Chapter 2 : Programming Basics

Comments

- can be short or long
- just enough so you know whats going on later

JS Grammar

- you don't need semicolon (;)
- but you should use just in case
- use as much white space as you need
- just keep it clean & easy to read

Reserved Words

"abstract, await, boolean, break, byte, case, catch, char, class, const, continue, debugger, default, delete, do, double, else, enum, export, extends, false, final, finally, float, for, function, goto, if, implements, import, in instanceof, int, interface, let, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, var, volatile, void, while, with, yield"

- they are used by the language itself
- this also includes undefined, NaN, Infinity

Primitive Data Types

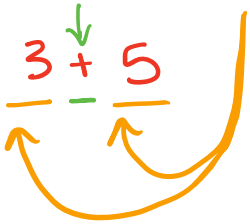
- String
- Symbol
- Number
- Boolean
- Undefined
- Null

- any value that isn't one of ↑ is an object.
- object can include arrays, functions, & object literals
- typeOf operator is for finding out the type of a value

- operator applies a operation to a value, which is known as an operand

typeOf 'hello'
operator operand

3 + 5



Variables

refer to a value stored in memory

Declaring & Assigning

- variables have to be declared before use
- const & let used to declare variables

"keyword const is used when the variable will not be reassigned to another value, whereas let is used if the variable might be reassigned later in the program"

- to assign value, use = operator
- variables using let can be reassigned another value later in program
- const stays constant & cannot be changed or reassigned & will result in error if you try

"If the variable references a non-primitive data type, such as an array, function or object, then using const will not make it immutable"

- meaning underlying data inside the object can change (known as mutating the object)

“const name = { value: 'Alexa' }; // an object
name.value = 'Siri'; // change the value
<< 'Siri”

- now the value property of the object referenced by the variable name was changed from 'Alexa' to 'Siri'
- Even if you use const, non-primitive data types can be mutated later in program

Scope

- using const & let to declare means they are block scoped, so their value exists only in the block they're declared
- scope refers to where a constant/variable is accessible by the program
- 2 common scopes:
  - local scopes
  - global scopes

Global Scope

- any variable declared outside of a block
- its accessible from any & everywhere in the program
- it is not considered good practice → keep the globals to a minimum

Local Scope

- blocks are used to create local scopes
- variables declared in blocks = only available in that block

If let or const are not used, the variable will have global scope & is available outside the block where its declared.

"Using let or const to declare variables will ensure they are block scoped, and it is good practice to always use them to declare variables."

Naming Constant & Variables

- try to give sensible names that describe what the variable represents

"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"

Variable naming

- vars (variables) that start w/ an underscore generally refer to private properties & methods
- $ character is also used by jQuery library, so best to avoid that
- const & vars are case sensitive, so ANSWER ≠ answer
- Camel case starts with lowercase, then each word is capitalized

  firstNameAndLastName
- Underscore separates each new word

  first_name_and_last_name
- JS built-in functions used camelCase!
- but most important is to be consistent

Direct Assignment & 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”

“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”

“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

a collection of characters, letters & symbols

Blackslashes

Used to escape special characters in strings:

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

String Properties & Methods

Properties are info about the object or value
Methods perform an action on the object or value
- either to change it or tell us something about it.

Wrapper Objects

"Technically, only objects have properties and methods. JavaScript overcomes this by creating wrapper objects for primitive data types. This all happens in the background, so for all intents and purposes it appears that primitive data types also have properties and 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. For example, every string has a length property that tells us how many characters are in the string:

```
const name = 'Alexa'; // declare and assign a variable
<< 'Alexa'

name.length; // retrieve the name variable's length property
<< 5
```

As you can see, this tells us that there are five characters in the string stored in the name constant."

Good Habits

- const variables & semicolons at end of line
- all properties of primitive data types are immutable, meaning unable to be changed

```
name.toUpperCase();
<< 'ALEXA'
```
] all capital letters

```
name.toLowerCase();
<< 'alexa'
```
] all lowercase

```
name.charAt(1);
<< 'l'
```
] to know which character is at a certain point

```
name.indexOf('A');
<< 0
```
] you want to find where a certain character or substring appears in a string

if character doesn't appear in string, -1 will be returned

```
name.lastIndexOf('a');
  << 4
```
want the last occurrance of a character or substring

```
name.includes('a');
  <<true
name.includes('Z');
  <<false
```
want to know if a string contains a certain character

```
name.startsWith('A');
  <<true
name.startsWith('a');
  <<false
```
check if a string starts w/ a certain character
*Be careful, it is case sensitive

```
name.endsWith('A');
  <<false
name.endsWith('a');
  <<true
```
check if a string ends w/ a certain character
*Be careful, it is case sensitive

```
'JavaScript'.concat('Ninja');
  <<'JavaScriptNinja'
'Hello'.concat(' ','World','!');
  <<Hello World!
```
used to concatenate 2 or more strings together

```
'Java' + 'Script' + ' ' + 'Ninja';
  <<'JavaScript Ninja'
```
shortcut is to concatenate using + operator

```
'   Hello World   '.trim();
  << 'Hello World"
```
.trim() will remove any whitespace from beginning or end of string

```
' \t\t JavaScript Ninja! \r'.trim();
  << 'JavaScript Ninja!"
```
... escaped tabs & carriage returns

```
'Hello'.repeat(2);
  <<'HelloHello'
```
repeat a string the stated # of times

Template Literals

- a special type of String that use backtick character, `, to deliminate the String

"They also allow interpolation of JavaScript code. This means that a JavaScript expression can be inserted inside a string and the result will be displayed"

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

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

- can also contain line breaks

`This is the start ...

... and this is the end`

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

Super-powered Strings

- template literals can be thought of as ←
- they behave the same as strings but can use interpolation

Symbols

- introduced as a new primitive value
- can be used to create unique variables
- only primitives that don't have a literal form
- only way to create them is use the `Symbol()` function

*recommended to add a description of the symbol inside the parentheses

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

- the typeOf operator should return a `typeof 'symbol'`

```
typeof uniqueID;
<< 'symbol'
```

- description acts as string representation of the symbol & is used to log symbols in the console

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

- manually access description using the `String()` function

```
String(uniqueID)
<< 'Symbol(this is a uniqueID)'
```

- it is possible for 2 variables to point @ same symbol if the `for()` method is used upon creation

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

"The main use-case for symbols is as object property keys, which will be covered in Chapter 5. The uniqueness of symbols, mean that it's impossible for the names of any properties to clash with each other if they are symbols"

Numbers

- can be integers (whole numbers)
- can be floating point numbers (decimals or floats)

```
<< typeof 42; // integer
<< 'number'

<< typeof 3.14159265; // floating point decimal
<< 'number'
```

JS doesn't distinguish between integers and floating point decimals

```
Number.isInteger(42);
<<< true
Number.isInteger(3.1415);
<<< false
```

.isInteger() checks if a number is an integer

Constructor Function for Numbers

- numbers also have constructor function

```
new Number(3)
<< [Number: 3]
```

- simply writing the number 3 is known as a number literal ← stick to these

Octal & Hexadecimal Numbers

- a number starts with a 0x, it's in hexadecimal (base 16) notation

```
0xAF // A represents 10, F represents 15
<< 175
```

- hex #'s are used for color codes
- If a # starts w/ a zero, followed by the letter o, it is considered to be in octal (base 8) notation

```
0o47; // 4 eights & 7 units
<<39
```

- If a # starts w/ a zero, followed by the letter b, it is considered to be in binary (base 2) notation

```
0b1010 // 1 eight, 0 fours, 1 two & 0 units
<<10
```

Exponential Notation

- shorthand for "multiply by 10 to the power of" (also called scientific notation)

```
1e6; // means 1 multiplied by 10 to
// the power 6 (a million)
<<1,000,000

2E3; // 2 multiplied by 10^3 (two thousand)
<<2000
```

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. Don't worry though, there are a few ways to deal with this, which we'll demonstrate with the toExponential() method; this returns the number as a string in exponential notation.

Use two dots `5..toExponential();`

put a space before the dot `5 .toExponential();`

always write integers as a decimal `5.0.toExponential();`

place the integer in parentheses `(5).toExponential();`

assign the number to a constant

```
const number = 5;
>> 5
number.toExponential();
```

all ↗ come to be: >> "5e+0"

toFixed() method rounds a # to a fixed # of decimal places

```
PI.toFixed(3);
<<"3.142"
```

value returned is string, not a #

toPrecision() method rounds a # to a fixed # of significant figures → also ↗

```
325678..toPrecision(2);
<< 3.3e+5"
2.459.toPrecision(2);
<< "2.5"
```

Arithmetic Operations

all the usual arithmetic operations can be used in JS.

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

% can calculate remainder of division

Changing the Value of Variables

Variable previously assigned numerical value can be increased by:

```
let points = 0;
<< 0
points = points + 10
<< 10
```

Shorthand ↑ uses += operator

```
points += 10
<< 20
```

equivalent compound assignment operators

```
points -= 5          // decreases points by 5
<< 15
points *= 2;         // doubles points
<< 30
points /= 3;         // divides value by 3
<< 10
points %= 7;         // changes the value of points to the remainder if its current value is divided by 7
<< 3
```

Incrementing Values

points ++ → increments by 1

“The main difference is the value that is returned by the operation. Both operations increase the value of the points variable by 1, but points++ will return the original value then increase it by 1, whereas ++points will increase the value by 1, then return the new value:

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:”
“points--; // returns 8, but has decreased points to 7
<< 8

--points; // decreases points to 6, then returns that value
<< 6”

Infinity

“Infinity is a special error value in JavaScript that is used to represent any number that is too big for the language to deal with. The biggest number that JavaScript can handle is 1.7976931348623157e+308:

1e308; // 1 with 308 zeroes!
<< 1e308

2e308; // too big!
<< Infinity

There is also a value -Infinity, which is used for negative numbers that go below -1.7976931348623157e+308:

-1e309;
<< -Infinity

The value of Infinity can also be obtained by dividing by zero:

1/0;
<< Infinity

The smallest number that JavaScript can deal with is 5e-324. Anything below this evaluates to either 5e-324 or zero:”

“5e-324; zero point (324 zeros) five
<< 5e-324

3e-325;
<< 5e-324

2e-325;
<< 0”

NaN

“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:

'hello' * 5;
<< NaN

The result returned by the typeof operator is rather ironic, however:

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

Checking a Value is a 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:

Number.isFinite(1/0);
<< false”

Type Coercian

when operands of an operator are different types

'2' * 8;
<< 16

- isn't always logical
- can cause confusion

'2' + 8;
<< 28

JS is Weakly Typed

You don't need to explicitly specify the data-type of a variable when declaring, BUT not doing so can lead to unexpected behaviors & hard to find bugs

Converting Between Strings and Numbers

Converting Strings to #'s

```
Number('23');
<< 23
```

tricks to convert string to # that use type coercion

```
const number = '5' * 1;
<< 5
typeof answer;
<< "answer"
```

or simply put a + operator in front

```
const answer = +'5';
<< 5
typeof answer;
<< 'number'
```

these methods are "hacky" & not recommended

Converting Numbers to Strings

preferred way is:

```
String(3);
<< '3'
```

Parsing Numbers

parseInt() ← converts string representation of numerical value back into a number.

```
parseInt('1010', 2); // Converts to binary, back to decimal
<< 10
```

If string starts w/ #, parseInt() will use the number & ignore any letters after

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

If you use parseInt() w/ a decimal, it removes anything after decimal point

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

*its NOT rounding, it just removes everything

```
parseInt('2.9', 10);
<< 2
```

parseFloat() converts strings to floating point decimal numbers

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

Undefined

- the value given to variables that haven't been assigned a value
- can also exist if object's property doesn't exist or a function has a missing parameter

NULL

- means no value
- null behaves like zero

```
10 + null;
<<10
10 + undefined;   // undefined is not a number
<<NaN
```

Booleans

- only two boolean values: true & false
- every value in JS has a boolean value & most of them are true (a.k.a. truthy values)

```
Boolean('hello');
<<true
Boolean(42);
<<true
Boolean(0);
<<false
```

- only 9 values are always false (aka falsy values)

* "" //double quoted empty string literal
* '' // single quoted empty string literal
* `` // empty string literal
* 0
* -0 //considered different to 0 by JS!
* NaN
* false
* null
* undefined

Logical Operators

! (Logical NOT)

- ! operator infront of a value converts it to Boolean & returns the opposite value
- truthy values return as falsy, vice versa, known as negation

```
!true // negating true returns false
<<false
!0; // 0 is falsy, so negating it returns true
<<true
```

- use double negation (!!) to find out if value is truthy or falsy → you're effectively negating the negation

`!! ' ';` <<false	`!! "hello";` <<true	`!! 3;` <<true	`!! NaN;` <<false
`!! "false";` <<true	`!! '0';` <<true		

&& (Logical AND)

- logical and operator works on 2 values (the operands)
  - only evaluates to true if ALL operands are truthy
  - value returned is last truthy value if all are true
  - first falsy value returned if at least one is falsy

```
'Shoes' && 18;  // both values are truthy
<< 18
'shoes' && 0;  // returns 0 because its falsy
<< 0
```

|| (Logical OR)

- only have to satisfy one rule
- works on 2 or more operands
- evaluates to true if any operands are true.
- only evaluates to false if both are falsy
- value returned is the first truthy value if any are true
- value returned is the last falsy is all are false

```
'shoes' || 0
<< 'shoes'
Nan || undefined;  // both NAN & undefined are falsy, so undefined is returned
<< undefined
```

Lazy Evaluation

- only check minimum # of criteria
- JS stops evaluating further operands once result is clear
- so logical AND stops reading as soon as it finds a false
- logical OR stops reading as soon as it finds a true

Bitwise Operators

- works w/ operands that are 32-bit integers

"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"

Bitwise NOT

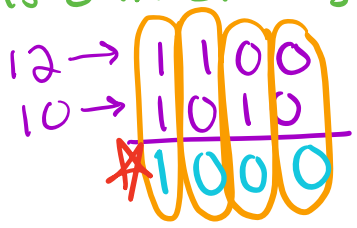
- NOT operator (~) converts # to 32-bit integer
- changes all 1s to 0s & all 0s to 1s, & returns new value

```
~2476;
<< -2477
```

Bitwise AND

- AND operator (&) converts both #s to binary & returns a # that in binary has a 1 in each position

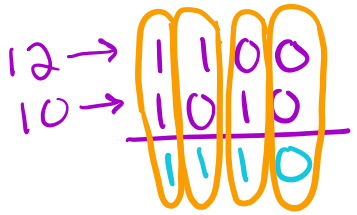
```
12 & 10; // in binary this is 1100 & 1010, so only the first digit is 1 in both cases. This returns 1000, which is 8 in binary
<< 8
```



Bitwise OR

- operator (|)
- converts to binary & compares, 1s stay 1s, 0s compared to 1s become 1s

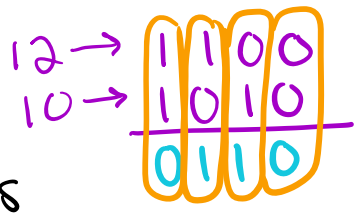
```
12 | 10;
<< 14
```



Bitwise XOR

- operator (^) stands for exclusive OR

```
12^10;
<<6
```



- don't use ^ for exponents

```
1^0;
<<1
```

```
true ^ true;
<<0
```

Bitwise Shift Operators

- operators << and >> moves binary representation a given number of places to right or left, which effectively multiplys or divides by power of 2

```
3 << 1; //multiply by 2
<<6
```

```
16 >>1; //divide by 2
<<8
```

```
5 << 3; // multiply by 2 cubed (8)
<<40
```

Comparison

Equality

- don't use (=) for comparing, this assigns
- use either a double equals operator, ==, known as "soft equality" or triple equals operator, ===, known as "hard equality"

Soft Equality

```
answer == 5;
<<true

answer == "5";
<<true
```

problem ↗

JS doesn't take data type into account

Hard Equality

- returns true ONLY if they are of the same data type

```
answer === 5;
<<true

answer === '5';
<<false

null === undefined;
<<false
```

} all correct

- one quirk:

```
NaN === NaN;
<<false
```

← only value not equal to itself

↖

get around this) with:

```
Number.isNaN(NaN);
<< true
```

JS Ninja should always use hard equality

Inequality

- soft inequality operator, !=
- hard inequality operator, !==

JS Ninja should always use hard inequality

Greater Than & Less Than

- greater than operator, >
- less than operator, <
- greater than or equal to, >=
- less than or equal to, <=
- no "hard" greater/less than so avoid errors by using combination of greater than operator, logical OR, & hard equality

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

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

- when these operators are used with strings, it checks alphabetical order

```
'apples' < 'bananas';
  >> true
```

*caution! results are case sensitive & upperCase is considered "less than" lowerCase

```
'apples' < 'Bananas';
  >> false
```

Quiz Ninja Project pg 106

Chapter Summary

“Comments are ignored by the program, but make your program”
“programs.
There are six primitive data types: strings, symbols, numbers, Booleans, undefined and null.
Non-primitive data types, such as arrays, functions and objects, all have a type of 'object'.
Variables point to values stored in memory and are declared using the const or let keywords.
Values are assigned to variables using the = operator.
Strings and numbers have various properties and methods that provide information about them.
Symbols are unique, immutable values.
Boolean values are either true or false.
There are only seven values that are false in JavaScript and these are known as 'falsy' values.
Data types can be converted into other data types.
Type coercion is when JavaScript tries to convert a value into another data type in order to perform an operation.
Logical operators can be used to check if compound statements are true or false.
Values can be compared to see if they are equal, greater than or less than other values.”

Chapter 3: Arrays, Logic, and Loops

Arrays

- an ordered list of values

```
const myArray = [];   <-- array literal
<< []
```

Initializing An Array

```
const heroes = [];
```

Find the value of element 0 in array:

```
heroes[0]
<< undefined
```

Adding Values to Arrays

```
heroes[0] = 'Superman';
heroes[0] = 'Batman';   <-- reassigning
heroes[1] = 'Wonder Woman';
heroes[2] = 'Flash';
heroes[5] = 'Aquaman';

heroes;
<< ['Batman', 'Wonder Woman', 'Flash',
    undefined, undefined, 'Aquaman']
```

Creating Array Literals

```
const avengers = ['Captain America',
'Iron Man', 'Thor', 'Hulk'];
>>['Captain America', 'Iron Man', 'Thor', 'Hulk']
```

Removing Values from Arrays

- delete operator will remove items from arrays

```
delete avengers[3];
<< true
```

NOW ↴:

```
avengers;
>>['Captain America', 'Iron Man', 'Thor', undefined]
```

Destructuring Arrays

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

- allows us to assign multiple values out of an array & present them as individual values

```
const [x, y] = [1, 2];
x
<< 1
y
<< 2
```

```
[x, y] = [y, x];
x
<< 2
y
<< 1
```

- also allows swapping the value of 2 variables over

Array Properties & Methods

```
const avengers = ['Captain America', 'Iron Man', 'Thor', 'Hulk', 'Hawkeye', 'Black Widow'];
avengers.length;
  << 6
avengers[avengers.length - 1];
  << 'Black Widow'
```

] find last item in the array

```
avengers.length = 8;
  << 8
avengers;
```

] length property is mutable; lengthen

```
['Captain America', 'Iron Man', 'Thor', 'Hulk', 'Hawkeye', 'Black Widow', undefined, undefined]
```

```
avengers.length = 3;
  << 3
avengers;
```

] shorten

```
['Captain America', 'Iron Man', 'Thor']
```

Pop, Push, Shift, & Unshift

- remove last item from array: pop()
- removes first item from array: shift()
- appends new value to end of array: push()
- appends new value to beginning of array: unshift()

Merging Arrays

- concat() can be used to merge arrays
- doesn't change arrays, just combines them
- spread operator (...) spreads out elements of that array

The join() Method

- join() used to turn the array into a string that comprises all the items in the array separated by commas.

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

- you can choose separator other than comma by placing it inside parentheses

```
avengers.join(&);
<< 'Captain America & Iron Man & ...'
```

Slicing & Splicing

- slice() creates a subarray, chopping out a slice of an original array

```
avengers.slice(2, 4) //starts @ 3rd item (index of 2)
                     and finishes at the 4th
  << ['Thor', 'Hulk']
```

- non-destructive - no items removed from array
- splice() removes items from array & inserts new items in their place

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

- destructive operation → changes the value of the array

```
avengers
<< ['Captain America', 'Iron Man', 'Thor',
'Scarlet Witch', 'Hawkeye', 'Black Widow']
```

- splice can also be used to insert values into an array @ specific index w/out removing items

```
avengers.splice(4, 0, 'Quicksilver');
<< []
```

empty because nothing was removed

- splice can also delete any index

```
avengers.splice(2, 1);
<< ['Thor']
```

Reverse

- reverse the order of array using `reverse()`

Sort

- sort order of array using `sort()`
- sorts alphabetically

Finding if a value is in an Array

- find particular value with `indexOf()`

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

- if item is not in array, it'll return -1
- find if value is just <u>in</u> array with `includes()`

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

- add extra parameter to indicate which index to start the search from:

avengers.includes('Captain America', 1);
<< false
// start search from second element in array

Multidimensional Arrays

- an array of 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”

“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

- data structure that represents a collection of unique values → can't include duplicate values

creating sets

- empty set is created with `new` operator and `Set()` constructor

```
const list = new Set();
```

Adding values to sets

- values can be placed in sets using `add` method

```
list.add(1);
<< Set {1}
```

- multi-adds

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

- Strings in sets then each character is added as a separate element

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

- to add separate words use `add()` method

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

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

Set Methods

- # of values in a set can be found w/ size()
- to check if a value is in a set : has()
  - this returns a bool value : true or false
- sets do NOT have index notation

```
jla[0]
<< undefined
```

Removing Values From Sets

- remove a value with delete()
- returns boolean value w/ true if it was removed, or false if it wasn't in the set
- clear() can remove ALL values from set

Converting Sets to Arrays

- by placing the set & spread operator directly inside array literal

"To demonstrate this, first we'll create a set of three items:

```
const shoppingSet = new Set().add('Apples').add('Bananas').add('Beans');

shoppingSet
<< Set { 'Apples', 'Bananas', 'Beans' }
```

Then we convert it into an array:

```
const shoppingArray = [...shoppingSet]

shoppingArray
<< [ 'Apples', 'Bananas', 'Beans' ]
```

It's also possible to use the Array.from() method to convert a set into an array. The following code would achieve the same result as using the spread operator above:

```
const shoppingSet = new Set().add('Apples').add('Bananas').add('Beans');

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 pg 133

"When objects are added to sets, they will be stored there as long as the set exists, even if the original reference to the object is removed. The technical term for this is the object is prevented from being garbage-collected, which can cause a memory leak"

Memory Leaks

A memory leak occurs when a program retains references to values that can no longer be accessed in its memory. This means that memory is being used to store values that are no longer required by the program, effectively wasting system resources.

Memory leaks can cause problems by gradually reducing the overall memory available, which can cause the program, or even the entire system, to run more slowly.

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 references to a 'dead object' thats had the original reference removed
- to create a weak set:

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

- still have has(), add(), & delete()

Maps pg 136

- convenient way of keeping a list of key & value pairs

Objects are limited to using strings for key values, whereas maps can use any data type as a key.
There is no efficient way to find the number of key-value pairs an object has, whereas this is easy to do with maps using the size property.
Objects have methods that can be called (see Chapter 5) and prototypes that can be used to create a chain of inheritance (see Chapter 12), whereas maps are solely focused on the storage and retrieval of key-value pairs.
The value of an object's properties can be accessed directly, whereas maps restrict you to using the get() method to retrieve any values

Creating Maps

```
const romanNumerals = new Map();
```

Adding Entries to Maps

- set() method can add key & value pair

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

return value shows the mapping with the key connected to the value using the 'hash rocket' symbol (=>)

Map Methods

- to look up a value, use the get() method:

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

- to check if a particular key is in map:

```
romanNumerals.has(5);
  << true
romanNumerals.has(10);
  << false
```

- map with multiple values can be created by nested array w/ parameters

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

heroes.size
<< 2
```

Removing Entries from Maps

- delete() can be used → returns bool value true if item was removed, false if it wasn't in the map
- clear() method removes all key & 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:

```
[...romanNumerals]
<< [ [ 1, 'I' ], [ 2, 'II' ], [ 3, 'III' ], [ 4, 'IV' ], [ 5, 'V' ] ]
```

... or the Array.from() method:

```
Array.from(romanNumerals)
<< [ [ 1, 'I' ], [ 2, 'II' ], [ 3, 'III' ], [ 4, 'IV' ], [ 5, 'V'] ]
```

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.

To create a weak map, the new operator is used along with the WeakMap() constructor:

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

Weak maps can use the has(), get(), set() and delete() methods in the same way as a regular map.

Weak maps and sets are useful for optimizing memory usage and avoiding memory leaks, but they’re also limited in that they don't have access to all the methods their regular counterparts have. 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 Statements

```
if (condition) {
  //code to run if condition === true
}
```

else can be used to add alternative

Ternary Operator

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

Switch Statements

- String lots of if & else statements together to make a logical decision tree

Loops

While Loops

- repeated run block while certain condition is true

Infinite Loops

- conditions have to be met at some point or loop will never end = infinite!

do ... while Loops

- similar to while, but condition comes after the block of code
- so block of code will always run at least once

For Loops

"The initialization code is run before the loop starts and is usually employed to initialize any variables used in the loop. The condition has to be satisfied for the loop to continue. The after code is what to do after each iteration of the loop, and it is typically used to increment a counter of some sort"

Nested for Loops

"You can place a loop inside another loop to create a nested loop. It will have an inner loop that will run all the way through before the next step of the outer loop occurs."

Looping over Arrays

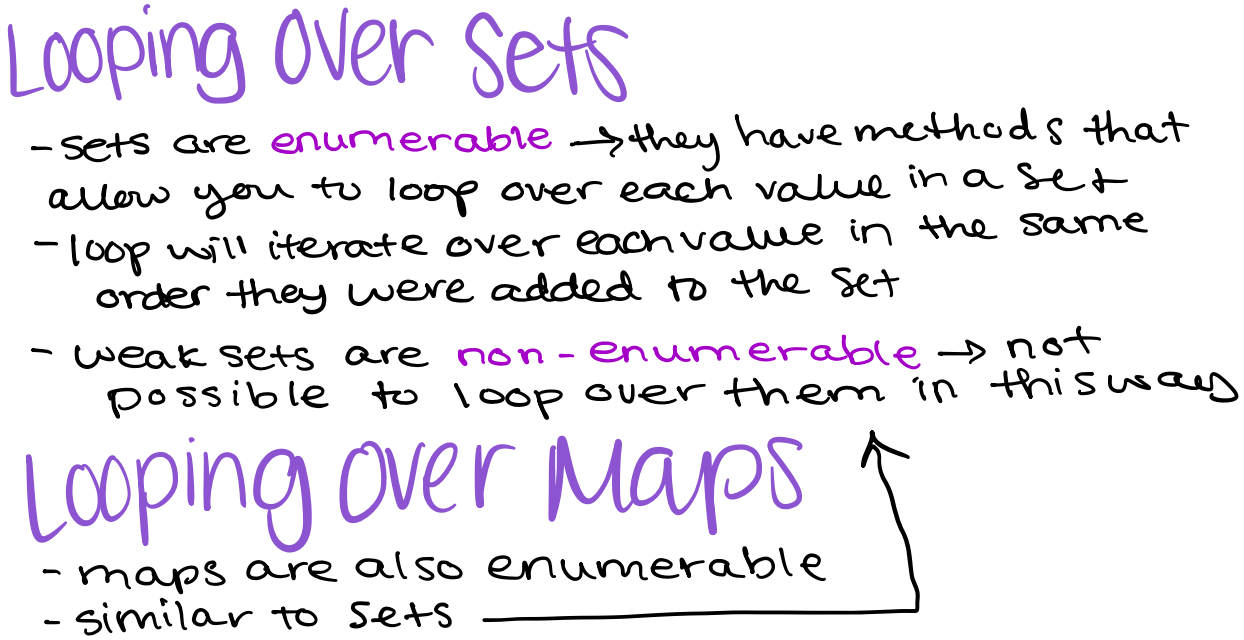
- for loop can be used to iterate over each value in an array

"Array indices start their numbering at zero, so make sure the value of the initial counter in the for loop also starts at zero. 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."

- ES6 introduced a for-of loop that implements slightly different syntax

“for(const value of avengers){
console.log(value);
}
<< 'Black Widow'
<< 'Captain America'
<< 'Hawkeye'
<< 'Iron Man'
<< 'Quicksilver'
<< 'Scarlet Witch”

“This replaces all of the setup of a ‘for’ loop with a variable (value in the example above) that represents the value of each element in the array. Note that this variable needs to be declared using const”



“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);
}
<< I
II
III
IV
V”
```

“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}`);
}
<< 1 in Roman numerals is I
2 in Roman numerals is II
3 in Roman numerals is III
4 in Roman numerals is IV
5 in Roman numerals is V”
```

- weak maps are non-enumerable → not possible to loop over them in this way

Quiz Ninja Project pg 154

Chapter Summary

Arrays are an ordered list of values
Multidimensional arrays are arrays that contain other arrays
Arrays have lots of methods that can be used to manipulate items in the array
Sets are new in ES6 and are ordered lists of non-duplicate values
Maps are new in ES6 and are ordered lists of key-value pairs
We can use an if and else statement to control the flow of code
The switch statement can be used instead of multiple if and else statements
A while loop and do ... while loop can be used to repeat a block of code while a condition is still true
A for loop works in a similar way to a while loop, but has a different syntax
A for-of loop can be used to iterate over an array
Sets and maps are enumerable, so can also be looped over using a for-of loop”

Chapter 4: Functions

“A function is a chunk of code that can be referenced by a name, and is almost like a small, self-contained mini program. Functions can help reduce repetition and make code easier to follow.”

“In JavaScript, functions are considered to be first-class objects. This means they behave in the same way as all the other primitive data types and objects in the language. They can be be assigned to variables, stored in arrays and can even be returned by another functions.”

Defining A Function

Function Declaration

```
function hello() {
console.log('Hello World!');
}
```

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!');
};”
```

Every Function Has a Name

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!");');
```

It's not recommended to declare functions in this way as there are numerous problems associated with placing the function body inside a string”

-ninja programmer should always declare functions using function literals, function declarations, or function expressions

Invoking a Function

to run the code inside the function body

```
hello();
<< 'Hello World!'
```

Return Values

all functions return a value

```
function howdy(){
  return 'Howdy World!';
}

const message = howdy();
<< 'HowdyWorld!'
```

Parameters & Arguments

"Parameters and arguments are terms that are often used interchangeably to represent values provided for the function as an input. There is a subtle difference though: any parameters a function needs are set when the function is defined. When a function is invoked, it is provided with arguments."

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;
}"
```

“Now let's invoke the arguments() function a couple of times to see the results:

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

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

- arguments is not an array → so no array methods like slice(), join(), or forEach()
  - 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”

Default Parameters

- to specify a default parameter, assign the default value to it in the function definition
- should always come after non-default parameters, otherwise default values will always have to be entered anyway

Arrow Functions

“Arrow functions can be identified by the 'arrow' symbol, => that gives them their name. The parameters come before the arrow and the main body of the function comes after. Arrow functions are always anonymous, so if you want to refer to them, you must assign them to a variable”

- advantages of arrow functions:

“They are much less verbose than normal function declarations.
Single parameters don't need putting into parentheses.
The body of the function doesn't need placing inside a block if it's only one line.
The return keyword isn't required if the return statement is the only statement in the body of the function.
They don't bind their own value of this to the function (we'll see why this is a particularly useful property when we cover objects later in the book).”

Function Hoisting

- action of moving all variable & function declarations to the top of the current scope
- all function definitions can be placed together

Variable Hoisting

- Variable declarations that use var are automatically moved to the top of the current scope.

Callbacks

- a function that is passed as an argument to another is known as a callback

"function sing(song) {
console.log(`I'm singing along to ${song}`);
}

sing('Let It Go')
<< 'I'm singing along to Let It Go"

"We can make the sing() function more flexible by adding a callback parameter:

function sing(song,callback) {
console.log(`I'm singing along to ${song}.`);
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"

Sorting Arrays with a Callback

"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() to loop through the array & invoke a callback function using each value as an argument
- map() → similar to ↑ = iterates over an array, takes a callback function as parameter that is invoked on each item in the array
  - the difference is it returns a new array that replaces each value w/ the return value of the callback function
- Reduce()

"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 callback function is used to describe how to combine each value of the array with the running total."

- Filter() → returns a new array that only contains items from the original array that return true when passed to the callback
  → provides useful way to find all truthy values from an array

-Chaining Iterators Together

"The various iterator functions can be used in combination to create some powerful transformations of data stored in arrays. This is achieved by a process called chaining methods together"

- iterator functions return an array, which lets another iterator be chained on the end & it'll be applied to new array

-improving the mean() function

"function mean(array) {
const total = array.reduce((a, b) => a + b);
return total/array.length;
}"

"Here is the code for the improved function that accepts a callback:

function mean(array,callback) {"
"if (callback) {
  array.map(callback);
  }
const total = array.reduce((a, b) => a + b);

```
return total/array.length;
}
```

Quiz Ninja Project

pg 192

Chapter Summary

“Functions are first-class objects that behave the same way as other values.

Function literals can be defined using the function declaration, or by creating a function expression by assigning an anonymous function to a variable.

All functions return a value. If this is not explicitly stated, the function will return undefined.

A parameter is a value that is written in the parentheses of a function declaration and can be used like a variable inside the function’s body.

An argument is a value that is provided to a function when it is invoked.

The arguments variable is an array-like object that allows access to each argument provided to the function using index notation.

The rest operator can be used to access multiple arguments as an array.

Default arguments can be supplied to a function by assigning them to the parameters.”

“Arrow functions are a new shorthand notation that can used for writing anonymous functions in ES6.

Function declarations can be invoked before they are defined because they are hoisted to the top of the scope, but function expressions cannot be invoked until after they are defined.

A callback is a function that is provided as an argument to another function”

Questions:

What is an instance you would want to use soft equality instead of hard equality?

Why not do away with soft equality? Not even teach it?