© Victor Tan 2022

JavaScript
Intro for Beginners
Lab 3

1 Objects

1.1 Basics

File to use: `objects-basic.js`

Objects are complex data types that allow us to group together related basic data types.
In JavaScript, an object is an unordered collection of key-value pairs. Each key-value pair is called a property. The key of a property is usually a string type, while its value can be of any type: a string, a number, an array, a function or even another object. The most common approach to create an object in JavaScript is through object literal notation.

1.2 Methods

File to use: `object-methods-basic.js`
Exercise: `objects-question.js`

When a function is a property of an object, we call that function a method. The method can be invoked in exactly the same way as a normal, standalone object.

1.3 Primitives vs references

File to use: `primitive-reference.js`

The data types we have covered earlier (boolean, string, number and so on) are called primitive data types. The way values of primitive data types are handled in memory is different from that of objects, which are considered reference values.

There are two areas of memory which the JavaScript engine allocates to hold variable values: the stack and the heap.
Static data is the data whose size is fixed at compile time. Static data includes:

- Primitive values (boolean, number, string, and others)
- Reference values that refer to objects

Because static data has a size that does not change, the JavaScript engine allocates a fixed amount of space to the static data and stores it on the stack.

The actual objects themselves are stored on the heap. The JavaScript engine doesn't allocate a fixed amount of memory for these objects, since we can keep adding properties into an object during run time. Instead, it'll allocate more space as needed.

The same concepts discussed above also applies when we are passing primitive values or objects to the parameters of functions.

2 Arrays

2.1 Basics

File to use: `arrays-basic.js`

Arrays are a collection of values. Each value is called an element of the array and is located at a specific position called an index. The index numbering for arrays start from 0. So, the 1st element of the array is stored at index 0, the 2nd element of the array is stored at index 1, and so on.

A JavaScript array has the following characteristics:

- An array can hold values of mixed types. For example, you can have an array that stores elements with the types number, string, and boolean.
- The size of an array is dynamic. You don't need to specify the array size upfront.

We can use a normal for loop to iterate through the contents of an array, or we can use the ES6 feature of for..of loop

2.2 Array operations

File to use: `array-operations.js`

Arrays are objects that have a variety of methods that allow us to perform useful operations on the elements contained in them. One of the most useful methods is the splice method that allows us to delete existing elements, insert new elements, and replace elements in an array

2.3 Higher order Array methods

File to use: `array-higher-order.js`

There are a variety of higher order array methods which accept another function as their parameter. The most widely used ones are map and filter.

2.4 Multi-Dimensional Arrays

File to use: `array-2-dimensional.js`
Exercise: `arrays-question.js`

Arrays can be nested, in that the element of an array can be another array. This level of nesting (array within an array within an array, etc, etc) can be carried on to an arbitrary level of depth to create a multi-dimensional array. The most common form of a multi-dimensional array is a 2-dimensional array.

A good way to visualize a 2D array is to enter the values into a table in an Excel spreadsheet.

3 Working with strings

File to use: `string-operations.js`
Exercise: `string-question.js`

A String is essentially an array of characters and we can access the individual characters using basic array notation.
The ES6 string template allows us to work with strings in a more flexible way.
In JavaScript, a string is an object which provides a variety of methods for us to do operations on the string.

4 Debugging Code in VS Code

VS Code provides built-in facilities that allow you to step through your program one statement at a time. This is very useful for identifying the specific statement / statements in a program where a bug was originally introduced.

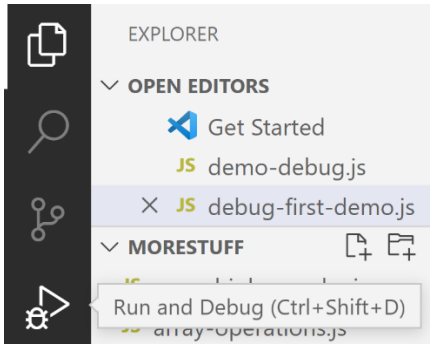
4.1 Demonstrating basic VS Code debugging facilities

File to use: `debug-first-demo.js`

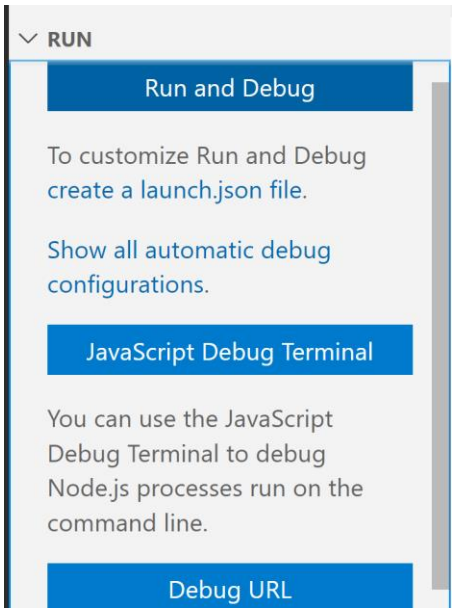
To start a debugging session, first place a breakpoint at a suitable point in the program code by clicking in the editor view just to the left of the line number of the statement.

```
42    console.log("Place your breakpoint at this point");
```

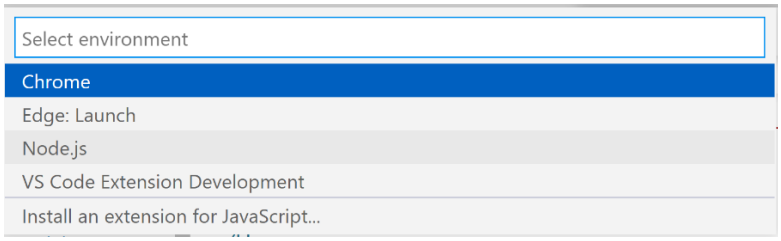
Select the Run and Debug option from the left most pane.



Three options for debugging appear in the left pane.



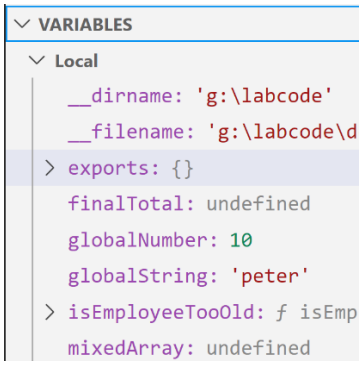
Select the Run and Debug option.
In the Drop down menu that appears from the menu bar as shown below, select Node.js



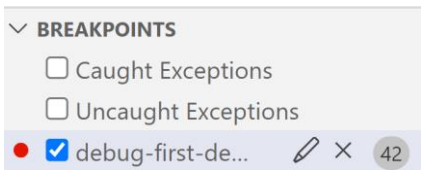
This transitions you into debugging mode. The orange highlight of the info bar at the bottom of the IDE indicates that you are now in debugging mode.

The left pane now consists of 5 main sections that you can minimize and maximize

1. Variables - shows you the content of the various variables, both at the local and global level. The global level variables shows universal variables that are part of the standard JavaScript library that is accessible to your code (not relevant for our program that does not use it). You should be able to see the various functions and variables declared in the program in the local section. Note that some of them are undefined as they have not yet been assigned a value at this point in the execution.



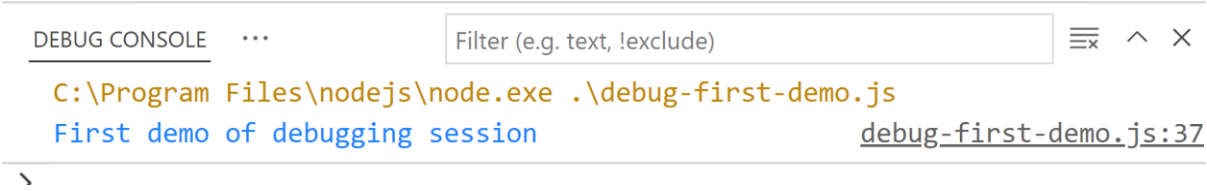
2. Watch - allows you to specifically identify a variable whose value you want to track. This is useful to focus on a few variables whose content you want to closely monitor out of the dozens of variables that are available in the variables section
3. Call Stack - indicates which function that the program is currently executing in.
4. Loaded scripts - indicates which JavaScript program (also called script) that is currently being debugged
5. Breakpoints - shows all the breakpoints that are set in the current program being debugged. Right now, we can see there is one active breakpoint at line 42, where execution is paused at.



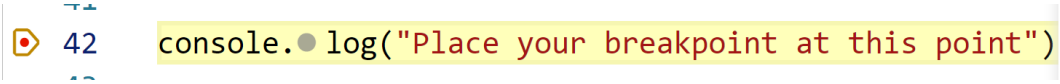
At the upper right hand corner of the IDE, you should also see a Debug Toolbar whose icons provide the following actions (shortcuts are also available):

Continue / Pause F5
Step Over F10
Step Into F11
Step Out Shift+F11
Restart Ctrl+Shift+F5
Stop Shift+F5

At the bottom of the IDE, you should be able to see the Debug Console, which shows the output from the various `console.log` statements in the program being debugged as you step through it.



At the moment, you will see that there is an arrow icon pointing at the statement with the breakpoint that you set earlier.

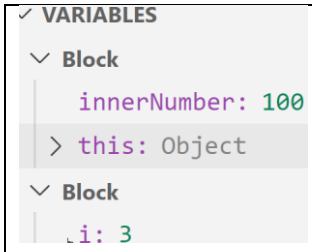


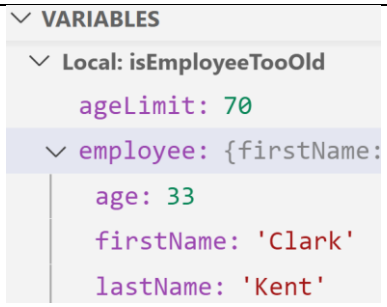
This means that the debugger has executed the program from the start until this current statement and is paused, waiting for directions from you on the next thing to perform.

For this first session of debugging, click on the Step Into button (shortcut F11) to step through the program one statement at a time until the end. The pointer will highlight the next statement in yellow to be executed when the Step Into button is clicked on, for e.g.

Observe carefully the output in the Debug Console as well as the content in the Variables, Watch and Call Stack sections on the left most pane. Some points to note:

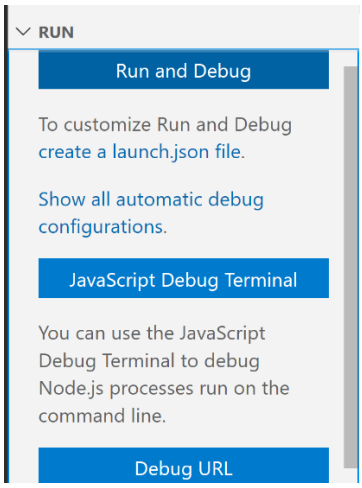
When you are in the body of a loop or method (considered a block), you will see a Block subsection in the Variables section showing the variables visible / accessible in that block. For e.g.



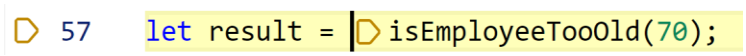


You can add in a few variables of your choice to monitor in the watch section. This can be global variables of the program, or local variables declared in a block (body of a loop or method).

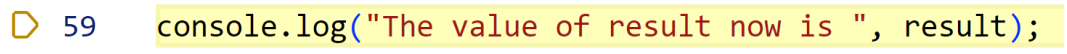
The debugging session completes when the debugger has completed executing all statements in the program. You should then see the left pane change back to its previous state:



Start another session of debugging in the same way as previously shown. Continue to step through the program line by line (using the Step Into button) until you reach the line when a call is to be made to the function `isEmployeeTooOld`



At that point click Step Over instead. This causes the debugger to execute that entire function without stepping through it statement by statement. So execution then continues immediately to the next line in the main program.



Continue stepping through the program line by line (using the Step Into button) until you reach the line when a call is to be made to the function

```
63    let finalTotal = sumNumbers(mixedArray);
```

At this point, click on Step Into as usual and you will move into the function body. Continue to click Step Into a few more times, then click on Step Out. This causes the debugger to complete the execution of the function immediately and exit it, returning the flow of control to the main program.

```
65    console.log("The sum of all numbers in mixedArray is "
```

Continue to click Step Into until the debugging session terminates.

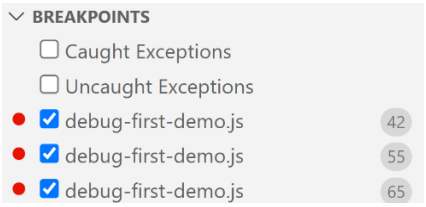
Start another session of debugging in the same way as previously shown. Continue to step through the program line by line (using the Step Into button) for a few times, then click the Restart button. This restarts the debugging session by running the program from the start until the first break point is reached. Continue to step through the program line by line (using the Step Into button) for a few times, then click the Stop button. This immediately terminates the debugging session.

Before you start the next session of debugging, add two more breakpoints to the program:

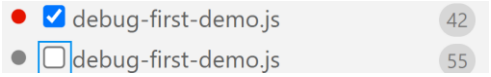
```
55    console.log("Exiting the loop and now ready to call

65    console.log("The sum of all numbers in mixedArray :
```

Start another session of debugging in the same way as previously shown. Once the debugging session starts, click the Continue icon. The debugger then continues executing all the statements in the program (including function calls) until it reaches the next breakpoint, and stops there. If you now press the click the Continue icon, the same action is repeated until there are no more breakpoints and the debugger will then execute the program until the end.

You can disable / remove the breakpoints in the Breakpoints section in the left pane by either checking their corresponding checkboxes or clicking on the x.



A disabled breakpoint will no longer function, but you can still see it as a greyed out circle in the editor view next to the line number.



On the other hand, a removed breakpoint will no longer be visible anymore.

4.2 Using debugging facilities to locate a bug

File to use: `debug-second-demo.js`
As an exercise, locate and correct the bug causing the incorrect program output.