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 used to reduce lengthy code, prevent coding errors, and pave the way for code reusability. This task begins with the `while` loop, which is the simplest loop type. We 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);
```

Although the numbers 1 to 10 **will** be printed by the code above, 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 each line of code, 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.

Consider the following code:

```
// Initialise the number to start at 0.
let number = 0;

// Set a condition for the loop to repeat itself until 10 is reached
while (number < 10) {
  number++; // Increment number by 1 to ensure the output starts at 1 not 0
  console.log(number); // Output the count from 1 to 10
}
```

If we run the program, the same result is produced, but looking at the code, we immediately see the advantage of using a loop. Instead of executing 10 different lines of code, line 7 executes 10 times. The original 10 lines of code have been reduced significantly. This has been achieved through the update statement in line 6, which adds 1 to the variable **`number`** each iteration until **`number == 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?

WHILE LOOP

A *while loop* is the most general loop statement. 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 the beginning of each iteration or repetition. If the test evaluates to true, the program passes control to the indented statements in the loop body; if false, control passes to the first statement after the loop body.

Syntax:

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

The following code shows a while statement that sums successive even integers (i.e., 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) becomes true and stops the process.

```
// Initialise variable to store the sum
let sum = 0;

// Initialise even integer to start the sum
let evenIntegar = 2;

// Set the condition for the loop to repeat as long as the sum is <= 250
while (sum <= 250) {
  sum += evenIntegar; // Sum = current value of sum plus the next even integer
  evenIntegar += 2; // Increment the even integer by 2 each time
  console.log(sum); // Return the sum
}
```

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 first example we discussed:

```
let number = 0;
while (number < 10) {
  number++;
  console.log(number);
}
```

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

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

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

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

DO WHILE LOOP

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

```
// Initialise the variable with a value of -10
let counter = -10;

// Return message until the condition is met
do {
  console.log("I've run at least once!");
  counter++; // Increment the counter by 1
} while (counter <= 1); // Loop will repeat as long as the counter is <= 1

// Returns the value of the counter once the loop ends
console.log("The result of the counter is " + counter);
```

Considering the code below, how many times will the statement "I've run at least once" be output? What will the value of that output be? If you're struggling to answer these questions, you might find that using something called a *trace table* could help you. The video **Dry-running algorithms with trace tables** introduces how to use this helpful tool within the first five minutes. It goes on to give more complex examples that you may wish to return to later in your learning journey.

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.

```
// Iterate through the loop 10 times
for (let i = 1; i <= 10; i++) {
  // Return the value of the variable after each iteration
  console.log(i);
}
```

Compare the *for loop* above 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 `number = 0;`.
2. **Increase the counter/control variable in the loop.** In the *for loop*, 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 `number++`.
3. **Specify a condition to control when the loop will end.** The condition of the *for loop* 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.

Here is a further explanation of the *for loop*. 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. The statement `i = 1` will be executed in the first iteration of the loop, so 1 will be output 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., continuing until `i=10`. At that point `i` is no longer 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*! The code in this example will only print the numbers 6, 7, 8, and 9 because numbers less than or equal to 5 are filtered out.

```
// Iterate through the loop less than 10 times
for (i = 1; i < 10; i++) {
  // Set condition to check if i is greater than 5
```

```
  if (i > 5) {
    // If the value is greater than 5, return the result.
    console.log(i);
  }
}
```

BREAK STATEMENT

A loop can 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 such as in the example below.

```
// Iterate through the loop less than 10 times
for (let i = 1; i < 10; i++) {
  // If the counter reaches 5, the loop breaks
  if (i === 5) {
    // Exit the loop
    break;
    }
    // Return the result at the point when the loop breaks
  console.log("Counter: " + i);
}
```

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. In this situation, the number of iterations will depend on the amount of data in the file as once the data runs out, there is nothing more to output. The task of reading from the file is part of the loop body, which becomes the place where the program discovers that the data is exhausted. When the end-of-file condition becomes true, a break statement ensures a neat exit from the loop.

Extra resource

Remember to make sure you use the correct syntax rules for each statement as you code using loops. This will ensure that your code not only executes without any unexpected errors, but is also easily readable and maintainable. For further insight into syntax rules in the context of using loops and iteration, see **here**.

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:

- Initialisation of the control variable
- Setting of a termination condition
- Updating of the control variable
- Repetition of the loop body

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. Conversely, for situations where a predetermined count is applicable, we usually use a *for loop*.

For example, if we want to create a table with *ten rows* comparing rand (R) amounts with their equivalent dollar ($) amounts, 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 means that *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) {
    indented statements(s);
  }
  indented statements(s);
}
```

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

```
while (condition) {
  while (condition) {
    indented statement(s);
  }
  indented 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) {
    indented statement(s);
  }
  indented statements(s);
}
```

The following program shows the potential of a nested loop. We'll step through what this program does in the next section, so if you're confused at this point don't panic - just read through the code slowly and try to follow what happens.

```
// Outer for loop for first multiplier
for (let firstNumber = 1; firstNumber < 6; firstNumber++) {
  // Inner for loop for second multiplier
  for (let secondNumber = 1; secondNumber < 6; secondNumber++) {
    // Return product of first multiplier and second multiplier
    console.log(
      String(firstNumber) +
        "*" +
        String(secondNumber) +
        "=" +
        String(firstNumber * secondNumber)
    );
  }
  // Return empty string between calculations of different multipliers
  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
```

If you're at all confused by nested loops, use a trace table to trace exactly what is happening within the code as each statement executes. We'll look at this tool in more detail in the next section.

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. If you watched the

video we recommended earlier, you probably already have the basic idea of what a trace table does. For further clarity, let's look back to the previous code example and create a trace table:

firstNumber	**secondNumber**	**firstNumber * secondNumber**
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 the `firstNumber` will remain the same value while the `secondNumber` loops through all its iterations. Then, once the outer loop iterates and its value finally increases, the inner loop restarts with another 5 iterations. This can be seen in the trace table, where the `secondNumber` cycles from 1-5 for each value of the `firstNumber`. Practice drawing trace tables for your upcoming tasks to assist you with the logic — they can be very helpful when coding gets tricky!

Instructions

Read and run the accompanying examples files provided before doing the Compulsory Task to become more comfortable with the concepts covered in this task.

Compulsory Task 1

Follow these steps:

- *Note: For this task, you will need to create an HTML file to get input from a user. Refer to your previous learning material for a refresher if needed.*
- 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. Refer to your previous learning material for a refresher if needed.*
- 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” OR “car is not a palindrome”`

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.