Hyperiondev

TASK

Control Structures - For and While Loops

Visit our website

Introduction

WELCOME TO THE CONTROL STRUCTURES - FOR AND WHILE LOOPS TASK!

In this task, you will be exposed to *loop structures* to understand how they can be utilised in reducing lengthy code, preventing coding errors, and paving the way for code reusability. This task begins with the *while loop*, which is the simplest loop of the group. You will then look at *for loops* and how loops can be nested within one another to solve more complex problems.

GET INTO THE LOOP OF THINGS

Loops are handy tools that enable programmers to do repetitive tasks with minimal effort. To count from 1 to 10, we could write the following program:

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
console.log(6);
console.log(7);
console.log(8);
console.log(9);
console.log(10);
```

The task will be completed correctly. The numbers 1 to 10 will be printed, but there are a few problems with this solution:

- **Efficiency:** repeatedly coding the same statements takes a lot of time.
- **Flexibility:** what if we wanted to change the start number or end number? We would have to go through and change them, adding extra lines of code where they're needed.
- **Scalability:** 10 repetitions are trivial, but what if we wanted 100 or even 100000 repetitions? The number of lines of code needed would be overwhelming and very tedious for a large number of iterations.
- **Maintenance:** where there is a large amount of code, the programmer is more likely to make a mistake.
- **Feature:** the number of tasks is fixed and doesn't change at each execution.

Using loops, we can solve all these problems. Once you get your head around them, they will be invaluable in solving many problems in programming.

Consider the following code:

```
let i = 0;
while (i < 10) {
    i++;                    // shorthand for i = i + 1
    console.log(i);
}
```

If we run the program, the same result is produced, but looking at the code, we immediately see the advantages of loops. Instead of executing 10 different lines of code, line 4 executes 10 times. 10 lines of code have been reduced to just 4. This is done through the update statement in line 3, that adds 1 to variable **`i`** each iteration until **`i == 10`**. Furthermore, we may change the number 10 to any number we like. Try it yourself, replace the 10 with another number.

Also note the first number that is output, and relate that back to the code. How could you rearrange the code, *without changing the initial value of* **`i`**, to output the numbers 0 to 9?

WHAT IS A WHILE LOOP?

A *while loop* is the most general form of loop statements. The *while statement* repeats its action until the controlling condition becomes false. In other words, the statements indented in the loop repeatedly execute "while" the condition is true (hence the name). The *while* statement begins with the keyword *while* followed by a boolean expression. The expression is tested before beginning each iteration or repetition. If the test is true, then the program passes control to the indented statements in the loop body; if false, control passes to the first statement after the body.

Syntax:

```
while (boolean expression) {
   statement(s);
}
```

The following code shows a while statement which sums successive even integers 2 + 4 + 6 + 8 + … until the total is greater than 250. An update statement increments *i* by 2 so that it becomes the next even integer. This is an event-controlled loop (as

opposed to counter-controlled loops, like the *for loop*) because iterations continue until some non-counter-related condition (event) stops the process.

```
let sum1 = 0
let i = 2                  // initial even integer for the sum
while (sum1 <= 250) {
    sum1 += i;
    i += 2;                // update statement, shorthand for i = i + 2
    console.log(sum1);
}
```

INFINITE LOOPS

A *while loop* runs the risk of running forever if the condition never becomes false. A loop that never ends is called an infinite loop. Creating an infinite loop will mean that your program will run indefinitely while never in fact moving to any code below the while loop - not a desirable outcome! Make sure that your loop condition eventually becomes false and that your loop is exited.

Consider the following code:

```
let a = 0;
while (a < 10) {
    a++;                   // shorthand for a = a + 1
    console.log(a);
}
```

In any loop, the following three steps are usually used:

1. **Declare a counter/control variable.** The code above does this when it says `a = 0;`. This creates a variable called `a` that contains the value zero.

2. **Increase the counter/control variable in the loop.** In the loop above, this is done with the instruction `a++` which increases `a` by one (*increments* **a**) with each pass of the loop.

3. **Specify a condition to control when the loop ends.** The condition of the while loop above is `a < 10`. This loop will carry on executing as long as `a` is less than ten. This loop will, therefore, execute 10 times. Would there be a difference in the output if the control variable, `a`, was incremented after the console.log statement? If so, what would the difference be?

THE DO WHILE CONDITIONAL

This structure has the same functionality as the *while loop*, with the exception of being guaranteed to iterate at least once. This is useful if you want your program to do something before variable evaluation actually begins.

```
let a = -10;
do {
    console.log("I've run at least once!");
    a++;
} while (a <= 1);
console.log("The result of a is " + a);
```

Considering the code above, how many times will the statement "I've run at least once" be output? What will the value of a that is output be?

WHAT IS A FOR LOOP?

This 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 in 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:

1. **Declare a counter/control variable.** The code below does this when it says `i = 1;`. This creates a variable called `i` that contains the value 1. Something similar was done with the *while loop* with the line `a = 0;`.
2. **Increase the counter/control variable in the loop.** In the *for* loop below, this is done with the instruction `i++` which increases `i` by one with each pass of the loop. Similarly, the *while loop* did this with the code `a++`.
3. **Specify a condition to control when the loop will end.** The condition of the *for loop* below is `i <= 10`. This loop will carry on executing as long as `i` is less than or equal to 10. This loop will, therefore, execute 10 times. The condition of the *while loop*, which also executed 10 times, was `a < 10`. Why is this the case?

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

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 in the range of 1 to 10 (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`**. Now **`i`** is not in the range (1,10), so the code will stop executing.

`i` is known as the index variable as it can tell you the iteration or repetition 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 (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, 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.

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 those situations, we usually use the *for loop*.

For example, if we want to create a table with *ten rows* comparing various rand amounts with their equivalent dollar amount, then we would use a *for* loop because we know that it must run ten times. However, if we 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 we cannot tell how many times we would have to run through the loop before we found 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. That is, 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);
}
```

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

```
while (condition) {
   while (condition) {
      statement(s);
   }
   statement(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.

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

The following program shows the potential of a nested loop:

```
for (x=1; x<6; x++) {
    for (y=1; y<6; y++) {
        console.log(String(x) + "*" + String(y) + "=" + String(x*y));
    }
    console.log("");
}
```

When the above code is executed, it produces 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
```

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 4 5	1 2 3 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 1-5 for the same 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 **example.js** 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 completely to be expected as you learn the keywords and syntax rules of this programming language. It is vital that you 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 really get stuck, you can contact an expert code reviewer for help.

Compulsory Task 1

Follow these steps:

- *Note: For this task, you will need to create an HTML file to get input from a user. If you need a refresher on how to do this, return to the **example.js** and **index.html** files in your previous task for a refresher.*
- Create a new JavaScript file called **swapping.js**.
- Write a program that asks the user to enter a number of at least 3 digits.
- Make use of a *for loop* to swap the second digit and last digit in the number
- Output the original number and the new number.

Compulsory Task 2

Follow these steps:

- *Note: For this task you will need to create an HTML file to get input from a user. If you need a refresher on how to do this, go back to the **example.js** and **index.html** files in your Task 2 folder for a refresher.*
- Create a JavaScript file called **palindrome.js**.
- Write a program that asks the user to enter a word.
- Using a *while loop* the program must determine whether or not the word is a palindrome, ie. whether it reads the same forwards and backwards, e.g. the word “racecar”.
- Output the given word and whether or not it is a palindrome.
- E.g. `“racecar is a palindrome”`

Compulsory Task 3

Follow these steps:

- *Note: For this task you will need to create an HTML file to get input from a user. If you need a refresher on how to do this, go back to the **example.js** and **index.html** files in your Task 2 folder for a refresher.*
- Create a new file called **while.js**.
- Write a program that always asks the user to enter a number.
- When the user enters -1, the program should stop requesting the user to enter a number,
- The program must then calculate the average of the numbers entered, excluding the -1.
- Make use of the *while loop* repetition structure to implement the program.
- Compile, save and run your file.

Compulsory Task 4

Follow these steps:

- *Note: For this task you will need to create an HTML file to get input from a user. If you need a refresher on how to do this, go back to the* ***example.js*** *and* ***index.html*** *files in your Task 2 folder for a refresher.*
- Create a JavaScript file in this folder called **quiz.js**.
- Create a *do-while loop* in which you ask the user a multiple choice question. If they choose the incorrect answer, they are asked if they want to try again, at which point they are asked the question again. The program ends when the user either guesses correctly, or they do not wish to guess again.
- Here is an example of the possible output:

```
What colour is the sky?
a. Purple
b. Pink
c. Blue
d. Yellow
Enter a, b, c or d: [user enters b]
This's incorrect! Would you like to try again? y/n: [user enters y]
[Question is asked again. User enters c]
That's correct! [Program exits]
```

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.