2

JavaScript Essentials

In this chapter, we will be dealing with some essential building blocks of JavaScript: variables and operators. We will start with variables, what they are, and which different variable data types exist. We need these basic building blocks to store and work with variable values in our scripts, making them dynamic.

Once we've got the variables covered, we will be ready to deal with operators. Arithmetic, assignment, and conditional and logical operators will be discussed at this stage. We need operators to modify our variables or to tell us something about these variables. This way we can do basic calculations based on factors such as user input.

Along the way, we'll cover the following topics:

- Variables
- Primitive data types
- Analyzing and modifying data types
- Operators

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

Variables

Variables are the first building block you will be introduced to when learning most languages. Variables are values in your code that can represent different values each time the code runs. Here is an example of two variables in a script:

```
firstname = "Maaike";
x = 2;
```

And they can be assigned a new value while the code is running:

```
firstname = "Edward";
x = 7;
```

Without variables, a piece of code would do the exact same thing every single time it was run. Even though that could still be helpful in some cases, it can be made much more powerful by working with variables to allow our code to do something different every time we run it.

Declaring variables

The first time you create a variable, you declare it. And you need a special word for that: `let`, `var`, or `const`. We'll discuss the use of these three arguments shortly. The second time you call a variable, you only use the name of the existing variable to assign it a new value:

```
let firstname = "Maria";
firstname = "Jacky";
```

In our examples, we will be assigning a value to our variables in the code. This is called "hardcoded" since the value of your variable is defined in your script instead of coming dynamically from some external input. This is something you won't be doing that often in actual code, as more commonly the value comes from an external source, such as an input box on a website that a user filled out, a database, or some other code that calls your code. The use of variables coming from external sources instead of being hardcoded into a script is actually the reason that scripts are adaptable to new information, without having to rewrite the code.

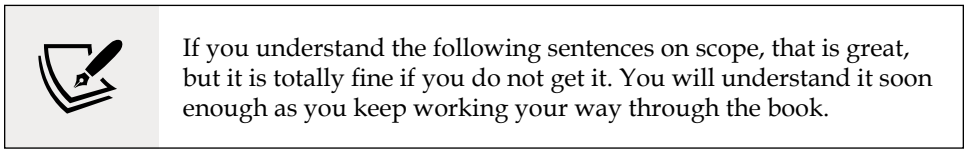
We have just established how powerful the variable building block is in code. Right now, we are going to hardcode variables into our scripts, and they therefore will not vary until a coder changes the program. However, we will soon learn how to make our variables take in values from outside sources.

let, var, and const

A variable definition consists of three parts: a variable-defining keyword (`let`, `var`, or `const`), a name, and a value. Let's start with the difference between `let`, `var`, or `const`. Here you can see some examples of variables using the different keywords:

```
let nr1 = 12;
var nr2 = 8;
const PI = 3.14159;
```

`let` and `var` are both used for variables that might have a new value assigned to them somewhere in the program. The difference between `let` and `var` is complex. It is related to scope.



If you understand the following sentences on scope, that is great, but it is totally fine if you do not get it. You will understand it soon enough as you keep working your way through the book.

`var` has **global scope** and `let` has **block scope**. `var`'s global scope means that you can use the variables defined with `var` in the entire script. On the other hand, `let`'s block scope means you can only use variables defined with `let` in the specific block of code in which they were defined. Remember, a block of code will always start with `{` and end with `}`, which is how you can recognize them.

On the other hand, `const` is used for variables that only get a value assigned once—for example, the value of pi, which will not change. If you try reassigning a value declared with `const`, you will get an error:

```
const someConstant = 3;
someConstant = 4;
```

This will result in the following output:

```
Uncaught TypeError: Assignment to constant variable.
```

We will be using `let` in most of our examples—for now, trust us that you should use `let` in most cases.

Naming variables

When it comes to naming variables, there are some conventions in place:

- Variables start with a lowercase letter, and they should be descriptive. If something holds an age, do not call it `x`, but `age`. This way, when you read your script later, you can easily understand what you did by just reading your code.
- Variables cannot contain spaces, but they can use underscores. If you use a space, JavaScript doesn't recognize it as a single variable.

We will be using camel case here. This means that when we want to use multiple words to describe a variable, we will start with a lowercase word, then use a capital for every new word after the first word—for example: `ageOfBuyer`.

Whatever the convention is in the place you are working, the key is consistency. If all naming is done in a similar format, the code will look cleaner and more readable, which makes it a lot easier to make a small change later.

The value of your variable can be anything. Let's start with the easiest thing variables can be: primitives.

Primitive data types

Now you know what variables are and why we need them in our code, it is time to look at the different types of values we can store in variables. Variables get a value assigned. And these values can be of different types. JavaScript is a loosely typed language. This means that JavaScript determines the type based on the value. The type does not need to be named explicitly. For example, if you declared a value of 5, JavaScript will automatically define it as a number type.

A distinction exists between primitive data types and other, more complex data types. In this chapter, we will cover the primitive type, which is a relatively simple data structure. Let's say for now that they just contain a value and have a type. JavaScript has seven primitives: String, Number, BigInt, Boolean, Symbol, undefined, and null. We'll discuss each of them in more detail below.

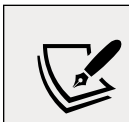
String

A string is used to store a text value. It is a sequence of characters. There are different ways to declare a string:

- Double quotes
- Single quotes
- Backticks: special template strings in which you can use variables directly

The single and double quotes can both be used like so:

```
let singleString = 'Hi there!';
let doubleString = "How are you?";
```



You can use the option you prefer, unless you are working on code in which one of these options has already been chosen. Again, consistency is key.

The main difference between single quotes and double quotes is that you can use single quotes as literal characters in double-quoted strings, and vice versa. If you declare a string with single quotes, the string will end as soon as a second quote is detected, even if it's in the middle of a word. So for example, the following will result in an error, because the string will be ended at the second single quote within `let's`:

```
let funActivity = 'Let's learn JavaScript';
```

`Let` will be recognized as a string, but after that, the bunch of characters that follow cannot be interpreted by JavaScript. However, if you declare the string using double quotes, it will not end the string as soon as it hits the single quote, because it is looking for another double quote. Therefore, this alternative will work fine:

```
let funActivity = "Let's learn JavaScript";
```

In the same way with double quotes, the following would not work:

```
let question = "Do you want to learn JavaScript? "Yes!"";
```

Again, the compiler will not distinguish between double quotes used in different contexts, and will output an error.

In a string using backticks, you can point to variables and the variable's value will be substituted into the line. You can see this in the following code snippet:

```
let language = "JavaScript";
let message = `Let's learn ${language}`;
console.log(message);
```

As you can see, you will have to specify these variables with a rather funky syntax—don't be intimidated! Variables in these template strings are specified between `${nameOfVariable}`. The reason that it's such an intense syntax is that they want to avoid it being something you would normally use, which would make it unnecessarily difficult to do so. In our case, the console output would be as follows:

```
Let's learn JavaScript
```

As you can see, the `language` variable gets replaced with its value: `JavaScript`.

Escape characters

Say we want to have double quotes, single quotes, and backticks in our string. We would have a problem, as this cannot be done with just the ingredients we have now. There is an elegant solution to this problem. There is a special character that can be used to tell JavaScript, "do not take the next character as you normally would." This is the escape character, a backslash.

In this example, the backslash can be used to ensure your interpreter doesn't see the single or double quote marks and end either string too early:

```
let str = "Hello, what's your name? Is it \"Mike\"?";
console.log(str);
let str2 = 'Hello, what\'s your name? Is it "Mike"?';
console.log(str2);
```

This logs the following to the console:

```
Hello, what's your name? Is it "Mike"?
Hello, what's your name? Is it "Mike"?
```

As you can see, both types of quote marks inside the strings have been logged without throwing an error. This is because the backslash before the quote character gives the quote character a different meaning. In this case, the meaning is that it should be a literal character instead of an indicator to end the string.

The escape character has even more purposes. You can use it to create a line break with `\n`, or to include a backslash character in the text with `\\`:

```
let str3 = "New \nline.";
let str4 = "I'm containing a backslash: \\!";
console.log(str3);
console.log(str4);
```

The output of these lines is as follows:

```
New
line.
I'm containing a backslash: \!
```

There are some more options, but we will leave them for now. Let's get back to primitive data types by looking at the number type.

Number

The number data type is used to represent, well, numbers. In many languages, there is a very clear difference between different types of numbers. The developers of JavaScript decided to go for one data type for all these numbers: number. To be more precise, they decided to go for a 64-bit floating-point number. This means that it can store rather large numbers and both signed and unsigned numbers, numbers with decimals, and more.

However, there are different kinds of numbers it can represent. First of all, integers, for example: 4 or 89. But the number data type can also be used to represent decimals, exponentials, octal, hexadecimal, and binary numbers. The following code sample should speak for itself:

```
let intNr = 1;
let decNr = 1.5;
let expNr = 1.4e15;
let octNr = 0o10; //decimal version would be 8
let hexNr = 0x3E8; //decimal version would be 1000
let binNr = 0b101; //decimal version would be 5
```

You don't need to worry about these last three if you're not familiar with them. These are just different ways to represent numbers that you may encounter in the broader field of computer science. The takeaway here is that all the above numbers are of the number data type. So integers are numbers, like these ones:

```
let intNr2 = 3434;
let intNr3 = -111;
```

And the floating points are numbers as well, like this one:

```
let decNr2 = 45.78;
```

And binary numbers are of the number data type as well, for example, this one:

```
let binNr2 = 0b100; //decimal version would be 4
```

We have just seen the number data type, which is very commonly used. But in some special cases, you will need an even bigger number.

BigInt

The limits of the number data type are between 2^{53}-1 and -(2^{53}-1). In case you were to need a bigger (or smaller) number, BigInt comes into play. A BigInt data type can be recognized by the postfix `n`:

```
let bigNr = 90071992547409920n;
```

Let's see what happens when we start to do some calculations between our previously made integer Number, `intNr`, and BigInt, `bigNr`:

```
let result = bigNr + intNr;
```

The output will be as follows:

```
Uncaught TypeError: Cannot mix BigInt and other types, use explicit
conversions
```

Uh-oh, a `TypeError`! It is very clear about what is going wrong. We cannot mix BigInt with the Number data type to perform operations. This is something to keep in mind for later when actually working with BigInt—you can only operate on BigInt with other BigInts.

Boolean

The Boolean data type can hold two values: `true` and `false`. There is nothing in between. This Boolean is used a lot in code, especially expressions that evaluate to a Boolean:

```
let bool1 = false;
let bool2 = true;
```

In the preceding example, you can see the options we have for the Boolean data type. It is used for situations in which you want to store a `true` or a `false` value (which can indicate on/off or yes/no). For example, whether an element is deleted:

```
let objectIsDeleted = false;
```

Or, whether the light is on or off:

```
let lightIsOn = true;
```

These variables suggest respectively that the specified object is not deleted, and that the specific light is on.

Symbol

Symbol is a brand new data type introduced in ES6 (we mentioned ECMA Script 6, or ES6, in *Chapter 1*, *Getting Started with JavaScript*). Symbol can be used when it is important that variables are not equal, even though their value and type are the same (in this case, they would both be of the symbol type). Compare the following string declarations to the symbol declarations, all of equal value:

```
let str1 = "JavaScript is fun!";
let str2 = "JavaScript is fun!";
console.log("These two strings are the same:", str1 === str2);

let sym1 = Symbol("JavaScript is fun!");
let sym2 = Symbol("JavaScript is fun!");
console.log("These two Symbols are the same:", sym1 === sym2);
```

And the output:

```
These two strings are the same: true
These two Symbols are the same: false
```

In the first half, JavaScript concludes that the strings are the same. They have the same value, and the same type. However, in the second part, each symbol is unique. Therefore, although they contain the same string, they are not the same, and output `false` when compared. These symbol data types can be very handy as properties of objects, which we will see in *Chapter 3*, *JavaScript Multiple Values*.

Undefined

JavaScript is a very special language. It has a special data type for a variable that has not been assigned a value. And this data type is undefined:

```
let unassigned;
console.log(unassigned);
```

The output here will be:

```
Undefined
```

We can also purposefully assign an `undefined` value. It is important you know that it is possible, but it is even more important that you know that manually assigning undefined is a bad practice:

```
let terribleThingToDo = undefined;
```

Alright, this can be done, but it is recommended to not do this. This is for a number of reasons—for example, checking whether two variables are the same. If one variable is undefined, and your own variable is manually set to undefined, they will be considered equal. This is an issue because if you are checking for equality, you would want to know whether two values are actually equal, not just that they are both undefined. This way, someone's pet and their last name might be considered equal, whereas they are actually both just empty values.

null

In the last example, we saw an issue that can be solved with a final primitive type, null. null is a special value for saying that a variable is empty or has an unknown value. This is case sensitive. You should use lowercase for null:

```
let empty = null;
```

To solve the issue we encountered with setting a variable as undefined, note that if you set it to null, you will not have the same problem. This is one of the reasons it is better to assign null to a variable when you want to say it is empty and unknown at first:

```
let terribleThingToDo = undefined;
let lastName;
console.log("Same undefined:", lastName === terribleThingToDo);

let betterOption = null;
console.log("Same null:", lastName === betterOption);
```

This outputs the following:

```
Same undefined: true
Same null: false
```

This shows that an automatically undefined variable, `lastName`, and a deliberately undefined variable, `terribleThingToDo`, are considered equal, which is problematic. On the other hand, `lastName` and `betterOption`, which was explicitly declared with a value of null, are not equal.

Analyzing and modifying data types

We have seen the primitive data types. There are some built-in JavaScript methods that will help us deal with common problems related to primitives. Built-in methods are pieces of logic that can be used without having to write JavaScript logic yourself.

We've seen one built-in method already: `console.log()`.

There are many of these built-in methods, and the ones you will be meeting in this chapter are just the first few you will encounter.

Working out the type of a variable

Especially with null and undefined, it can be hard to determine what kind of data type you are dealing with. Let's have a look at `typeof`. This returns the type of the variable. You can check the type of a variable by entering `typeof`, then either a space followed by the variable in question, or the variable in question in brackets:

```
testVariable = 1;
variableTypeTest1 = typeof testVariable;
variableTypeTest2 = typeof(testVariable);
console.log(variableTypeTest1);
console.log(variableTypeTest2);
```

As you might assume, both methods will output `number`. Brackets aren't required because technically, `typeof` is an operator, not a method, unlike `console.log`. But, sometimes you may find that using brackets makes your code easier to read. Here you can see it in action:

```
let str = "Hello";
let nr = 7;
let bigNr = 12345678901234n;
let bool = true;
let sym = Symbol("unique");
let undef = undefined;
let unknown = null;

console.log("str", typeof str);
console.log("nr", typeof nr);
console.log("bigNr", typeof bigNr);
console.log("bool", typeof bool);
console.log("sym", typeof sym);
console.log("undef", typeof undef);
console.log("unknown", typeof unknown);
```

Here, in the same `console.log()` print command, we are printing the name of each variable (as a string, declared with double quotes), then its type (using `typeof`). This will produce the following output:

```
str string
nr number
bigNr bigint
bool boolean
```

```
sym symbol
undef undefined
unknown object
```

There is an odd one out, and that is the null type. In the output you can see that `typeof null` returns `object`, while in fact, null truly is a primitive and not an object. This is a bug that has been there since forever and now cannot be removed due to backward compatibility problems. Don't worry about this bug, as it won't affect our programs—just be aware of it, since it will go nowhere anytime soon, and it has the potential to break applications.

Converting data types

The variables in JavaScript can change types. Sometimes JavaScript does this automatically. What do you think the result of running the following code snippet will be?

```
let nr1 = 2;
let nr2 = "2";
console.log(nr1 * nr2);
```

We try to multiply a variable of type Number with a variable of type String. JavaScript does not just throw an error (as many languages would), but first tries to convert the string value to a number. If that can be done, it can execute without any problem as if two numbers were declared. In this case, `console.log()` will write `4` to the console.

But this is dangerous! Guess what this code snippet does:

```
let nr1 = 2;
let nr2 = "2";
console.log(nr1 + nr2);
```

This one will log `22`. The plus sign can be used to concatenate strings. Therefore, instead of converting a string to a number, it is converting a number to a string in this example, and clubbing the two strings together—`"2"` and `"2"` make `"22"`. Luckily, we do not need to rely on JavaScript's behavior when converting data types. There are built-in functions we can use to convert the data type of our variable.

There are three conversion methods: `String()`, `Number()`, and `Boolean()`. The first one converts a variable to type String. It pretty much takes any value, including undefined and null, and puts quotes around it.

The second one tries to convert a variable to a number. If that cannot be done logically, it will change the value into NaN (not a number). `Boolean()` converts a variable to a Boolean. This will be true for everything except for null, undefined, 0 (number), an empty string, and NaN. Let's see them in action:

```
let nrToStr = 6;
nrToStr = String(nrToStr);
console.log(nrToStr, typeof nrToStr);

let strToNr = "12";
strToNr = Number(strToNr);
console.log(strToNr, typeof strToNr);

let strToBool = "any string will return true";
strToBool = Boolean(strToBool);
console.log(strToBool, typeof strToBool);
```

This will log the following:

```
6 string
12 number
true boolean
```

This might seem pretty straightforward, but not all of the options are equally obvious. These, for example, are not what you might think:

```
let nullToNr = null;
nullToNr = Number(nullToNr);
console.log("null", nullToNr, typeof nullToNr);

let strToNr = "";
strToNr = Number(strToNr);
console.log("empty string", strToNr, typeof strToNr);
```

The preceding code snippet will log the following to the console:

```
null 0 number
empty string 0 number
```

As you can see, an empty string and null will both result in the number 0. This is a choice that the makers of JavaScript made, which you will have to know—it can come in handy at times when you want to convert a string to 0 when it is empty or null.

Next, enter the following snippet:

```
let strToNr2 = "hello";
strToNr2 = Number(strToNr2);
console.log(strToNr2, typeof strToNr2);
```

The result that will be logged to the console is:

```
NaN number
```

Here, we can see that anything that can't be interpreted as a number by simply removing the quotes will evaluate as `NaN` (not a number).

Let's continue with the following code:

```
let strToBool2 = "false";
strToBool2 = Boolean(strToBool2);
console.log(strToBool2, typeof strToBool2);

let strToBool = "";
strToBool = Boolean(strToBool);
console.log(strToBool, typeof strToBool);
```

Finally, this one will log the following:

```
true boolean
false boolean
```

This output shows that any string will return `true` when converted to a Boolean, even the string `"false"`! Only an empty string, null, and undefined will lead to a Boolean value of `false`.

Let's tease your brain a little bit more. What do you think this one will log?

```
let nr1 = 2;
let nr2 = "2";
console.log(nr1 + Number(nr2));
```

This one logs `4`! The string gets converted to a number before it executes the plus operation, and therefore it is a mathematical operation and not a string concatenation. In the next sections of this chapter, we will discuss operators in more depth.

Practice exercise 2.1

What are the types of these variables listed below? Verify this with `typeof` and output the result to the console:

```
let str1 = 'Laurence';
let str2 = "Svekis";
let val1 = undefined;
let val2 = null;
let myNum = 1000;
```

Operators

After seeing quite a few data types and some ways to convert them, it is time for the next major building block: operators. These come in handy whenever we want to work with the variables, modify them, perform calculations on them, and compare them. They are called operators because we use them to operate on our variables.

Arithmetic operators

Arithmetic operators can be used to perform operations with numbers. Most of these operations will feel very natural to you because they are the basic mathematics you will have come across earlier in life already.

Addition

Addition in JavaScript is very simple, we have seen it already. We use + for this operation:

```
let nr1 = 12;
let nr2 = 14;
let result1 = nr1 + nr2;
```

However, this operator can also come in very handy for concatenating strings. Note the added space after `"Hello"` to ensure the end result contains space characters:

```
let str1 = "Hello ";
let str2 = "addition";
let result2 = str1 + str2;
```

The output of printing `result1` and `result2` will be as follows:

```
26
Hello addition
```

As you can see, adding numbers and strings lead to different results. If we add two different strings, it will concatenate them into a single string.

Practice exercise 2.2

Create a variable for your name, another one for your age, and another one for whether you can code JavaScript or not.

Log to the console the following sentence, where `name`, `age` and `true/false` are variables:

```
Hello, my name is Maaike, I am 29 years old and I can code JavaScript:
true.
```

Subtraction

Subtraction works as we would expect it as well. We use - for this operation. What do you think gets stored in the variable in this second example?

```
let nr1 = 20;
let nr2 = 4;
let str1 = "Hi";
let nr3 = 3;
let result1 = nr1 - nr2;
let result2 = str1 - nr3;
console.log(result1, result2);
```

The output is as follows:

```
16 NaN
```

The first result is `16`. And the second result is more interesting. It gives `NaN`, not an error, but just simply the conclusion that a word and a number subtracted is not a number. Thanks for not crashing, JavaScript!

Multiplication

We can multiply two numeric values with the * character. Unlike some other languages, we cannot successfully multiply a number and a string in JavaScript.

The result of multiplying a numeric and a non-numeric value is NaN:

```
let nr1 = 15;
let nr2 = 10;
let str1 = "Hi";
let nr3 = 3;
let result1 = nr1 * nr2;
let result2 = str1 * nr3;
console.log(result1, result2);
```

Output:

```
150 NaN
```

Division

Another straightforward operator is division. We can divide two numbers with the / character:

```
let nr1 = 30;
let nr2 = 5;
let result1 = nr1 / nr2;
console.log(result1);
```

The output is as follows:

```
6
```

Exponentiation

Exponentiation means raising a certain base number to the power of the exponent, for example, x^y. This can be read as x to the power of y. It means that we will multiply x by itself y number of times. Here is an example of how to do this in JavaScript—we use ** for this operator:

```
let nr1 = 2;
let nr2 = 3;
let result1 = nr1 ** nr2;
console.log(result1);
```

We get the following output:

```
8
```

The result of this operation is 2 to the power of 3 (2 * 2 * 2), which is `8`. We're going to avoid going into a mathematics lesson here, but we can also find the root of a number by using fractional exponents: for example, the square root of a value is the same as raising it to the power of 0.5.

Modulus

This is one that often requires a little explanation. Modulus is the operation in which you determine how much is left after dividing a number by another number in its entirety. The amount of times the number can fit in the other number does not matter here. The outcome will be the remainder, or what is left over. The character we use for this operation is the `%` character. Here are some examples:

```
let nr1 = 10;
let nr2 = 3;
let result1 = nr1 % nr2;
console.log(`${nr1} % ${nr2} = ${result1}`);

let nr3 = 8;
let nr4 = 2;
let result2 = nr3 % nr4;
console.log(`${nr3} % ${nr4} = ${result2}`);

let nr5 = 15;
let nr6 = 4;
let result3 = nr5 % nr6;
console.log(`${nr5} % ${nr6} = ${result3}`);
```

And the output:

```
10 % 3 = 1
8 % 2 = 0
15 % 4 = 3
```

The first one is `10 % 3`, where 3 fits 3 times into 10, and then 1 is left. The second one is `8 % 2`. This results in 0, because 2 can fit 4 times into 8 without having anything left. The last one is `15 % 4`, where 4 fits 3 times into 15. And then we have 3 left as a result.

This is something that would happen in your head automatically if I asked you to add 125 minutes to the current time. You will probably do two things: integer division to determine how many whole hours fit into 125 minutes, and then 125 modulo 60 (in JavaScript terms, `125 % 60`) to conclude that you'll have to add 5 more minutes to the current time. Say our current time is 09:59, you will probably start by adding 2 hours, and get to 11:59, and then add 5 minutes, and then you will perform another modulus operation with 59 and 5, adding 1 more hour to the total and having 4 minutes left: 12:04.

Unary operators: increment and decrement

The last two operators of our arithmetic operator section are probably new to you, if you are new to programming (or only familiar with another programming language). These are the increment and decrement operators. A term we use here is **operand**. Operands are subject to the operator. So, if we say `x + y`, *x* and *y* are operands.

We only need one operand for these operators, and therefore we also call them unary operators. If we see `x++`, we can read this as $x = x + 1$. The same is true for the decrement operators: `x--` can be read as $x = x - 1$:

```
let nr1 = 4;
nr1++;
console.log(nr1);

let nr2 = 4;
nr2--;
console.log(nr2);
```

The output is as follows:

```
5
3
```

Prefix and postfix operators

We can have the increment operator after the operand (`x++`), in which case we call this the **postfix unary operator**. We can also have it before (`++x`), which is the **prefix unary operator**. This does something different though—the next few lines might be complicated, so do not worry if you need to read it a few times and have a good look at the examples here.

The postfix gets executed after sending the variable through, and then after that, the operation gets executed. In the following example, `nr` gets incremented by 1 *after* logging. So the first logging statement is still logging the old value because it has not been updated yet. It has been updated for the second log statement:

```
let nr = 2;
console.log(nr++);
console.log(nr);
```

The output is as follows:

```
2
3
```

The prefix gets executed *before* sending the variable through, and often this is the one you will need. Have a look at the following example:

```
let nr = 2;
console.log(++nr);
```

We get the following output:

```
3
```

Alright, if you can figure out what the next code snippets logs to the console, you should really have a handle on it:

```
let nr1 = 4;
let nr2 = 5;
let nr3 = 2;
console.log(nr1++ + ++nr2 * nr3++);
```

It outputs `16`. It will do the multiplication first, according to the basic mathematical order of operations. For multiplying, it uses 6 (prefix, so 5 is incremented before multiplying) and 2 (postfix, so 2 is only incremented after execution, meaning it won't affect our current calculation). This comes down to 12. And then `nr1` is a postfix operator, so this one will execute after the addition. Therefore, it will add 12 to 4 and become 16.

Combining the operators

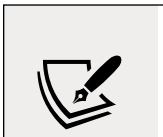
These operators can be combined, and it works just as it does in math. They get executed in a certain order, and not necessarily from left to right. This is due to a phenomenon called operator precedence.

There is one more thing to take into account here, and that is grouping. You can group using `(` and `)`. The operations between the parentheses have the highest precedence. After that, the order of the operations takes place based on the type of operation (highest precedence first) and if they are of equal precedence, they take place from left to right:

Name	Symbol	Example
Grouping	`(...)`	`(x + y)`
Exponentiation	`**`	`x ** y`
Prefix increment and decrement	`--, ++`	`--x, ++y`
Multiplication, division, modulus	`*, /, %`	`x * y, x / y, x % y`
Addition and subtraction	`+, -`	`x + y, x - y`

Practice exercise 2.3

Write some code to calculate the hypotenuse of a triangle using the Pythagorean theorem when given the values of the other two sides. The theorem specifies that the relation between the sides of a right-angled triangle is $a^2 + b^2 = c^2$.



The Pythagorean theorem only applies to right-angled triangles. The sides connected to the 90-degree angle are called the adjacent and opposite sides, represented by a and b in the formula. The longest side, not connected to the 90-degree angle, is called the hypotenuse, represented by c.

You can use `prompt()` to get the value for a and b. Write code to get the value from the user for `a` and `b`. Then square the values of both `a` and `b` before adding them together and finding the square root. Print your answer to the console.

Assignment operators

We have seen one assignment operator already when we were assigning values to variables. The character for this basic assignment operation is `=`. There are a few others available. Every binary arithmetic operator has a corresponding assignment operator to write a shorter piece of code. For example, *x* += 5 means *x* = *x* + 5, and *x* **= 3 means *x* = *x* ** 3 (x to the power of 3).

In this first example we declare a variable x, and set it to 2 as an initial value:

```
let x = 2;
x += 2;
```

After this assignment operation, the value of x becomes 4, because x += 2 is the same as $x = x + 2$:

In the next assignment operation, we will subtract 2:

```
x -= 2;
```

So, after this operation the value of x becomes 2 again ($x = x - 2$). In the next operation, we are going to multiply the value by 6:

```
x *= 6;
```

When this line has been executed, the value of x is no longer 2, but becomes 12 ($x = x * 6$). In the next line, we are going to use an assignment operator to perform a division:

```
x /= 3;
```

After dividing x by 3, the new value becomes 4. The next assignment operator we will use is exponentiation:

```
x **= 2;
```

The value of x becomes 16, because the old value was 4, and 4 to the power of 2 equals 16 (4 * 4). The last assignment operator we will talk about is the modulus assignment operator:

```
x %= 3;
```

After this assignment operation, the value of x is 1, because 3 can fit 5 times into 16 and then leaves 1.

Practice exercise 2.4

Create variables for three numbers: *a*, *b*, and *c*. Update these variables with the following actions using the assignment operators:

- Add *b* to *a*
- Divide *a* by *c*

- Replace the value of c with the modulus of *c* and *b*
- Print all three numbers to the console

Comparison operators

Comparison operators are different from the operators we have seen so far. The outcome of the comparison operators is always a Boolean, true, or false.

Equal

There are a few equality operators that determine whether two values are equal. They come in two flavors: equal value only, or equal value and data type. The first one returns true when the values are equal, even though the type is different, while the second returns true only when the value and the type are the same:

```
let x = 5;
let y = "5";
console.log(x == y);
```

The double equals operator, two equal signs, means that it will only check for equal value and not for data type. Both have the value `5`, so it will log `true` to the console. This type of equality is sometimes called loose equality.

The triple equals operator, written as three equal signs, means that it will evaluate both the value and the data type to determine whether both sides are equal or not. They both need to be equal in order for this statement to be true, but they are not and therefore the following statement outputs `false`:

```
console.log(x === y);
```

This is sometimes also called strict equality. This triple equals operator is the one you should most commonly be using when you need to check for equality, as only with this one can you be sure that both variables are really equal.

Not equal

Not equal is very similar to equal, except it does the opposite—it returns true when two variables are not equal, and false when they are equal. We use the exclamation mark for not equal:

```
let x = 5;
let y = "5";
console.log(x != y);
```

This will log `false` to the console. If you are wondering what is going on here, take a look again at the double and triple equals operators, because it is the same here. However, when there is only one equals sign in a not-equal operator, it is comparing loosely for non-equality. Therefore, it concludes that they are equal and therefore not equal should result in false. The one with two equals signs is checking for strict non-equality:

```
console.log(x !== y);
```

This will conclude that since `x` and `y` have different data types, they are not the same, and will log `true` to the console.

Greater than and smaller than

The greater than operator returns true if the left-hand side is greater than the right-hand side of the operation. We use the `>` character for this. We also have a greater than or equal to operator, `>=`, which returns `true` if the left-hand side is greater than or equal to the right-hand side.

```
let x = 5;
let y = 6;
console.log(y > x);
```

This one will log `true`, because `y` is greater than `x`.

```
console.log(x > y);
```

Since `x` is not greater than `y`, this one will log `false`.

```
console.log(y > y);
```

`y` is not greater than `y`, so this one will log `false`.

```
console.log(y >= y);
```

This last one is looking at whether `y` is greater than or equal to `y`, and since it is equal to itself, it will log `true`.

It might not surprise you that we also have smaller than (`<`) and smaller than or equal to operators (`<=`). Let's have a look at the smaller than operator, as it is very similar to the previous ones.

```
console.log(y < x);
```

This first one will be `false`, since `y` is not smaller than `x`.

```
console.log(x < y);
```

So, this second one will log `true`, because `x` is smaller than `y`.

```
console.log(y < y);
```

`y` is not smaller than `y`, so this one will log `false`.

```
console.log(y <= y);
```

This last one looks at whether `y` is smaller than or equal to `y`. It is equal to `y`, so it will log `true`.

Logical operators

Whenever you want to check two conditions in one, or you need to negate a condition, the logical operators come in handy. You can use and, or, and not.

And

The first one we will have a look at is and. If you want to check whether `x` is greater than `y` and `y` is greater than `z`, you would need to be able to combine two expressions. This can be done with the `&&` operator. It will only return `true` if both expressions are true:

```
let x = 1;
let y = 2;
let z = 3;
```

With these variables in mind, we are going to have a look at the logical operators:

```
console.log(x < y && y < z);
```

This will log `true`, you can read it like this: if `x` is smaller than `y` and `y` is smaller than `z`, it will log `true`. That is the case, so it will log `true`. The next example will log `false`:

```
console.log(x > y && y < z);
```

Since `x` is not greater than `y`, one part of the expression is not true, and therefore it will result in `false`.

Or

If you want to get true if either one of the expressions is true, you use or. The operator for this is `||`. These pipes are used to see if either one of these two is true, in which case the whole expression evaluates to `true`. Let's have a look at the or operator in action:

```
console.log(x > y || y < z);
```

This will result in `true`, whereas it was `false` with `&&`. This is because only one of the two sides needs to be true in order for the whole expression to evaluate to `true`. This is the case because `y` is smaller than `z`.

When both sides are false, it will log `false`, which is the case in the next example:

```
console.log(x > y || y > z);
```

Not

In some cases you will have to negate a Boolean. This will make it the opposite value. It can be done with the exclamation mark, which reads as not:

```
let x = false;
console.log(!x);
```

This will log true, since it will simply flip the value of the Boolean. You can also negate an expression that evaluates to a Boolean, but you would have to make sure that the expression gets evaluated first by grouping it.

```
let x = 1;
let y = 2;
console.log(!(x < y));
```

`x` is smaller than `y`, so the expression evaluates to `true`. But, it gets negated due to the exclamation mark and prints `false` to the console.

Chapter project

Miles-to-kilometers converter

Create a variable that contains a value in miles, convert it to kilometers, and log the value in kilometers in the following format:

```
The distance of 130 kms is equal to 209.2142 miles
```

For reference, 1 mile equals 1.60934 kilometers.

BMI calculator

Set values for height in inches and weight in pounds, then convert the values to centimeters and kilos, outputting the results to the console:

- 1 inch is equal to 2.54 cm
- 2.2046 pounds is equal to 1 kilo

Output the results. Then, calculate and log the BMI: this is equal to weight (in kilos) divided by squared height (in meters). Output the results to the console.

Self-check quiz

1. What data type is the following variable?

```
const c = "5";
```

2. What data type is the following variable?

```
const c = 91;
```

3. Which one is generally better, line 1 or line 2?

```
let empty1 = undefined; //line 1
let empty2 = null; //line 2
```

4. What is the console output for the following?

```
let a = "Hello";
a = "world";
console.log(a);
```

5. What will be logged to the console?

```
let a = "world";
let b = `Hello ${a}!`;
console.log(b);
```

6. What is the value of a?

```
let a = "Hello";
a = prompt("world");
console.log(a);
```

7. What is the value of `b` output to the console?

```
let a = 5;
let b = 70;
let c = "5";
b++;
console.log(b);
```

8. What is the value of `result`?

```
let result = 3 + 4 * 2 / 8;
```

9. What is the value of `total` and `total2`?

```
let firstNum = 5;
let secondNum = 10;
firstNum++;
secondNum--;
let total = ++firstNum + secondNum;
console.log(total);
let total2 = 500 + 100 / 5 + total--;
console.log(total2);
```

10. What is logged to the console here?

```
const a = 5;
const b = 10;
console.log(a > 0 && b > 0);
console.log(a == 5 && b == 4);
console.log(true ||false);
console.log(a == 3 || b == 10);
console.log(a == 3 || b == 7);
```

Summary

In this chapter, we dealt with the first two programming building blocks: variables and operators. Variables are special fields that have a name and contain values. We declare a variable by using one of the special variable-defining words: `let`, `var`, or `const`. Variables enable us to make our scripts dynamic, store values, access them later, and change them later. We discussed some primitive data types, including strings, numbers, Booleans, and Symbols, as well as more abstract types such as undefined and null. You learned how to determine the type of a variable using the `typeof` word. And you saw how you can convert the data type by using the built-in JavaScript methods `Number()`, `String()`, and `Boolean()`.

Then we moved on and discussed operators. Operators enable us to work with our variables. They can be used to perform calculations, compare variables, and more. The operators we discussed included arithmetic operators, assignment operators, comparison operators, and logical operators.

After this chapter, you are ready to deal with more complex data types, such as arrays and objects. We'll cover these in the next chapter.