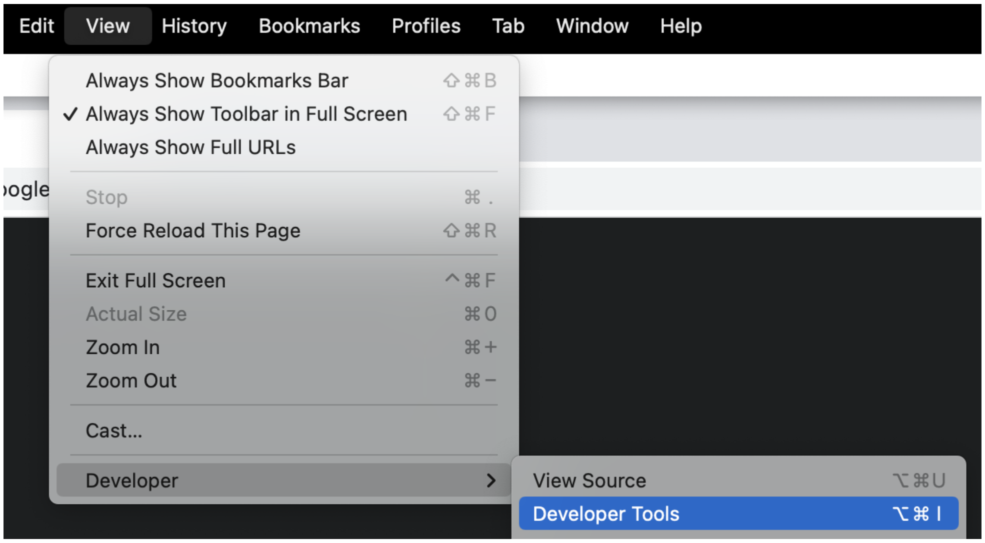
Introduction to Debugging

Setup Notes

The code can be cloned / downloaded from:
https://github.com/diva-lonial-vgw/debugging-demo

To set up Chrome so we can begin debugging, we will need to firstly open dev tools. We can do this using

- Command+Option+I (Mac)
- Control+Shift+I (Windows, Linux)
- OR in the toolbar, click View -> Developer -> Developer Tools



In Developer Tools, go to the sources tab -> filesystem -> add folder to workspace, and add the downloaded folder.

Right click on `get-started.html` & click copy link address. Paste this in a new tab.

Before we begin this exercise, there are a few 'gotchas' specific to chrome Developer Tools.

- You may need to save and / or refresh if you run into issues not being able to create or remove breakpoints.
- Keep an eye on the file you are editing. Chrome dev tools may open a VM version of the file you are editing. Don't edit this one - your changes will not be reflected.

Multiply

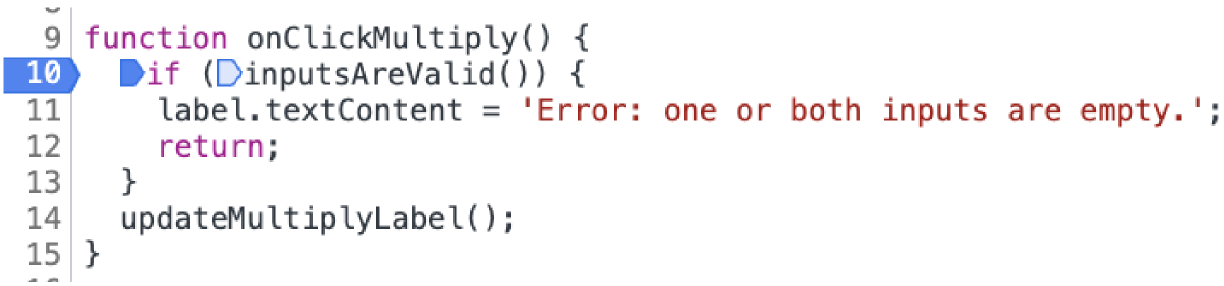
Stepping Through Code

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 :

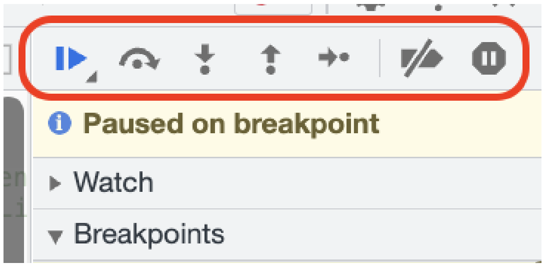


```
 9  function onClickMultiply() {
10    if (inputsAreValid()) {
11      label.textContent = 'Error: one or both inputs are empty.';
12      return;
13    }
14    updateMultiplyLabel();
15  }
```

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

Step inside ↓ to continue debugging inside the inputsAreValid() 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

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

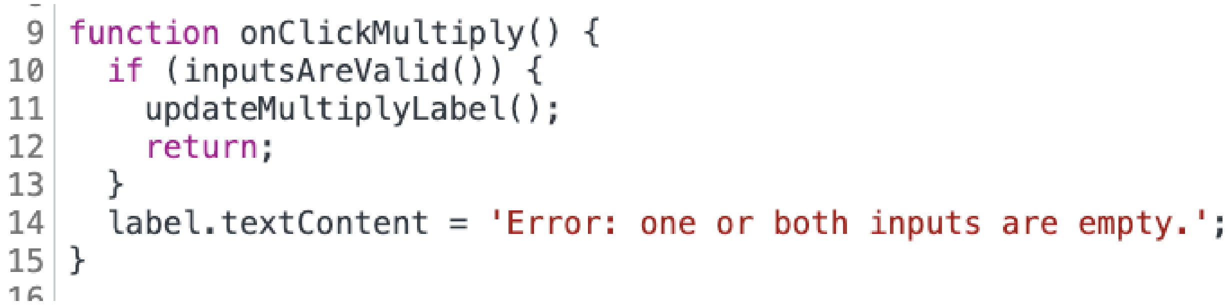
Step over once, and we can see the function is returning true, as both numbers have values - which is correct! The bug must be something different. We can return back to the function that called this, by clicking step out ⬆.

NOTE: Step out can be especially useful when debugging inside long functions, when 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.

inputsAreValid() is resolving to true, so we are executing what is inside the if block.

When inputsAreValid() is true, we want to multiply the 2 numbers - instead we are setting an error message. This means the contents inside the blocks need to be switched, as per the following;

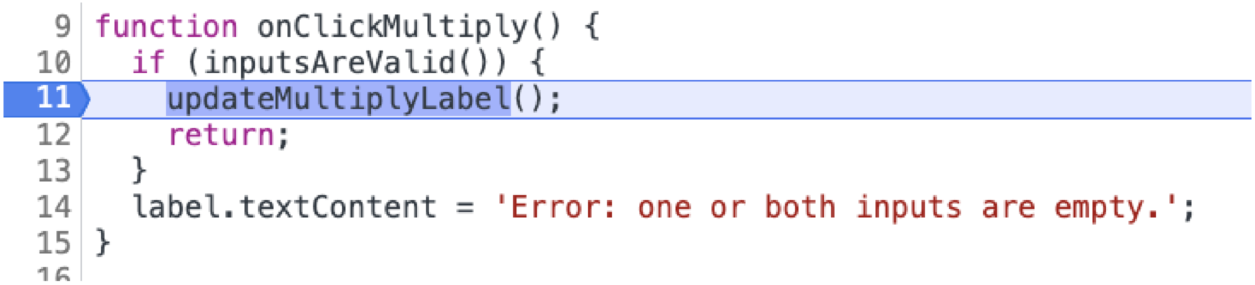


```
function onClickMultiply() {
  if (inputsAreValid()) {
    updateMultiplyLabel();
    return;
  }
  label.textContent = 'Error: one or both inputs are empty.';
}
```

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.

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



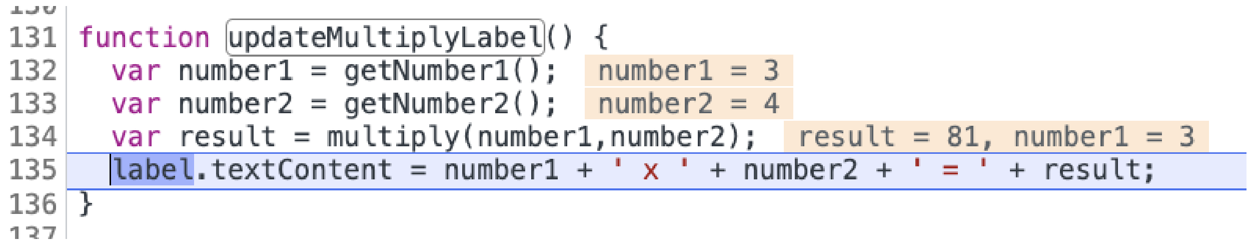
```
function onClickMultiply() {
  if (inputsAreValid()) {
    updateMultiplyLabel();
    return;
  }
  label.textContent = 'Error: one or both inputs are empty.';
}
```

This time, let’s step inside ⬇ to inspect what is happening inside the function.

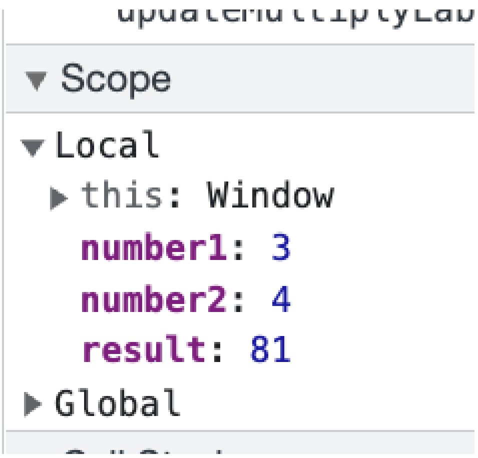
Scope Pane & Call Stack

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.

Step over ↷ the first 3 lines. Look at the variables number1 and number2 next to the code (or in the scope pane), it looks like we are getting the numbers correctly this time. However, we can also see that the result has not been correctly calculated.



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



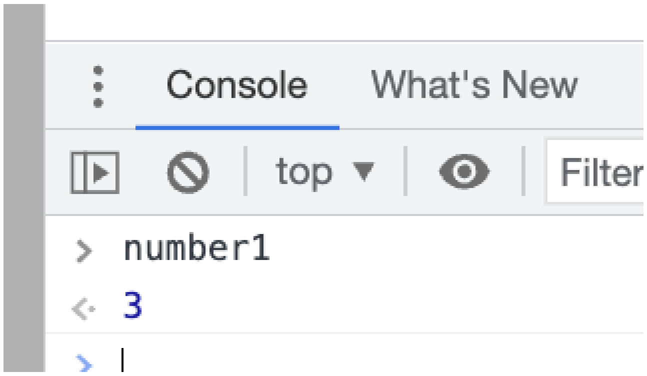
```
updateMultiplyLab
▼ Scope
▼ Local
  ▶ this: Window
    number1: 3
    number2: 4
    result: 81
▶ Global
```

Looks like this multiply function is not doing the correct thing!

Console

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

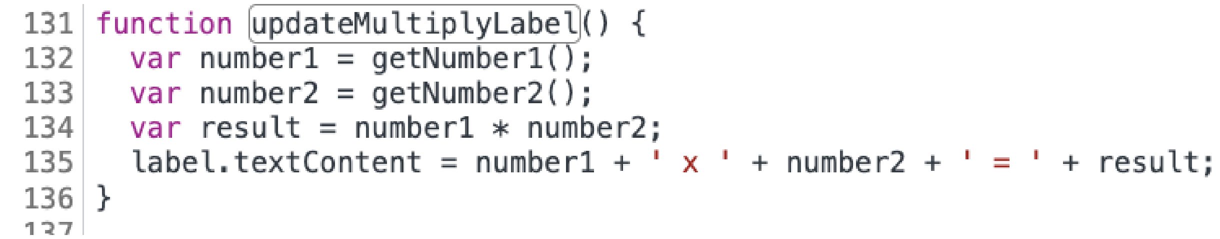
For example, if we type number1 into the console, and hit enter, it will log the number the script is using (the same number we can see in the scope pane).



```
> number1
< 3
>
```

We can use this to test 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.

Update the code with the fix as per the following image, and resume execution of the script by clicking ▶.



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

The multiply functionality finally works as expected!

Divide

Event Listener Breakpoints

Enter a number in the number 1 input, and another number in the number 2 input.

Click the divide button - note how it seems to add them together - instead of dividing them!

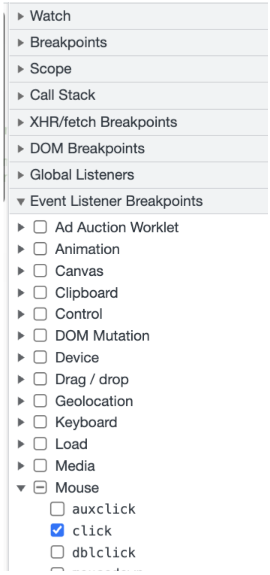
Set a breakpoint on the onClickDivide() function - and try clicking the divide button - the breakpoint doesn't seem to work!

Let's try using a different type of breakpoint to investigate.

In the **Debugging** pane, click **Event Listener Breakpoints** to expand the section.

Next to the **Mouse** event category, click **Expand** ▶. You should see a list of mouse events, such as **click** and **mousedown**.

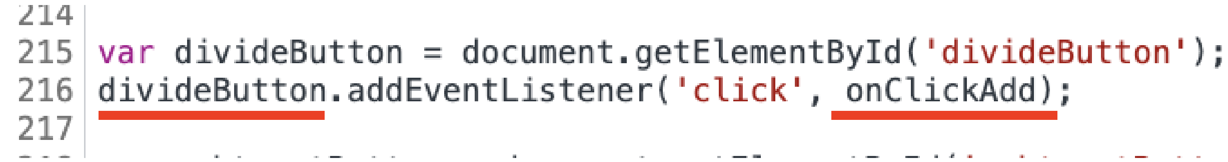
Check the **click** checkbox. The debugger will now automatically pause on any click event. The debugging pane should look like this:



▸ Watch
▸ Breakpoints
▸ Scope
▸ Call Stack
▸ XHR/fetch Breakpoints
▸ DOM Breakpoints
▸ Global Listeners
▾ Event Listener Breakpoints
▸ ☐ Ad Auction Worklet
▸ ☐ Animation
▸ ☐ Canvas
▸ ☐ Clipboard
▸ ☐ Control
▸ ☐ DOM Mutation
▸ ☐ Device
▸ ☐ Drag / drop
▸ ☐ Geolocation
▸ ☐ Keyboard
▸ ☐ Load
▸ ☐ Media
▾ ⊟ Mouse
☐ auxclick
☑ click
☐ dblclick

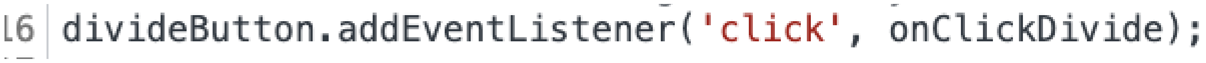
Click the Divide button again, and look where the execution has paused, or the call Stack pane on the right. We can see we are inside the onClickAdd function! It looks like when we click, the right function isn't being called (ie. the button isn't being mapped to the right function).

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.

Watch Expressions

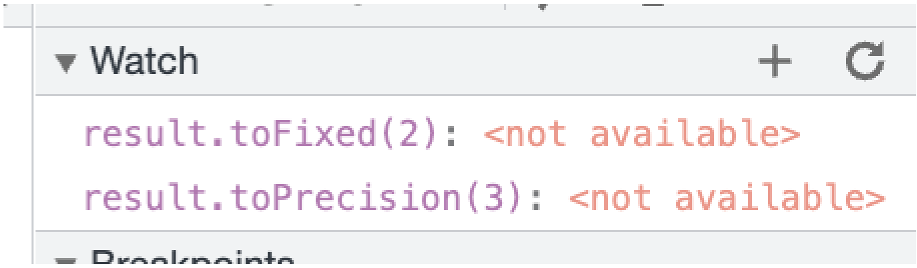
While we are here, let's use the divide button a bit more.

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

We can use watch expressions to quickly compare and decide how we want to format this result.

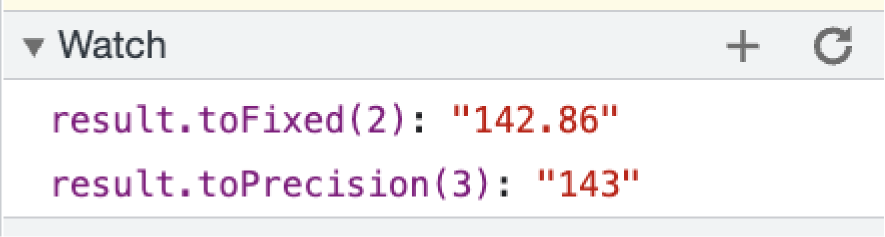
Create a couple of watch expressions like
`result.toFixed(2)`
And / Or
`result.toPrecision(3)`

Set a breakpoint on the first line of the update divide label function & divide 2 numbers.
Note how initially the results are unavailable.



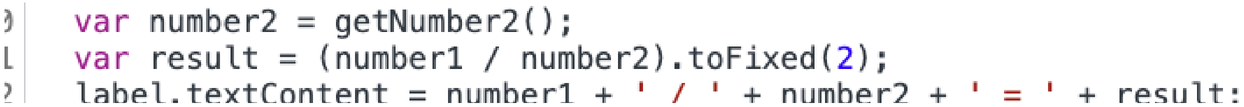
```
result.toFixed(2): <not available>
result.toPrecision(3): <not available>
```

Step through 3 lines, until the result is evaluated. Note how the watch pane is updated as follows;



```
result.toFixed(2): "142.86"
result.toPrecision(3): "143"
```

Find a format you like and update the code.

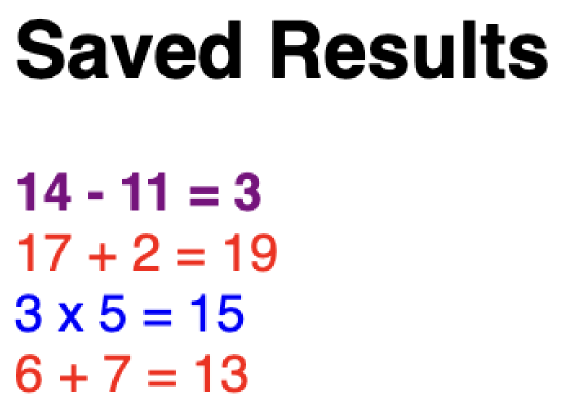


```
    var number2 = getNumber2();
    var result = (number1 / number2).toFixed(2);
    label.textContent = number1 + ' / ' + number2 + ' = ' + result;
```

Save

Exception Breakpoints

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 chosen 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 click add. Click the save button - nothing happens!

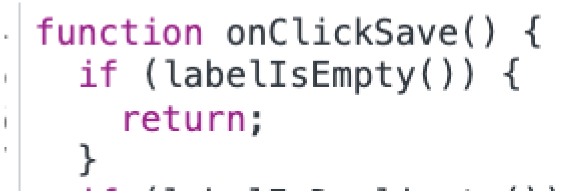
Check the console - it looks like we are seeing an exception being thrown!

This could be a good chance to try out exception breakpoints, which 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 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 - update it as follows;



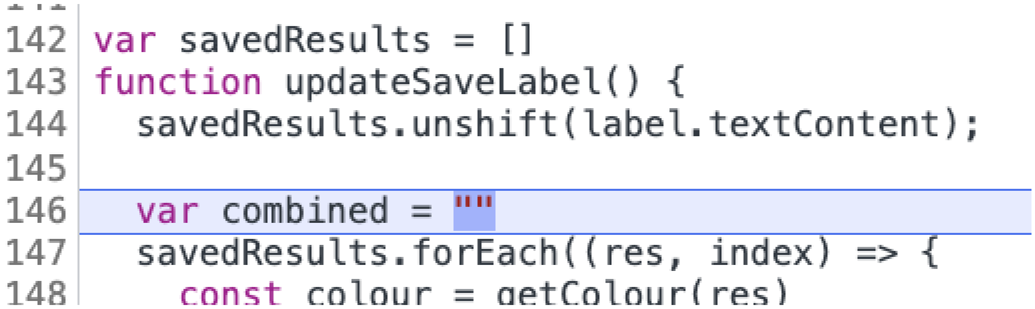
```
function onClickSave() {
  if (labelIsEmpty()) {
    return;
  }
```

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 get more familiar with what the code is trying to do. In particular, look inside the `updateSaveLabel` function.

Note: To look inside the loop, you may need to use Step Into!

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



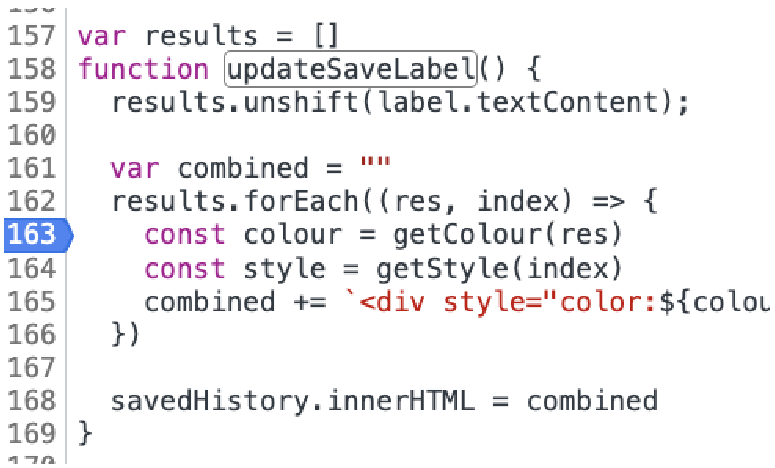
```
var savedResults = []
function updateSaveLabel() {
  savedResults.unshift(label.textContent);

  var combined = ""
  savedResults.forEach((res, index) => {
    const colour = getColour(res)
```

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, the text is transparent.

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
}
```

Conditional Breakpoints

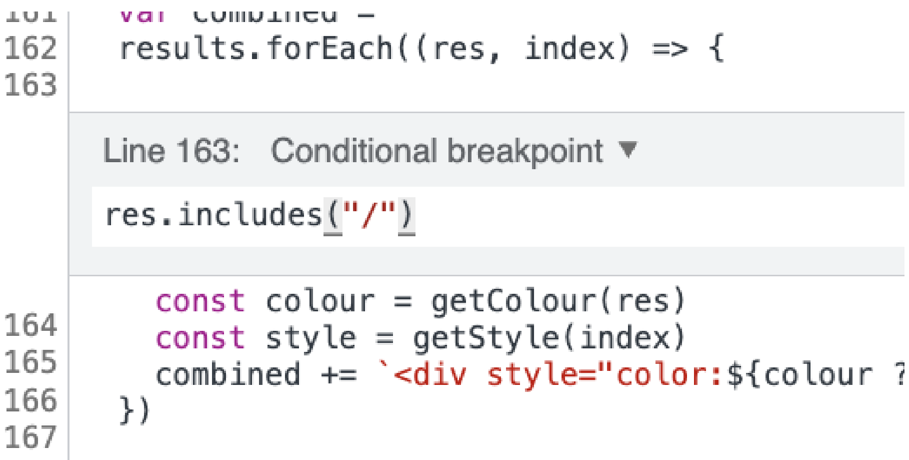
After saving a few results, you may see it can be annoying to have to skip breakpoints for the results we know don't have a problem (ie adding or multiplying). This could be a good chance to try out conditional breakpoints.

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 the expression `res.includes("/")`.

This will only pause execution if it's a divide expression.



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

Line 163: Conditional breakpoint ▼

```
res.includes("/")
```

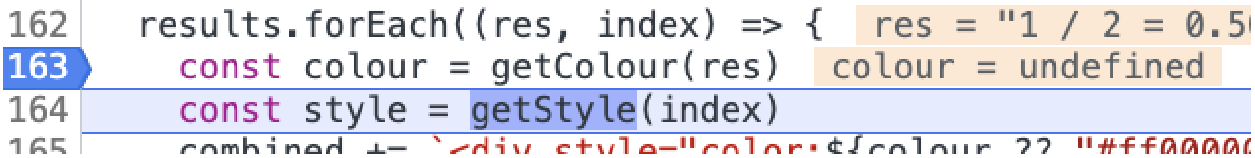
```
    const colour = getColour(res)
    const style = getStyle(index)
    combined += `<div style="color:${colour ?
  })
```

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

Divide 2 NEW numbers (they must be new - as we can't save duplicate results).

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.



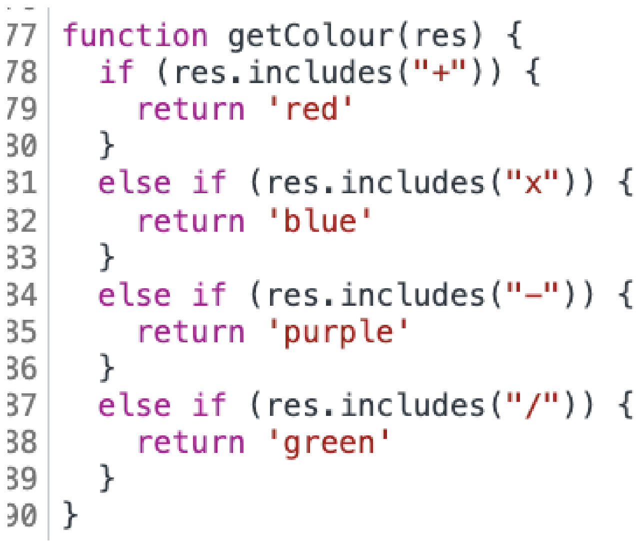
```
  results.forEach((res, index) => {  res = "1 / 2 = 0.5
    const colour = getColour(res)  colour = undefined
    const style = getStyle(index)
    combined += `<div style="color:${colour ?? "#ff0000
```

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!

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.

Here we can see there's no statement to handle the divide ("/") case. Let's add an additional case for res.includes("/"), returning "green".



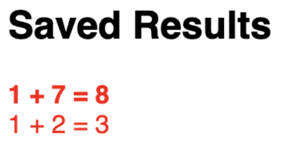
```
function getColour(res) {
  if (res.includes("+")) {
    return 'red'
  }
  else if (res.includes("x")) {
    return 'blue'
  }
  else if (res.includes("-")) {
    return 'purple'
  }
  else if (res.includes("/")) {
    return 'green'
  }
}
```

Bonus

To complete the exercise, there are still a few bugs left to fix with the subtract, clear and ANS button. The bugs are listed below - but you may find it useful to try and find the bugs yourself too, as this is a big part of debugging!

Try and use the concepts covered thus far, to fix these on your own.

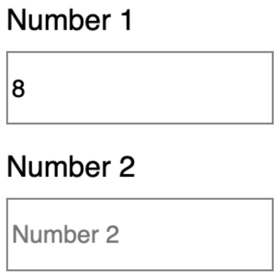
- The subtract button does not
    - calculate the correct value
    - Show the correct label
- The clear all functionality
    - Does not clear one of the inputs
    - Breaks the save button, after the first time it's used. (you will need to fix the previous bug to recreate this.)
- The ANS button should
    - 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

- It turns out, when we added our fix for parseInt() in the demo earlier, we had introduced another bug! Try adding empty inputs, and see what happens.
- If we enter 0 into both inputs and multiply, we get an Error: one or both inputs are empty message.