2

DATA TYPES AND VARIABLES

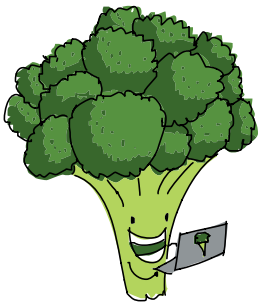
Programming is all about manipulating data, but what *is* data? *Data* is information that we store in our computer programs. For example, your name is a piece of data, and so is your age. The color of your hair, how many siblings you have, where you live, whether you're male or female—these things are all data.

In JavaScript, there are three basic types of data: numbers, strings, and Booleans. Numbers are used for representing, well, numbers! For example, your age can be represented as a number, and so can your height. Numbers in JavaScript look like this:

```
5;
```

Strings are used to represent text. Your name can be represented as a string in JavaScript, as can your email address. Strings look like this:

```
"Hi, I'm a string";
```



Booleans are values that can be true or false. For example, a Boolean value about you would be whether you wear glasses. Another could be whether you like broccoli. A Boolean looks like this:

```
true;
```

There are different ways to work with each data type. For example, you can multiply two numbers, but you can't multiply two strings. With a string, you can ask for the first five characters. With Booleans, you can check to see whether two values are both true. The following code example illustrates each of these possible operations.

```
99 * 123;
12177
"This is a long string".slice(0, 4);
"This"
true && false;
false
```

All data in JavaScript is just a combination of these types of data. In this chapter, we'll look at each type in turn and learn different ways to work with each type.

NOTE *You may have noticed that all of these commands end with a semicolon (;). Semicolons mark the end of a particular JavaScript command or instruction (also called a* statement*), sort of like the period at the end of a sentence.*

NUMBERS AND OPERATORS

JavaScript lets you perform basic mathematical operations like addition, subtraction, multiplication, and division. To make these calculations, we use the symbols `+`, `-`, `*`, and `/`, which are called *operators*.

You can use the JavaScript console just like a calculator. We've already seen one example, adding together 3 and 4. Let's try something harder. What's 12,345 plus 56,789?

```
12345 + 56789;
69134
```

That's not so easy to work out in your head, but JavaScript calculated it in no time.

You can add multiple numbers with multiple plus signs:

```
22 + 33 + 44;
99
```

JavaScript can also do subtraction . . .

```
1000 - 17;
983
```

and multiplication, using an asterisk . . .

```
123 * 456;
56088
```

and division, using a forward slash . . .

```
12345 / 250;
49.38
```

You can also combine these simple operations to make something more complex, like this:

```
1234 + 57 * 3 - 31 / 4;
1397.25
```

Here it gets a bit tricky, because the result of this calculation (the answer) will depend on the order that JavaScript does

each operation. In math, the rule is that multiplication and division always take place before addition and subtraction, and JavaScript follows this rule as well.

Figure 2-1 shows the order JavaScript would follow. First, JavaScript multiplies `57 * 3` and gets `171` (shown in red). Then it divides `31 / 4` to get `7.75` (shown in blue). Next it adds `1234 + 171` to get `1405` (shown in green). Finally it subtracts `1405 - 7.75` to get `1397.25`, which is the final result.

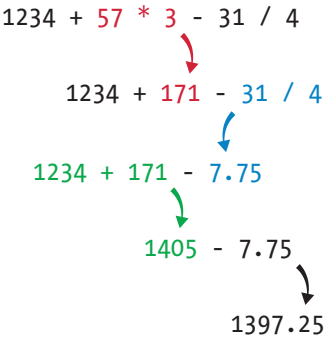


Figure 2-1: The order of operations: multiplication, division, addition, subtraction

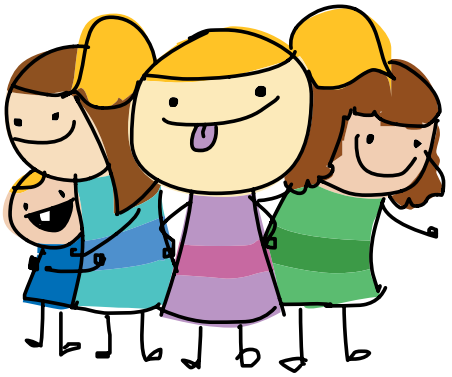
What if you wanted to do the addition and the subtraction first, before doing the multiplication and division? For example, say you have 1 brother and 3 sisters and 8 candies, and you want to split the candies equally among your 4 siblings? (You've already taken your share!) You would have to divide 8 by your number of siblings.

Here's an attempt:

```
8 / 1 + 3;
11
```

That can't be right! You can't give each sibling 11 candies when you've only got 8! The problem is that JavaScript does division before addition, so it divides 8 by 1 (which equals 8) and then adds 3 to that, giving you 11. To fix this and make JavaScript do the addition first, we can use *parentheses*:

```
8 / (1 + 3);
2
```



That's more like it! Two candies to each of your siblings. The parentheses force JavaScript to add 1 and 3 *before* dividing 8 by 4.

TRY IT OUT!

Let's say your friend is trying to use JavaScript to work out how many balloons to buy. She's throwing a party and wants everyone to have 2 balloons to blow up. There were originally 15 people coming, but then she invited 9 more. She tries the following code:

```
15 + 9 * 2;
33
```

But that doesn't seem right.

The problem is that the multiplication is happening before the addition. How would you add parentheses to make sure that JavaScript does the addition first? How many balloons does your friend really need?

VARIABLES

JavaScript lets you give names to values using *variables*. You can think of a variable as a box that you can fit one thing in. If you put something else in it, the first thing goes away.

To create a new variable, use the keyword `var`, followed by the name of the variable. A *keyword* is a word that has special meaning in JavaScript. In this case, when we type `var`, JavaScript knows that we are about to enter the name of a new variable. For example, here's how you'd make a new variable called `nick`:

```
var nick;
undefined
```

We've created a new variable called `nick`. The console spits out `undefined` in response. But this isn't an error! That's just what JavaScript does whenever a command doesn't return a value. What's a return value? Well, for example, when you typed `12345 + 56789;`, the console returned the value `69134`. Creating a variable in JavaScript doesn't return a value, so the interpreter prints `undefined`.

To give the variable a value, use the equal sign:

```
var age = 12;
undefined
```

Setting a value is called *assignment* (we are assigning the value `12` to the variable `age`). Again, `undefined` is printed, because we're creating another new variable. (In the rest of my examples, I won't show the output when it's `undefined`.)

The variable `age` is now in our interpreter and set to the value `12`. That means that if you type `age` on its own, the interpreter will show you its value:

```
age;
12
```

Cool! The value of the variable isn't set in stone, though (they're called *variables* because they can *vary*), and if you want to update it, just use `=` again:

```
age = 13;
13
```

This time I didn't use the `var` keyword, because the variable `age` already exists. You need to use `var` only when you want to *create* a variable, not when you want to change the value of a variable. Notice also, because we're not creating a new variable, the value `13` is returned from the assignment and printed on the next line.

This slightly more complex example solves the candies problem from earlier, without parentheses:

```
var numberOfSiblings = 1 + 3;
var numberOfCandies = 8;
numberOfCandies / numberOfSiblings;
2
```

First we create a variable called `numberOfSiblings` and assign it the value of `1 + 3` (which JavaScript works out to be `4`). Then we create the variable `numberOfCandies` and assign `8` to it. Finally, we write `numberOfCandies / numberOfSiblings`. Because `numberOfCandies` is `8` and `numberOfSiblings` is `4`, JavaScript works out `8 / 4` and gives us `2`.

NAMING VARIABLES

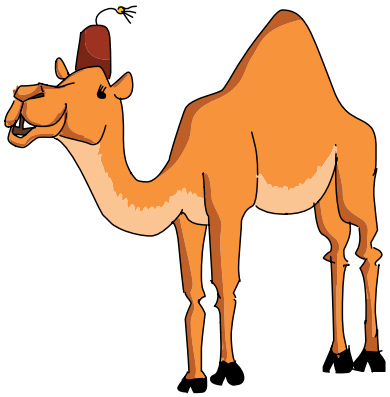
Be careful with your variable names, because it's easy to misspell them. Even if you just get the capitalization wrong, the JavaScript interpreter won't know what you mean! For example, if you accidentally used a lowercase *c* in `numberOfCandies`, you'd get an error:

```
numberOfcandies / numberOfSiblings;
ReferenceError: numberOfcandies is not defined
```

Unfortunately, JavaScript will only do exactly what you ask it to do. If you misspell a variable name, JavaScript has no idea what you mean, and it will display an error message.

Another tricky thing about variable names in JavaScript is that they can't contain spaces, which means they can be difficult to read. I could have named my variable `numberofcandies` with no capital letters, which makes it even harder to read because it's not clear where the words end. Is this variable "numb erof can dies" or "numberofcan dies"? Without the capital letters, it's hard to tell.

One common way to get around this is to start each word with a capital letter as in `NumberOfCandies`. (This convention is called *camel case* because it supposedly looks like the humps on a camel.)



The standard practice is to have variables start with a lowercase letter, so it's common to capitalize each word except for the first one, like this: `numberOfCandies`. (I'll follow this version of the camel case convention throughout this book, but you're free to do whatever you want!)

CREATING NEW VARIABLES USING MATH

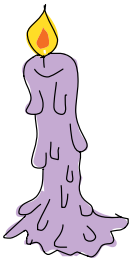
You can create new variables by doing some math on older ones. For example, you can use variables to find out how many seconds there are in a year—and how many seconds old you are! Let's start by finding the number of seconds in an hour.

SECONDS IN AN HOUR

First we create two new variables called `secondsInAMinute` and `minutesInAnHour` and make them both `60` (because, as we know, there are 60 seconds in a minute and 60 minutes in an hour). Then we create a variable called `secondsInAnHour` and set its value to the result of multiplying `secondsInAMinute` and `minutesInAnHour`. At ❶, we enter `secondsInAnHour`, which is like saying, "Tell me the value of `secondsInAnHour` right now!" JavaScript then gives you the answer: it's `3600`.

```
var secondsInAMinute = 60;
var minutesInAnHour = 60;
var secondsInAnHour = secondsInAMinute * minutesInAnHour;
❶ secondsInAnHour;
3600
```

SECONDS IN A DAY



Now we create a variable called `hoursInADay` and set it to `24`. Next we create the variable `secondsInADay` and set it equal to `secondsInAnHour` multiplied by `hoursInADay`. When we ask for the value `secondsInADay` at ❶, we get `86400`, which is the number of seconds in a day.

```
var hoursInADay = 24;
var secondsInADay = secondsInAnHour * hoursInADay;
❶ secondsInADay;
86400
```

SECONDS IN A YEAR

Finally, we create the variables `daysInAYear` and `secondsInAYear`. The `daysInAYear` variable is assigned the value `365`, and the variable `secondsInAYear` is assigned the value of `secondsInADay` multiplied by `daysInAYear`. Finally, we ask for the value of `secondsInAYear`, which is `31536000` (more than 31 million)!

```
var daysInAYear = 365;
var secondsInAYear = secondsInADay * daysInAYear;
secondsInAYear;
31536000
```

AGE IN SECONDS

Now that you know the number of seconds in a year, you can easily figure out how old you are in seconds (to the nearest year). For example, as I'm writing this, I'm 29:

```
var age = 29;
age * secondsInAYear;
914544000
```

To figure out your age in seconds, enter the same code, but change the value in `age` to *your* age. Or just leave out the `age` variable altogether and use a number for your age, like this:

```
29 * secondsInAYear;
914544000
```

I'm more than 900 million seconds old! How many seconds old are you?

INCREMENTING AND DECREMENTING

As a programmer, you'll often need to increase or decrease the value of a variable containing a number by 1. For example, you might have a variable that counts the number of high-fives you received today. Each time someone high-fives you, you'd want to increase that variable by 1.

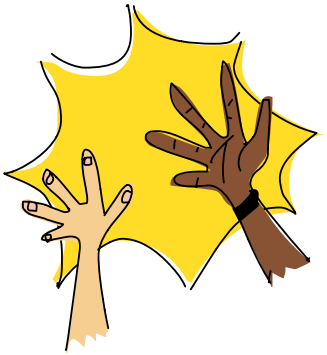
Increasing by 1 is called *incrementing*, and decreasing by 1 is called *decrementing*. You increment and decrement using the operators `++` and `--`.

```
var highFives = 0;
++highFives;
1
++highFives;
2
--highFives;
1
```

When we use the `++` operator, the value of `highFives` goes up by 1, and when we use the `--` operator, it goes down by 1. You can also put these operators *after* the variable. This does the same thing, but the value that gets returned is the value *before* the increment or decrement.

```
highFives = 0;
highFives++;
0
highFives++;
1
highFives;
2
```

In this example, we set `highFives` to `0` again. When we call `highFives++`, the variable is incremented, but the value that gets printed is the value *before* the increment happened. You can see at the end (after two increments) that if we ask for the value of `highFives`, we get `2`.



+= (PLUS-EQUALS) AND -= (MINUS-EQUALS)

To increase the value of a variable by a certain amount, you could use this code:

```
var x = 10;
x = x + 5;
x;
15
```

Here, we start out with a variable called `x`, set to 10. Then, we assign `x + 5` to `x`. Because `x` was 10, `x + 5` will be 15. What we're doing here is using the old value of `x` to work out a new value for `x`. Therefore, `x = x + 5` really means "add 5 to `x`."

JavaScript gives you an easier way of increasing or decreasing a variable by a certain amount, with the `+=` and `-=` operators. For example, if we have a variable `x`, then `x += 5` is the same as saying `x = x + 5`. The `-=` operator works in the same way, so `x -= 9` would be the same as `x = x - 9` ("subtract 9 from `x`"). Here's an example using both of these operators to keep track of a score in a video game:

```
var score = 10;
score += 7;
17
score -= 3;
14
```

In this example, we start with a score of 10 by assigning the value `10` to the variable `score`. Then we beat a monster, which increases `score` by 7 using the `+=` operator. (`score += 7` is the same as `score = score + 7`.) Before we beat the monster, `score` was `10`, and 10 + 7 is 17, so this operation sets `score` to `17`.

After our victory over the monster, we crash into a meteor and `score` is reduced by 3. Again, `score -= 3` is the same as `score = score - 3`. Because `score` is `17` at this point, `score - 3` is `14`, and that value gets reassigned to `score`.

TRY IT OUT!

There are some other operators that are similar to `+=` and `-=`. For example, there are `*=` and `/=`. What do you think these do? Give them a try:

```
var balloons = 100;
balloons *= 2;
???
```

What does `balloons *= 2` do? Now try this:

```
var balloons = 100;
balloons /= 4;
???
```

What does `balloons /= 4` do?