JAVASCRIPT:

SCOPE:

Now check the value of c outside the block:

c;<< 5

In this example, c is defined inside the block, but because this is done without using let or const, it has global scope and is also available outside the block.

In the last example we'll create a local variable, d, inside a block that is not accessible outside the block.

In this example, d is defined inside the block, but by using const so it has local scope and is only accessible inside the block. When we try to log the value of d outside the block, it causes an error because d is not defined globally.

NAMING VARIABLES:

Constant and variable names can start with any upper or lower-case letter, an underscore, \_, or dollar character, $. They can also contain numbers, but cannot start with them.

Constant and variable names are case sensitive, so ANSWER is different to Answer and answer.

DIRECT ASSIGNMENT AND ASSIGNMENT BY REFERENCE:

When you assign a primitive value to a variable, any changes you make are made directly to that value:

*const* a = 1;*let* b = a; *// a = 1, b = 1*

b = 2; *// a = 1, b = 2*

In the example above, a references the primitive number 1. We then assign b to the same value as a. At this point a and b both have the same value of 1. Then we reassign the value of 2 to b, but the a still has a value of 1.

But if you assign a non-primitive value to a variable, then this is done by *reference*, so any changes that are subsequently made will affect *all* references to that object:

*const* c = { value: 1 };*let* d = c; *// c.value = 1, d.value = 1*d.value = 2; *// c.value = 2, d.value = 2*

In the example above, the change to the value property of d also results in the value property of c changing as well. This is because the variables c and d are both referencing the *same* object, so any changes to one of them will also affect the other.

STRINGS:

If you want to use double quote marks inside a string literal, you need to use single quote marks to enclose the string. And if you want to use an apostrophe in your string, you will need to use double quote marks to enclose the string, otherwise the apostrophe will terminate the string, causing an error:

The backslash is used to escape special characters in strings such as:

* Single quote marks \'
* Double quote marks \"
* End of line \n
* Carriage return \r
* Tab \t

If you want to actually write a backslash, you need to escape it with another backslash:

"This is a backslash \\"

<< "This is a backslash \"

STRING PROPERTIES & METHODS:

We can access the properties of a string using dot notation. This involves writing a dot followed by the property we are interested in.

name.length;

An alternative notation you can use to access a primitive data type's properties are square brackets:

name['length'];

All properties of primitive data types are immutable, meaning they’re unable to be changed.

A method is an action that a primitive data type or object can perform. To call a method, we use the dot operator (.) followed by the name of the method, followed by parentheses

name.toUpperCase();

If you want to know which character is at a certain position, you can use the charAt() method:

name.charAt(1);<< 'l'

counting usually starts at zero in programming!).

name.indexOf('A');

If a character doesn’t appear in the string, -1 will be returned:

If we want the last occurrence of a character or substring, we can use the lastIndexOf() method:

name.lastIndexOf('a');

name.includes('a');

<< true

To check if a string starts with a certain character, we can use the startsWith() method. Be careful though, it's case-sensitive:

And we can use the similar endsWith() method to check if a string ends with a particular character:

The concat() method can be used to concatenate two or more strings together:

'JavaScript'.concat('Ninja');<< 'JavaScriptNinja'

'Hello'.concat(' ','World','!');<< 'Hello World!'

'Java' + 'Script' + ' ' + 'Ninja';<< 'JavaScript Ninja'

The trim() method will remove any whitespace from the beginning and end of a string:

' Hello World '.trim(); *// the space in the middle will be preserved*<< 'Hello World'

' \t\t JavaScript Ninja! \r'.trim(); *// escaped tabs and carriage returns are also removed*<< 'JavaScript Ninja!'

ES6 also introduced the repeat() method that will repeat a string the stated number of times:

'Hello'.repeat(2);<< 'HelloHello'

TEMPLATE LITERALS:

**Template literals** are a special types of string that were introduced in ES6. Template literals use the backtick character, `, to deliminate the string, as shown in the example below:

`Hello!`;

This has the advantage of being able to use both types of quote mark within the string:

`She said, "It's Me!"`

They also allow interpolation of JavaScript code.

*const* name = `Siri`;`Hello ${ name }!`;<< 'Hello Siri!'

*const* age = 39;`I will be ${ age + 1 } next year`;<< 'I will be 40 next year'

Template literals can also contain line breaks, which are all converted into a Line Feed (\n):

`This is the start ...

.... and this is the end`

<< 'This is the start ...\n\n\n.... and this is the end'

If you want to place a backtick inside a template literal, then it needs to be escaped in the usual way, using a backslash:

SYMBOLS:

Symbols were introduced as a new primitive value in ES6. They can be used to create unique values, which helps to avoid any naming collisions.

Symbols are the only primitives that don't have a literal form. The only way to create them is to use the Symbol() function:

*const* uniqueID = Symbol();

It is recommended to add a description of the symbol inside the parentheses:

*const* uniqueID = Symbol('this is a unique ID');

The description acts as a string representation of the symbol and is used to log the symbol in the console, making it useful for debugging purposes:

console.log(uniqueID);<< Symbol(*this* is a unique ID)

You can manually access the description using the String() function:

String(uniqueID)<< 'Symbol(this is a unique ID)'

It is possible for two variables to point to the same symbol if the for() method is used when the symbol is created:

*const* A = Symbol.*for*('shared symbol');*const* B = Symbol.*for*('shared symbol');

The variables A and B now both point to the same symbol in memory. In this case the description 'shared symbol' also acts as a shared identifier for the symbol.

NUMBERS:

As you can see in the examples above, JavaScript doesn’t distinguish between integers and floating point decimals ― they are both given the type of 'number'

ES6 provides a handy method called Number.isInteger() that can be used to check if a number is an integer:

NUMBER METHODS:

Numbers also have some built-in methods, although you need to be careful when using the dot notation with number literals that are integers because JavaScript will confuse the dot with a decimal point.

there are a few ways to deal with this

5..toExponential();>> "5e+0"

5 .toExponential(); >> "5e+0"

5.0.toExponential(); >> "5e+0"

5 .toExponential(); >> "5e+0"

Assign the number to a constant:

PI.toFixed(3); *// only one dot is needed when using constants or variables*<< "3.142"

Note that the value is returned as a string, rather than a number.

The toPrecision() method rounds a number to a fixed number of significant figures that is once again returned as a string (and often using exponential notation):

325678..toPrecision(2);

<< "3.3e+5"

2.459.toPrecision(2);

<< "2.5"

CHANGING THE VALUE OF VARIABLES:

A shorthand for doing this is to use the **compound assignment operator**, +=:

points -= 5; // decreases points by 5

<< 15

points \*= 2; // doubles points

<< 30

points /= 3; // divides value of points by 3

<< 10

points %= 7; // changes the value of points to the remainder if its current value is divided by 7

<< 3

INCREMENTING THE VALUE OF VARIABLES:

points++; *// will return 6, then increase points to 7*<< 6

++points; *// will increase points to 8, then return it*<< 8

There is also a -- operator that works in the same way:

NaN is an error value that is short for "Not a Number". It is used when an operation is attempted and the result isn’t numerical, like if you try to multiply a string by a number, for example:

*typeof* NaN; *// when is a number not a number?!?*<< 'number'

You can check if a value is a number that can be used by using the Number.isFinite() method. This will return true if the value is a number that isn't Infinity, -Infinity or NaN:

CONVERTING STRINGS TO NUMBERS:

Number('23');<< 23

Number('hello');<<< NaN

For example we can multiply a numerical string by 1, which will coerce it into a number:

*const* answer = '5' \* 1;<< 5

Another trick is to simply place a + operator in front of it:

*const* answer = +'5';<< 5

CONVERTING NUMBERS TO STRINGS:

String(3);<< '3'

Another option is to use the toString() method:

10..toString();<< '10'

PARSING NUMBERS:

There is also a useful function called parseInt() that can be used to convert a string representation of a numerical value back into a number. You can specify the base of the number you are trying to convert, for example:

parseInt('1010',2); *// converts from binary, back to decimal*<< 10

parseInt('omg',36);<< 31912

parseInt('23',10);<< 23

If a string starts with a number, the parseInt function will use this number and ignore any letters that come afterwards:

*const* address = '221B Baker Street';parseInt(address, 10)<< 221

This is different to the Number function, which returns NaN:

Number(address);<< NaN

And if you use parseInt with a decimal, it will remove anything after the decimal point:

parseInt('2.4',10);<< 2

Be careful not to think that this is rounding the number to the nearest integer; it simply removes the part after the decimal point, as seen in this example:

There is also a similar function called parseFloat() that converts strings into floating point decimal numbers:

parseFloat('2.9',10);<< 2.9

NULL:

10 + *null*; *// null behaves like zero* << 10

10 + *undefined*; *// undefined is not a number* << NaN

null is coerced to be 0, making the sum possible whereas undefined is coerced to NaN, making the sum impossible to perform.

BOOLEANS:

To find the Boolean value of something, you can use the Boolean function like so:

Only 9 values are always false and these are known as **falsy** values:

\* "" *// double quoted empty string literal*

\* '' *// single quoted empty string literal*

\* `` *// empty template literal*

\* 0

\* -0 *// considered different to 0 by JavaScript!*

\* NaN

\* false

\* *null*

\* *undefined*

LOGICAL OPERATORS:

Placing the ! operator in front of a value will convert it to a Boolean and return the opposite value

You can use double negation (!!) to find out if a value is truthy or falsy (it is a shortcut to using the Boolean function we employed earlier because you are effectively negating the negation):

Imagine a nightclub that only allows people inside if they are wearing shoes AND (&&) over 18. This is an example of a logical AND condition: anybody going into the club must satisfy both conditions before they are allowed in.

Now imagine that the club relaxes its rules and allows people in if they wear shoes OR (||)they're over 18. This means they only have to satisfy one of the rules to be allowed in. This is an example of a logical OR condition.

LAZY EVALUATION:

This means it stops evaluating any further operands once the result is clear.

BITWISE OPERATORS:

Bitwise operators work with operands that are 32-bit integers. These are numbers written in binary (base two) that have 32 digits made up of just 0 s and 1 s. Here are some examples

JavaScript will convert any values used with bitwise operators into a 32-bit integer then carry out the operation. It then changes it back into a base 10 integer to display the return value.

COMPARISON:

~2476; NOT

12 & 10 AND

12 | 10; OR

12 ^ 10; XOR

The ^ character is often used as an informal notation for exponents, so be careful not to use this mistakenly when programming in JavaScript. For example 2^3 will not return 8. There is a specific Math method for doing this that is covered in Chapter 5, and ES2017 introduced the exponent operator \*\*, that allows you to write it as 2\*\*3.

When using non-integer values, this evaluates to 1 if either operands are truthy and evaluates to 0 if both operands are truthy or both are falsy:

1 ^ 0; *// The first operand is truthy*<< 1

true ^ true; *// if both operands are true then the result is false*<< 0

SOFT EQUALILTY

We can check if answer is in fact equal to 5 using the soft, or lenient, equality operator ==, like so:

HARD EQUALITY

The hard, or strict, equality operator, ===, tests for equality but only returns true if and only if they are of the same data type:

NaN === NaN;<< false EXTRANGE VALUE

Because of this, the new Number.isNaN() method should always be used to check if a value is NaN.

A JavaScript ninja should always use hard equality when testing if two values are equal. This will avoid the problems caused by JavaScript’s type coercion.

If you want to check whether a number represented by a string is equal to a number, you should convert it to a number yourself explicitly rather than relying on type coercion happening in the background:

Number('5') === 5

<< true

This can come in handy when you’re checking values entered in a form as these are usually always submitted as strings.

!== INEQUALITY

As with equality, a ninja programmer should use the hard inequality operator as this will give more reliable results unaffected by type coercion.

As you can see, type coercion means that strings can be confused with numbers. Unfortunately, there are no "hard" greater-than or equal-to operators, so an alternative way to avoid type coercion is to use a combination of the greater-than operator, logical OR, and a hard equality:

8 > 8 || 8 === 8;<< true

8 > '8' || 8 === '8';<< false

These operators can also be used with strings, which will be alphabetically ordered to check if one string is "less than" the other:

'apples' < 'bananas';>> true

Be careful, though, as the results are case-sensitive, and upper-case letters are considered to be "less than" lower-case letters:

'apples' < 'Bananas';>> false

ARRAYS, LOGIC, AND LOOPS.

Here we can see that the sixth item (with an index of 5) has been filled with the string 'Aquaman'. This has made the array longer than it was before, so all the other unused slots in the array are filled by the value undefined.

The delete operator will remove an item from an array:

*delete* avengers[3];<< true

If we look at the avengers array, we can see that the fourth entry, 'Hulk' (with an index of 3), has indeed been removed ... but it has been replaced by a value of undefined:

DESTRUCTURING ARRAYS:

is the concept of taking values out of an array and presenting them as individual values.

Destructuring allows us to assign multiple values at the same time, using arrays:

*const* [x,y] = [1,2];

Even though the assignment is made using arrays, each individual variable exists on its own outside the array. We can see this by checking the value of each variable:

x<< 1

y<< 2

Destructuring also gives us a neat way of swapping the value of two variables over:

[x,y] = [y,x];x<< 2

y<< 1

METHODS AND PROPERTIES:

The length property is mutable, meaning you can manually change it:

As you can see, if you make the array longer, the extra slots will be filled in with undefined:

If you make the array shorter than it already is, all the extra elements will be removed completely.

To remove the last item from an array, we can use the pop() method:

The shift() method works in a similar way to the pop() method, but this removes the first item in the array:

The push() method appends a new value to the end of the array.

The unshift() method is similar to the push() method, but this appends a new item to the beginning of the array:

MERGIN ARRAYS:

The concat() method can be used to merge an array with one or more arrays:

avengers.concat(['Hulk','Hawkeye', 'Black Widow']);

Note that this does not change the avengers array, it simply creates another array combining the two arrays. You can use assignment to update the avengers array to this new array:

An alternative is to use the new spread operator that was added to ES6. The spread operator is three dots, ... that are placed in front of an array, with the effect of spreading out the elements of that array. This can be used to spread the elements of two arrays and put them together in a new array, like so:

avengers = [ ...avengers, ...['Hulk','Hawkeye', 'Black Widow'] ];<< ['Captain America', 'Iron Man', 'Thor', 'Hulk', 'Hawkeye', 'Black Widow']

The join() method can be used to turn the array into a string that comprises all the items in the array, separated by commas:

You can choose a separator other than a comma by placing it inside the parentheses. Let’s try using an ampersand:

avengers.join(' & ');<< 'Captain America & Iron Man & Thor & Hulk & Hawkeye & Black Widow'

SLICING AND SPLICING:

avengers.slice(2,4) *// starts at the third item (index of 2) and finishes at the fourth (the item with index 4 is not included)*<< ['Thor', 'Hulk']

Note that this operation is non-destructive ― no items are actually removed from the array, as we can see if we take a look at the avengers array.

The splice() method removes items from an array then inserts new items in their place. For example, the following code removes the string 'Hulk' and replaces it with 'Scarlett Witch'::

avengers.splice(3, 1, 'Scarlet Witch');<< ['Hulk']

This is a destructive operation as it changes the value of the array, as we can see below:

The first number in the parentheses tells us the index at which to start the splice. In the example we started at index 3, which is the fourth item in the array ('Hulk'). The second number tells us how many items to remove from the array. In the example, this was just one item. Every value after this is then inserted into the array in the same place the other items were removed. In this case, the string 'Scarlet Witch' is inserted into the array, starting at the fourth item. Notice the splice() method returns the items removed from the array as a subarray. So in the example, it returned the array ['Hulk'].

The splice() method can also be used to insert values into an array at a specific index without removing any items, by indicating that zero items are to be removed:

We saw earlier that we can use the delete operator to remove an item from an array. Unfortunately, this leaves a value of undefined in its place. If you want to remove a value completely, you can use the splice() method with a length of 1 and without specifying any values to add:

avengers.splice(2,1); *// will remove the item at index 2 (i.e. the third item in the array)*<< [ 'Thor' ]

We can reverse the order of an array using the reverse() method:

We can sort the order of an array using the sort() method:

It is alphabetical order by default for String objects. Note that this also changes the order of the array permanently.

Numbers are also sorted alphabetically (that is, by their first digit, rather than numerically), so 9 will come after 10 when you try to sort an array of numbers:

[5, 9, 10].sort();<< [10, 5, 9]

FINDING IF A VALUE IS IN AN ARRAY

We can find out if an array contains a particular value using the indexOf() method to find the first occurrence of a value in an array. If the item is in the array, it will return the index of the first occurrence of that item:

If the item is not in the array, it will return -1:

avengers.indexOf('Iron Man');<< 3

ES6 also introduced the includes() method. This returns a boolean value depending on whether the array contains a particular element or not:

avengers.includes('Iron Man');<< true

You can also add an extra parameter to indicate which index to start the search from:

avengers.includes('Black Widow', 1); *// will start the search from the second element in the array*<< false

MULTIDIMENSIONAL ARRAYS:

*const* coordinates = [[1,3],[4,2]];<< [[1,3],[4,2]]

To access the values in a multidimensional array, we use two indices: one to refer to the item’s place in the outer array, and one to refer to its place in the inner array:

coordinates[0][0]; *// The first value of the first array*<< 1

coordinates[1][0]; *// The first value of the second array*<< 4

coordinates[0][1]; *// The second value of the first array*<< 3

coordinates[1][1]; *// The second value of the second array*<< 2

\*EL PRIMER INDEX INDICA EL ELEMENTO DE LA ARRAY MAS GRANDE, Y VA EN ORDEN.

The spread operator that we met earlier can be used to flatten multi-dimensional arrays. Flattening an array involves removing all nested arrays so all the values are on the same level in the array. You can see an example of a flattened array below:

*const* summer = ['Jun', 'Jul', 'Aug'];

*const* winter = ['Dec', 'Jan', 'Feb'];

*const* nested = [ summer, winter ];

<< [ [ 'Jun', 'Jul', 'Aug' ], [ 'Dec', 'Jan', 'Feb' ] ]

*const* flat = [...summer, ...winter];

<< [ 'Jun', 'Jul', 'Aug', 'Dec', 'Jan', 'Feb' ]

SETS:

A set is a data structure that represents a collection of unique values, so it cannot include any duplicate values.

Sets offer a useful way to keep track of data without having to check if any values have been duplicated. It's also quick and easy to check if a particular value is in a set,

There is, at the time of writing, no literal notation for creating sets.

const list = new Set();

ADDING VALUES TO SETS:

list.add(2).add(3).add(4);<< Set { 1, 2, 3, 4 }

If you try to add a value that is already contained in the set, then the operation is simply ignored:

*const* numbers = *new* Set([1,2,3]);

If any values are repeated in the array, then they will only appear once in the set:

This gives a convenient way of eliminating any duplicate values from an array in a single operation.

If a string is used as the argument then each character will be added as a separate element, with any repeated characters ignored:

*const* letters = *new* Set('hello');letters<< Set { 'h', 'e', 'l', 'o' }

If you want to add separate words, you need to use the add() method:

*const* words = *new* Set().add('the').add('quick').add('brown').add('fox')

words<< Set { 'the', 'quick', 'brown', 'fox' }

*const* jla = *new* Set().add('Superman').add('Batman').add('Wonder Woman');

jla<< Set { 'Superman', 'Batman', 'Wonder Woman' }

All non-primitive values, such as arrays and objects, are considered unique values, even if they contain the same values. On the face of it, this appears to allow duplicate values appear in a set:

*const* arrays = *new* Set().add([1]).add([1]);

arrays<< Set { [ 1 ], [ 1 ] }

[1] === [1];<< false

Type coercion is not used when values are added to a set, so the string '2' will be added as a new entry, even if the number 2 is already an element of the set:

*const* mixedTypes = *new* Set().add(2).add('2');

mixedTypes<< Set { 2, '2' }

SET METHODS:

The number of values in a set can be found using the size() method:

The has() method can be used to check if a value is in a set. This returns a boolean value of true or false:

jla.has('Superman');

The has() method that sets use is a very efficient operation and much faster than using the includes() or indexOf()

Sets do not have index notation for inspecting individual entries, so you can't find the value of the first element in a set like this:

REMOVING VALUES FROM SETS:

The delete() method can be used to remove a value from a set. This returns a boolean value of true if the value was removed from the set, or false if the value wasn't in the set and couldn't be removed:

The clear() method can be used to remove all values from a set:

CONVERTING SETS TO ARRAYS:

A set can be converted into an array by placing the set, along with the **spread operator** directly inside an array literal.

It's also possible to use the Array.from() method to convert a set into an array.

*const* shoppingArray = Array.*from*(shoppingSet);

By combining this use of the spread operator with the ability to pass an array to the new Set() constructor, we now have a convenient way to create a copy of an array with any duplicate values removed:

*const* duplicate = [3, 1, 4, 1, 5, 9, 2, 6 ,5,3,5,9];<< [ 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9 ]

*const* nonDuplicate = [...*new* Set(repeatedArray)];<< [ 3, 1, 4, 5, 9, 2, 6 ]

WEAK SETS:

Most modern programming language, including JavaScript, employ various dynamic memory management techniques such as **garbage collection**, which is the process of automatically removing items from memory that are no longer required by the program. Some languages, such as C++, require the programmer to manually manage memory by removing items from memory once they are finished with.

Weak sets avoid this situation by garbage collecting any references to a 'dead object' that’s had its original reference removed.

*let* array = [1,2,3];*const* strong = *new* Set().add(array);

array = *null*; *// remove reference to the original*

strong<< Set { [ 1, 2, 3 ] }

array = [...strong][0];

array<< [1,2,3]

*const* weak = *new* WeakSet();

Only non-primitive data types can be added to weak sets. Trying to add primitive values will throw a type error:

Apart from these restrictions, weak sets behave in the same way as regular sets, and have the has(), add(), and delete() methods that work in the same way.

*const* array = [1,2,3];weak.add(array);<< WeakSet {}

Because weak maps use weak references to objects, they don't have access to a list of values they contain. This makes the return value in the example look as though the weak set is empty, when, in fact it isn't.

weak.has(array);<< true

weak.delete(array);<< true

MAPS:

CREATING MAPS:

*const* romanNumeals = *new* Map();

ADDING ENTRIES TO MAPS:

romanNumerals.set(1,'I');<< Map { 1 => 'I' }

Multiple items can be added to the set by repeatedly calling the set() method in one go:

MAP METHODS:

romanNumerals.get(4);<< 'IV'

The has() method can be used to check if a particular key is in a map.

romanNumerals.has(5);<< true

A map can be created with multiple values by using a nested array as a parameter:

*const* heroes = *new* Map([ ['Clark Kent','Superman'],['Bruce Wayne', 'Batman']]);

heroes.size<< 2

REMOVING ENTRIES FROM MAPS:

The delete() method can be used to remove a key and value pair from a map. This returns a boolean value of true if the value was removed or false if it wasn't in the map. To delete a specific value, you need to specify the key in parentheses:

The clear() method will remove all key and value pairs from a map:

CONVERTING MAPS TO ARRAYS:

Maps can be converted into a nested array of key-value pairs in a similar way to sets; using either the spread operator:

... or the Array.from() method:

WEAK MAPS:

**Weak maps** work in the same way as weak sets. They are the same as maps, except their keys cannot be primitives, and the garbage collector will automatically remove any dead entries when the reference to the original object is deleted.

*const* weak = *new* WeakMap();

For example, you cannot use the size() method to see how many entries they contain. The choice of which to use will depend on what you plan to use them for.

LOGIC:

IF:

ELSE:

TERNARY OPERATOR:

condition ? (*//code to run if condition is true) : (//code to run if condition is false)*

*const* n = 5;n%2 === 0 ? console.log('n is an even number') : console.log('n is an odd number');<< 'n is an odd number'

We could make the example even shorter by placing the ternary operator inside a template string:

console.log(`n is a ${(n%2 === 0)? 'even' : 'odd'} number`);

SWITCH:

*switch* (number) {

*case* 4:

console.log('You rolled a four');

*break*;

*default*:

console.log('You rolled a number less than four');

break;

}

a ninja programmer always finishes a case block with a break!

The default keyword is used at the end for any code than needs to be run if none of the cases are true.

LOOPS:

*let* bottles = 11;

*while* (--bottles){

console.log(`There were ${bottles} green bottles, hanging on a wall. And if one green bottle should accidentally fall, there'd be ${bottles-1} green bottles hanging on the wall`);}

The reason this code works is because the loop will continue while the bottles variable is true, and before each loop, the value of the bottles variable decreases by 1. When the bottles variable reaches 0, it is not true anymore (remember that 0 is a falsy value) so the loop will stop. Notice that you have to start with one more bottle (11) as it will be decreased by one even before the first block is run.

INFINITE LOOPS:

DO…WHILE LOOPS:

A do ... while loop is similar to a while loop. The only difference is that the condition comes after the block of code:

FOR LOOPS:

*for* (*let* bottles = 10 ; bottles > 0 ; bottles--) {}

Each part of a for loop are optional, and the code could be written as:

*for* ( ; bottles > 0 ; ) {}

NESTED FOR LOOPS:

LOOPING OVER ARRAYS:

*for*(*let* i=0, max=avengers.length; i < max; i++)

We want the loop to continue until it reaches the length of the array; this can be set as the variable max in the initialization part of the for loop, then the condition becomes i < max. This is preferable to using i < avengers.length because then the length of the avengers array would have to be calculated after every pass through the loop. This might not sound all that important, but it can make a big difference to the speed of the program when using large arrays.

*for*(*const* value *of* avengers){console.log(value);} FOR … OF LOOP

LOOPING OVER SETS:

*for*(*const* letter *of* letters) {console.log(letter);}

<< h

e

l

o

Note that weak sets are **non-enumerable**, so it's not possible to loop over them in this way.

LOOPING OVER MAPS:

Every map object has a keys() method lets us iterate over each key with the following for-of loop:

*for*(*const* key *of* romanNumerals.keys()) { console.log(key);}

<< 1

2

3

4

5

There is also a values() method that lets us iterate over the values in a similar way:

*for*(*const* value *of* RomanNumerals.values()) { console.log(value);}

If you want to access both the key and the value, you can use the entries() method:

*for*(*const* [key,value] *of* RomanNumerals.entries()) { console.log(`${key} in Roman numerals is ${value}`);}

Note that weak maps are also **non-enumerable**, so it isn't possible to loop over them using any of the methods shown above.

FUNCTIONS:

They can be be assigned to variables, stored in arrays and can even be returned by another functions.

DEFINING A FUNCTION:

FUNCTION DECLARATIONS:

FUNCTION EXPRESSIONS:

Another way of defining a function literal is to create a function expression. This

assigns an anonymous function to a variable:

*const* goodbye = *function*(){ console.log('Goodbye World!');};

Alternatively, we can create a named function expression instead:

*const* goodbye = *function* bye(){ console.log('Goodbye World!');};

Notice also that the example ends with a semicolon. This finishes the assignment statement, whereas a normal function declaration ends in a block (there's no need for semicolons at the end of blocks).

All functions have a read-only property called name, which can be accessed like so:

FUNCTION() CONSTRUCTORS:

A function can also be declared using the constructor Function(). The body of the function is entered as a string, as shown in this example:

*const* hi = *new* Function('console.log("Hi World!");');

A ninja programmer should always declare functions using function literals, function declarations or function expressions. These two ways of creating functions are similar, although there are some subtle differences that will be covered later in the chapter. Some people prefer function declarations as they are akin to how functions are declared in other languages. Others prefer function expressions because it is clear that functions are just another value assigned to a variable, rather than a special feature of the language.

INVOKING A FUNCTION:

[Don’t Repeat Yourself,](http://en.wikipedia.org/wiki/Don%27t_repeat_yourself" \t "_blank) or DRY, is a principle of programming that specifies that every part of a program should only be written once. This avoids duplication and means there’s no need to keep multiple pieces of code up to date and in sync.

If you have assigned a function to a variable, you need to place parentheses after the variable to invoke it as a function:

goodbye();<< 'Goodbye World!'

Remember: you need parentheses to invoke a function ― either by name or by reference to the variable it is assigned to. If you skip the parentheses, you are simply referencing the function itself rather than invoking it, as you can see here:

goodbye;<< [Function: goodbye]

All that has been returned is the function definition that the variable goodbye is pointing to, rather than running the code. This can be useful if you want to assign the function to another variable, like so:

seeya = goodbye;<< [Function: goodbye]

RETURN VALUES:

A function that doesn’t explicitly return anything (such as all the examples we have seen so far) will return undefined by default.

This means we can now assign a function invocation to a variable, and the value of that variable will be the return value of that function:

*const* message = howdy();<< 'Howdy World!'

PARAMETERS AND ARGUMENTS:

If too many arguments are provided when a function is invoked, the function will work as normal and the extra arguments will be ignored (although they can be accessed using the arguments object that is discussed in the next section):

VARIABLE NUMBERS OF ARGUMENTS:

Every function has a special variable called arguments. This is an array-like object that contains every argument passed to the function when it is invoked. We can create a simple function called arguments() that will return the arguments object so we can see what it contains:

*function* arguments(){ *return* arguments;}

arguments('hello', NaN);<< { '0': 'hello', '1': NaN }

arguments(1,2,3,4,5);<< { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }

As you can see, the arguments object that is returned contains every value that was entered. These can then be accessed using an index notation like we did with arrays, so the first argument would be accessed using arguments[0].

The problem is that arguments is not an array. It has a length property and you can read and write each element using index notation, but it doesn’t have array methods such as slice(), join(), and forEach().

A much better option is to use the rest operator. This was introduced in ES6 and can be used to deal with multiple arguments by creating an array of arguments that are available inside the body of the function.

To use the rest operator, simply place three dots in front of the last parameter in a function declaration. This will then collect all the arguments entered into an array. For example, the following function will have access to an array of all the arguments entered:

*function* rest(...args){

*return* args;

}

The args parameter is an actual array, and has access to the same methods. For example we can use a for-of loop to iterate over each value given as an argument:

*function* rest(...args){ *for*(arg *of* args){ console.log(arg); }}

IMPROVED MEAN FUNCTION:

*function* mean(...values) {

*let* total = 0;

*for*(*const* value *of* values) {

total += value;

}

*return* total/values.length;}

DEFAULT PARAMETERS:

 These are values that will be used by the function if no arguments are provided when it is invoked.

*function* hello(name='World') { console.log(`Hello ${name}!);}

Default parameters should always come after non-default parameters, otherwise default values will always have to be entered anyway. Consider the following function for calculating a discounted price in a store:

*function* discount(price, amount=10) { *return* price\*(100-amount)/100;}

ARROW FUNCTIONS:

*const* square = x => x\*x; ONE PARAMETER

*const* add = (x,y) => x + y; TWO OR MORE PARAMETERS

*const* hello = () => alert('Hello World!'); NO PARAMETERS

MORE COMPLICATED FUNCTIONS NEED THE BRACES AND THE RETURN KEYWORD:

*const* tax = (salary) => {

*const* taxable = salary - 8000;

*const* lowerRate = 0.25 \* taxable;

taxable = taxable - 20000;

*const* higherRate = 0.4 \* taxable;

*return* lowerRate + higherRate;

}

Arrow functions make perfect candidates for short, anonymous functions, and you will often see them used later in the book.

FUNCTION HOISTING:

A function can be used before declaring it.

VARIABLE HOISTING:

A variable can be hoisted to the top of the block when declaring it with VAR. Otherwise it is not hoisted.

This is the major difference between the two ways of defining function literals and it may influence your decision regarding which one to use.

A function declaration can’t be used before declaring it. A function literal can be used anywhere whenever.

CALLBACKS:

The callback is provided as a parameter, then invoked inside the body of the function.

There is nothing to actually define a parameter as a callback, so if a function isn't provided as an argument, then this code won't work. It is possible to check if an argument is a function using the following code:

*if*(*typeof*(callback) === 'function'){ callback();}

Note that the callback dance is passed as an argument without parentheses. This is because the argument is only a reference to the function. The actual callback is invoked in the body of the function, where parentheses are used.

A function can also take an anonymous function as a callback.

sing('Let It Go',()=>{ console.log("I'm standing on my head.");});<< 'I'm singing along to Let It Go.''I'm standing on my head.'

SORTING ARRAYS WITH A CALLBACK:

Sorting numbers:

*function* numerically(a,b){ *return* a-b;}

> [1,3,12,5,23,18,7].sort(numerically);<< [1, 3, 5, 7, 12, 18, 23]

So how do you sort an array of numerical values? The answer is to provide a callback function to the sort() method that tells it how to compare two values, a and b. The callback function should return the following:

* A negative value if a comes before b
* 0 if a and b are equal
* A positive value if a comes after b

ARRAY ITERATORS:

FOREACH():

With callback functions (arrow functions)

colors.forEach( (color,index) =>

console.log(`Color at position ${index} is ${color}`) );

<< "Color at position 0 is Red" "Color at position 1 is Green" "Color at position 2 is Blue"

MAP():

The map() method is very similar to the forEach() method. It also iterates over an array, and takes a callback function as a parameter that is invoked on each item in the array.

 This is often used to process data returned from databases in array form, such as adding HTML tags to plain text.

The difference is that it returns a new array that replaces each value with the return value of the callback function

[1,2,3].map( square )<< [1, 4, 9]

[1,2,3].map( x => 2 \* x);<< [2,4,6]

['red','green','blue'].map( color => `<p> ${color.toUpperCase()}</p>` );<< ['<p>RED</p>', '<p>GREEN</p>', '<p>BLUE</p>']

Notice in this and the previous example, the anonymous function takes a parameter, color, which refers to the item in the array.

It's quite common for callbacks to only used the first, index, parameter, but the next example shows all three parameters being used:

['red','green','blue'].map( (color, index, array) => `Element ${index} is ${color}. There are ${array.length} items in total.` );<< [ 'Element 0 is red. There are 3 items in total.','Element 1 is green. There are 3 items in total.','Element 2 is blue. There are 3 items in total.' ]

REDUCE():

The reduce() method is another method that iterates over each value in the array, but this time it cumulatively combines each result to return just a single value.

The first parameter represents the accumulated value of all the calculations so far, and the second parameter represents the current value in the array. The following example shows how to sum an array of numbers:

[1,2,3,4,5].reduce( (acc,val) => prev + val );<< 15

The reduce() method also takes a second parameter after the callback, which is the initial value of the accumulator, acc. For example, we could total the numbers in an array, but starting at 10, instead of zero:

[1,2,3,4,5].reduce( (acc,val) => acc + val, 10 ); *// <---- second parameter of 10 here*<< 25

FILTER():

The filter() method returns a new array that only contains items from the original

array of numbers to just the even numbers using the following code:

*const* numbers = [ 2, 7, 6, 5, 11, 23, 12 ]

numbers.filter(x => x%2 === 0 ); *// this returns true if the number is even*

<< [ 2, 6, 12 ]

The filter() method provides a useful way of finding all the truthy values from an array:

*const* array = [ 0, 1, '0', false, true, 'hello' ];array.filter(Boolean);<< [ 1, '0', true, 'hello' ]

This uses the fact that the Boolean() function will return the boolean representation of a value, so only truthy values will return true and be returned by the filter() method.

To find all the falsy values, the following filter can be used:

array.filter(x => !x);[ 0, false ]

There are other array methods that use callbacks that are worth investigating such as reduceRight(), every(), find() and some().

CHAINING ITERATORS TOGETHER:

Chaining works because the iterator functions return an array, which means that another iterator function can then be chained on to the end and it will be applied to the new array.

[1,2,3].map( x => x\*x ).reduce((acc,x) => acc + x );<< 14

QUESTIONS:

I have no questions for this week’s reading. Everything has been explained well in the reading.