var and let the difference:

In JavaScript, var is one keyword used to declare variables. These variable declarations “float” all the way to the top. This is known as *variable hoisting*. Variables declared at the bottom of the script will not be the last thing executed in a JavaScript program during runtime.

Here’s an example:

function scope1(){

var top = "top";  
 bottom = "bottom";

console.log(bottom);

var bottom;

}

scope1(); *// prints "bottom" - no error*

How does this work? The previous is the same as writing the following:

function scope1(){

var top = "top";  
 var bottom;  
 bottom = "bottom"

console.log(bottom);

}

scope1(); *// prints "bottom" - no error*

The bottom variable declaration, which was at the last line in the function, is floated to the top, and logging the variable works.

Another keyword that can be used to declare a variable is let. Any variables declared this way are in the closest block scope (meaning within the {} they were declared in).

function scope3(print){

if(print){

let insideIf =’12’;

console.log(insideIf); }

scope3(true); *// prints ''*

In this example, nothing is logged to the console because the insideIf variable is available only inside the if statement block.

Variable Types

In JavaScript, there are seven primitive data types: boolean, number, string, undefined, object, function, and symbol.

**Objects**

var o1 = {};  
var o2 = {};  
o1 == o2 *// returns false* 5 o1 === o2 *// returns false*

Although these objects are equivalent (same properties and values), they are not equal. Namely, the variables have different addresses in memory.

This is why most JavaScript applications use utility libraries such as lodash1 or underscore,2 which have the isEqual(*object1*, *object2*) function to check two objects or values strictly. This occurs via implementation of some property-based equality checking where each property of the object is compared.

var function1 = function(){console.log(2)};

var function2 = function(){console.log(2)};

console.log(function1 == function2); *// prints 'false'*

Although the two functions perform the same operation, the functions have different addresses in memory, and therefore the equality operator returns false. The primitive equality check operators, == and ===, can be used only for strings and numbers. To implement an equivalence check for objects, each property in the object needs to be checked.

**Number System**

JavaScript uses a 32-bit floating-point representation for numbers, as shown in Figure 3-1. In this example, the value is 0.15625. The sign bit (the 31st bit) indicates that the number is negative if the sign bit is 1. The next 8 bits (the 30th to 23rd bits) indicate the exponent value, e. Finally, the remaining 23 bits represent the fraction value.

With decimal fractions, this floating-point number system causes some rounding errors in JavaScript. For example, 0.1 and 0.2 cannot be represented precisely.

Hence, 0.1 + 0.2 === 0.3 yields false.

0.1 + 0.2 === 0.3; *// prints 'false'*

To really understand why 0.1 cannot be represented properly as a 32-bit floating- point number, you must understand binary. Representing many decimals in binary requires an infinite number of digits. This is because binary numbers are represented by 2*n* where *n* is an integer.

# **Integer Rounding**

Since JavaScript uses floating point to represent all numbers, integer division does not work. Integer division in programming languages like Java simply evaluates division expressions to their quotient.

# Number.EPSILON

Number.EPSILON returns the smallest interval between two representable numbers. This is useful for the problem with floating-point approximation.

function numberEquals(x, y) {

return Math.abs(x - y) < Number.EPSILON;

}

numberEquals(0.1 + 0.2, 0.3); *// true*

This function works by checking whether the difference between the two numbers is smaller than Number.EPSILON. Remember that Number.EPSILON is the smallest difference between two *representable* numbers. The difference between 0.1+0.2 and 0.3 will be smaller than Number.EPSILON.

# **Maximums**

Number.MAX\_SAFE\_INTEGER returns the largest integer.

Number.MAX\_SAFE\_INTEGER + 1 === Number.MAX\_SAFE\_INTEGER + 2

This returns true because it cannot go any higher. However, it does not work for floating-point decimals.

Number.MAX\_SAFE\_INTEGER + 1.111 === Number.MAX\_SAFE\_INTEGER + 2.022;

*// false*

Number.MAX\_VALUE returns the largest floating-point number possible. Number.MAX\_VALUE is equal to 1.7976931348623157e+308.

Number.MAX\_VALUE + 1 === Number.MAX\_VALUE + 2; *// true*

Unlike Number.MAX\_SAFE\_INTEGER, this uses double-precision floating-point representation and works for floating points as well.

Number.MAX\_VALUE + 1.111 === Number.MAX\_VALUE + 2.022; *// true*

# Minimums

Number.MIN\_SAFE\_INTEGER returns the smallest integer. Number.MIN\_SAFE\_INTEGER is equal to -9007199254740991.

Number.MIN\_SAFE\_INTEGER - 1 === Number.MIN\_SAFE\_INTEGER - 2; *// true* This returns true because it cannot get any smaller. However, it does not work for floating-point decimals.

Number.MIN\_SAFE\_INTEGER - 1.111 === Number.MIN\_SAFE\_INTEGER - 2.022;

*// false*

Number.MIN\_VALUE returns the smallest floating-point number possible.

Number.MIN\_VALUE is equal to 5e-324. This is not a negative number since it is the smallest *floating-point* number possible and means that Number.MIN\_VALUE is actually bigger than Number.MIN\_- SAFE\_INTEGER.

Number.MIN\_VALUE is also the closest floating point to zero. 1 Number.MIN\_VALUE - 1 == -1; *// true*

This is because this is similar to writing 0 - 1 == -1.

**Infinity**

The only thing greater than Number.MAX\_VALUE is Infinity, and the only thing smaller than Number.MAX\_SAFE\_INTEGER is -Infinity.

Infinity > Number.MAX\_SAFE\_INTEGER; *// true* 2 -Infinity < Number.MAX\_SAFE\_INTEGER *// true;* 3 -Infinity -32323323 == -Infinity -1; *// true*