number 1

      8

      Number 2

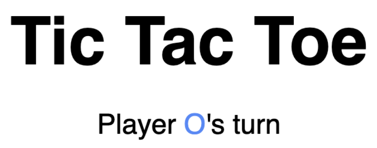
      Number 2
    - Right now it just shows undefined.

BONUS

If these bugs were simple, there is a more complex Tic Tac Toe scenario included in the exercise_2 folder. Once again, all the bugs should be localised to the index.js file.

Try and fix the following bugs;

- We seem to be stuck on Player X's turn. After Player X has played, we should see the label updated to the top as per the following;



Tic Tac Toe

Player O's turn

  - Note: If you are looking for variables inside the scope pane, you may need to look inside the 'Closure' subsection.

- Player X is able to overwrite Player O's move.

- When player X wins, we see a "Player O won" message, and vice versa.