TASK

Beginner Control Structures — FOR LOOPS

Visit our website

Introduction

WELCOME TO THE BEGINNER CONTROL STRUCTURES — LOOPS TASK!

In this task, you will be exposed to the **for loop** structure to understand how it can be utilised in reducing lengthy code, preventing coding errors, and paving the way for code reusability. This task begins with recapping the while loop, which is the simplest loop in the group. You will then look at **for loops** and how loops can be nested within one another to solve more complex problems. Ready? Let's go!

WHAT IS A FOR LOOP?

A **for loop** is similar to a while loop. Either a **for loop** or a while loop can be used to repeat instructions. However, unlike a while loop, the number of repetitions in a **for loop** is known ahead of time. A **for loop** is a counter-controlled loop. It begins with a start value and counts up to an end value. In this way, it allows for counter-controlled repetition to be written in a really compact and clear way. Therefore, **for loops** tend to be easier to read.

In Javascript, a **for loop** has the following syntax:

```
for (initialization statement; stopping condition; iteration statement) {
      statement(s);
}
```

1. **The Initialization statement.** This will indicate where the loop begins by declaring/referencing a counter/index variable.
2. **The stopping condition.** The stopping condition acts as a conditional statement that will stop the loop when the expression we passed into the loop in this position evaluates to false.
3. **The iteration statement.** The iteration statement is where the counter variable that will be updated after each iteration in the loop is completed.

The for loop has very similar functionality to a while loop. You will notice that the for loop in the example below actually does all the same things as the first while loop we looked at previously, just using a different format. Compare the for loop below with the first while loop we looked at previously, and notice the three steps they both have in common:

```
for (let i = 1; i < 10; i++) {
       console.log(i);
};

// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Compare the syntax of each statement. Remember to make sure you use the correct syntax rules for each statement as you code using loops.

In the for loop above, while the variable **`i`** (which is an integer) is within the range from 1 to 9 (i.e. either 1, 2, 3, 4, 5, 6 ... or 9), the indented code in the body of the loop will execute. **`i = 1`** in the first iteration of the loop. So 1 will be logged in the first iteration of this code. Then the code will run again, this time with **`i=2`**, and 2 will be printed out...etc, until **`i=10`**. When **`i`** reaches 10 it is no longer <10, and so the condition evaluates to false and the code stops executing.

`i` is known as the counter/index variable as it can tell you the iteration or number of repetitions that the loop is on. In each iteration of the *for loop*, the code indented inside is repeated.

You can use an **if statement** within a for loop!

```
for (let i = 1; i < 10; i++) {
    if (i > 5) {
        console.log(i);
    }
}
```

The code in the example above will only print the numbers 6, 7, 8 and 9 because numbers less than or equal to 5 are filtered out.

A loop could also contain a **break statement**. Within a loop body, a break statement causes an immediate exit from the loop to the first statement after the loop body. The break allows for an exit at any intermediate statement in the loop.

```
break;
```

Using a break statement to exit a loop has limited but important applications. For example, as we discussed previously, a program may use a loop to input data from a file. The number of iterations depends on the amount of data in the file. The task

of reading from the file is part of the loop body, which becomes the place where the program discovers that data is exhausted. When the end-of-file condition becomes true, a break statement exits the loop.

```
let i = Number(prompt("Please enter a number"));  // User Enters 1

for ( i ; i < 10; i+=2) {
    console.log(i);
    if (i == 7) {
        break;
    }
}

// Output: 1, 3, 5, 7
```

LOOPS AND ARRAYS

For loops are quite convenient when it comes to looping over data structures. At this level, the data structure you will work with the most is an **array**. Arrays can be looked at as containers that hold more than one value - for example a list of car names, a list of numbers, etc. The most common use case of for loops in arrays would be when we want to perform certain functions on all the elements within an array.

To loop through an array, we have to change the structure of our stopping condition to loop through the length of the array. The arrays' length can be evaluated with the **.length** property. The .length property will return an integer value that represents the number of items in the array. This is always 1 more than the integer value of the last element (because array elements are numbered from zero). To loop through the entire array, we use the condition **`i < array.length`**

Look at the example below.

```
const colors = ["yellow", "blue", "orange", "green", "red"]; //set up array

for (let i = 0; i < colors.length; i++){
    // now output the value stored in the array at position i
     console.log(colors[i]);
}
```

We need to keep in mind that `i` is an iterator variable so it represents an integer value that is incremented after each iteration. We then use this to index each item in the loop according to its position in the array.

Also, as previously mentioned, arrays are zero-indexed (their indexes start from zero) so the index of the last element in the array is equivalent to the length of the array minus 1 (since the array will be starting from 0). In the loop above we have set our stopping condition to be when `i` reaches the value of the length of the array, so our for loop iterates over the exact number of items in the list, stopping at `colors[length-1]`.

The output from the code example above will be:

```
yellow
blue
orange
green
red
```

WHICH LOOP TO CHOOSE?

How do we know which looping structure — the **for loop** or the **while loop** — is more appropriate for a given problem? The answer lies with the kind of problem we are facing. After all, both these loops have the four components that were introduced to you, namely:

- The initialisation of the control variable
- The termination condition
- Updating the control variable
- The body to be repeated

A **while loop** is generally used **when we don't know how many times to run** through the loop. Usually, the logic of the solution will decide when we break out of the loop, and not a count that we have worked out before the loop has begun. For situations where we **know when we start how many times the loop needs to run**, we usually use the **for loop**.

For example, if you want to create a table with *ten rows* comparing various rand amounts with their equivalent dollar amount, then you will need to use a **for** loop because you know that it must run ten times. However, if you want to determine how many perfect squares there are below a certain number entered by a user, then we would want to use a **while** loop because (unless you're a maths savant)

you cannot tell how many times you will have to run through the loop before you find the solution.

NESTED LOOPS

A **nested loop** is simply a loop within a loop. Each time the outer loop is executed, the inner loop is executed right from the start through to its terminating condition. In other words, all the iterations of the inner loop are executed with each iteration of the outer loop.

The syntax for a nested for loop in another for loop is as follows:

```
for (iterating_var in sequence) {
   for (iterating_var in sequence) {
      statements(s);
   }
   statements(s);
}
```

You can put any type of loop inside of any other kind of loop. For example, a for loop can be inside a while loop, or vice versa as in the example below.

```
for (iterating_var in sequence) {
  while (condition) {
      statement(s);
  }
  statements(s);
}
```

The following program shows the potential of a nested loop:

```
for (let x = 1; x < 6; x++) {   // outer loop will execute 5 times
    /* inner loop  will execute 5 times for every iteration of outer loop,
       so  5 * 5 = 25 times in total */
    for (let y = 1 ; y < 6; y++) {
        console.log(String(x) + "*" + String(y) + "=" + String(x*y));
    }
    console.log("");
}
```

When the above code is executed, it produces the following result :

```
1 * 1 = 1
1 * 2 = 2
```

```
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10

3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15

4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20

5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
```

From the result above, can you see how each group of calculations corresponds to each outer loop iteration, and each line within any group corresponds to the inner loop iterations? Work through the code carefully until this is clear.

TRACE TABLES

As you start to write more complex code, trace tables can be a really useful way of desk-checking your code to make sure the logic of it makes sense. We do this by creating a table where we fill in the values of our variables for each iteration. This is particularly useful when we are iterating through nested loops. For example, let's look back to the code above and create a trace table:

x	y	x*y
1	1 2 3	1 2 3

	4 5	4 5
2	1 2 3 4 5	2 4 6 8 10
3	1 2 3 4 5	3 6 9 12 15
4	1 2 3 4 5	4 8 12 16 20
5	1 2 3 4 5	5 10 15 20 25

As you can see, because of the nature of nested loops, the inner loop iterates 5 times before the outer loop is iterated again. That means that **`x`** will remain the same value while **`y`** loops through all its iterations. Then, once the outer loop iterates, the inner loop restarts with another 5 iterations. This is represented by the trace table, where **`y`** cycles from 1-to 5 for each value of **`x`**. Practise drawing trace tables for your upcoming tasks to assist you with the logic — they can be very helpful when coding gets tricky!

Instructions

Open **Task 7's example.js loop examples** in Visual Studio Code and read through the comments before attempting these tasks.

Getting to grips with JavaScript takes practice. You will make mistakes in this task. This is to be expected as you learn the keywords and syntax rules of this programming language. You must learn to debug your code. To help with this remember that you can:

- Use either the JavaScript console or Visual Studio Code (or another editor of your choice) to execute and debug JavaScript in the next few tasks.

- Remember that if you get stuck, you can contact an expert code reviewer for help.

Compulsory Task 1

Follow these steps:

- Create a new JavaScript file in this folder called **numberMayhem.js**.
- Create a while loop that will display count down from 20 to 0.
- Next, create a loop (any kind) that will display all the even numbers between 1 and 20.
- Now, create a loop (any kind) that will produce the following output:

  *

  **

  ***

  ****

  *****

Compulsory Task 2

Follow these steps:

- Create a new JavaScript file called **vehicles.js**.
- Create an array named vehicles and populate it with 5 different cars of your choosing.
- Make use of a for loop to output (console.log) each car within the vehicles array after the string “I would love to drive a ……” .
- Your output should read as “ I would love to drive a [nameOfCar] ” repeated 5 times, once for each car in the array.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.