5 Loops

We are starting to get a good basic grasp of JavaScript. This chapter will focus on a very important control flow concept: loops. Loops execute a code block a certain number of times. We can use loops to do many things, such as repeating operations a number of times and iterating over data sets, arrays, and objects. Whenever you feel the need to copy a little piece of code and place it right underneath where you copied it from, you should probably be using a loop instead.

We will first discuss the basics of loops, then continue to discuss nesting loops, which is basically using loops inside loops. Also, we will explain looping over two complex constructs we have seen, arrays and objects. And finally, we will introduce two keywords related to loops, `break` and `continue`, to control the flow of the loop even more.

> There is one topic that is closely related to loops that is not in this chapter. This is the built-in `foreach` method. We can use this method to loop over arrays, when we can use an arrow function. Since we won't discuss these until the next chapter, `foreach` is not included here.

These are the different loops we will be discussing in this chapter:

- `while` loop
- `do while` loop
- `for` loop
- `for in`
- `for of` loop

Note: exercise, project, and self-check quiz answers can be found in the *Appendix*.

while loops

The first loop we will discuss is the **while loop**. A `while` loop executes a certain block of code as long as an expression evaluates to `true`. The snippet below demonstrates the syntax of the `while` loop:

```
while (condition) {
  // code that gets executed as long as the condition is true
}
```

The `while` loop will only be executed as long as the condition is `true`, so if the condition is `false` to begin with, the code inside will be skipped.

Here is a very simple example of a `while` loop printing the numbers 0 to 10 (excluding 10) to the console:

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

The output will be as follows:

```
1
2
3
4
5
6
7
8
9
```

These are the steps happening here:

1. Create a variable, `i`, and set its value to zero
2. Start the `while` loop and check the condition that the value of `i` is smaller than 10

3. Since the condition is true, the code logs `i` and increases `i` by 1
4. The condition gets evaluated again; 1 is still smaller than 10
5. Since the condition is true, the code logs `i` and increases `i` by 1
6. The logging and increasing continues until `i` becomes 10
7. 10 is not smaller than 10, so the loop ends

We can have a `while` loop that looks for a value in an array, like this:

```
let someArray = ["Mike", "Antal", "Marc", "Emir", "Louiza", "Jacky"];
let notFound = true;

while (notFound && someArray.length > 0) {
  if (someArray[0] === "Louiza") {
    console.log("Found her!");
    notFound = false;
  } else {
    someArray.shift();
  }
}
```

It checks whether the first value of the array is a certain value, and when it is not, it deletes that value from the array using the `shift` method. Remember this method? It removes the first element of the array. So, by the next iteration, the first value has changed and is checked again. If it stumbles upon the value, it will log this to the console and change the Boolean `notFound` to `false`, because it has found it. That was the last iteration and the loop is done. It will output:

```
Found her!
false
```

Why do you think the `&& someArray.length > 0` is added in the `while` condition? If we were to leave it out, and the value we were looking for was not in the array, we would get stuck in an infinite loop. This is why we make sure that we also end things if our value is not present, so our code can continue.

But we can also do more sophisticated things very easily with loops. Let's see how easy it is to fill an array with the Fibonacci sequence using a loop:

```
let nr1 = 0;
let nr2 = 1;
let temp;
```

```
fibonacciArray = [];

while (fibonacciArray.length < 25) {
  fibonacciArray.push(nr1);
  temp = nr1 + nr2;
  nr1 = nr2;
  nr2 = temp;
}
```

In the Fibonacci sequence, each value is the sum of the two previous values, starting with the values 0 and 1. We can do this in a `while` loop as stated above. We create two numbers and they change every iteration. We have limited our number of iterations to the length of the `fibonacciArray`, because we don't want an infinite loop. In this case the loop will be done as soon as the length of the array is no longer smaller than 25.

We need a temporary variable that stores the next value for `nr2`. And every iteration we push the value of the first number to the array. If we log the array, you can see the numbers getting rather high very quickly. Imagine having to generate these values one by one in your code!

```
[
     0,     1,     1,     2,     3,
     5,     8,    13,    21,    34,
    55,    89,   144,   233,   377,
   610,   987,  1597,  2584,  4181,
  6765, 10946, 17711, 28657, 46368
]
```

Practice exercise 5.1

In this exercise we will create a number guessing game that takes user input and replies based on how accurate the user's guess was.

1. Create a variable to be used as the max value for the number guessing game.
2. Generate a random number for the solution using `Math.random()` and `Math.floor()`. You will also need to add 1 so that the value is returned as 1-[whatever the set max value is]. You can log this value to the console for development to see the value as you create the game, then when the game is complete you can comment out this console output.

3. Create a variable that will be used for tracking whether the answer is correct or not and set it to a default Boolean value of `false`. We can update it to be `true` if the user guess is a match.
4. Use a `while` loop to iterate a prompt that asks the user to enter a number between 1 and 5, and convert the response into a number in order to match the data type of the random number.
5. Inside the `while` loop, check using a condition to see if the prompt value is equal to the solution number. Apply logic such that if the number is correct, you set the status to `true` and break out of the loop. Provide the player with some feedback as to whether the guess was high or low, and initiate another prompt until the user guesses correctly. In this way we use the loop to keep asking until the solution is correct, and at that point we can stop the iteration of the block of code.

do while loops

In some cases, you really need the code block to be executed at least once. For example, if you need valid user input, you need to ask at least once. The same goes for trying to connect with a database or some other external source: you will have to do so at least once in order for it to be successful. And you will probably need to do so as long as you did not get the result you needed. In these cases, you can use a **do while loop**.

Here is what the syntax looks like:

```
do {
  // code to be executed if the condition is true
} while (condition);
```

It executes what is within the `do` block, and then after that it evaluates the `while`. If the condition is `true`, it will execute what is in the `do` block again. It will continue to do so until the condition in the `while` changes to `false`.

We can use the `prompt()` method to get user input. Let's use a `do while` loop to ask the user for a number between 0 and 100.

```
let number;
do {
  number = prompt("Please enter a number between 0 and 100: ");
} while (!(number >= 0 && number < 100));
```

Here is the output; you will have to enter the number in the console yourself here.

```
Please enter a number between 0 and 100: > -50
Please enter a number between 0 and 100: > 150
Please enter a number between 0 and 100: > 34
```

Everything behind the `>` is user input here. The `>` is part of the code; it is added by the console to make the distinction between console output (`Please enter a number between 0 and 100`) and the console input (`-50`, `150`, and `34`) clearer.

It asks three times, because the first two times the number was not between 0 and 100 and the condition in the `while` block was true. With `34`, the condition in the `while` block became false and the loop ended.

Practice exercise 5.2

In this exercise, we will create a basic counter that will increase a dynamic variable by a consistent step value, up to an upper limit.

1. Set the starting counter to 0
2. Create a variable, `step`, to increase your counter by
3. Add a `do while` loop, printing the counter to the console and incrementing it by the `step` amount each loop
4. Continue to loop until the counter is equal to 100 or more than 100

for loops

for loops are special loops. The syntax might be a little bit confusing at first, but you will find yourself using them soon, because they are very useful.

Here is what the syntax looks like:

```
for (initialize variable; condition; statement) {
  // code to be executed
}
```

Between the parentheses following the `for` statement, there are three parts, separated by semi-colons. The first one initializes the variables that can be used in the `for` loop. The second one is a condition: as long as this condition is true, the loop will keep on iterating. This condition gets checked after initializing the variables before the first iteration (this will only take place when the condition evaluates to true). The last one is a statement. This statement gets executed after every iteration. Here is the flow of a `for` loop:

1. Initialize the variables.
2. Check the condition.
3. If the condition is true, execute the code block. If the condition is false, the loop will end here.
4. Perform the statement (the third part, for example, `i++`).
5. Go back to step 2.

This is a simple example that logs the numbers 0 to 10 (excluding 10) to the console:

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

It starts by creating a variable, `i`, and sets this to `0`. Then it checks whether `i` is smaller than 10. If it is, it will execute the log statement. After this, it will execute `i++` and increase `i` by one.

> If we don't increase `i`, we will get stuck in an infinite loop, since the value of `i` would not change and it would be smaller than 10 forever. This is something to look out for in all loops!

The condition gets checked again. And this goes on until `i` reaches a value of 10. 10 is not smaller than 10, so the loop is done executing and the numbers 0 to 9 have been logged to the console.

We can also use a `for` loop to create a sequence and add values to an array, like this:

```
let arr = [];
for (let i = 0; i < 100; i++) {
  arr.push(i);
}
```

This is what the array looks like after this loop:

```
[
   0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
  12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
  36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
  72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
  84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
  96, 97, 98, 99
]
```

Since the loop ran the block of code 100 times, starting with an initial value of 0 for `i`, the block of code will add the incrementing value into the array at the end of the array. This results in an array that has a count of 0–99 and a length of 100 items. Since arrays start with an index value of zero, the values in the array will actually match up with the index values of the items in the array.

Or we could create an array containing only even values:

```
let arr = [];
for (let i = 0; i < 100; i = i + 2) {
  arr.push(i);
}
```

Resulting in this array:

```
[
   0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20,
  22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
  44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64,
  66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
  88, 90, 92, 94, 96, 98
]
```

Most commonly, you will see `i++` as the third part of the `for` loop, but please note that you can write any statement there. In this case, we are using `i = i + 2` to add 2 to the previous value every time, creating an array with only even numbers.

Practice exercise 5.3

In this exercise we will use a `for` loop to create an array that holds objects. Starting with creating a blank array, the block of code within the loop will create an object that gets inserted into the array.

1. Setup a blank array, `myWork`.
2. Using a `for` loop, create a list of 10 objects, each of which is a numbered lesson (e.g. Lesson 1, Lesson 2, Lesson 3….) with an alternating `true/false` status for every other item to indicate whether the class will be running this year. For example:

   ```
   name: 'Lesson 1', status: true
   ```

3. You can specify the status by using a ternary operator that checks whether the modulo of the given lesson value is equal to zero and by setting up a Boolean value to alternate the values each iteration.
4. Create a lesson using a temporary object variable, containing the name (`lesson` with the numeric value) and predefined status (which we set up in the previous step).
5. Push the objects to the `myWork` array.
6. Output the array to the console.

Nested loops

Sometimes it can be necessary to use a loop inside a loop. A loop inside a loop is called a nested loop. Often it is not the best solution to the problem. It could even be a sign of poorly written code (sometimes called "code smell" among programmers), but every now and then it is a perfectly fine solution to a problem.

Here is what it would look like for `while` loops:

```
while (condition 1) {
  // code that gets executed as long as condition 1 is true
  // this loop depends on condition 1 being true
    while (condition 2) {
      // code that gets executed as long as condition 2 is true
    }
}
```

Nesting can also be used with `for` loops, or with a combination of both `for` and `while`, or even with all kinds of loops; they can go several levels deep.

An example in which we might use nested loops would be when we want to create an array of arrays. With the outer loop, we create the top-level array, and with the inner loop we add the values to the array.

```
let arrOfArrays = [];
for (let i = 0; i < 3; i++){
  arrOfArrays.push([]);
  for (let j = 0; j < 7; j++) {
    arrOfArrays[i].push(j);
  }
}
```

When we log this array like this:

```
console.log(arrOfArrays);
```

We can see that the output is an array of arrays with values from `0` up to `6`.

```
[
  [
    0, 1, 2, 3, 4, 5, 6
  ],
  [
    0, 1, 2, 3, 4, 5, 6
  ],
  [
    0, 1, 2, 3, 4, 5, 6
  ]
]
```

We used the nested loops to create an array in an array, meaning we can work with rows and columns after having created this loop. This means nested loops can be used to create tabular data. We can show this output as a table using the `console.table()` method instead, like so:

```
console.table(arrOfArrays);
```

This will output:

```
┌─────────┬───┬───┬───┬───┬───┬───┬───┐
│ (index) │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
├─────────┼───┼───┼───┼───┼───┼───┼───┤
│    0    │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
│    1    │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
│    2    │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
└─────────┴───┴───┴───┴───┴───┴───┴───┘
```

Let's put this into practice in the next exercise.

Practice exercise 5.4

In this exercise we will be generating a table of values. We will be using loops to generate rows and also columns, which will be nested within the rows. Nested arrays can be used to represent rows in a table. This is a common structure in spreadsheets, where each row is a nested array within a table and the contents of these rows are the cells in the table. The columns will align as we are creating an equal number of cells in each row.

1. To create a table generator, first create an empty array, `myTable`, to hold your table data.
2. Set variable values for the number of rows and columns. This will allow us to dynamically control how many rows and columns we want within the table. Separating the values from the main code helps make updates to the dimensions easier.
3. Set up a `counter` variable with an initial value of `0`. The counter will be used to set the content and count the values of the cells within the table.
4. Create a `for` loop with conditions to set the number of iterations, and to construct each row of the table. Within it, set up a new temporary array (`tempTable`) to hold each row of data. The columns will be nested within the rows, generating each cell needed for the column.
5. Nest a second loop within the first to count the columns. Columns are run within the row loop so that we have a uniform number of columns within the table.

6. Increment the main counter each iteration of the inner loop, so that we track a master count of each one of the cells and how many cells are created.
7. Push the counter values to the temporary array, `tempTable`. Since the array is a nested array representing a table, the values of the counter can also be used to illustrate the cell values next to each other in the table. Although these are separate arrays representing new rows, the value of the counter will help illustrate the overall sequence of cells in the final table.
8. Push the temporary array to the main table. As each iteration builds a new row of array items, this will continue to build the main table in the array.
9. Output into the console with `console.table(myTable)`. This will show you a visual representation of the table structure.

Loops and arrays

If you are not convinced of how extremely useful loops are by now, have a look at loops and arrays. Loops make life with arrays a lot more comfortable.

We can combine the `length` property and the condition part of the `for` loop or `while` loop to loop over arrays. It would look like this in the case of a `for` loop:

```
let arr = [some array];
for (initialize variable; variable smaller than arr.length; statement)
{
  // code to be executed
}
```

Let's start with a simple example that is going to log every value of the array:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i ++){
  console.log(names[i]);
}
```

This will output:

```
Chantal
John
Maxime
Bobbi
Jair
```

We use the `length` property to determine the maximum value of our index. The index starts counting at 0, but the length does not. The index is always one smaller than the length. Hence, we loop over the values of the array by increasing the length.

In this case we aren't doing very interesting things yet; we are simply printing the values. But we could be changing the values of the array in a loop, for example, like this:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i ++){
  names[i] = "hello " + names[i];
}
```

We have concatenated `hello` with the beginnings of our names. The array is changed in the loop and the array will have this content after the loop has executed:

```
[
  'hello Chantal',
  'hello John',
  'hello Maxime',
  'hello Bobbi',
  'hello Jair'
]
```

The possibilities are endless here. When an array comes in somewhere in the application, data can be sent to the database per value. Data can be modified by value, or even filtered, like this:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i ++){
  if(names[i].startsWith("M")){
    delete names[i];
    continue;
  }
  names[i] = "hello " + names[i];
}
console.log(names);
```

The `startsWith()` method just checks whether the string starts with a certain character. In this case it checks whether the name starts with the string `M`.

Don't worry, we will cover this function and many more in detail in *Chapter 8, Built-in JavaScript Methods*.

The output is:

```
[
  'hello Chantal',
  'hello John',
  <1 empty item>,
  'hello Bobbi',
  'hello Jair'
]
```

You'll have to be careful here though. If we were to remove the item instead of deleting it and leaving an empty value, we would accidentally skip the next value, since that value gets the index of the recently deleted one and `i` is incremented and moves on to the next index.

What do you think this one does:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i++){
  names.push("...")
}
```

Your program gets stuck in an infinite loop here. Since a value gets added every iteration, the length of the loop grows with every iteration and `i` will never be bigger than or equal to `length`.

Practice exercise 5.5

Explore how to create a table grid that contains nested arrays as rows within a table. The rows will each contain the number of cells needed for the number of columns set in the variables. This grid table will dynamically adjust depending on the values for the variables.

1. Create a grid array variable.
2. Set a value of `64` for the number of cells.
3. Set a counter to `0`.

4. Create a global variable to be used for the `row` array.
5. Create a loop that will iterate up to the number of cells you want in the array, plus one to include the zero value. In our example, we would use 64+1.
6. Add an outer `if` statement, which uses modulo to check if the main counter is divisible by 8 or whatever number of columns you want.
7. Inside the preceding `if` statement, add another `if` statement to check if the row is undefined, indicating whether it is the first run or whether the row is complete. If the row has been defined, then add the row to the main grid array.
8. To finish off the outer `if` statement, if the counter is divisible by 8, clear the `row` array—it has already been added to the grid by the inner `if` statement.
9. At the end of the for loop, increment of the main counter by 1.
10. Set up a temporary variable to hold the value of the counter and push it to the `row` array.
11. Within the loop iteration, check if the value of the counter is equal to the total number of columns you want; if it is, then add the current row to the grid.
12. Please note that the extra cell will not be added to the grid since there aren't enough cells to make a new row within the condition that adds the rows to the grid. An alternative solution would be to remove the +1 from the loop condition and add `grid.push(row)` after the loop is completed, both of which will provide the same solution output.
13. Output the grid into the console.

for of loop

There is another loop we can use to iterate over the elements of an array: the **for of loop**. It cannot be used to change the value associated with the index as we can do with the regular loop, but for processing values it is a very nice and readable loop.

Here is what the syntax looks like:

```
let arr = [some array];
for (let variableName of arr) {
  // code to be executed
  // value of variableName gets updated every iteration
  // all values of the array will be variableName once
}
```

So you can read it like this: "For every value of the array, call it `variableName` and do the following." We can log our `names` array using this loop:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let name of names){
  console.log(name);
}
```

We need to specify a temporary variable; in this case we called it `name`. This is used to put the value of the current iteration in, and after the iteration, it gets replaced with the `next` value. This code results in the following output:

```
Chantal
John
Maxime
Bobbi
Jair
```

There are some limitations here; we cannot modify the array, but we could write all the elements to a database or a file, or send it somewhere else. The advantage of this is that we cannot accidentally get stuck in an infinite loop or skip values.

Practice exercise 5.6

This exercise will construct an array as it loops through the incrementing values of x. Once the array is done, this exercise also will demonstrate several ways to output array contents.

1. Create an empty array
2. Run a loop 10 times, adding a new incrementing value to the array
3. Log the array into the console
4. Use the `for` loop to iterate through the array (adjust the number of iterations to however many values are in your array) and output into the console
5. Use the `for of` loop to output the value into the console from the array

Loops and objects

We have just seen how to loop over the values of an array, but we can also loop over the properties of an object. This can be helpful when we need to go over all the properties but don't know the exact properties of the object we are iterating over.

Looping over an object can be done in a few ways. We can use the `for in` loop to loop over the object directly, or we can convert the object to an array and loop over the array. We'll consider both in the following sections.

for in loop

Manipulating objects with loops can also be done with another variation of the `for` loop, the **for in loop**. The `for in` loop is somewhat similar to the `for of` loop. Again here, we need to specify a temporary name, also referred to as a key, to store each property name in. We can see it in action here:

```
let car = {
  model: "Golf",
  make: "Volkswagen",
  year: 1999,
  color: "black",
};

for (let prop in car){
  console.log(car[prop]);
}
```

We need to use the prop of each loop iteration to get the value out of the `car` object. The output then becomes:

```
Golf
Volkswagen
1999
black
```

If we just logged the prop, like this:

```
for (let prop in car){
  console.log(prop);
}
```

This is what our output would look like:

```
model
make
year
color
```

As you can see, all the names of the properties get printed, and not the values. This is because the `for in` loop is getting the property names (keys) and not the values. The `for of` is doing the opposite; it is getting the values and not the keys.

This `for in` loop can also be used on arrays, but it is not really useful. It will only return the indices, since these are the "keys" of the values of the arrays. Also, it should be noted that the order of execution cannot be guaranteed, even though this is usually important for arrays. It is therefore better to use the approaches mentioned in the section on loops and arrays.

Practice exercise 5.7

In this exercise, we will experiment with looping over objects and internal arrays.

1. Create a simple object with three items in it.
2. Using the `for in` loop, get the properties' names and values from the object and output them into the console.
3. Create an array containing the same three items. Using either the `for` loop or the `for in` loop, output the values from the array into the console.

Looping over objects by converting to an array

You can use any loop on objects, as soon as you convert the object to an array. This can be done in three ways:

- Convert the keys of the object to an array
- Convert the values of the object to an array
- Convert the key-value entries to an array (containing arrays with two elements: object key and object value)

Let's use this example:

```
let car = {
  model: "Golf",
  make: "Volkswagen",
  year: 1999,
  color: "black",
};
```

If we want to loop over the keys of the object, we can use the `for in` loop, as we saw in the previous section, but we can also use the `for of` loop if we convert it to an array first. We do so by using the `Object.keys(nameOfObject)` built-in function. This takes an object and grabs all the properties of this object and converts them to an array.

To demonstrate how this works:

```
let arrKeys = Object.keys(car);
console.log(arrKeys);
```

This will output:

```
[ 'model', 'make', 'year', 'color' ]
```

We can loop over the properties of this array like this using the `for of` loop:

```
for(let key of Object.keys(car)) {
  console.log(key);
}
```

And this is what it will output:

```
model
make
year
color
```

Similarly, we can use the `for of` loop to loop over the values of the object by converting the values to an array. The main difference here is that we use `Object.values(nameOfObject)`:

```
for(let key of Object.values(car)) {
  console.log(key);
}
```

You can loop over these arrays in the same way you loop over any array. You can use the length and index strategy like this in a regular `for` loop:

```
let arrKeys = Object.keys(car);
for(let i = 0; i < arrKeys.length; i++) {
  console.log(arrKeys[i] + ": " + car[arrKeys[i]]);
}
```

And this will output:

```
model: Golf
make: Volkswagen
year: 1999
color: black
```

More interesting is how to loop over both arrays at the same time using the `for of` loop. In order to do so, we will have to use `Object.entries()`. Let's demonstrate what it does:

```
let arrEntries = Object.entries(car);
console.log(arrEntries);
```

This will output:

```
[
  [ 'model', 'Golf' ],
  [ 'make', 'Volkswagen' ],
  [ 'year', 1999 ],
  [ 'color', 'black' ]
]
```

As you can see, it is returning a two-dimensional array, containing key-value pairs. We can loop over it like this:

```
for (const [key, value] of Object.entries(car)) {
  console.log(key, ":", value);
}
```

And this will output:

```
model : Golf
make : Volkswagen
year : 1999
color : black
```

Alright, you have seen many ways to loop over objects now. Most of them come down to converting the object to an array. We can imagine that at this point you could use a break. Or maybe you'd just like to continue?

break and continue

break and **continue** are two keywords that we can use to control the flow of execution of the loop. `break` will stop the loop and move on to the code below the loop. `continue` will stop the current iteration and move back to the top of the loop, checking the condition (or in the case of a `for` loop, performing the statement and then checking the condition).

We will be using this array of `car` objects to demonstrate `break` and `continue`:

```
let cars = [
  {
    model: "Golf",
    make: "Volkswagen",
    year: 1999,
    color: "black",
  },
  {
    model: "Picanto",
    make: "Kia",
    year: 2020,
    color: "red",
  },
  {
    model: "Peugeot",
    make: "208",
    year: 2021,
    color: "black",
  },
  {
    model: "Fiat",
    make: "Punto",
    year: 2020,
    color: "black",
  }
];
```

We will first have a closer look at `break`.

break

We have already seen **break** in the `switch` statement. When `break` was executed, the `switch` statement ended. This is not very different when it comes to loops: when the `break` statement is executed, the loop will end, even when the condition is still true.

Here is a silly example to demonstrate how `break` works:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
  if (i === 4) {
    break;
  }
}
```

It looks like a loop that will log the numbers 0 to 10 (again excluding 10) to the console. There is a catch here though: as soon as `i` equals `4`, we execute the `break` command. `break` ends the loop immediately, so no more loop code gets executed afterward.

We can also use `break` to stop looping through the array of cars when we have found a car that matches our demands.

```
for (let i = 0; i < cars.length; i++) {
  if (cars[i].year >= 2020) {
    if (cars[i].color === "black") {
      console.log("I have found my new car:", cars[i]);
      break;
    }
  }
}
```

As soon as we run into a car with the year 2020 or later and the car is black, we will stop looking for other cars and just buy that one. The last car in the array would also have been an option, but we did not even consider it because we found one already. The code snippet will output this:

```
I have found my new car: { model: 'Peugeot', make: '208', year: 2021,
color: 'black' }
```

However, often it is not a best practice to use `break`. If you can manage to work with the condition of the loop to break out of the loop instead, this is a much better practice. It prevents you getting stuck in an infinite loop, and the code is easier to read.

If the condition of the loop is not an actual condition, but pretty much a run-forever kind of statement, the code gets hard to read.

Consider the following code snippet:

```
while (true) {
  if (superLongArray[0] != 42 && superLongArray.length > 0) {
    superLongArray.shift();
  } else {
    console.log("Found 42!");
    break;
  }
}
```

This would be better to write without `break` and without something terrible like `while(true)`; you could do it like this:

```
while (superLongArray.length > 0 && notFound) {
  if (superLongArray[0] != 42) {
    superLongArray.shift();
  } else {
    console.log("Found 42!");
    notFound = false;
  }
}
```

With the second example, we can see the conditions of the loop easily, namely the length of the array and a `notFound` flag. However, with `while(true)` we are kind of misusing the while concept. You want to specify the condition, and it should evaluate to `true` or `false`; this way your code is nice to read. If you say `while(true)`, you're actually saying forever, and the reader of your code will have to interpret it line by line to see what is going on and when the loop is ended by a workaround `break` statement.

continue

`break` can be used to quit the loop, and **continue** can be used to move on to the next iteration of the loop. It quits the current iteration and moves back up to check the condition and start a new iteration.

Here you can see an example of `continue`:

```
for (let car of cars){
  if(car.color !== "black"){
    continue;
  }
  if (car.year >= 2020) {
    console.log("we could get this one:", car);
  }
}
```

The approach here is to just skip every car that is not black and consider all the others that are not older than make year 2020 or later. The code will output this:

```
we could get this one: { model: 'Peugeot', make: '208', year: 2021,
color: 'black' }
we could get this one: { model: 'Fiat', make: 'Punto', year: 2020,
color: 'black' }
```

Be careful with `continue` in a `while` loop. Without running it, what do you think the next code snippet does?

```
// let's only log the odd numbers to the console
let i = 1;
while (i < 50) {
  if (i % 2 === 0){
    continue;
  }
  console.log(i);
  i++;
}
```

It logs `1`, and then it gets you stuck in an infinite loop, because `continue` gets hit before the value of `i` changes, so it will run into `continue` again, and again, and so on. This can be fixed by moving the `i++` up and subtracting 1 from `i`, like this:

```
let i = 1;
while (i < 50) {
  i++;
  if ((i-1) % 2 === 0){
    continue;
  }
  console.log(i-1);
}
```

But again, there is a better way without `continue` here. The chance of error is a lot smaller:

```
for (let i = 1; i < 50; i = i + 2) {
  console.log(i);
}
```

And as you can see it is even shorter and more readable. The value of `break` and `continue` usually comes in when you are looping over large data sets, possibly coming from outside your application. Here you'll have less influence to apply other types of control. Using `break` and `continue` is not a best practice for simple basic examples, but it's a great way to get familiar with the concepts.

Practice exercise 5.8

This exercise will demonstrate how to create a string with all the digits as it loops through them. We can also set a value to skip by adding a condition that will use `continue`, skipping the matching condition. A second option is to do the same exercise and use the `break` keyword.

1. Set up a string variable to use as output.
2. Select a number to skip, and set that number as a variable.
3. Create a `for` loop that counts to 10.
4. Add a condition to check if the value of the looped variable is equal to the number that should be skipped.
5. If the number is to be skipped in the condition, `continue` to the next number.
6. As you iterate through the values, append the new count value to the end of the main output variable.
7. Output the main variable after the loop completes.
8. Reuse the code, but change the `continue` to `break` and see the difference. It should now stop at the skip value.

break, continue, and nested loops

`break` and `continue` can be used in nested loops as well, but it is important to know that when `break` or `continue` is used in a nested loop, the outer loop will not break.

We will use this array of arrays to discuss `break` and `continue` in nested loops:

```
let groups = [
  ["Martin", "Daniel", "Keith"],
  ["Margot", "Marina", "Ali"],
  ["Helen", "Jonah", "Sambikos"],
];
```

Let's break down this example. We are looking for all the groups that have two names starting with an `M`. If we find such a group, we will log it.

```
for (let i = 0; i < groups.length; i++) {
  let matches = 0;

for (let j = 0; j < groups[i].length; j++) {
  if(groups[i][j].startsWith("M")){
      matches++;
    } else {
      continue;
    }
  if (matches === 2){
      console.log("Found a group with two names starting with an M:");
      console.log(groups[i]);
      break;
    }
  }
}
```

We first loop over the top-level arrays and set a counter, `matches`, with a start value of `0`, and for each of these top-level arrays, we are going to loop over the values. When a value starts with an M, we increase `matches` by one and check whether we have found two matches already. If we find two Ms, we break out of the inner loop and continue in our outer loop. This one will move on to the next top-level array, since nothing is happening after the inner loop.

If the name does not start with an M, we do not need to check for `matches` being `2`, and we can continue to the next value in the inner array.

Take a look at this example: what do you think it will log?

```
for (let group of groups){
  for (let member of group){
    if (member.startsWith("M")){
      console.log("found one starting with M:", member);
```

```
        break;
      }
    }
  }
```

It will loop over the arrays, and for every array it will check the value to see if it starts with an M. If it does, the inner loop will break. So, if one of the arrays in the array contains multiple values starting with M, only the first one will be found, since the iteration over that array breaks and we continue to the next array.

This one will output:

```
found one starting with M: Martin
found one starting with M: Margot
```

We can see that it finds Margot, the first one from the second array, but it skips Marina, because it is the second one in the array. And it breaks after having found one group, so it won't loop over the other elements in the inner array. It will continue with the next array, which doesn't contain names starting with an M.

If we wanted to find groups that have a member with a name that starts with an M, the previous code snippet would have been the way to go, because we are breaking the inner loop as soon as we find a hit. This can be useful whenever you want to make sure that an array in a data set contains at least one of something. Because of the nature of the `for of` loop, it won't give the index or place where it found it. It will simply break, and you have the value of the element of the array to use. If you need to know more, you can work with counters, which are updated every iteration.

If we want to see whether only one of all the names in the array of arrays starts with an M, we would have to break out of the outer loop. This is something we can do with labeled loops.

break and continue and labeled blocks

We can break out of the outer loop from inside the inner loop, but only if we give a label to our loop. This can be done like this:

```
outer:
for (let group of groups) {
  inner:
  for (let member of group) {
    if (member.startsWith("M")) {
      console.log("found one starting with M:", member);
```

```
        break outer;
      }
    }
  }
```

We are giving our block a label by putting a word and a colon in front of a code block. These words can be pretty much anything (in our case, "outer" and "inner"), but not JavaScript's own reserved words, such as `for`, `if`, `break`, `else`, and others.

This will only log the first name starting with an `M`:

```
found one starting with M: Martin
```

It will only log one, because it is breaking out of the outer loop and all the loops end as soon as they find one. In a similar fashion you can continue the outer loop as well.

Whenever you want to be done as soon as you find one hit, this is the option to use. So, for example, if you want check for errors and quit if there aren't any, this would be the way to go.

Chapter project

Math multiplication table

In this project, you will create a math multiplication table using loops. You can do this using your own creativity or by following some of the following suggested steps:

1. Set up a blank array to contain the final multiplication table.
2. Set a `value` variable to specify how many values you want to multiply with one another and show the results for.
3. Create an outer `for` loop to iterate through each row and a `temp` array to store the row values. Each row will be an array of cells that will be nested into the final table.
4. Add an inner `for` loop for the column values, which will push the multiplied row and column values to the `temp` array.
5. Add the temporary row data that contains the calculated solutions to the main array of the final table. The final result will add a row of values for the calculations.

Self-check quiz

1. What is the expected output for the following code?

```
let step = 3;

for (let i = 0; i < 1000; i += step) {
    if (i > 10) {
        break;
    }
    console.log(i);
}
```

2. What is the final value for `myArray`, and what is expected in the console?

```
const myArray = [1,5,7];
for(el in myArray){
    console.log(Number(el));
    el = Number(el) + 5;
    console.log(el);
}
console.log(myArray);
```

Summary

In this chapter we introduced the concept of loops. Loops enable us to repeat a certain block of code. We need some sort of condition when we loop, and as long as that condition is true, we'll keep looping. As soon as it changes to false, we end our loop.

We have seen the `while` loop, in which we just insert a condition, and as long as that condition is true we keep looping. If the condition is never true, we won't even execute the loop code once.

This is different for the `do while` loop. We always execute the code once, and then we start to check a condition. If this condition is true, we execute the code again and do so until the condition becomes false. This can be useful when working with input from outside, such as user input. We would need to request it once, and then we can keep on requesting it again until it is valid.

Then we saw the `for` loop, which has a slightly different syntax. We have to specify a variable, check a condition (preferably using that variable, but this is not mandatory), and then specify an action to be executed after every iteration. Again, it's preferable for the action to include the variable from the first part of the `for` loop. This gives us code that is to be executed as long as a condition is true.

We also saw two ways to loop over arrays and objects, `for in` and `for of`. The `for in` loop loops over keys and `for of` loops over values. They go over every element in a collection. The advantage of these loops is that JavaScript controls the execution: you can't miss an element or get stuck in an infinite loop.

Lastly, we saw `break` and `continue`. We can use the `break` keyword to end a loop immediately and the `continue` keyword to end the current iteration and go back to the top and start the next iteration, if the condition is still true, that is.

In the next chapter we are going to be adding a really powerful tool to our JavaScript toolbox: functions! They allow us to take our coding skills to the next level and structure our code better.