case of true || X, no matter what X is—even if it’s an expression that does something terrible—the result will be true, and X is never evaluated. The same goes for false && X, which is false and will ignore X. This is called short-circuit evaluation.

A single var statement may define multiple variables. The definitions must be separated by commas.

var one = 1, two = 2;

console.log(one + two);

// → 3

**KEYWORDS**

break case catch class const continue debugger

default delete do else enum export extends false

finally for function if implements import in

instanceof interface let new null package private

protected public return static super switch this

throw true try typeof var void while with yield

---------------------

do {

var name = prompt("Who are you?");

} while (!name);

console.log(name);

This program will force you to enter a name. It will ask again and again until it gets something that is not an empty string. Applying the ! operator will convert a value to Boolean type before negating it, and all strings except "" convert to true.

-------------------------

FOR LOOP

for (var current = 20; ; current++) {

2

if (current % 7 == 0)

3

break;

4

}

5

console.log(current);

6

// → 21

The for construct in the example does not have a part that checks for the end of the loop. This means that the loop will never stop unless the break statement inside is executed.

For counter += 1 and counter -= 1, there are even shorter equivalents: counter++ and counter--.

switch (prompt("What is the weather like?")) {

case "rainy":

console.log("Remember to bring an umbrella.");

break;

case "sunny":

console.log("Dress lightly.");

case "cloudy":

console.log("Go outside.");

break;

default:

console.log("Unknown weather type!");

break;

}

Chapter3

Power function

var power = function(base, exponent) {

var result = 1;

for (var count = 0; count < exponent; count++)

result \*= base;

return result;

};

console.log(power(2, 10));

// → 1024

# Chapter 3

console.log("The future says:", future());

function future() {

return "We STILL have no flying cars.";

}

This code works, even though the function is defined *below* the code that uses it. This is because function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope. This is sometimes useful because it gives us the freedom to order code in a way that seems meaningful, without worrying about having to define all functions above their first use.

The upside is that this behavior can be used to have a function take “optional” arguments. For example, the following version of power can be called either with two arguments or with a single argument, in which case the exponent is assumed to be two, and the function behaves like square.

function power(base, exponent) {

if (exponent == undefined)

exponent = 2;

var result = 1;

for (var count = 0; count < exponent; count++)

result \*= base;

return result;

}

console.log(power(4));

// → 16

console.log(power(4, 3));

// → 64

function multiplier(factor) {

return function(number) {

return number \* factor;

};

}

var twice = multiplier(2);

console.log(twice(5));

// → 10

In the example, multiplier returns a frozen chunk of code that gets stored in the twice variable. The last line then calls the value in this variable, causing the frozen code (return number \* factor;) to be activated. It still has access to the factor variable from the multiplier call that created it, and in addition it gets access to the argument passed when unfreezing it, 5, through its number parameter.

RECURSION

function power(base, exponent) {

if (exponent == 0)

return 1;

else

return base \* power(base, exponent - 1);

}

console.log(power(2, 3));

// → 8

Consider this puzzle: by starting from the number 1 and repeatedly either adding 5 or multiplying by 3, an infinite amount of new numbers can be produced. How would you write a function that, given a number, tries to find a sequence of such additions and multiplications that produce that number? For example, the number 13 could be reached by first multiplying by 3 and then adding 5 twice, whereas the number 15 cannot be reached at all.

Here is a recursive solution:

function findSolution(target) {

function find(start, history) {

if (start == target)

return history;

else if (start > target)

return null;

else

return find(start + 5, "(" + history + " + 5)") ||

find(start \* 3, "(" + history + " \* 3)");

}

return find(1, "1");

}

console.log(findSolution(24));

// → (((1 \* 3) + 5) \* 3)