Use the JavaScript Console to Check the Value of a Variable

Both Chrome and Firefox have excellent JavaScript consoles, also known as DevTools, for debugging your JavaScript.

You can find Developer tools in your Chrome's menu or Web Console in Firefox's menu. If you're using a different browser, or a mobile phone, we strongly recommend switching to desktop Firefox or Chrome.

The `console.log()` method, which "prints" the output of what's within its parentheses to the console, will likely be the most helpful debugging tool. Placing it at strategic points in your code can show you the intermediate values of variables. It's good practice to have an idea of what the output should be before looking at what it is. Having check points to see the status of your calculations throughout your code will help narrow down where the problem is.

Here's an example to print 'Hello world!' to the console:

```
console.log('Hello world!');
```

Use typeof to Check the Type of a Variable

You can use `typeof` to check the data structure, or type, of a variable. This is useful in debugging when working with multiple data types. If you think you're adding two numbers, but one is actually a string, the results can be unexpected. Type errors can lurk in calculations or function calls. Be careful especially when you're accessing and working with external data in the form of a JavaScript Object Notation (JSON) object.

Here are some examples using `typeof`:

```
console.log(typeof ""); // outputs "string"
console.log(typeof 0); // outputs "number"
console.log(typeof []); // outputs "object"
console.log(typeof {}); // outputs "object"
```

JavaScript recognizes six primitive (immutable) data types: `Boolean`, `Null`, `Undefined`, `Number`, `String`, and `Symbol` (new with ES6) and one type for mutable items: `Object`. Note that in JavaScript, arrays are technically a type of object.

Catch Misspelled Variable and Function Names

The `console.log()` and `typeof` methods are the two primary ways to check intermediate values and types of program output. Now it's time to get into the common forms that bugs take. One syntax-level issue that fast typers can commiserate with is the humble spelling error.

Transposed, missing, or mis-capitalized characters in a variable or function name will have the browser looking for an object that doesn't exist - and complain in the form of a reference error. JavaScript variable and function names are case-sensitive.

Catch Unclosed Parentheses, Brackets, Braces and Quotes

Another syntax error to be aware of is that all opening parentheses, brackets, curly braces, and quotes have a closing pair. Forgetting a piece tends to happen when you're editing existing code and inserting items with one of the pair types. Also, take care when nesting code blocks into others, such as adding a callback function as an argument to a method.

One way to avoid this mistake is as soon as the opening character is typed, immediately include the closing match, then move the cursor back between them and continue coding. Fortunately, most modern code editors generate the second half of the pair automatically.

Catch Mixed Usage of Single and Double Quotes

JavaScript allows the use of both single (`'`) and double (`"`) quotes to declare a string. Deciding which one to use generally comes down to personal preference, with some exceptions.

Having two choices is great when a string has contractions or another piece of text that's in quotes. Just be careful that you don't close the string too early, which causes a syntax error.

Here are some examples of mixing quotes:

```
// These are correct:
const grouchoContraction = "I've had a perfectly wonderful evening, but this wasn't it.";
const quoteInString = "Groucho Marx once said 'Quote me as saying I was mis-quoted.'";
// This is incorrect:
```

```
const uhOhGroucho = 'I've had a perfectly wonderful evening, but
this wasn't it.';
```

Of course, it is okay to use only one style of quotes. You can escape the quotes inside the string by using the backslash (`\`) escape character:

```
// Correct use of same quotes:
const allSameQuotes = 'I\'ve had a perfectly wonderful evening, but
this wasn\'t it.';
```

Catch Use of Assignment Operator Instead of Equality Operator

Branching programs, i.e. ones that do different things if certain conditions are met, rely on `if`, `else if`, and `else` statements in JavaScript. The condition sometimes takes the form of testing whether a result is equal to a value.

This logic is spoken (in English, at least) as "if x equals y, then ..." which can literally translate into code using the `=`, or assignment operator. This leads to unexpected control flow in your program.

As covered in previous challenges, the assignment operator (`=`) in JavaScript assigns a value to a variable name. And the `==` and `===` operators check for equality (the triple `===` tests for strict equality, meaning both value and type are the same).

The code below assigns `x` to be 2, which evaluates as `true`. Almost every value on its own in JavaScript evaluates to `true`, except what are known as the "falsy" values: `false`, `0`, `""` (an empty string), `NaN`, `undefined`, and `null`.

```
let x = 1;
let y = 2;
if (x = y) {
  // this code block will run for any value of y (unless y were
originally set as a falsy)
} else {
  // this code block is what should run (but won't) in this example
}
```

Catch Missing Open and Closing Parenthesis After a Function Call

When a function or method doesn't take any arguments, you may forget to include the (empty) opening and closing parentheses when calling it. Often times the result of a function call is saved in a variable for other use in your code. This error can be detected by logging variable values (or their types) to the console and seeing that one is set to a function reference, instead of the expected value the function returns.

The variables in the following example are different:

```
function myFunction() {
  return "You rock!";
}
let varOne = myFunction; // set to equal a function
let varTwo = myFunction(); // set to equal the string "You rock!"
```

Catch Arguments Passed in the Wrong Order When Calling a Function

Continuing the discussion on calling functions, the next bug to watch out for is when a function's arguments are supplied in the incorrect order. If the arguments are different types, such as a function expecting an array and an integer, this will likely throw a runtime error. If the arguments are the same type (all integers, for example), then the logic of the code won't make sense. Make sure to supply all required arguments, in the proper order to avoid these issues.

Catch Off By One Errors When Using Indexing

Off by one errors (sometimes called OBOE) crop up when you're trying to target a specific index of a string or array (to slice or access a segment), or when looping over the indices of them. JavaScript indexing starts at zero, not one, which means the last index is always one less than the length of the item. If you try to access an index equal to the length, the program may throw an "index out of range" reference error or print `undefined`.

When you use string or array methods that take index ranges as arguments, it helps to read the documentation and understand if they are inclusive (the item at the given index is part of what's returned) or not. Here are some examples of off by one errors:

```javascript
let alphabet = "abcdefghijklmnopqrstuvwxyz";
let len = alphabet.length;
for (let i = 0; i <= len; i++) {
  // loops one too many times at the end
  console.log(alphabet[i]);
}
for (let j = 1; j < len; j++) {
  // loops one too few times and misses the first character at index 0
  console.log(alphabet[j]);
}
for (let k = 0; k < len; k++) {
  // Goldilocks approves - this is just right
  console.log(alphabet[k]);
}
```

Use Caution When Reinitializing Variables Inside a Loop

Sometimes it's necessary to save information, increment counters, or re-set variables within a loop. A potential issue is when variables either should be reinitialized, and aren't, or vice versa. This is particularly dangerous if you accidentally reset the variable being used for the terminal condition, causing an infinite loop.

Printing variable values with each cycle of your loop by using `console.log()` can uncover buggy behavior related to resetting, or failing to reset a variable.

Prevent Infinite Loops with a Valid Terminal Condition

The final topic is the dreaded infinite loop. Loops are great tools when you need your program to run a code block a certain number of times or until a condition is met, but they need a terminal condition that ends the looping. Infinite loops are likely to freeze or crash the browser, and cause general program execution mayhem, which no one wants.

There was an example of an infinite loop in the introduction to this section - it had no terminal condition to break out of the `while` loop inside `loopy()`. Do NOT call this function!

```
function loopy() {
  while(true) {
    console.log("Hello, world!");
  }
}
```

It's the programmer's job to ensure that the terminal condition, which tells the program when to break out of the loop code, is eventually reached. One error is incrementing or decrementing a counter variable in the wrong direction from the terminal condition. Another one is accidentally resetting a counter or index variable within the loop code, instead of incrementing or decrementing it.