Here is another example. If the age is below 18, log to the console that access is denied, otherwise log to the console that the person is allowed to come in:

```
if(age < 18) {
  console.log("We're very sorry, but you can't get in under 18");
} else {
  console.log("Welcome!");
}
```

There is a common coding mistake related to `if` statements. I have made it in the following code snippet. Can you see what this code does?

```
let hobby = "dancing";

if(hobby == "coding"){
  console.log("** I love coding too! **");
} else {
  console.log("** Can you teach me that? **");
}
```

It will log the following:

```
** I love coding too! **
```

That might surprise you. The problem here is the single equal sign in the `if` statement. Instead of evaluating the condition, it is assigning `coding` to `hobby`. And then it is converting `coding` to a Boolean, and since it is not an empty string, it will become true, so the if block will be executed. So, always remember to use the double equal sign in this case.

Let's test our knowledge with a practice exercise.

Practice exercise 4.1

1. Create a variable with a Boolean value.
2. Output the value of the variable to the console.
3. Check whether the variable is true and if so, output a message to the console, using the following syntax:

```
if(myVariable){
//action
}
```

```
} else {
    cost = 7;
    message ="A ticket is 7 dollars.";
}

console.log(message);
console.log("Your Total cost "+cost);
```

Chances are that you will think the code is easier to read than the written template. In that case, nicely done! You are really starting to think like a JavaScript developer already.

The code gets executed top to bottom, and only one of the blocks will be executed. As soon as a true expression is encountered, the other ones will be ignored. This is why we can also write our sample like this:

```
if(age < 3){
  console.log("Access is free under three.");
} else if(age < 12) {
  console.log("With the child discount, the fee is 5 dollars");
} else if(age < 65) {
  console.log("A regular ticket costs 10 dollars.");
} else if(age >= 65) {
  console.log("A ticket is 7 dollars.");
}
```

Practice exercise 4.2

1. Create a prompt to ask the user's age
2. Convert the response from the prompt to a number
3. Declare a message variable that you will use to hold the console message for the user
4. If the input age is equal to or greater than 21, set the `message` variable to confirm entry to a venue and the ability to purchase alcohol
5. If the input age is equal to or greater than 19, set the `message` variable to confirm entry to the venue but deny the purchase of alcohol
6. Provide a default else statement to set the `message` variable to deny entry if none are true
7. Output the response `message` variable to the console

Conditional ternary operators

We did not actually discuss this very important operator in our section on operators in *Chapter 2, JavaScript Essentials*. This is because it helps to understand the if else statement first. Remember that we had a unary operator that was called a unary operator because it only had one operand? This is why our ternary operator has its name; it has three operands. Here is its template:

```
operand1 ? operand2 : operand3;
```

`operand1` is the expression that is to be evaluated. If the value of the expression is `true`, `operand2` gets executed. If the value of the expression is `false`, `operand3` gets executed. You can read the question mark as "then" and the colon as "else" here:

```
expression ? statement for true : statement associated with false;
```

The template for saying it in your head should be:

*if *operand1*, then *operand2*, else *operand3**

Let's have a look at a few examples:

```
let access = age < 18 ? "denied" : "allowed";
```

This little code snippet will assign a value to access. *If* age is lower than 18, *then* it will assign the value `denied`, *else* it will assign the value `allowed`. You can also specify an action in a ternary statement, like this:

```
age < 18 ? console.log("denied") : console.log("allowed");
```

This syntax can be confusing at first. The template of what to say in your head while reading it can really come to the rescue here. You can only use these ternary operators for very short actions, so it's best practice to use the ternary operator in these instances as it makes code easier to read. However, if the logic contains multiple comparison arguments, you'll have to use the regular if-else.

Practice exercise 4.3

1. Create a Boolean value for an ID variable
2. Using a ternary operator, create a message variable that will check whether their ID is valid and either allow a person into a venue or not
3. Output the response to the console

```
    break;
  case "Breakfast":
    console.log("It is 7:00AM");
    break;
  case "Drive to work":
    console.log("It is 8:00AM");
    break;
  case "Lunch":
    console.log("It is 12:00PM");
    break;
  case "Drive home":
    console.log("It is 5:00PM");
    break;
  case "Dinner":
    console.log("It is 6:30PM");
    break;
  default:
    console.log("I cannot determine the current time.");
    break;
}
```

If the value of the activity was to be something that is not specified as a case, for example, "Watch Netflix," it would log the following to the console:

```
I cannot determine the current time.
```

Practice exercise 4.4

As discussed in *Chapter 1, Getting Started with JavaScript*, the JavaScript function `Math.random()` will return a random number in the range of 0 to less than 1, including 0 but not 1. You can then scale it to the desired range by multiplying the result and using `Math.floor()` to round it down to the nearest whole number; for example, to generate a random number between 0 and 9:

```
// random number between 0 and 1
let randomNumber = Math.random();
// multiply by 10 to obtain a number between 0 and 10
randomNumber = randomNumber * 10;
// removes digits past decimal place to provide a whole number
RandomNumber = Math.floor(randomNumber);
```

For the values `F` and `D`, the same thing is happening. This is also true for `C` and `B`. When the value of `grade` is either `C` or `B`, it will log the following to the console:

```
You've passed!
```

This is more readable than the alternative if-else statement:

```
if(grade === "F" || grade === "D") {
  console.log("You've failed!");
} else if(grade === "C" || grade === "B") {
  console.log("You've passed!");
} else if(grade === "A") {
  console.log("Nice!");
} else {
  console.log("I don't know this grade.");
}
```

Practice exercise 4.5

1. Create a variable called `prize` and use a prompt to ask the user to set the value by selecting a number between 0 and 10
2. Convert the prompt response to a number data type
3. Create a variable to use for the output message containing the value "`My Selection: `"
4. Using the switch statement (and creativity), provide a response back regarding a prize that is awarded depending on what number is selected
5. Use the switch break to add combined results for prizes
6. Output the message back to the user by concatenating your prize variable strings and the output message string

Chapter projects

Evaluating a number game

Ask the user to enter a number and check whether it's greater than, equal to, or less than a dynamic number value in your code. Output the result to the user.

```
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.

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
}
```

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.

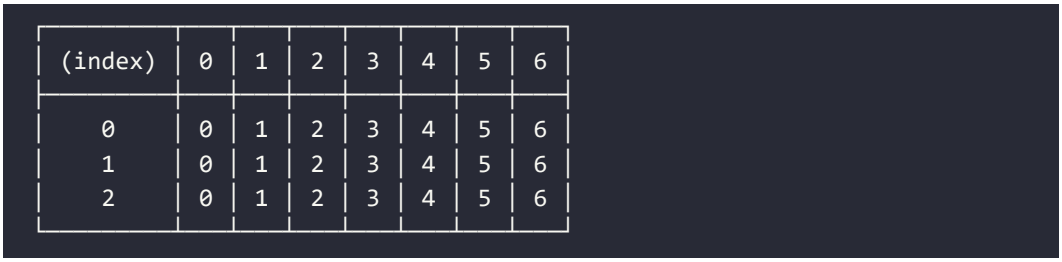
Nested loops

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
    }
}
```

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.

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`.

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.

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",
};
```

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.

```
        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.