Hyperiondev

TASK

Javascript II:

Conditional Statements and Looping

Visit our website

Introduction

WELCOME TO THE SECOND JAVASCRIPT TASK!

This task will introduce you to 2 types of JavaScript control structures: *if statements* and *loops*. These are important structures that can be used to greatly enhance the functionality of the code you write. 'If' statements allow your code to make decisions and loops allow your code to repeat instructions.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

CONTROL STRUCTURES IN JAVASCRIPT

Usually, a program is executed one instruction at a time in order. The *control structures* within JavaScript allow the flow of your program to change within a unit of code or function. These statements can determine whether or not given statements are executed, or they can repeat the execution of a block of code.

DECISION OR SELECTION CONTROL STRUCTURE

Decision-making is a basic building block for all applications. Most programs do not simply go through a set of instructions, like a recipe. There is a higher logic to the code that takes into account various options and possible divergence in the execution of a program. In this task, we will learn how to write programs that can make decisions.

THE IF CONDITIONAL

Often you will want to perform different actions based on different conditions. In order to do this, you will need to use a conditional statement. The **if statement** is an example of a conditional statement. As the name suggests, it is essentially a question. It enables us to compare two or more variables or scenarios and perform a certain action based on the outcome of that comparison.

If statements contain a condition. The condition is the part of the *if statement* in brackets in the example below. Conditions are statements that can only evaluate to true or false. If the condition is true, then the indented statements are executed. If the condition is false, then the indented statements are skipped.

In JavaScript, *if statements* have the following general syntax:

```
if (condition) {
        indented statements;
}
```

Below is an example of an **if** statement in JavaScript. A condition, (`a === b`), is evaluated and, if it is true, then the block of code in the braces below will execute.

```
let a = 10;
let b = 10;

if (a === b) {
   console.log("The variable a is to equal the variable b.");
}
```

An if statement has two parts to it:

1. The condition in brackets which can either be true or false and is checked to determine whether to execute the statements in the body of the 'if' statement.
2. The body of code that contains statements is enclosed by curly brackets/braces (**{ }**) which are executed if the condition evaluates to true. If the condition evaluates to false, then the statements are ignored and the statement directly after the if statement is executed.

Note that three equals signs are used in variable comparison for JavaScript. The symbol **===** is a comparison operator. Comparison operators are similar to the normal operators that we encountered in prior lessons, such as +, -, * and /. The difference between the mathematical operators and comparison operators is that comparison operators evaluate to a boolean value of true or false, and not a numeric value. The table below shows some more comparison operators.

COMPARISON OPERATORS

As a programmer, it's important not to forget the basic logical commands. We use comparison operators to compare values or variables in programming. These operators work well with *if statements* and *loops* to control what goes on in our programs. You can see comparison operators in use in the example code files for this task. The full set of comparison operators is tabulated below.

Operator	Description	Example
>	greater than	Condition: 12 > 1 Result: True
<	less than	Condition: 12 < 1 Result: False
>=	greater than or equal to	Condition: 12 >= 1 Result: True
<=	less than or equal to	Condition: 12 <= 12 Result: True
==	equals	Condition: 12 == 1 Result: False
===	Equal value and equal type	Condition: 12 === twelve Result: False

!=	does not equal	Condition: 12 != 1 Result: True

Take note that the symbol we use for 'equal to' is '==', and not '='. This is because '==' literally means 'equals' (e.g. if (*i == 4)* means if *i* is equal to 4). We use '==', two equals signs, in conditional statements to check if the values are the same. Similarly, we use '===', 3 equals signs, in conditional statements to check if the value *and* type are the same (e.g. 5 === 5 would return true, but 5 === "5" would return false).

On the other hand, '=', a single equals sign, is used to assign a value to a variable (e.g. *i = "blue"* means I assign the value of *"blue"* to *i*). It is a subtle but important difference. You're welcome to check back in the previous tasks if you feel like you need a refresher on variables.

ELSE STATEMENTS

If statements are one of the most important concepts in programming, but on their own, they are a bit limited. The *else statement* represents an alternative path for the flow of logic if the condition of the if statement turns out to be *false*.

Imagine if you were hungry and you sent your friend to the shop to buy chocolate. When they get to the shop, they find no chocolate and just leave because you told them of no alternatives. They would have to keep coming back for instructions unless you provided them with an alternative. If you said "If there is chocolate, buy some, otherwise buy any other bag of sweets" then even if chocolate was out of stock, your friend would be likely to return with something for you.

Instead of us having many 'if' statements to test each scenario, we can add an 'else' statement to give us a single alternative.

In JavaScript, the general if-else syntax is:

```
if (condition) {
    indented statements;
}
else {
    indented statements;
}
```

If the condition turns out to be *false*, the statements in the indented block following the *if* statement are skipped and the statements in the indented block following the *else* statement are executed. Continuing our previous example, the

logic behind your friend's trip to the shop could be expressed in pseudo-code something like:

```
if (there is chocolate){
        buy some chocolate;
}
else {
        buy any other bag of sweets;
}
```

Now that you've got the basic idea, take a look at the following JavaScript example:

```
let num = 10;
if (num < 12) {
    console.log("the variable num is lower than 12");
}
else {
    console.log("the variable num is greater than or equal to 12");
}
```

Now instead of nothing happening if the condition of the if statement is *False* (*num* ends up being greater than or equal to 12), the else statement will be executed.

Another example of using an else statement with an if statement can be found below. The value that the variable *hour* holds determines which string is assigned to *greeting*.

```
if (hour < 18) {
     greeting = "Good morning";
}
else {
     greeting = "Good evening";
}
```

We are faced with decisions like this on a daily basis. For instance, if it is cold outside you would likely wear a jacket. However, if it is not cold you might not find a jacket necessary. This is a type of branching. If one condition is true, you do one thing and if the condition is false, you do something else. As you have seeb, this type of branching decision making can be implemented in JavaScript using 'if-else' statements.

ELSE IF STATEMENTS

The last piece of the puzzle when it comes to if statements is called an *else-if statement*. With this statement, we can add more conditions to the if statement. This empowers us to test multiple parameters in the same statement.

Unlike in an *if/else statement*, you can have multiple *else-if* statements in an *if/else-if/else* (read "if, else-if, else") statement. If the condition of the if statement is *false*, the condition of the next else-if statement is checked. If the first else-if statement condition is also *false*, the condition of the next else-if statement is checked, and then the next, etc. If all the else-if conditions are *false*, the *else statement* with its indented block of statements is executed.

In JavaScript, *if/else-if/else statements* have the following syntax:

```
if (condition1) {
      indented statements;
}
else if (condition2) {
      indented statements;
}
else if (condition3) {
      indented statements;
}
else if (condition4) {
      indented statements;
}
else {
      indented statements;
}
```

Look at the following example:

```
let num = 10;

if (num >12) {
    console.log("the variable num is greater than 12");
}
else if (num > 10) {
    console.log("the variable num is greater than 10");
}
else if (num < 5) {
    console.log("the variable num is less than 5");
}
else {
    console.log("the variable num is between 5 and 10");
```

```
}
```

Remember that you can combine if, else, and else-if into one big conditional statement.

Some important points to note on the syntax of if/else-if/else statements:

- Make sure that the if/else-if/else statements end with an open curly bracket and the code in that block ends with a closing curly bracket.
- Ensure that your indentation is done correctly (i.e. statements that are part of a certain control structure's 'code block' need the same indentation).
- To have an *else if* you must have an *if* above it.
- To have an *else* you must have an *if*, or another *else if*, above it.
- You can't have an *else* without an *if* — think about it!
- You can have many *else if* statements under an *if*, but only one *else* right at the bottom. It's the fail-safe statement that executes if the other *if/else-if* statements fail!

Sometimes you may need to use logical operators: AND (**&&**), OR (**||**) or NOT (**!**). The AND and OR operators can be used to combine boolean expressions. The resulting expression will still evaluate to either true or false. For example, consider the code below:

```
let num = 12;
if (num >= 10 && num <= 15){
    console.log(num + " is a value between 10 and 15");
}
```

The boolean expression (**num >= 10 && num <= 15**) evaluates to true in the example above because both the expression **num >= 10** AND the expression **num <= 15** are true. This is an example of a conjunction operation where both conditions need to be true for the whole statement to be true. The symbol for the AND operator is **&&.**

To illustrate how the OR operator is used, consider a real-life example: you could buy a very nice car if you have enough money OR if someone gives you the money as a gift OR if you can get a loan.

```
let lotsOfMoney = false;
let receivedGift = false;
```

```
let loanApproved = true;

if (lotsOfMoney || receivedGift || loanApproved){
    console.log("Can purchase a car");
} else {
       console.log("Sorry! Can't afford a car");
}
```

The example is a disjunction operation where at least one of the conditions needs to be true for the whole statement to be true. The boolean expression (`lotsOfMoney || receivedGift || loanApproved`) is true because at least one of the expressions that make up that compound expression is true. The `||` symbols are used for the OR operation.

Think about it

- Without running the code, what will the output of the above code block be?
- Would anything change if the value of lotsOfMoney was changed to true?
- If we changed the operators in the logical expression to ANDs (`&&`) rather than ORs (`||`), but kept the values of the variables the same as they are in the original example, what would the output be?

Try this:

- Open the JavaScript Console.
- Copy and paste the code above into the console.
- Execute the code and take note of the output. Make sure you understand why your code produced the output it did.

THE SWITCH CONDITIONAL

Now that you are comfortable using conditional statements, we can simplify them a bit into a switch statement. This structure allows multiple conditions to be evaluated with a much more efficient and structured layout than using a long series of *else-if* statements. The syntax of the **switch** statement is given below:

```
switch(expression)
{
       case 1:
              Do something;
              break;
       case 2:
              Do something else;
              break;
       default:
              Else do this;
              break;
```

```
}
```

See the example of the **switch** statement in action below. The example statement uses the *getDay* function to get the current day of the week. If `Date().getDay() == 0`, then the variable 'day' is assigned the value "Sunday", else if `Date().getDay() == 1`, then the variable 'day' is assigned the value "Monday" etc.

```
let day = new Date().getDay();
/*"new Date().getDay()" is built-in JavaScript code. It returns a value from 0 to 6 corresponding to the dates of the week, starting from Sunday. To see more about how this works, visit this site: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/getDay */
console.log("The value of day is " + day);
switch (day) {
    case 0:
       day = "Sunday";
        break;
    case 1:
       day = "Monday";
        break;
    case 2:
       day = "Tuesday";
        break;
    case 3:
       day = "Wednesday";
        break;
    case 4:
       day = "Thursday";
        break;
    case 5:
       day = "Friday";
        break;
    case 6:
       day = "Saturday";
}
console.log("Today is " + day);
```

SPOT CHECK 1

Let's see what you can remember from this section.

1. Put the following lines of code in the correct order:

```
else if (eyeColour == "green") {
if (eyeColour == "blue") {
else {
```

```
else if (eyeColour == "hazel") {
   console.log("I don't know what colour eyes you have.");
   console.log("You have hazel eyes.");
   console.log("You have green eyes.");
   console.log("You have blue eyes.");
let eyeColour = "brown";
}
}
}
}
```

2. What are you expecting the output to be?

REPETITION OR LOOPING CONTROL STRUCTURES

We have seen a number of statements that allow us to write code that reacts differently, based on whether a condition is true or not. We now consider control structures that allow you to repeat sections of code.

GET INTO THE LOOP OF THINGS

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

```
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 and 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 1000 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, one 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, you'll find loops are invaluable in solving many problems in programming!.

THE WHILE CONDITIONAL

This structure will repeatedly execute a block of code for as long as the evaluation condition is met. For example, consider the code below. The instructions contained in the braces { }, will be repeated for as long as (or *while*) the variable 'a' is less than or equal to 10. As soon as 'a' is no longer less than or equal to 10 (i.e. as soon as 'a' equals 11), the instructions will no longer be repeated. This repetition of code is known as ***looping***.

Syntax:

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

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 (in other words, increments *i*) until *i == 10*. Furthermore, we may change the

number 10 to any number we like, making it really easy to change the number of times the loop runs. Try it yourself - replace the 10 with another number.

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 (and sometimes statements) for each iteration of a loop.

Try this:
Fill out the table below to trace the execution of the code in the above example through the iterations of the *while loop*.

Iteration of the loop	**Starting value of *i***	**What is printed to the console**
1st	0	1
2nd	1	2
3rd	2	3
4th	3	4
5th	4	5
6th	5	6
7th	6	7
8th	7	8
9th	8	9
10th	9	10

Let's look at the example again and change it slightly.

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

Notice that the starting value of *i* is now 1. Consider the following questions (you can draw a trace table similar to the one we used above to help you work these out if necessary):

- What is the highest number the loop above will output to the console?
- How many values will the loop output to the console?
- Without changing the starting value of *i*, could you make a change to the code to get the same results output to the console as in the original example, and what would this change be?

Let's look at the example and change it slightly again.

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

Notice that the starting value of *i* is now back to 0, but the incrementing of *i* now takes place after the console output code. Consider the following questions (you can once again draw a trace table similar to the one we used above to help you work these out if necessary):

- What is the highest number the loop will output to the console?
- How many values will the loop output to the console?
- Without changing the starting value of *i*, could you make a change to the code to get the same results output to the console as in the original example, and what would this change be?

If you've worked through these questions properly, you should now have a relatively good, deep understanding of the flow of control through the while loop with different parameters. Let's consider some more examples.

The following code shows a while loop which sums successive even integers 2 + 4 + 6 + 8 + … until the total is greater than 250. An update statement increments *a* 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* or the previous *while loop* example) because iterations continue until some non-counter-related condition (event) stops the process.

```
let a = 2;                  //initial even integer
let sum = 0;                // initial even integer for the sum
while (sum <= 250) {        //loop until the total is greater than 250
    sum+= a;                //update statement, shorthand for sum = sum + a
    console.log(sum);     // output the sum
    a+= 2;              // update statement, shorthand, increment by 2
}
```

Try this:

- Fill out the following trace table, tracking what happens in each statement on each iteration of the loop. The first few iterations have been done for you.

Loop iteration	**a**	**sum**	**sum<=250**	**sum+=a**	**console.log(sum)**	**a+=2**
1st	2	0	true	2	2	4
2nd	4	2	true	6	6	6
3rd	6	6	true	12	12	8
4th	8	12	true	20	20	10
5th						
6th						
7th						
8th						
9th						
10th						
11th						
12th						
13th						
14th						
15th						
16th						
17th			false - loop thus does not run again			

Think about it

- What would you have to change in the code, and to what, to make the last value output to the console be 240?

Try this:

- Open the JavaScript Console.
- Copy and paste the code above into the console.

- Execute the code and take note of the output. Compare the output to the trace table you have filled in above and ensure that the values correlate. Make sure you understand why your code produced the output it did.

INFINITE LOOPS

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 with each pass of the loop.
3. **Specify a condition to control when the loop ends.** The condition of the for 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.

A *while loop* **runs the risk of running forever** if the condition that should control when the loop ends never becomes false. A loop that never ends is called an infinite or non-terminating loop. Creating an infinite loop is not desirable! Here's an example (don't run this in the console):

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

FIll in the trace table below for this code for just the first 4 iterations of the loop.

Loop iteration	a	b	a<b	a++	console.log(a)	b+=a

1st	0	1	true	1	1	2
2nd						
3rd						
4th						

What happens? Can you see why the loop will continue to run and never stop? When you are coding, **make sure that your loop condition eventually becomes false and that your loop is exited**.

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 value of a when the loop finally terminates is " + a);
```

THE 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. Here is the *while loop* again, for comparison purposes:

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

Now, here is a for loop that does the same thing with slightly different syntax:

```
for (i = 1; i < 11; i++) {
      console.log(i);
```

```
}
```

Compare the code used in the first while loop and the for loop, both pictured above, and note 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 `i = 0;`. - the counter simply started at zero instead of 1.

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. The while loop did this with the exact same code, `i++`.

3. **Specify a condition to control when the loop will end.** The condition of the for loop below is `i < 11`. This loop will carry on executing as long as *i* is less than 10. This loop will, therefore, execute 10 times. The condition of the while loop was `i < 10`, but because it's counter started at zero, it also executed 10 times.

Compare the syntax of each statement in the two loops. 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 … up to 10), 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 to the console (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. until *i=10*. At this point *i* increments to 11, so it is no longer in the range (1,10), and 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* (or in fact any loop for that matter) as shown below:

```
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 5 are filtered out.

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.

```
break;
```

Here's an example of using a break statement to exit from a loop depending on a conditional:

```
for (i=1; i<10; i++) {
    if (i > 5) {
        console.log(i);
    }
    if (i == 8) {
        break;
    }
 }
```

The code in the example above will only print the numbers 6, 7, and 8 but *not* 9 - numbers less than 5 remain filtered out, and the second *if statement* applies a *break* statement to exit from the loop when the value of *i* has become 8. Note that because the console.log statement is before the conditional to test if *i* is 8 and then break, 8 *is* printed. If we changed the code as shown in the example below, placing the break statement *before* the console.log statement, it would only output 6 and 7 because the flow of control would break out of the loop before 8 was printed.

```
for (i=1; i<10; i++) {
   if (i == 8) {
       break;
   }

   if (i > 5) {
        console.log(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. The number of iterations can be set to depend 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 can be used to exit 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:

- Initialisation of control variable
- Termination condition
- Update control variable
- 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.

SPOT CHECK 2

Let's see what you can remember from this section.

1. When would you use a *for loop*?
2. When would you use a *while loop*?
3. When would you use a *do while loop*?
4. How can you avoid an *infinite* loop?

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 are 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 6

	3 4 5	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 can be seen in the trace table, where *y* cycles repeatedly 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 *all* the examples for this task in VSCode 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 VSCode (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 a code reviewer for help.

Compulsory Task 1

Follow these steps:

- Create a .js file called **controlStructures.js**. You don't have to create an HTML page for this task. You can write all your output to the console.
- Declare three variables called "num1", "num2", and "num3". Assign each variable a number value.
- Now write a conditional statement that compares "num1" and "num2" and displays the larger value.
- Write a conditional statement that determines whether "num1" is odd or even and displays the result. (Tip: remember the modulus operator %)
- Next, write a JavaScript conditional statement to sort the three numbers from largest to smallest.
- Write a JavaScript *for loop* that will display the numbers 0 - 20.
- Now write a JavaScript *while loop* that will display the numbers 0 - 20.
- Next, create a *for loop* that will display all even numbers between 1 and 20.
- Create a *for loop* that will produce the following output:

  *

  **

  ***

  ****

  *****

  Use nested loops (i.e. a loop within a loop). *Tip: Refer to the example of nested loops in example.js if you need help to see how a nested loop works.*

- Write a for loop to compute the greatest common divisor (GCD - also called the Highest Common factor (HCF)) of two positive integers of your choice.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Completed the task(s)?

Ask an expert to review your work!

Review work

Things to look out for:

1. Make sure that you have installed and set up all programs correctly. You have set up **Dropbox** correctly if you are reading this, but **Visual Studio Code** may not be installed correctly.

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.

SPOT CHECK 1 ANSWERS:

1.

```
let eyeColour = "brown";

if (eyeColour == "blue") {
   console.log("You have blue eyes.");
}
else if (eyeColour == "green") {
   console.log("You have green eyes.");
}
else if (eyeColour == "hazel") {
   console.log("You have hazel eyes.");
}
else {
   console.log("I don't know what colour eyes you have.");
}
```

2. The output would be: “I don’t know what colour eyes you have.”

SPOT CHECK 2 ANSWERS:

1. You would use a *for loop* when you know the exact number of times that the loop should run.
2. You would use a *while loop* when you’re not sure how many times the loop must run, but you know when the loop needs to stop.
3. You would use a *do while loop* when you’re not sure how many times the loop must run, but you know when the loop needs to stop *and you need the loop to run at least once*.
4. You can avoid an infinite loop by making sure that your loop has a condition that will cause the loop to end. Without it, the loop can theoretically go on forever.