Institute of Data

Software Engineering

Module 3 Intermediate JavaScript

Intermediate JavaScript

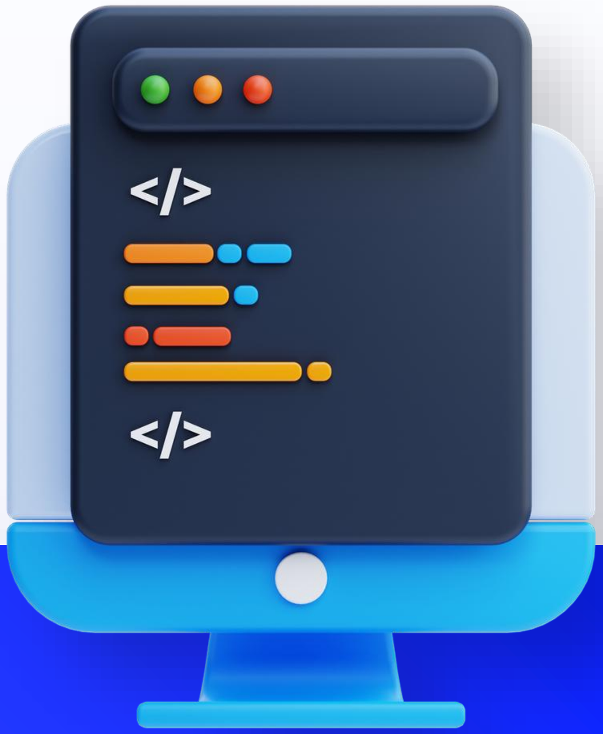
- Primitives
- Numbers
- Strings
- Arrays
- Iterables
- Map and Set
- Destructuring assignment
- Efficient algorithms
- Date and time
- JSON

Primitive data types

To recap, a **primitive** data type is a variable that stores a **single** value.

Examples include **numbers** (with and without decimals), **strings** and **booleans**. They take up very little space in memory and are efficient and useful for storing data in many simple cases.

Primitives are fundamental building blocks in programming, and can be used to compose more **complex** data types such as **arrays** and **objects**.



Methods of primitive

Primitives as objects

Number, **boolean** and **string** primitives can still use certain special **functions** (built into their **prototypes**) in the same way that objects can:

```
const n = 1.23456; // primitive floating point number
console.log(n.toFixed(2)) // 1.23 - fixed to 2 decimal places
console.log(n.toPrecision(10)) // 1.234560000 - fills or rounds to the exact number of digits

const hello = 'hello world' // primitive string
console.log(hello.toUpperCase()) // HELLO WORLD
console.log(hello.endsWith('world')) // true
```

The MDN has a full list of Number methods and String methods that can be used on these primitive variable data types. We will look at these later.

Primitive-like Object

As well as treating **primitives like objects** by using their prototype functions, we can also treat **objects like primitives** by defining special methods.

String-like Object

Since objects are so generic, treating them as strings usually results in the generic **[object Object]** message. We can override this by creating a custom **toString** method to specify the string format of our object:

```
const user = {
    name: 'John';
}

console.log("User: " + user);
// User: [object Object]
```

```
const user = {
    name: 'John',
    toString() {
        return this.name; // custom string value
    }
}

console.log("User: " + user); // User: John
```

Primitive-like Object

Number-like Object

We can also customise how an object behaves like a **number** by defining a valueOf method.

```
const apple = {
    name: 'Apple',
    category: 'Granny Smith',
    price: 1.2,
    valueOf() { // without this special function, we can’t multiply the object below
        return this.price;
    }
}

console.log(apple * 2) // 2.4
```

Numbers

Decimal Numbers

Variations on syntax for representing

very large numbers

```
const billion1 = 1000000000 // 9 zeros - hard to read
const billion2 = 1_000_000_000 // easier to read,
underscores ignored
const billion3 = 1e9 // exponential format

console.log(billion1 === billion2) // true
console.log(billion2 === billion3) // true
```

Variations on syntax for representing

very small numbers

```
const microSecs1 = 0.000001 // 6 decimal places - hard
to read
const microSecs2 = 0.000_001 // easier to read,
underscores ignored
const microSecs3 = 1.e-6 // exponential format

console.log(microSecs1 === microSecs2) // true
console.log(microSecs2 === microSecs3) // true
```

Numbers

Hexadecimal Numbers

Hexadecimal numbers use **base 16** (0-9 and A-F) instead of our standard **base 10** (0-9) for decimals. They are widely used in JavaScript to represent **colours**, **encode characters**, and for many other things. So naturally, there exists a shorter way to write them: 0x and then the hexadecimal number.

```
const r = 0xff;
const g = 0xff;
const b = 0xff;

const white = `rgb(${r}, ${g}, ${b})`
console.log(white) // rgb(255, 255, 255)
```

Numbers

Binary and Octal Numbers

Binary (**base 2** : 0 and 1) and octal (**base 8** : 0-7) numeral systems are rarely used, but are supported using the `0b` and `0o` prefixes.

Any number in JS beginning with a 0 is also interpreted as octal, which can lead to issues if treating phone numbers as **numbers** instead of **strings**:

```
const mobile = 041234567
console.log(mobile) // 8730999 - decimal equivalent

const binary = 0b11111111 // binary form of 255
const octal = 0o377 // octal form of 255

console.log(binary) // 255
console.log(binary === octal) // true
```

Numbers

Base conversion

toString(base) on numbers returns a **string** representation in the **number** system with the given **base**. The default base is **10** but it can range from **from 2 to 36**.

base=16 is used for hex colours, character encodings etc. There are 16 possible digits - **0..9 and A..F**.

base=2 is mostly for debugging bitwise operations, with 2 possible digits - **0 and 1**.

base=36 is the maximum, with 36 possible digits - **0..9 and A..Z**.

```
const ff = 255
const ee = 238
const dd = 221
console.log(ff.toString(16)) // ff

//convert from rgb colour code to hexadecimal
console.log(`#${ff.toString(16)}${ee.toString(16)}${dd.t
oString(16)}`) // #ffeedd
```

Numbers

Imprecise calculations

Internally, numbers are represented as **double precision floating point numbers**, following the international IEEE 754 standard. This uses exactly **64 bits of memory** to store a **number**: **52** for storing the **number** (between 0 and 1), **11** to store the **exponent** (power of 2 to multiply the **number**) and **1** bit is for the **sign**.

- When the number is too big, it **overflows the 64-bit** storage and cannot be accurately represented, so it converts to **Infinity**:

```
const toobig = 1e350 // 1 * 10^350 - overflows storage

console.log(toobig) // Infinity
console.log(Number.MAX_VALUE) // 1.7976931348623157e+308
```

Numbers

Imprecise calculations

- Because floating point numbers **can’t be accurately represented in binary**, (https://floating-point-gui.de/) some operations lead to **unexpected results**:

```
const point1 = 0.1; const point2 = 0.2;
console.log(`${point1} + ${point2} = ${point1 + point2}`) // 0.1 + 0.2 = 0.30000000000000004
```

- Precision is also lost when the number of digits reaches **16 or more**, or for values greater than 253. In JS, integer numbers are accurate up to 15 digits.

```
// Numeric literals with absolute values equal to 2^53 or greater are too large to be represented
accurately as integers.
console.log(9_999_999_999_999_999) // 16 digits, prints as 10000000000000000
console.log(Number.MAX_SAFE_INTEGER) // 9_007_199_254_740_991
```

Numbers

Tests: isFinite and isNaN

- **isNaN(value)** converts its argument to a number and then tests it for being NaN (**not a number**). We need to use this because **NaN != NaN**:

```
console.log(isNaN(NaN)) // true
console.log(NaN == NaN) // false
```

- **isFinite(value)** converts its argument to a number and returns true if it is a **regular number**, not **NaN**/**Infinity**/-**Infinity**

```
console.log(isFinite(1/3)) // true, regular number
console.log(isFinite("string")) // false, because converts to NaN
console.log(isFinite(Infinity)) // false, because represents infinite number
```

Numbers

parseInt and parseFloat (soft conversion)

- Numeric conversion using a unary plus + or Number() is strict. If a value cannot be interpreted exactly as a number, it fails and results in **NaN**:

```
console.log( +"100px" ) // NaN - 100px is not a valid number
```

- parseInt and parseFloat 'read' numbers from a string until they can't. Instead of an error, the **total number is returned**. The function parseInt returns an integer, whilst parseFloat will return a **floating-point (decimal) number**.

```
console.log( parseInt("150px") ) // 150 - stops at 'px'
console.log( parseFloat("2.5em") ) // 2.5 - stops at 'em'
console.log( parseFloat("12.34.56") ) // 12.34 - stops at second invalid decimal
console.log( parseInt("a123") ) // NaN - can't parse the 'a' so stops
```

Strings

Definition

Textual data is stored as **strings**. The internal format for strings is always the UTF-16 character encoding system, which uses 16 bit values.

Special characters

A backslash precedes a special character in an '**escape sequence**'. We can create multiline strings by using a **newline character**, written as `\n` , which denotes a **line break**. Use `\t` for a tab space:

```
const guestList = `Guests: \n\t- John \n\t- Pete \n\t- Mary`
console.log(guestList)
```

```
// Guests:
//          - John
//          - Pete
//          - Mary
```

Strings

Other special characters

Character	Description
\r	Carriage return: \r\n to represent a line break in windows
\’, \”	Use a single or double quote inside a single/double quoted string
\\	Backslash
\xXX	Unicode **character** with the given hexadecimal Unicode XX. e.g. \x7A is the same as z.
\uXXXX	A Unicode **symbol** with the hex code XXXX in **UTF-16** encoding. e.g. \u00A9 is a Unicode for the copyright symbol ©. It must be exactly 4 hex digits.
\u{X…XXXXXX} (1 to 6 hex characters)	A Unicode symbol with the given UTF-32 encoding. Some rare characters are encoded with **two Unicode symbols**, taking **4 bytes**. This way we can insert long codes, e.g. \u{1F60D} is a smiling face symbol 😍

Strings

Comparing strings

All strings are encoded using **UTF-16**. That means each **character** has a corresponding **numeric code**. When two strings are compared using < or >, under the hood, JavaScript **converts each character into a numeric code** and compares them mathematically.

- str.codePointAt(pos) returns the **numeric code** for the char at position pos.

```
console.log( 'Z'.codePointAt(0) ) // 90
```

- String.fromCodePoint(code) creates a character by its **numeric code**.

```
console.log( String.fromCodePoint(90) ) // Z
```

Strings

Manipulating strings

JS has a wide variety of built-in functions for **manipulating strings**. Some of the most-used operations are to check their **length**, to concatenate them using **+**, checking for the existence or location of **substrings** with the **indexOf()** method, changing **case**, or extracting substrings with the **substring()** method.

```
const string = 'I could be anything you like'

console.log(string.length) // 28 - string is 28 characters long
console.log(string.indexOf('anything')) // 11 - starting at 0 for 'I'
console.log(string.substring(20)) // 'you like' - substring starting at position 20
console.log(string.toUpperCase()) // I COULD BE ANYTHING YOU LIKE
console.log(string + ' plus more') // I could be anything you like plus more
```

Strings

Manipulating strings

Other useful string functions include **startsWith()** and **endsWith()**, **split()**, **slice()**, **replace()** and **replaceAll()** and **trim()**.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

console.log(sentence.startsWith('The')) // true - case sensitive
console.log(sentence.endsWith('dog')) // false - needs the full stop
console.log(sentence.split(' ')) // splits into multiple strings using the given separator
// ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']
console.log(sentence.slice(4, 10)) // quick - gets the section between chars 4 and 10
console.log(sentence.replace('quick', 'slow')) // The slow brown fox jumps over the lazy dog.
console.log("  extra     spaces  ".trim()) // extra     spaces - trims whitespace from start and end
```

Try it! Create your own string and experiment with several string manipulation functions.

Arrays

Definition

Arrays are a data structure to store an **ordered collection**. Arrays are often used in JavaScript to store and manipulate **lists of related items**. They can contain numbers, strings, objects and even other nested arrays.

See the MDN reference on arrays for complete and up-to-date details.

Declaration - To create an array:

```
const arr1 = new Array(1,2,3); // constructor, not often used
const arr2 = [1, 2, 3]; // array literal, much more common

console.log(arr1); // [ 1, 2, 3 ] - both do the same
console.log(arr2); // [ 1, 2, 3 ] - both do the same
```

Arrays

Arrays can be used to implement two common data structures in programming:

Queue data structure, FIFO (First-In-First-Out)

- push appends an element to the **end**.
- shift gets an element from the **beginning**, advancing the queue, so that the second element becomes the first.

Stack data structure, LIFO (Last-In-First-Out)

- push adds an element to the **end** (or the top).
- pop takes an element from the **end**.

Arrays

Methods that work with the end of the array:

- pop extracts the **last** element of the array and returns it:

```
const fruits = ['Apple', 'Orange', 'Pear']
const lastFruit = fruits.pop() // removes and returns the last item
console.log(lastFruit); // Pear
console.log(fruits); // [ 'Apple', 'Orange' ]
```

- push appends an element to the **end** of the array:

```
fruits.push('Banana') // adds to the end of the array
console.log(fruits); // [ 'Apple', 'Orange', 'Banana' ]
```

Arrays

Methods that work with the beginning of the array:

- shift extracts the **first** element of the array and returns it:

```
const fruits = ['Apple', 'Orange', 'Pear']
const firstFruit = fruits.shift() // removes and returns the first item
console.log(firstFruit) // Apple
console.log(fruits) // [ 'Orange', 'Pear' ]
```

- unshift adds the element to the **beginning** of the array:

```
fruits.unshift('Banana') // adds to the beginning of the array
console.log(fruits) // [ 'Banana', 'Orange', 'Pear' ]
```

Arrays

Internals

Arrays are not **primitives**, but special types of objects **stored by reference**. They are **resizable** and can contain a mix of data types. They are **indexed numerically** beginning at 0, using **square brackets**.

```
const fruits1 = ['Apple', 'Orange', 'Pear']
const fruits2 = fruits1 // refers to same memory location

fruits1.push('Banana') // add new item to the end of fruits1
console.log(fruits2) // [ 'Apple', 'Orange', 'Pear', 'Banana' ]
// fruits2 reflects the same change made to fruits1 since they both reference the same memory location

console.log('fruit at index 0: ' + fruits1[0]) // Apple - accessed using numeric index 0
console.log('fruit at index 1: ' + fruits1[1]) // Orange - accessed using numeric index 1
```

Arrays

Multidimensional arrays

Arrays can have items that are also arrays. We can use this ability for **multidimensional** arrays, for example to **store matrices**.

```
const matrix = [ // 3x3 matrix
    [1, 2, 3], // row 0
    [4, 5, 6], // row 1
    [7, 8, 9]  // row 2
]

console.log(matrix[1][1]) // 5, the value in row 1, column 1
```

Use double square brackets to access array elements in nested/internal arrays, starting with the outermost array (row).

Arrays

toString method

Arrays have their own internal implementation of the toString method that returns a **comma-separated** list of elements.

```
const numbers = [1, 2, 3]
const strings = ["one", "two", "three"]
const empty = []

console.log('Numbers: ' + numbers) // Numbers: 1,2,3
console.log('Strings: ' + strings) // Strings: one,two,three
console.log('Empty: ' + empty) // Empty:
```

Arrays do not have an internal valueOf, they implement **only** toString conversion, so empty array [] becomes an **empty string**.

Arrays

Arrays are a commonly used data type, and have many very useful built-in methods. Knowing how to use these effectively will help you enormously as a programmer. Make good use of the official MDN Array Reference and practice these as you go:

slice	get a segment of an array
map	alter each element in the array
splice	insert/remove items in an array
forEach	access each element in turn
sort	sort the elements by a condition
concat	combine arrays into a larger array
find	get the first element matching condition/s
filter	get multiple elements matching condition/s
reduce	calculate a single value from all elements
reverse	reorder array elements in opposite order
indexOf	get the index of a given array element
lastIndexOf	get the index of a given array element
join	create a string by joining elements together with a separator character

*The **teal** methods will change (or mutate) the original array, while the **blue** methods will return a new independent value, which is usually preferred.*

Arrays

slice method

- Syntax

```
arr.sliced([end], [end])
```

- Slice returns a **new array** copying to it all items from index start to end (**not including end**). **Negative** indexes work backwards from **array end**.

```
const sliceArray = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]
const sliced = sliceArray.slice(0, 3) // start at the beginning, get items up to index 3
const endSliced = sliceArray.slice(-3) // start at the end, get last 3 items

console.log(sliced) // [ 'red', 'orange', 'yellow' ]
console.log(endSliced) // [ 'blue', 'indigo', 'violet' ]
console.log(sliceArray) // ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

Arrays

splice method

- Syntax

```
arr.splice(start[, deleteCount, elem1,  … , elemN])
```

- Splice can **insert**, **remove and replace** items in an array. It modifies the original array, **starting** from the **index** start, **removes** deleteCount items and then **inserts** elem1, …, elemN at their place. It returns the removed elements:

```
const spliceArray = ["I", "study", "JavaScript", "right", "now"]
const removed = spliceArray.splice(0, 3, "Let's", "dance")

console.log(removed) // [ 'I', 'study', 'JavaScript' ] - 3 elements starting at index 0 are removed
console.log(spliceArray) // [ "Let's", 'dance', 'right', 'now' ] - 2 new elements are inserted
```

Arrays

concat method

- Syntax

```
arr.concat(arg1, arg2…)
```

- Concat creates a **new array** including values from other arrays and additional items. It accepts any number of arguments: either **arrays** or **values**.

```
const array1 = [1,2,3]
const array2 = [4,5,6]
const array3 = [7,8,9]

const combined = array1.concat(array2, array3)
console.log(combined) // [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
console.log(combined.concat(10, 11)) // [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
```

Arrays

indexOf method

- Syntax

```
arr.indexOf(item, from)
```

- It looks for item **starting** from index from (or 0 if omitted) and returns the **index** where it was found, otherwise -1.

```
const marks = ['A+', 'C+', 'B-', 'D', 'B+', 'C+', 'B-']

let bIndex = marks.indexOf('B-')
let eIndex = marks.indexOf('E')

// first index is always 0
console.log(marks[bIndex] + ' is at index: ' + bIndex) // B- is at index: 2
console.log(eIndex) // -1, since not found
```

Arrays

lastIndexOf method

- Syntax

```
arr.lastIndexOf(item, from)
```

- The same as indexOf, but it looks from right to left until a match is found.

```
const marks = ['A+', 'C+', 'B-', 'D', 'B+', 'C+', 'B-']

let bIndexFirst = marks.indexOf('B-')
let bIndexLast = marks.lastIndexOf('B-')

console.log(marks[bIndexFirst] + ' is first at: ' + bIndexFirst) // B- is first at: 2
console.log(marks[bIndexLast] + ' is last at: ' + bIndexLast) // B- is last at: 6
```

Arrays

join method

- Syntax

```
arr.join(separator)
```

- It returns a string of arr items joined by separator between them. When separator is omitted, the arr elements are separated with a **comma** (,)

```
const elements = ['Wind', 'Water', 'Fire', 'Air']

console.log( elements.join() ) // Wind,Water,Fire,Air
console.log( elements.join(' ') ) // Wind Water Fire Air
console.log( elements.join('; ') ) // Wind; Water; Fire; Air
```

Arrays

forEach method

- Syntax

```
arr.forEach(function(item, index, array) {
   // . . . Do something with item
});
```

- forEach runs a **function** for **every element** of the array. This function can take 3 parameters (named as needed): the array **item**, the array **index**, and the **array** itself. We always use the **item**, sometimes also the **index**, rarely the **array**.

```
const hobbits = ['Bilbo', 'Frodo', 'Samwise', 'Merry', 'Pippin']
hobbits.forEach((hobbit, index) => { // usually we use an arrow function here
    console.log(`#${index}: ${hobbit}`) // runs once for every item in array
})
```

Arrays

find method

- Syntax

```
const result = arr.find(function(item, index, array) {
   // . . . If true is returned, item is returned and iteration is stopped, for
falsy scenario returns indefined
});
```

- The function passed to **find** is called for each array element. Its arguments can be renamed. item is the element, index is its index (optional), array is the **array** itself (optional). The function evaluates a given condition - when true, the matched item is returned.

```
const products = [
    { id: 1, title: 'Sleeveless Tee', price: 23.95, category: 'Shirts' },
    { id: 2, title: "Men's Hoodie", price: 54.95, category: 'Winter' },
    { id: 3, title: "Denim Jeans", price: 49.95, category: 'Pants' }
]
// we usually use an arrow function - runs once for each array element until
condition is true
let jeans = products.find(product => product.title == 'Denim Jeans') // returns
matching item
let shirt = products.find(product => product.category == 'Shirts') // returns
matching item
console.log(jeans) // { id: 3, title: 'Denim Jeans', price: 49.95, category:
'Pants' }
console.log(shirt) // { id: 1, title: 'Sleeveless Tee', price: 23.95, category:
'Shirts' }
```

Arrays

filter method

- Syntax

```
const results = arr.filter(function(item, index, array) {
   // . . . If true is pushed to results and the iteration continues, and returns empty array if nothing found
});
```

- Syntax is similar to find, but filter returns **an array** of **all** matching elements.

```
const products = [
    { id: 1, title: 'Sleeveless Tee', price: 23.95, category: 'Shirts' },
    { id: 2, title: "Men's Hoodie", price: 54.95, category: 'Winter' },
    { id: 3, title: "Denim Jeans", price: 49.95, category: 'Pants' },
    { id: 4, title: "Ladies Tee", price: 25.95, category: 'Shirts' }
]
// we usually use an arrow function - runs once for each element, returns array of matches
let shirts = products.filter(product => product.category == 'Shirts')
let under50 = products.filter(product => product.price < 50)
console.log(shirts) // 2 matching items in shirts array
console.log(under50) // 3 matching items in under50 array
```

Institute of Data

Arrays

map method

- Syntax

```
const result = arr.map(function(item, index, array) {
   // return the new value instead of item
});
```

- It **transforms** the array by calling the **function**, which returns a transformation of each **item**, in a **new array of results**.

```
let titles = products.map(product => product.title)
let h2titles = products.map(product => '<h2>'+product.title+'</h2>')
let raisedPrices = products.map(product => ({...product, price: product.price + 5}) )
console.log(titles) // [ 'Sleeveless Tee', "Men's Hoodie", 'Denim Jeans', 'Ladies Tee' ]
console.log(h2titles) // [ '<h2>Sleeveless Tee</h2>', "<h2>Men's Hoodie</h2>", '<h2>Denim Jeans</h2>', '<h2>Ladies
Tee</h2>' ]
console.log(raisedPrices) // all prices increased by $5
```

Arrays

sort method

- Syntax

```
arr.sort(function compare(firstEl, secondE1) {..})
```

- Sorts the array **in place**, changing its **element order**. It also returns the sorted array, but the returned value is usually **ignored**, as arr itself is modified.

- If compareFn is omitted, the array elements are sorted as strings.

```
products.sort( (product1, product2) => product1.price - product2.price )
console.log(products) // original array is modified to new low-high price sorting order: 1,4,3,2
products.sort( (p1, p2) => p1.title > p2.title ? 1 : -1 )
console.log(products) // original array is modified to new title sorting order: 3,4,2,1

const numbers = [4,8,1,5,66,23,41]
console.log( numbers.sort() ) // [ 1, 23, 4, 41, 5, 66, 8 ] : string comparisons
console.log( numbers.sort((num1, num2) => num1 - num2) ) // [ 1, 4, 5, 8, 23, 41, 66 ]
```

Arrays

sort method

- The comparison function provided to **sort** should take two parameters (eg. **a** and **b**), and return a positive value if **a** is greater than **b**, a negative value if **a** is less than **b**, and 0 if they are equal. This is why we often use **minus** for calculating numerical comparisons.

```
const stringNums = ["1", "81", "41", "102", "35", "1004"]
console.log( stringNums.sort() ) // [ '1', '1004', '102', '35', '41', '81' ] : string comparisons
console.log( stringNums.sort((a, b) => a - b) ) // [ '1', '35', '41', '81', '102', '1004' ]
```

To keep a sorted array independent to the original, we need to **clone it first**:

```
const stringNums = ["1", "81", "41", "102", "35", "1004"]
const sortedNums = [...stringNums].sort()
console.log(stringNums) // [ '1', '81', '41', '102', '35', '1004' ]
console.log(sortedNums) // [ '1', '1004', '102', '35', '41', '81' ]
```

Arrays

reverse method

- Syntax

```
arr.reverse();
```

- It **reverses the order** of elements in arr, modifies the orginal array, and returns a **reference** to arr.

```
const elements = ['Wind', 'Water', 'Fire', 'Air']
const reversed1 = elements.reverse() // modifies the original
const reversed2 = [...elements].reverse() // clone first to preserve the original

console.log(elements) // [ 'Air', 'Fire', 'Water', 'Wind' ]
console.log(reversed2) // [ 'Wind', 'Water', 'Fire', 'Air' ] (reversed twice)
```

Arrays

reduce method

- Syntax

```
const value = arr.reduce(function(accumulator, item, index, array) {
   // . . .
},[initial]);
```

- When function is applied, the **result** of the **previous function call** is passed to the next one as the **first argument**, in the accumulator that stores the **combined result** of previous executions. Finally, this **single accumulated value** is returned.

```
const products = [
    { id: 1, title: 'Sleeveless Tee', price: 23.95, category: 'Shirts', quantity: 2 },
    { id: 2, title: "Men's Hoodie", price: 54.95, category: 'Winter', quantity: 3 },
    { id: 3, title: "Denim Jeans", price: 49.95, category: 'Pants', quantity: 5 }
]   // initial (below) should be 0 for reliability in calculating totals
const totalPrice = products.reduce((currentTotal, currentProduct) => currentTotal + currentProduct.price, 0)
const totalQty = products.reduce((currentQty, currentProduct) => currentQty + currentProduct.quantity, 0)
console.log(totalPrice) // 128.85000000000002
console.log(totalQty) // 10
```

Arrays

- The previous array functions (and there are more) can be **combined (chained) together** to simplify complex tasks on a list of items.

```
const products = [
    { id: 1, title: 'Sleeveless Tee', price: 23.95, category: 'Shirts', quantity: 2 },
    { id: 2, title: "Men's Hoodie", price: 54.95, category: 'Winter', quantity: 3 },
    { id: 3, title: "Denim Jeans", price: 49.95, category: 'Pants', quantity: 5 }
]
// get the titles of all products over $25:
const over25Titles = products.filter(prod => prod.price > 25).map(prod => prod.title)
console.log(over25Titles) // [ "Men's Hoodie", 'Denim Jeans' ]

// list product ids and titles in descending order of price:
const hiloProducts = [...products].sort((p1, p2) => p1.price - p2.price).reverse()
                    .map(prod => ({id: prod.id, title: prod.title}))
console.log(hiloProducts)
// [ {id: 2, title: "Men's Hoodie"}, {id: 3, title: 'Denim Jeans'}, {id: 1, title: 'Sleeveless Tee'} ]
```

Iterables

Iterable objects are special objects (such as arrays) which can **step through** each one in a **series of multiple values** in a for .. of loop.

- **String**, **Array**, **Map and Set** are all **built-in iterables**, so they can be used in a `for .. of` loop.
- Standard objects are **not iterable**. Custom objects can be **made iterable** by implementing the iterable protocol, but this is rarely done.

```javascript
const animalsArr = ['tiger', 'lion', 'elephant', 'giraffe']
for (let animal of animalsArr) { console.log(animal); } // prints each animal in turn

const animalObj = { name: 'tiger', genus: 'panthera', class: 'mammal' }
for (let property of animalObj) { console.log(property); } // TypeError: animalObj is not iterable
```

Arrays

Static Method Array.from

- Syntax

```
array.from(arraylike, mapFn, thisArg)
array.from(iterable, mapFn, thisArg)
```

- Conversion **from Array-like object** or **iterable to Array** (creates shallow copy)

```
console.log( Array.from("string") ) // [ 's', 't', 'r', 'i', 'n', 'g' ]
console.log( Array.from(new Set(['cat', 'bat', 'sat', 'cat', 'bat'])) ) // [ 'cat', 'bat', 'sat' ]
console.log( Array.from(new Map([[1, 'one'], [2, 'two'], [3, 'three']])) )
// [ [ 1, 'one' ], [ 2, 'two' ], [ 3, 'three' ] ]

function makeArray() {
    return Array.from(arguments);
}
console.log( makeArray(1, 2, 3) ); // [ 1, 2, 3 ]
```

Map

Map is a **collection** of **keyed data items**, very much like an **Object**. Objects use only **string** keys, but a Map allows keys and values of **any type**.

Methods and properties

- new Map(): creates the map
- map.set(key, value): stores the value by the key, and returns map itself
- map.size: returns the current element **count**

```
const exampleMap = new Map() // create new empty map object
exampleMap.set(1, 'number one') // 'set' adds a new key-value pair to the map
exampleMap.set('1', 'string one') // maps support keys of different types
exampleMap.set(true, 'true') // can have boolean keys
exampleMap.set({name: 'John'}, {phone: '0412345678'}) // object keys also valid
exampleMap.set('1', 'second string one') // overwrites previous value if key exists
console.log(exampleMap.size) // 4 - number of items in the map
console.log(exampleMap)
// Map(4) { 1 => 'number one', '1' => 'second string one', true => 'true',
          { name: 'John' } => { phone: '0412345678' } }
```

Map

Methods and properties continued

- map.get(key): returns the **value** with the matching key, or **undefined** if key doesn't exist in map
- map.has(key): returns **true** if the key exists, **false** otherwise
- map.delete(key): **removes** the value with the matching key
- map.clear(): removes **everything** from the map

```
console.log( exampleMap.get('1') ) // second string one - gets value for matching key
console.log( exampleMap.get(2) ) // undefined - key doesn't exist so no value
console.log( exampleMap.has(1) ) // true - key does exist
console.log( exampleMap.delete(true) ) // true - removes item and returns true if successful

exampleMap.clear() // removes all items from map
console.log( exampleMap ) // Map(0) {}
```

Map Iteration

- map.keys(): returns an **iterable** (not an array, but similar) for **keys**
- map.values(): returns an **iterable** (not an array, but similar) for **values**
- map.entries(): returns an **iterable** for **entries** [key, value]. Used by default in for .. of.

```
const recipeMap = new Map([
    ['flour', '1 cup'],
    ['milk', '1/2 cup'],
    ['eggs', 2],
    ['butter', '50g']
])
for (let ingredient of recipeMap.keys()) {
    console.log(ingredient) // flour, milk, eggs, butter
}
for (let quantity of recipeMap.values()) {
    console.log(quantity) // 1 cup, 1/2 cup, 2, 50g
}
for (let item of recipeMap) { // same as recipeMap.entries()
    console.log(item) // ['flour', '1 cup'], (and so on)
}
```

Map

Conversions with Object

- Object.fromEntries

Create an **Object** from Map.entries(). Since Objects and Maps are quite similar, we can **convert** from one to the other:

```
const priceMap = new Map([
    ['banana', 1],
    ['pineapple', 2],
    ['watermelon', 5]
])

const priceObject = Object.fromEntries(priceMap)
console.log(priceObject) // { banana: 1, pineapple: 2, watermelon: 5 }
```

Map

Conversions with Object

- Object.entries

Create a **Map** from an **Object** using Object.entries(object) inside the Map constructor.

```
const priceObject = { banana: 1, pineapple: 2, watermelon: 5 }

const priceMap = new Map( Object.entries(priceObject) )
console.log(priceMap) // Map(3) { 'banana' => 1, 'pineapple' => 2, 'watermelon' => 5 }
console.log(priceMap.get('banana')) // 1
```

Map

Maps are a versatile, efficient data structure for managing key-value data, and provide several advantages over regular objects and arrays:

- **Speed**: Map is faster for adding, searching, and deleting data, ideal for frequently updated data.
- **Flexibility**: It allows any data type as a key, including objects and functions, unlike basic objects.
- **Useful methods**: Built-in methods like has, get, and forEach simplify data handling.

Some common, practical uses for **Maps** include:

- Caching (or ‘memoising’) frequently accessed or expensive-to-calculate data
- Organizing data with non-string keys

Map - Caching Example

```
// Simulate fetching external data, which can be slow
function fetchExternalData(id) {
    console.log(`Fetching data for ID: ${id}`);
    const data = `Data for ID: ${id}`; // Simulated data
    return data;
}

// Create a Map for caching
const cache = new Map();

function getCachedData(id) {
    // Check if data is already in the cache
    if (cache.has(id)) {
        return cache.get(id); // return cached value, no expensive lookup
    }

    // If not in cache, fetch "external" data and store in cache for next time
    const data = fetchExternalData(id);
    cache.set(id, data);
    return data;
}

// Example usage
console.log('#1: ' + getCachedData(1)); // First time: fetches "external" data and caches result
console.log('#2: ' + getCachedData(1)); // Other times: can fetch result from cache, much faster
```

Set

A Set is a special type of collection: a **"set of values"** (without keys), where each value is **unique** and may occur only once.

Methods

- new Set(iterable): creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
- set.add(value): adds a value, returns the set itself.
- set.size: is the elements count.

```
const names = new Set(['Pedro', 'Oliver', 'Jack', 'Mateo'])
names.add('Mateo')
names.add('Oliver')
names.add('Bruno')
console.log(names.size) // 5 - only the unique names
console.log(names) // Set(5) { 'Pedro', 'Oliver', 'Jack', 'Mateo', 'Bruno' }
```

Set

Methods continued

- set.delete(value): **removes** the value, returns **true** if value existed at the moment of the call, otherwise **false**.
- set.has(value): returns **true** if the value exists in the set, otherwise **false**.
- set.clear(): removes **everything** from the set.

```
const names = new Set(['Pedro', 'Oliver', 'Jack', 'Mateo'])

console.log( names.delete('Jack') ) // true - successful delete
console.log( names.has('Jack') ) // false - Jack no longer exists in set
console.log( names.has('Mateo') ) // true - Mateo does exist in set
names.clear()
console.log(names) // Set(0) {}
```

Set

Iteration over Set

- **Loop over** items in a **set** either with for .. of or using forEach.

```
const names = new Set(['Pedro', 'Oliver', 'Jack', 'Mateo'])

// traditional style of for loop - works because Sets are iterable
for (let name of names) {
    console.log(name)
}

// more concise for simple operations, newer syntax using arrow function
names.forEach(name => console.log(name))
```

Set

Sets are a powerful, efficient data structure in JavaScript for managing collections of unique values, providing several advantages over regular arrays:

- **Uniqueness**: Set automatically ensures that all values are unique, preventing duplicates without needing to check if they already exist
- **Efficiency**: Operations like adding, searching, and deleting values in a Set are typically faster than with arrays, especially when dealing with large datasets.
- **Useful methods**: Built-in methods like has, get, and forEach make it easy to manage and interact with the data.

Some common, practical uses for **Sets** include:

- Easily filtering out duplicate values from datasets
- Keeping track of items that should be stored without repetition, like user actions, permissions, tags or visited URLs.

Efficient Algorithms

When choosing and making use of **data structures**, functions and constructs such as **for loops**, there are some key things to consider to ensure your code is efficient and maintains good performance even when under load. Understanding these will come with experience, so revisit these concepts as your code grows.

1. Time Complexity (Big O Notation):

Big O notation is a type of formula capturing the ‘order of complexity’ that shows how much longer an algorithm takes as the amount of data it handles increases. It can help to analyse and compare the efficiency of different algorithms. Examples:

- O(1): Constant time, like accessing an array element.
- O(n): Linear time, like iterating through an array.
- O(n^2): Quadratic time, like nested loops (to be avoided).

Efficient Algorithms

2. Space Complexity:

As well as being aware of how long your code will take to run, we also want to control how much memory an algorithm uses as the input size grows.

Why it matters: Efficient algorithms use minimal memory (no unnecessary variables), which is crucial for performance, especially in environments with limited resources.

3. Avoiding Unnecessary Operations:

Efficient algorithms aim to minimise redundant work, like checking the same value multiple times or recalculating results, which saves both time and space.

Example: Use a Set to track unique items instead of searching an array repeatedly, or cache expensive calculations instead of performing them every time.

Efficient Algorithms

4. Using Built-In Methods:

JavaScript’s built-in methods like map and `filter` are optimised for efficiency.

Example: Use `filter` to clean up an array instead of manual loops.

5. Choosing the Right Data Structure:

The efficiency of an algorithm can depend on the data structure used, so think about what kind of data you need to manage and choose appropriately.

Example: Use Set for unique values, Map for key-value pairs, Array for ordered lists.

6. Early Exits:

Exiting or returning from loops/functions as soon as a result is found will save time.

Why it matters: Reduces unnecessary processing.

Destructuring assignment

Destructuring assignment is a **special syntax** that allows us to **"unpack"** arrays or objects into a **bunch of variables**.

Array destructuring

- It **"destructures"** by **copying** items into variables.

```
const mj = ['Michael', 'Jordan']
const [mjFirst, mjLast] = mj // destructure (unpack) array on right into separate variables on left

console.log(mjFirst, mjLast) // Michael Jordan
```

- It **ignores** elements using **commas**.

```
const [jcFirst, jcLast, , , jcPlace] = ['Julius', 'Caesar', 'Consul', 'of the', 'Roman', 'Republic']
console.log(`${jcFirst} ${jcLast} is a ${jcPlace}`) // Julius Caesar is a Roman
```

Destructuring assignment

Array destructuring

- It works with any **iterable** on the **right-side**.

```
const [a, b, c] = "abc" // strings are iterable, so can break into characters
const [ one, two, three ] = new Set([1, 2, 3]) // Sets are iterable, so can be destructured
const [ [type, quantity] ] = new Map([ ['apple', 4] ]) // Maps are iterable too
// now we have 8 individual variables: a, b, c, one, two, three, type, quantity
console.log(a, b, c, one, two, three, type, quantity) // a b c 1 2 3 apple 4
```

- It **assigns** to anything at the **left-side**.

```
const monarch = {}; // empty object
[ monarch.title, monarch.name ] = "King Charles".split(' '); // store array pieces in object properties
console.log(monarch); // { title: 'King', name: 'Charles' }
```

Destructuring assignment

Array destructuring

- When looping with .entries() we can destructure into **key** and **value** variables

```
const teeProduct = { id: 1, title: 'Sleeveless Tee', price: 23.95, category: 'Shirts' }
// key and value are just variable names, could be anything
for (let [key, value] of Object.entries(teeProduct)) {
    console.log(`${key}: ${value}`) // id: 1, title: Sleeveless Tee, price: 23.95 ...
}
```

- Swap **variables** trick

```
let student = 'James', teacher = 'Andrew';
[student, teacher] = [teacher, student]

console.log(student) // Andrew
console.log(teacher) // James
```

Destructuring assignment

The array rest ...

Usually, if the array is longer than the list at the left, the **“extra”** items are **omitted**. If we’d like to capture them, we can add a parameter that gets **‘the rest’** using three dots **...**

```
const [jcFirst, jcLast, ...jcTitles] = ['Julius', 'Caesar', 'Consul', 'of the', 'Roman', 'Republic']
console.log( jcTitles ) // [ 'Consul', 'of the', 'Roman', 'Republic' ]
console.log( jcTitles.length ) // 4
```

Default values

We can also provide a default value for any extra variable without a match on the right:

```
const [jcFirst, jcLast, ...jcTitles] = ['Julius', 'Caesar', 'Consul', 'of the', 'Roman', 'Republic']
console.log( jcTitles ) // [ 'Consul', 'of the', 'Roman', 'Republic' ]
console.log( jcTitles.length ) // 4
```

Destructuring assignment

Object destructuring

Objects can be destructured similarly to arrays - we just use **curly brackets** instead of **square brackets**. There are some other differences as well:

- An **existing** object, at the **right side** can be split into variables. The left side contains an **object-like "pattern"** for **corresponding properties** based on name:

```
// property names (keys) on right are matched to variable names on left
let { width, height, title } = { title: 'My Component', height: 100, width: 200 }
console.log(width, height, title) // 200 100 My Component
```

- Assign default value to **missing property** (same as arrays)

```
let { width = 200, height = 100, title} = { title: 'My Component' }
console.log(width, height, title) // 200 100 My Component
```

Destructuring assignment

Object destructuring

- Smart function parameters

There are times when a function has **many parameters**, most of which are optional. We can pass **parameters as an object**, and the function immediately **destructures them into variables**.

```
function displayComponent({height = 200, width = 100, title}) {
    console.log(`<div style="width:${width}px; height:${height}px"><h2>${title}</h2></div>`)
}

displayComponent({width:200, title:'My Awesome Component'})
displayComponent({title:'My Amazing Component'})
displayComponent({width:300, height:300, title:'My Average Component'})
```

Destructuring assignment

Object destructuring

- The rest pattern **...** .

We can use the … **rest** pattern when **destructuring objects** in the same way that we can use it for arrays, to **unpack** remaining properties into a **new object**.

```
let options = { width: 200, height: 100, title: 'My Component' }
let { title, ...rest } = options
console.log(title) // My Component
console.log(rest) // { width: 200, height: 100 }
```

Date and time

Internally all dates in JavaScript are converted to and stored as a **number**, which is the total milliseconds (1/1000th of a second) since the **epoch** time: **Jan 1st 1970 UTC+0**.

Creation

- Without arguments, the **Date** constructor creates a new Date for the current time:

```
const now = new Date()
console.log( now ) // 2023-03-26T11:45:59.096Z
console.log( +now ) // 1679832116638 - number of milliseconds since epoch
```

- With a single argument new Date(milliseconds), it creates a **Date object** with the time equal to the number of **milliseconds** after the epoch:

```
const epoch = new Date(0) // 0 milliseconds since Jan 1 1970
const jan2_1970 = new Date(1000 * 60 * 60 * 24) // a full day in milliseconds after Jan 1
console.log(epoch) // 1970-01-01T00:00:00.000Z
console.log(jan2_1970) // 1970-01-02T00:00:00.000Z
```

Date and time

Creation

- new Date(datestring) parses a string into a **Date object**, which is the same as Date.parse. Strings should be **yyyy-mm-dd**. Strings including a time assume a **local timezone**, whereas strings without a time assume a **UTC timezone** (**Z**).
- To specify a timezone, add **+hh:mm** after the time to provide the **difference** between UTC and the desired timezone.

```
const christmas = new Date('2023-12-25') // assumes UTC timezone if time not included
console.log(christmas) // 2023-12-25T00:00:00.000Z - Z indicates UTC timezone, GMT+0

const nyeLocal = new Date('2023-12-31 23:59:59') // assumes local timezone if time is included
const nyeUTC = new Date('2023-12-31 23:59:59+00:00') // specific timezone specified (UTC)
console.log(nyeLocal) // 2023-12-31T13:59:59.000Z - stored internally as UTC so now hours are different
console.log(nyeUTC) // 2023-12-31T23:59:59.000Z - UTC before midnight, no longer local timezone
```

Date and time

Creation

- new Date(year, month, day, hours, minutes, seconds, ms) creates the date with the given components in the **local time zone**. Only the first **two are mandatory**.
  - The year must have **4 digits**: 2013 is okay, 98 is not.
  - The **month** count **starts with 0 (Jan)**, up to 11 (Dec).
  - The day parameter is actually the day of month. If absent then 1 is assumed.
  - If hours/minutes/seconds/ms are absent, they are assumed to be equal to 0.

```
const boxingDay = new Date(2023, 11, 26) // month 11 is December, assumes local timezone
console.log(boxingDay) // 2023-12-25T14:00:00.000Z - so hours are different in UTC

const remembranceDay = new Date( 2023, 10, 11, 11, 11 ) // month 10 is November, assumes local timezone
console.log(remembranceDay) // 2023-11-11T01:11:00.000Z - so hours are different in UTC
```

Date and time

Get date components. Replace get with set when updating.

- getFullYear(): Get the year (4 digits)
- getMonth(): Get the month, from **0 to 11**
- getDate(): Get the day of month, from **1 to 31**
- getHours(), getMinutes(), getSeconds(), getMilliseconds(): Get the corresponding time components from the given Date object (as integers)
- getDay(): Get the day of week, from **0 (Sunday)** to **6 (Saturday)**. The first day is always Sunday.
- getTime(): Return the timestamp for the date, a number of milliseconds passed from the **1st of January 1970 UTC+0**.
- getTimezoneOffset(): Return the difference between **UTC** and the **local time zone, in minutes**.

Date and time

Displaying Dates

Since all JavaScript dates are stored internally as **UTC time**, we often need to convert to a local timezone when **displaying** them.

The **toLocaleString()** method returns a string with a locale and language-sensitive representation of this date including the time. **toLocaleDateString()** and **toLocaleTimeString()** display just the date and time portions of the date respectively.

```
const christmas = new Date('2023-12-25') // assumes UTC timezone if time not included
console.log( christmas.toLocaleDateString() ) // 25/12/2023 - dd/mm/yyyy if in Australia/NZ
console.log( christmas.toLocaleString('ko-KR', { timeZone: 'Asia/Seoul' }) )
// 2023. 12. 25. 오전 9:00:00 - both timezone and language are converted to Korean

const nyeLocal = new Date('2023-12-31 23:59:59') // assumes local timezone if time is included
console.log( nyeLocal.toLocaleString() ) // 31/12/2023, 11:59:59 pm - default to local TZ
```

JSON

JSON (JavaScript Object Notation) is a general format used to represent values and objects. It is described in the **RFC 4627** standard, and is often used to **send data** between two separate applications via an **API**. It looks similar to JS object syntax, but all keys need to be double quoted as strings.

- JSON.stringify method converts **objects** into **JSON**.

```
const student = {
    name: 'Sita',
    age: 28,
    courses: ['HTML', 'CSS', 'JS'],
    occupation: null
}
console.log( JSON.stringify(student) )
//{"name":"Sita","age":28,"courses":["HTML","CSS","JS"],"occupation":null}
```

JSON

- **JSON** is a **data-only** language-independent specification, so some **JavaScript-specific object properties** are **skipped** by JSON.stringify. Namely:
  - Function properties (methods)
  - Symbolic keys and values
  - Properties that store undefined

```
const book = {
    title: "Gone With The Wind",
    printTitle() { // ignored
        console.log(this.title)
    },
    releaseDate: undefined // ignored
}

console.log(JSON.stringify(book)) // {"title":"Gone With The Wind"}
```

JSON

- No circular references with JSON.stringify. JSON can't convert **recursive structures** into strings, and will throw an error if you try:

```
const room = {
    number: 23
}
const meetup = {
    title: "Strategy Conference",
    participants: ['Chris', 'Tina'],
}

meetup.place = room; // meetup references room
room.occupiedBy = meetup; // room references meetup

JSON.stringify(meetup); // TypeError: Converting circular structure to JSON
```

JSON

Excluding and transforming: replacer

- The second argument in JSON.stringify(value[, replacer, space]) is an **Array of properties to encode** or a **mapping function** ***function(key, value)***.
- replacer use case: **filter out** circular references

```
console.log( JSON.stringify(meetup, ['title', 'participants']) ); // just stringify the properties in
the array: {"title":"Strategy Conference","participants":["Chris","Tina"]}
```

```
console.log( JSON.stringify(meetup, function(key, value) {
    if (key != '' && value == meetup) return undefined // skip references to current object
    else if (typeof value == 'function') return value.toString() // stringify functions
    return value // otherwise return original value unchanged
}, 2) ); // use 2 spaces for nicer formatting
```

JSON

JSON.stringify : custom “toJSON”

Like toString for **string conversion**, an object may provide the method toJSON for **to-JSON** conversion. JSON.stringify automatically calls it whenever **available**.

```
const room = {
    number: 23, toJSON() { return this.number }
}
const meetup = {
    title: "Strategy Conference", participants: ['Chris', 'Tina']
}
meetup.place = room; // meetup references room
room.occupiedBy = meetup; // room references meetup

console.log( JSON.stringify(meetup) ); // no more circular references as room stringifies to 23
// {"title":"Strategy Conference","participants":["Chris","Tina"],"place":23}
```

JSON

JSON.parse method

- Converts a string back into an object by parsing it. Syntax:

```
JSON.parse(str, [review]);
```

- str is the **JSON string** to be **decoded**, reviver is an **optional function(key, value)** used to **transform** the value. Usually the **reviver** function is not needed.

```
const meetup = {
    title: "Strategy Conference", participants: ['Chris', 'Tina'], date: '2023-06-01'
}
const meetupString = JSON.stringify(meetup) // convert object to string
const meetupParsed = JSON.parse(meetupString, (key, value) => { // convert string to object
    if ( !isNaN(Date.parse(value)) ) return new Date(value) // if valid date, create Date object
    return value;
})
console.log(meetupParsed) // { title, participants: (as above), date: 2023-06-01T00:00:00.000Z }
```

JSON

JSON.parse for deep cloning

We said earlier that destructuring can be used to **shallow clone** an object - ideal when the original object contains no nested properties and needs to be independent.

A **deep clone** is a copy of an object that not only duplicates its **top-level properties**, but also creates new instances of any **nested objects or arrays**. With JSON.stringify() we can transform an object into a string, then back into a new object using JSON.parse():

```
const box1 = {
    size: 'large',
    dimensions: { width: 50, length: 70, height: 30, units: 'cm' },
    items: [ 'glasses', 'plates', 'cutlery' ]
}
const boxString = JSON.stringify(box1) // convert object to string
const box2 = JSON.parse(boxString) // convert string back to new object

// how could we check to make sure both boxes are the same but independent?
```

Revision

- What are some alternatives to decimal numbers and why might they be useful?
- What might cause a calculated number to lose precision in JS?
- What are some of the built-in string manipulation functions in JS?
- What is an array?
- What are some of the built-in array manipulation functions in JS?
- What is an iterable data type, and what are some examples?
- What are some ways we can iterate in JS?
- What are some differences between a Map and a Set?
- What does destructuring mean, and which data types can use it?
- How are dates represented internally in JS?