Testing with Expected Inputs

These are inputs that you expect the function to handle in its normal operation. If you're writing a function to calculate the square of a number, for instance, typical inputs would be numbers.

```
function square() {
    let num = prompt('Give me a number')

    alert(num * num)
}
```

If you expected the user to input a number from 1 to 9, you would test the code with a few numbers within that range.

Your commit message could look something like this:

> Summary: Expected testing
> Description: Tested the square() function with expected inputs of 2, 5, and 7 and got the correct outputs of 4, 25, and 49.

Testing with Boundary Inputs

These are inputs at the extremes of what you expect the function to handle. If you're writing a function to check whether a number is within a certain range, you would test it with numbers right at the boundaries. This is because for many programs, bugs or unexpected behaviour can occur for these values.

If you had the code above, and you expected a number from 1 to 9, you would test 1 and 9 as inputs specifically.

Your commit message could look something like this:

> Summary: Boundary testing
> Description: Tested the square() function with boundary inputs of 1 and 9, and got the correct outputs of 1 and 81.

Testing with Invalid Inputs

These are inputs that your code shouldn't be able to handle. It's important to test these to ensure your code fails gracefully and doesn't produce incorrect results or crash the entire program.

For code that expects a number, invalid inputs could include a string, or null (nothing).

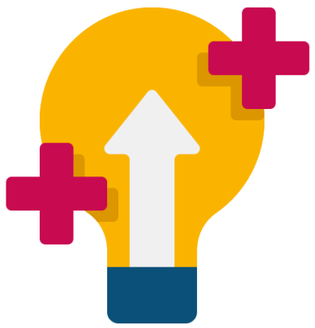
To improve your code further, you should write your code so that it handles invalid input correctly. It should provide a message to the user, or give them another chance to input the correct value.

```
function square() {
    let num = prompt('Give me a number')

    if (num > 0 && num < 10) {
      alert(num * num)
    } else {
      alert('That is outside the range.')
    }
}
```

```
function square() {
    let num = prompt('Give me a number')

    if (typeof num == 'number') {
      alert(num * num)
    } else {
      alert('You need to enter a number.')
    }
}
```

Your commit message could look something like this:

> Summary: Invalid testing
> Description: Tested the square() function with invalid inputs of -50, 0, 10, 1000, “hello” and nothing and got the correct error message output for each one..



Coding with JS

06

Improvements

Improving code is an important step towards maintaining a robust, flexible, and sustainable codebase. It's not just about making the code work—it's also about making it readable, maintainable, and scalable.

Inside are some key principles and practices to improve your code.

- Eliminating Magic Numbers
- Writing Comments
- Testing Code

Eliminate Magic Numbers - Use Constants

Magic numbers are numbers that appear in your code without any explanation. For example, if you've written a line like this:

```
let total = total * 7
```

It's not clear what `7` is. Why is it `7` and not `6` or `8`?

To someone else reading your code - or to you, when you come back to your code after a while - it won't be clear why you've chosen that number.

To eliminate magic numbers, we can define them as *constants* at the beginning of the program, giving them a meaningful name.

Constants are variables whose values don't change once they're set. In JavaScript, constants are declared using the const keyword followed by the variable name and value.

For instance, if the `7` was representing the number of days in a week, you could write:

```
const daysInTheWeek = 7
```

at the top of your code and then use **`daysInTheWeek`** instead of the unexplained `7`.

```
let total = total * daysInTheWeek
```

Constants make your code more readable and maintainable because they provide meaningful names for values that won't change. When you use constants instead of hardcoding values, it's easier to understand the purpose of those values, and if you need to change the value, you can do it in one place rather than hunting through your code for all instances where it was used.

Eliminate Magic Numbers - Use Derived Values

The use of derived values, such as arrayName.length(), can help to eliminate the need for these magic numbers by calculating the number as the code is running.

For instance, suppose you have an array of students in a class, and you want to loop through the array to print each student's name. A non-ideal way (with magic numbers) would look like this:

```
let students = ['Alice', 'Barbara', 'Charlie', 'Dorothy', 'Eva']

let index = 0
while (i < 5) {
    alert(students[index])
    index = index + 1
}
```

In this case, `5` is a magic number. It represents the number of students, but it's hardcoded, making it unclear what it means and potentially leading to errors if the size of the students array changes.

A better way to write this code would be to use the derived value **`students.length`** instead of the magic number:

```
let index = 0
while (i < students.length {
    alert(students[index])
    index = index + 1
}
```

Now the loop will automatically adjust to the size of the students array. If you add or remove students from the array, the loop will still work correctly without needing to adjust the `5` to a new number.

```
let students = ['Alice', 'Barbara', 'Charlie', 'Dorothy', 'Eva']

let studentIndex = prompt('Give me an index number.')

if (studentIndex >= students.length) {
  alert('That is not a valid index number.')
}
```

Write Comments

Comments are a key part of making your code understandable to others, as well as to your future self.

They should explain why the code is doing something, not what it's doing. The code itself shows what it's doing.

Here are some guidelines on writing good comments:

1. Start with a general comment at the top of your code file describing what the code does.
2. Comment on each function to explain what it does, what its inputs and outputs are.
3. Write comments for complex code blocks to explain how they work.
4. Avoid unnecessary comments for code that is self-explanatory.

Here are some examples of bad comments:

- *`// Input.`*
- *`// Adds a number.`*
- *`// Checks.`*

Here are some examples of good comments:

- *`// Calculates the Fibonacci sequence up to the specified maximum.`*
- *`// Loops over the array and adds GST to all of the sale prices.`*
- *`// Shows an error message for any invalid inputs.`*