The For Loop

Get familiar with the For loop along with some optimization techniques regarding loops.

Introduction

You will now learn another loop that is perfect for iteration: the *for* loop. This loop combines several lines of code into one. The execution of the for loop is identical to the while loop:

```
initialization_of_loop_variable;
while ( condition ) {
    statements;
    increment_loop_variable;
}

for ( initialization_of_loop_variable; condition; increment_loop_variable ) {
    statements;
}
```

The for loop has everything you need in one line. You don't have to mix your loop body with the increment of the loop variable. The execution of the for loop is as follows:

1. `initialization_of_loop_variable`
2. check `condition`. If `condition` is falsy, exit the for loop
3. `statements` in the loop body
4. `increment_loop_variable`, then go back to step 2

Rewrite the `sumArray` function using a for loop. Here is the example with the `while` loop:

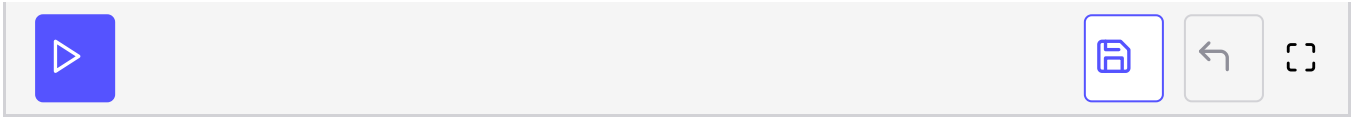
```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    let i = 0;
    while ( i < values.length ) {
```

```
        sum += values[i];
        i += 1;
    }

    return sum;
}

sumArray( numbers );
```



Don’t continue reading before you solve this exercise. Please change the while loop into a for loop.

I hope you liked the exercise. To verify your solution, check this code:

```
for ( let i = 0; i < values.length; i += 1 ) {
    sum += values[i];
}
```

This code should go in place of

```
let i = 0;
    while ( i < values.length ) {
        sum += values[i];
        i += 1;
    }
```

Notice you can declare variables in the initializer part of the for loop. The rest of the code works exactly like the `while` loop.

Let’s simplify the for loop a bit further: you can write `i++` or `++i` in place of `i += 1`:

```
for ( let i = 0; i < values.length; ++i ) {
    sum += values[i];
}
```

Some optimization techniques

In some programming competitions, people tend to count downwards instead of upwards. The reason is that computing `values.length` costs some nanoseconds more than checking if the value of `i` is positive:

```
for ( let i = values.length - 1; i >= 0; --i ) {
    sum += values[i];
}
```

These wannabe geniuses don't stop there by the way. They go even further:

```
for ( let i = values.length; i; sum += values[--i] );
```

Weird, isn't it? Instead of iterating from `12` to `0`, we iterate from `13` to `1`. The condition `i` is `true` as long as its value stays non-zero. This is the easiest check a computer can make with an integer.

Suppose `i` is `13`. `values[-i]` is interpreted as `values[12]`, because first we decrease the value of `i`, then we equate the expression `-i` to the new value of `i`. This is a unique case when writing `i-` would have been incorrect, because our array does not even have an element at index `13`.

Don't worry if you don't feel like understanding this optimization technique. Once you read this paragraph for the tenth time, you will most likely get it. It's normal. Oh, and by the way, I discourage using such optimizations. Readability always trumps these slight edges. Sometimes you can save seconds in your code, so don't bother with the nanoseconds. It's like being penny-wise and pound-fool.

Another reason to avoid looking smart and optimizing your code is that JavaScript interpreters actually optimize code for you. This is a very easy and straightforward optimization. You don't have to bother doing it yourself. Just stick to this version of the for loop:

```
for ( let i = 0; i < values.length; ++i ) {
    sum += values[i];
}
```