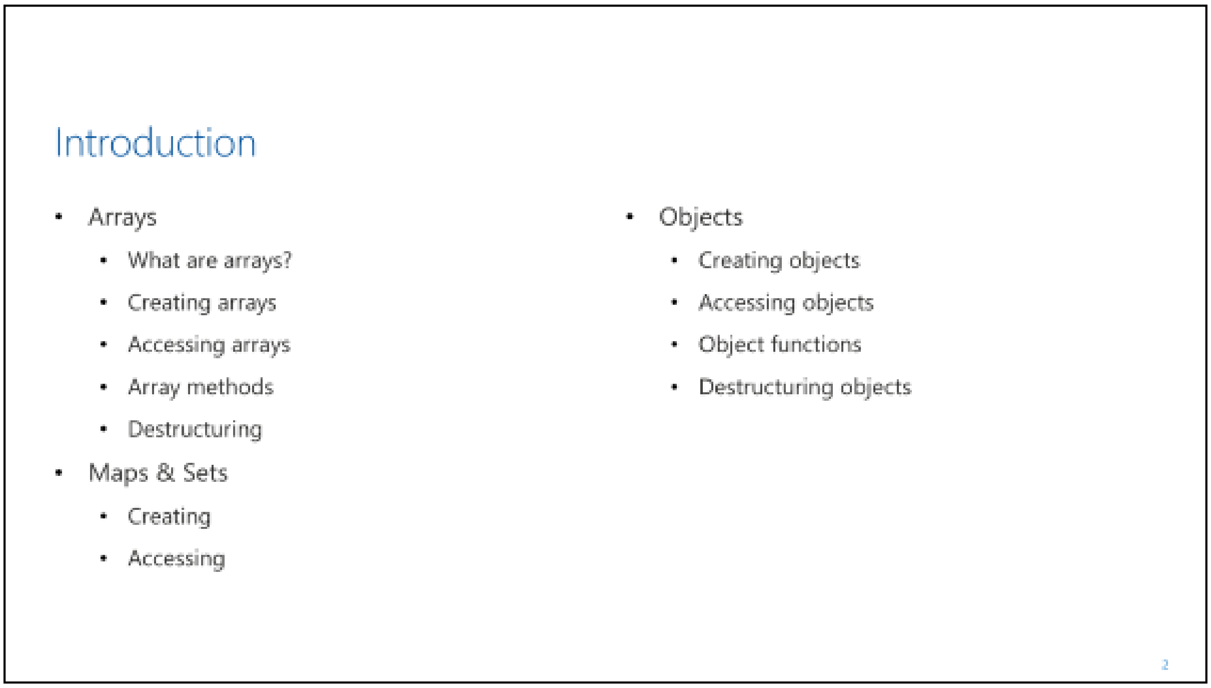
Collections

PROGRAMMING WITH JAVASCRIPT

QA



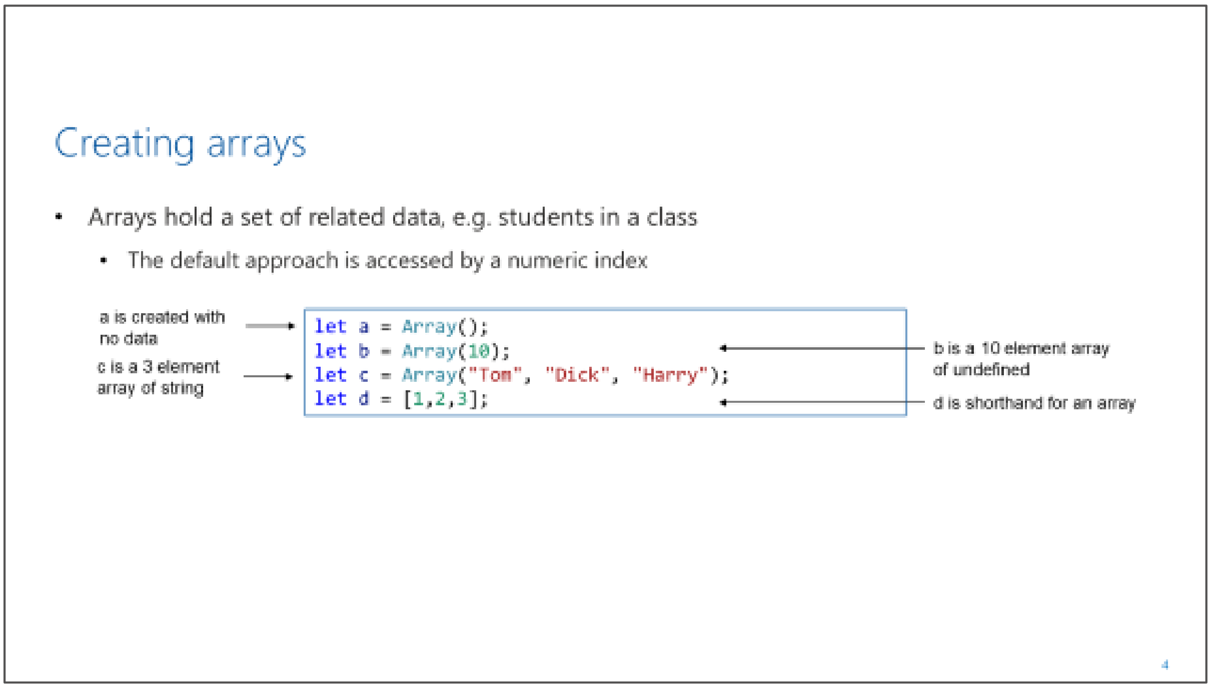
Introduction

- Arrays
  - What are arrays?
  - Creating arrays
  - Accessing arrays
  - Array methods
  - Destructuring
- Maps & Sets
  - Creating
  - Accessing
- Objects
  - Creating objects
  - Accessing objects
  - Object functions
  - Destructuring objects

2

Arrays

PROGRAMMING WITH JAVASCRIPT



Creating arrays

- Arrays hold a set of related data, e.g. students in a class
  - The default approach is accessed by a numeric index

```
let a = Array();
let b = Array(10);
let c = Array("Tom", "Dick", "Harry");
let d = [1,2,3];
```

a is created with no data
c is a 3 element array of string
b is a 10 element array of undefined
d is shorthand for an array
4

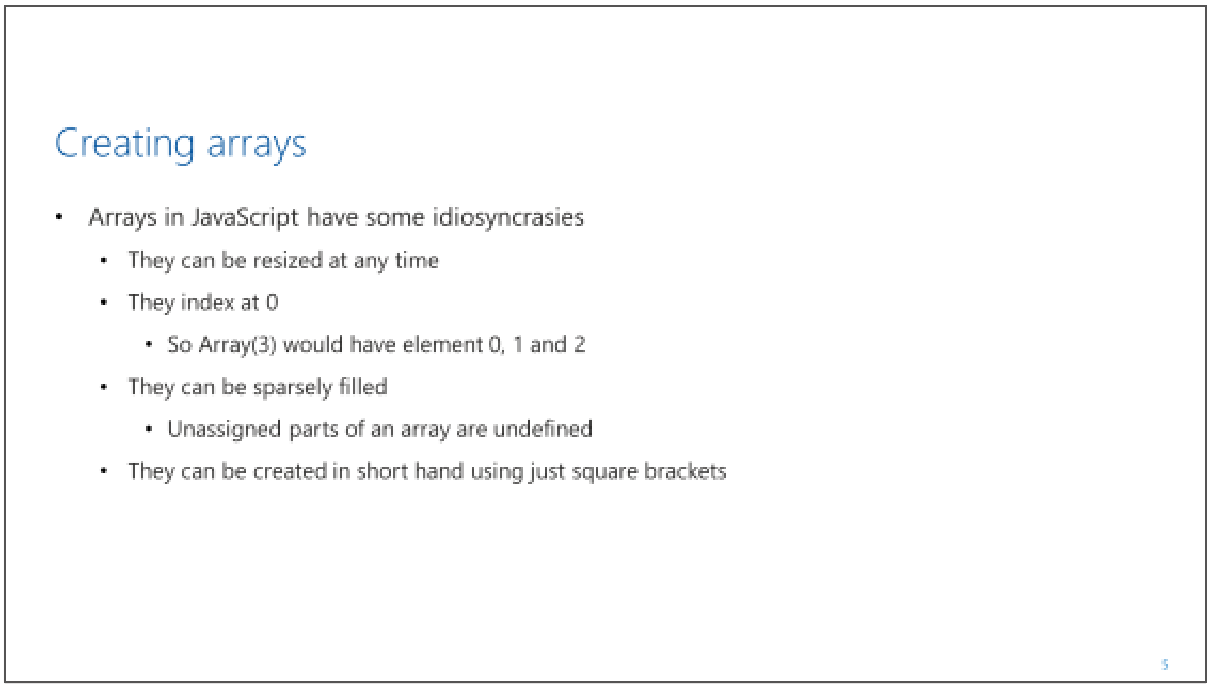
As with the rest of JavaScript, the elements in arrays are loosely-typed. This means that each separate element of an array can contain a different type. This is useful and dangerous at once and brings us back to the consideration that type matters.

Array indices in JavaScript start at 0. Arrays have a `length` property equal to the length of the array; note that an array of length of 6 will have elements 0 to 5.

JavaScript arrays are *sparse* – this means that there can be holes in an array. The example above shows an array that initially has values assigned to elements number 0 and 5. This results in an array that contains only two elements. Elements numbered 1 to 4 exist but, if referenced, will yield the special value `undefined`.

Note that once a value has been assigned to an element of an array, that array element is now defined, and cannot be removed. Assigning `null` to it changes the value but does not remove the array element. In fact, simply *referring* to the element is enough to create it.

Arrays in JavaScript will grow dynamically from its initial size. It is also possible to shrink an array by assigning a smaller value to the `length` property than it currently holds. Do not assume that elements with indexes greater than or equal to the new value exist.



Creating arrays

- Arrays in JavaScript have some idiosyncrasies
  - They can be resized at any time
  - They index at 0
    - So Array(3) would have element 0, 1 and 2
  - They can be sparsely filled
    - Unassigned parts of an array are undefined
  - They can be created in short hand using just square brackets

5

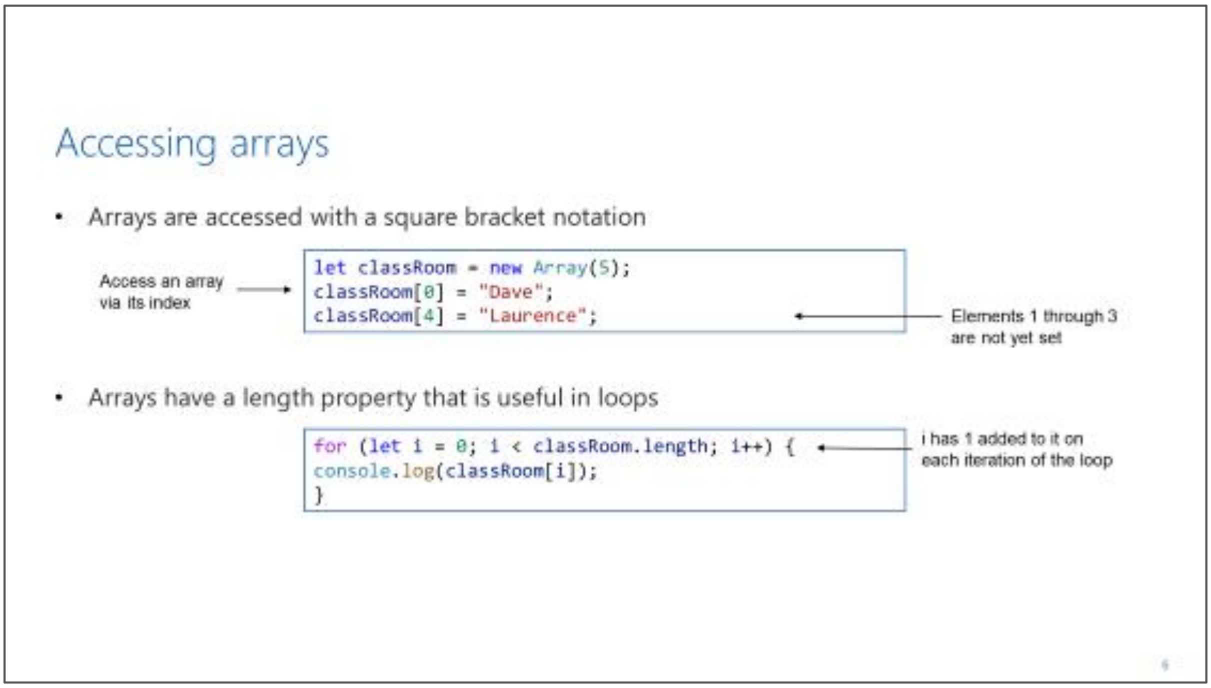
As with the rest of JavaScript, the elements in arrays are loosely-typed. This means that each separate element of an array can contain a different type. This is useful and dangerous at once and brings us back to the consideration that type matters.

Array indices in JavaScript start at 0. Arrays have a `length` property equal to the length of the array; note that an array of length of 6 will have elements 0 to 5.

JavaScript arrays are *sparse* – this means that there can be holes in an array. The example above shows an array that initially has values assigned to elements number 0 and 5. This results in an array that contains only two elements. Elements numbered 1 to 4 exist but, if referenced, will yield the special value `undefined`.

Note that once a value has been assigned to an element of an array, that array element is now defined, and cannot be removed. Assigning `null` to it changes the value but does not remove the array element. In fact, simply *referring* to the element is enough to create it.

Arrays in JavaScript will grow dynamically from its initial size. It is also possible to shrink an array by assigning a smaller value to the `length` property than it currently holds. Do not assume that elements with indexes greater than or equal to the new value exist.



Accessing arrays

- Arrays are accessed with a square bracket notation

```
let classRoom = new Array(5);
classRoom[0] = "Dave";
classRoom[4] = "Laurence";
```

Access an array via its index
Elements 1 through 3 are not yet set

- Arrays have a length property that is useful in loops

```
for (let i = 0; i < classRoom.length; i++) {
console.log(classRoom[i]);
}
```

i has 1 added to it on each iteration of the loop

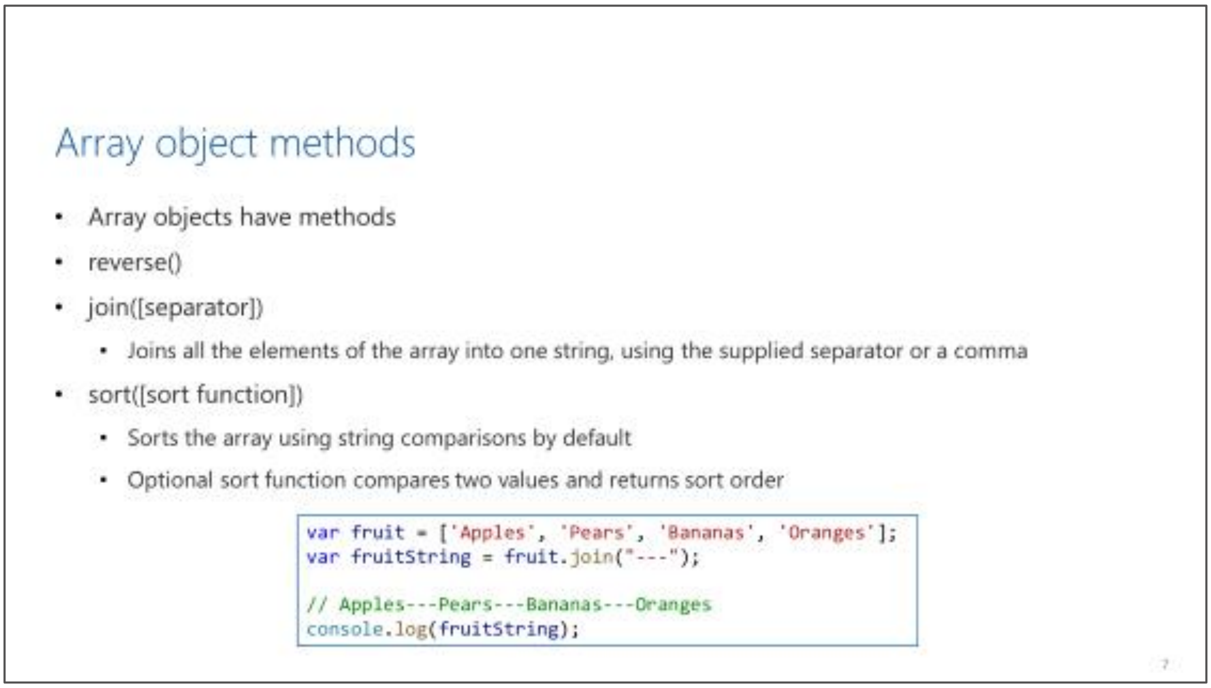
Arrays allow us to store a related set of data. Arrays store data in a list of elements; we access a particular elements by specifying the name of the array and the element index within square brackets, e.g.

```
myArray[0];
```

The first element of the array is always accessed through [0].

In the above example, a 5 element array is created and elements 1 through 3 are not set. In this situation, they will have a value of `undefined`. If you remember what we have learnt from the module on types, this makes perfect sense. They have been created but not initialised.

Arrays also have a length property. This will always reflect the exact number of elements an array contains. As you can see, this is immensely useful as it allows us to create loops that will always run as many times as is needed.



Array object methods

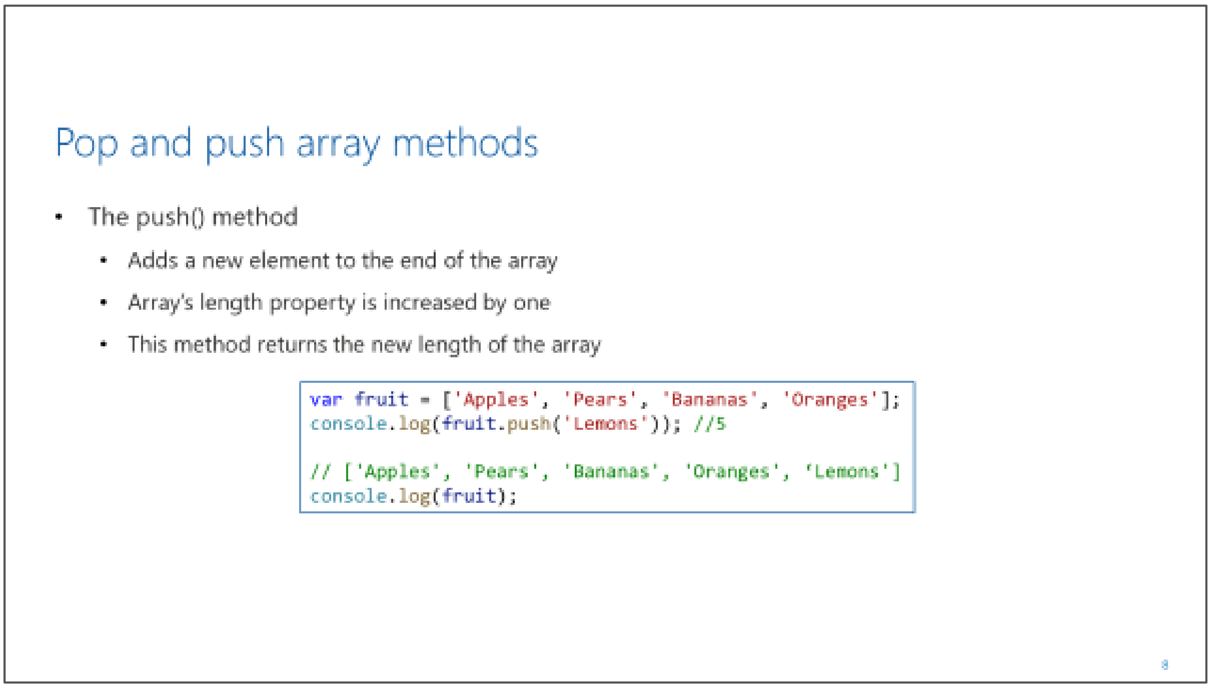
- Array objects have methods
- reverse()
- join([separator])
  - Joins all the elements of the array into one string, using the supplied separator or a comma
- sort([sort function])
  - Sorts the array using string comparisons by default
  - Optional sort function compares two values and returns sort order

```
var fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
var fruitString = fruit.join("---");

// Apples---Pears---Bananas---Oranges
console.log(fruitString);
```

7

Array objects have various methods:

- **`reverse`** reverses the order of the elements in the array, so that the last element in the array becomes the first, and so on. This method operates directly on the array itself, rather than returning a new array
- **`join`** returns a string formed by joining together all the elements in the array using the supplied separator. If no separator is supplied, a comma is used. The separator may be any string (including an empty one). Undefined array elements are represented by a null string, meaning that two or more separators will appear next to each other
- **`Sort`** sorts the array. If no function argument is supplied, the array elements are temporarily converted to strings and sorted in standard dictionary order. To make the sort generic, it is possible to supply a function as an argument. This function will be called by the sort routine as necessary to compare two values in the array. The function should have the form: `compare(a, b)` and should return a value less than 0 if a should be sorted less than b, 0 if they are equal, and greater than 0 if a should be sorted greater than b



Pop and push array methods

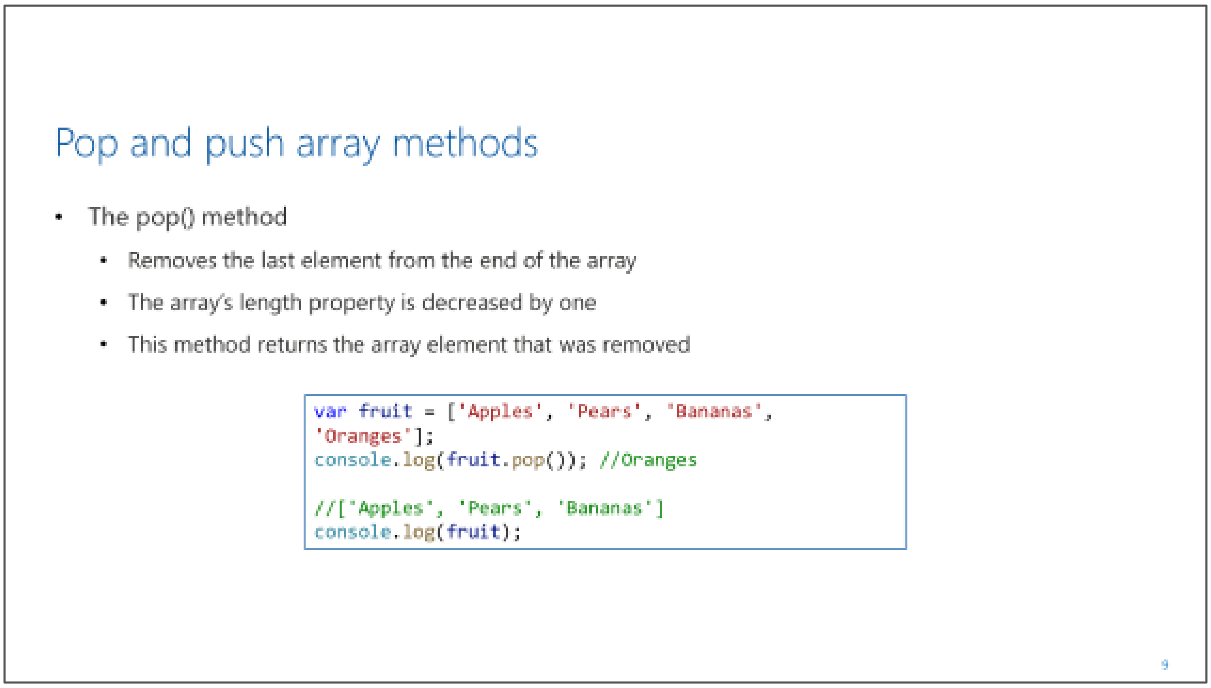
- The push() method
  - Adds a new element to the end of the array
  - Array's length property is increased by one
  - This method returns the new length of the array

```
var fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.push('Lemons')); //5

// ['Apples', 'Pears', 'Bananas', 'Oranges', 'Lemons']
console.log(fruit);
```

8

The JavaScript `Array.push()` method adds a new element to the end of the array. When doing so, the array's length property increases by one. After adding the new element to the end of the array, this method returns the new length of the array.

The JavaScript `Array.pop()` method removes the last element from the end of the array. When doing so, the array's length property decreases by one. After removing the last element from the end of the array, this method returns the array element that was removed.



Pop and push array methods

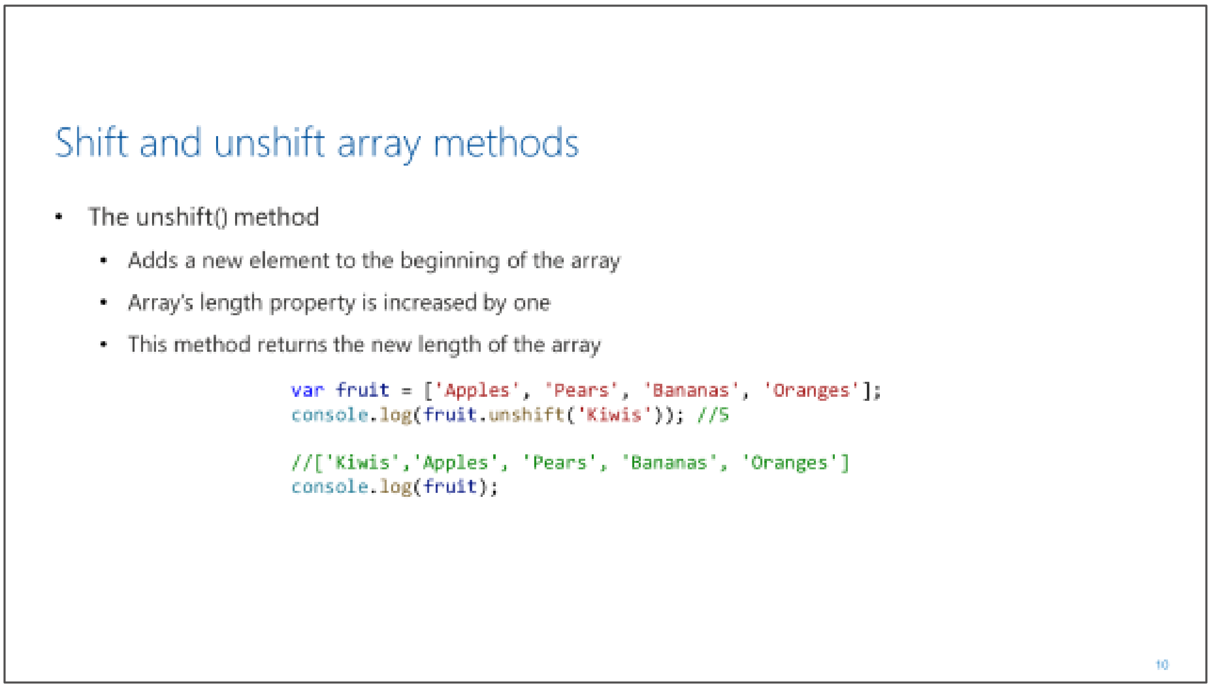
- The pop() method
  - Removes the last element from the end of the array
  - The array's length property is decreased by one
  - This method returns the array element that was removed

```
var fruit = ['Apples', 'Pears', 'Bananas',
'Oranges'];
console.log(fruit.pop()); //Oranges

//['Apples', 'Pears', 'Bananas']
console.log(fruit);
```

9

The JavaScript `Array.push()` method adds a new element to the end of the array. When doing so, the array's length property increases by one. After adding the new element to the end of the array, this method returns the new length of the array.

The JavaScript `Array.pop()` method removes the last element from the end of the array. When doing so, the array's length property decreases by one. After removing the last element from the end of the array, this method returns the array element that was removed.



Shift and unshift array methods

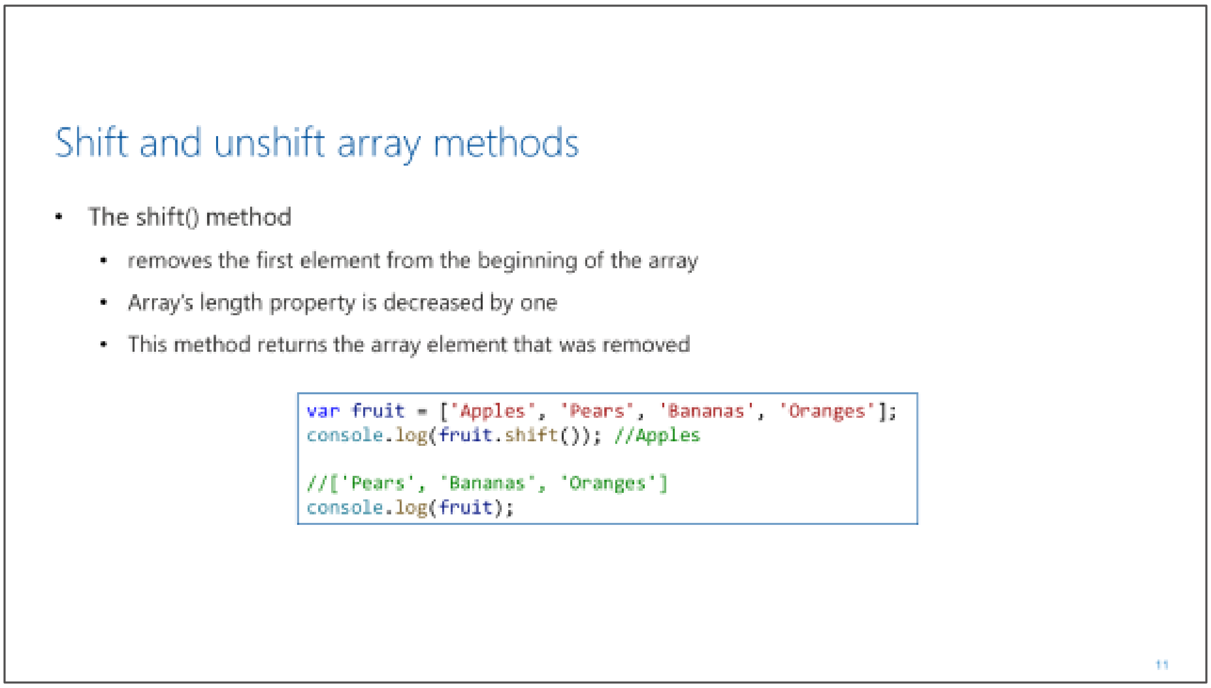
- The unshift() method
  - Adds a new element to the beginning of the array
  - Array's length property is increased by one
  - This method returns the new length of the array

```
var fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.unshift('Kiwis')); //5

//['Kiwis','Apples', 'Pears', 'Bananas', 'Oranges']
console.log(fruit);
```

10

The JavaScript `Array.shift()` method removes the first element from the beginning of the array. When doing so, the array's length property decreases by one. After removing the first element from the beginning of the array, this method returns the array element that was removed.

The JavaScript `Array.unshift()` method adds a new element to the beginning of the array. When doing so, the array's length property increases by one. After adding the new element to the beginning of the array, this method returns the new length of the array.



Shift and unshift array methods

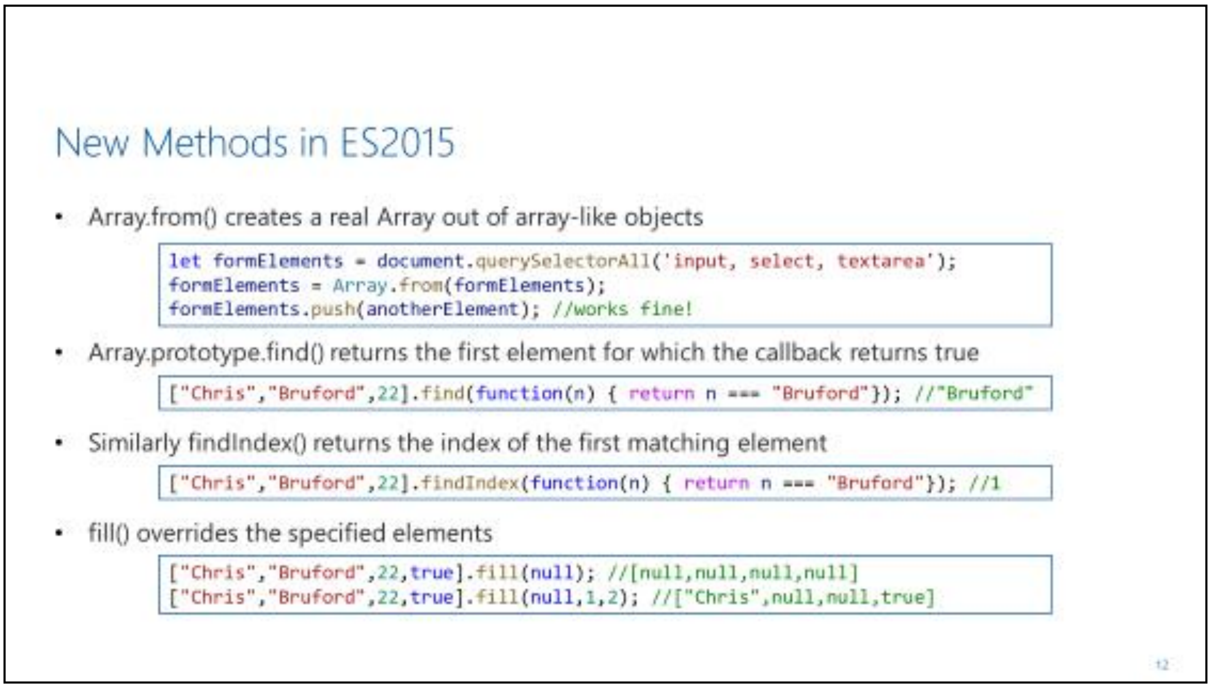
- The shift() method
  - removes the first element from the beginning of the array
  - Array's length property is decreased by one
  - This method returns the array element that was removed

```
var fruit = ['Apples', 'Pears', 'Bananas', 'Oranges'];
console.log(fruit.shift()); //Apples

//['Pears', 'Bananas', 'Oranges']
console.log(fruit);
```

11

The JavaScript `Array.shift()` method removes the first element from the beginning of the array. When doing so, the array's length property decreases by one. After removing the first element from the beginning of the array, this method returns the array element that was removed.

The JavaScript `Array.unshift()` method adds a new element to the beginning of the array. When doing so, the array's length property increases by one. After adding the new element to the beginning of the array, this method returns the new length of the array.



New Methods in ES2015

- Array.from() creates a real Array out of array-like objects

```
let formElements = document.querySelectorAll('input, select, textarea');
formElements = Array.from(formElements);
formElements.push(anotherElement); //works fine!
```

- Array.prototype.find() returns the first element for which the callback returns true

```
["Chris","Bruford",22].find(function(n) { return n === "Bruford"}); //"Bruford"
```

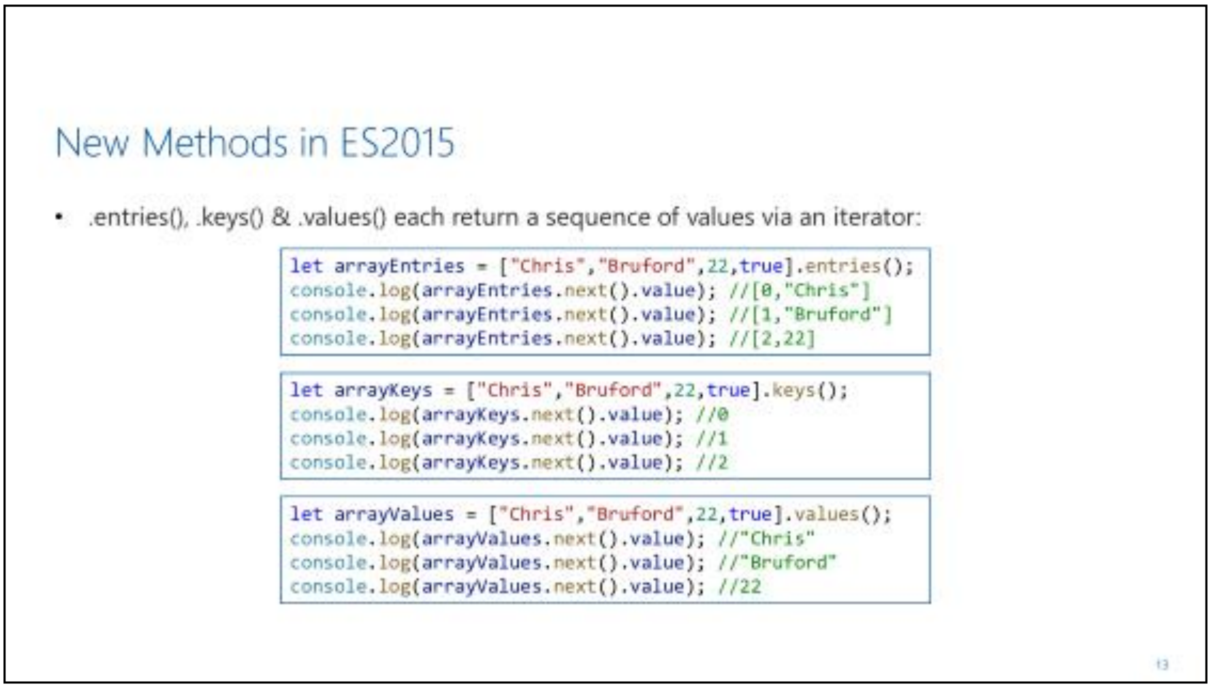
- Similarly findIndex() returns the index of the first matching element

```
["Chris","Bruford",22].findIndex(function(n) { return n === "Bruford"}); //1
```

- fill() overrides the specified elements

```
["Chris","Bruford",22,true].fill(null); //[null,null,null,null]
["Chris","Bruford",22,true].fill(null,1,2); //["Chris",null,null,true]
```

12



New Methods in ES2015

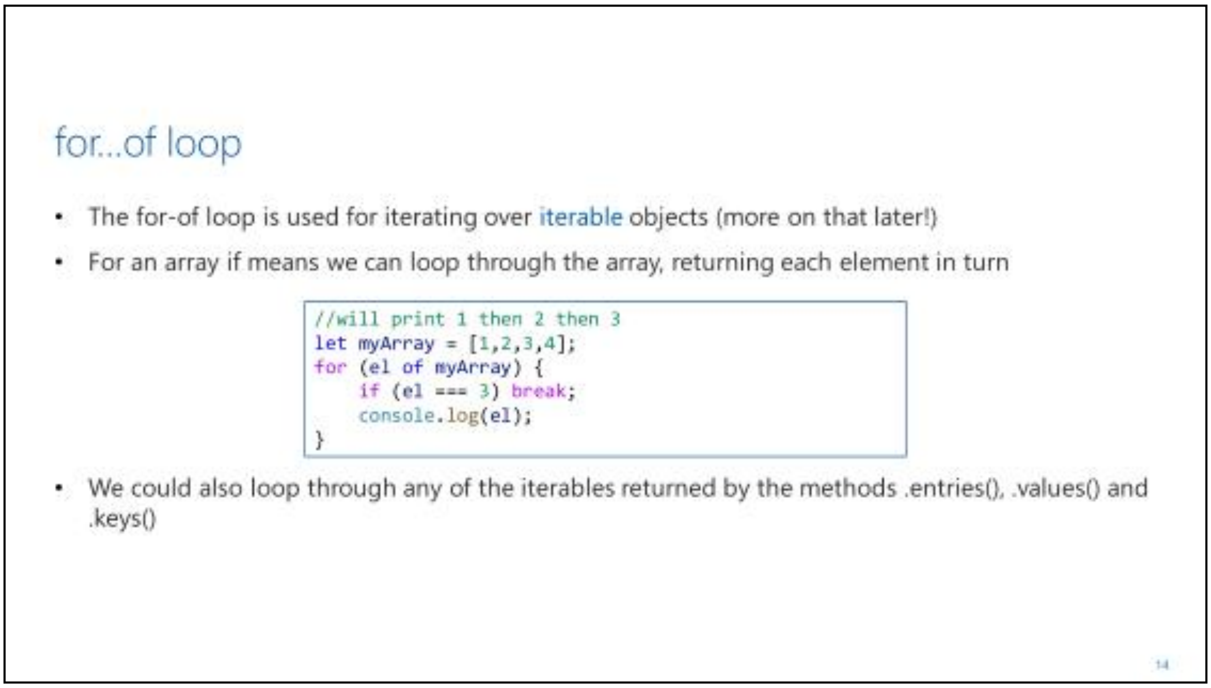
- .entries(), .keys() & .values() each return a sequence of values via an iterator:

```
let arrayEntries = ["Chris","Bruford",22,true].entries();
console.log(arrayEntries.next().value); //[0,"Chris"]
console.log(arrayEntries.next().value); //[1,"Bruford"]
console.log(arrayEntries.next().value); //[2,22]
```

```
let arrayKeys = ["Chris","Bruford",22,true].keys();
console.log(arrayKeys.next().value); //0
console.log(arrayKeys.next().value); //1
console.log(arrayKeys.next().value); //2
```

```
let arrayValues = ["Chris","Bruford",22,true].values();
console.log(arrayValues.next().value); //"Chris"
console.log(arrayValues.next().value); //"Bruford"
console.log(arrayValues.next().value); //22
```

13



for...of loop

- The for-of loop is used for iterating over iterable objects (more on that later!)
- For an array if means we can loop through the array, returning each element in turn

```
//will print 1 then 2 then 3
let myArray = [1,2,3,4];
for (el of myArray) {
    if (el === 3) break;
    console.log(el);
}
```

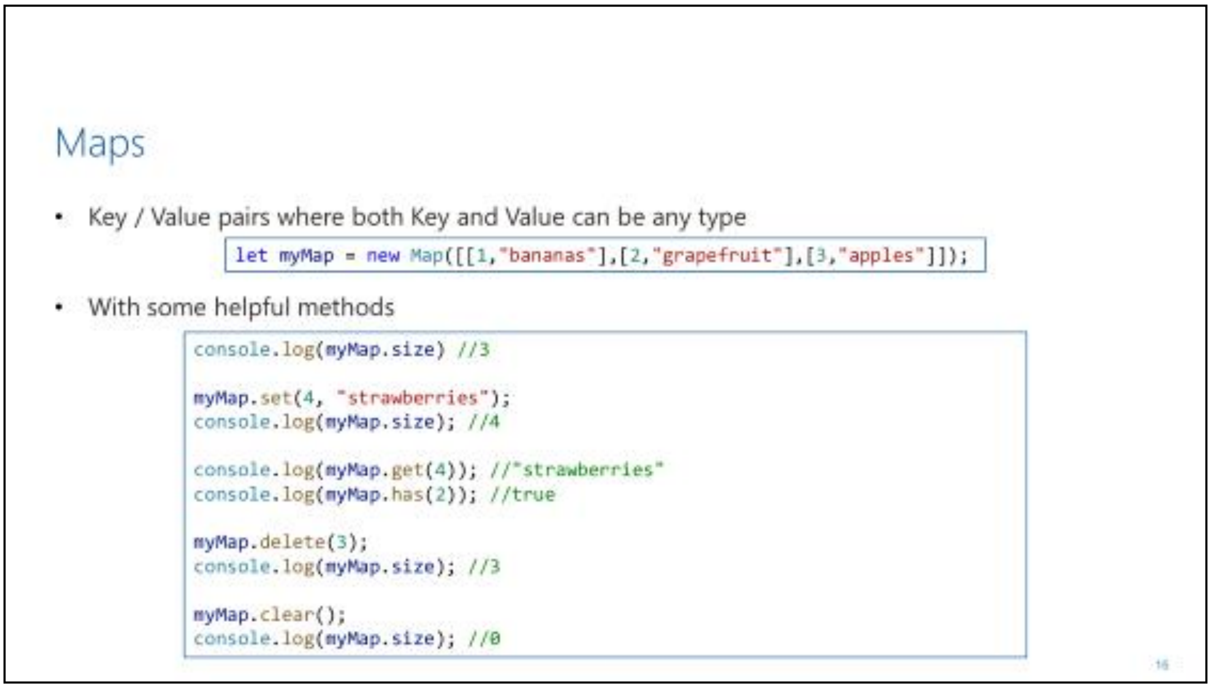
- We could also loop through any of the iterables returned by the methods .entries(), .values() and .keys()

14

Exercise – Part 1

- Creating and Managing arrays

15



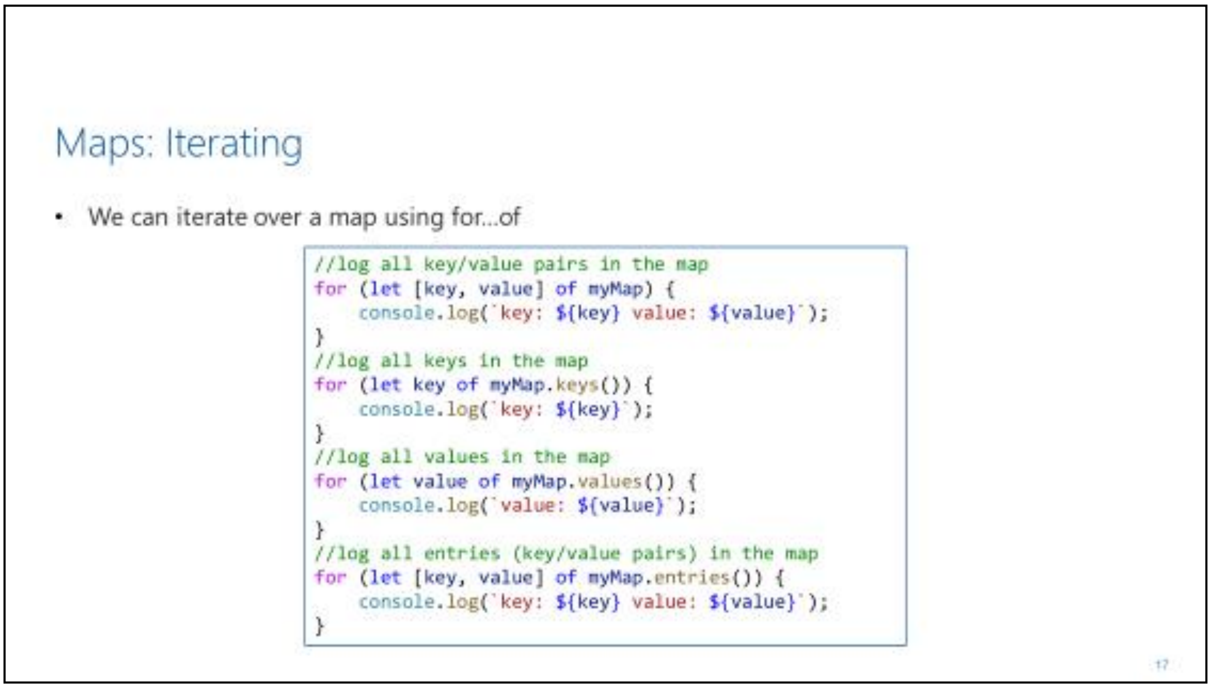
Maps

- Key / Value pairs where both Key and Value can be any type

```
let myMap = new Map([[1,"bananas"],[2,"grapefruit"],[3,"apples"]]);
```

- With some helpful methods

```
console.log(myMap.size) //3

myMap.set(4, "strawberries");
console.log(myMap.size); //4

console.log(myMap.get(4)); //"strawberries"
console.log(myMap.has(2)); //true

myMap.delete(3);
console.log(myMap.size); //3

myMap.clear();
console.log(myMap.size); //0
```

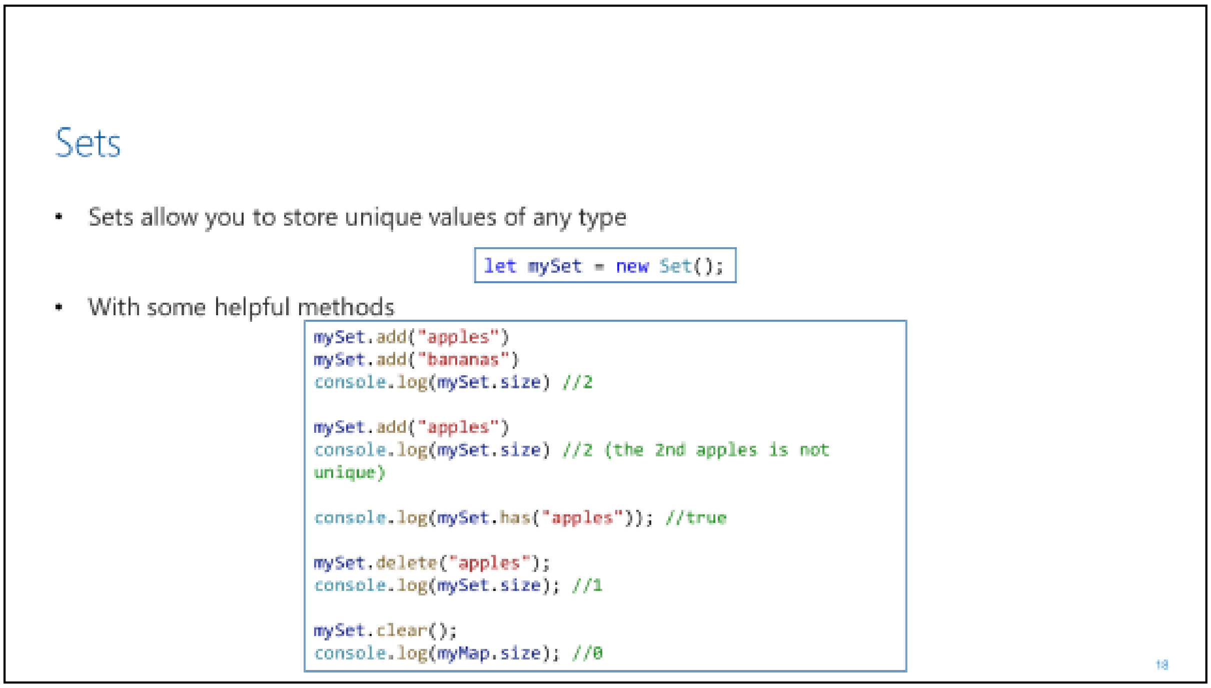


Maps: Iterating

- We can iterate over a map using for...of

```
//log all key/value pairs in the map
for (let [key, value] of myMap) {
    console.log(`key: ${key} value: ${value}`);
}
//log all keys in the map
for (let key of myMap.keys()) {
    console.log(`key: ${key}`);
}
//log all values in the map
for (let value of myMap.values()) {
    console.log(`value: ${value}`);
}
//log all entries (key/value pairs) in the map
for (let [key, value] of myMap.entries()) {
    console.log(`key: ${key} value: ${value}`);
}
```

17



Sets

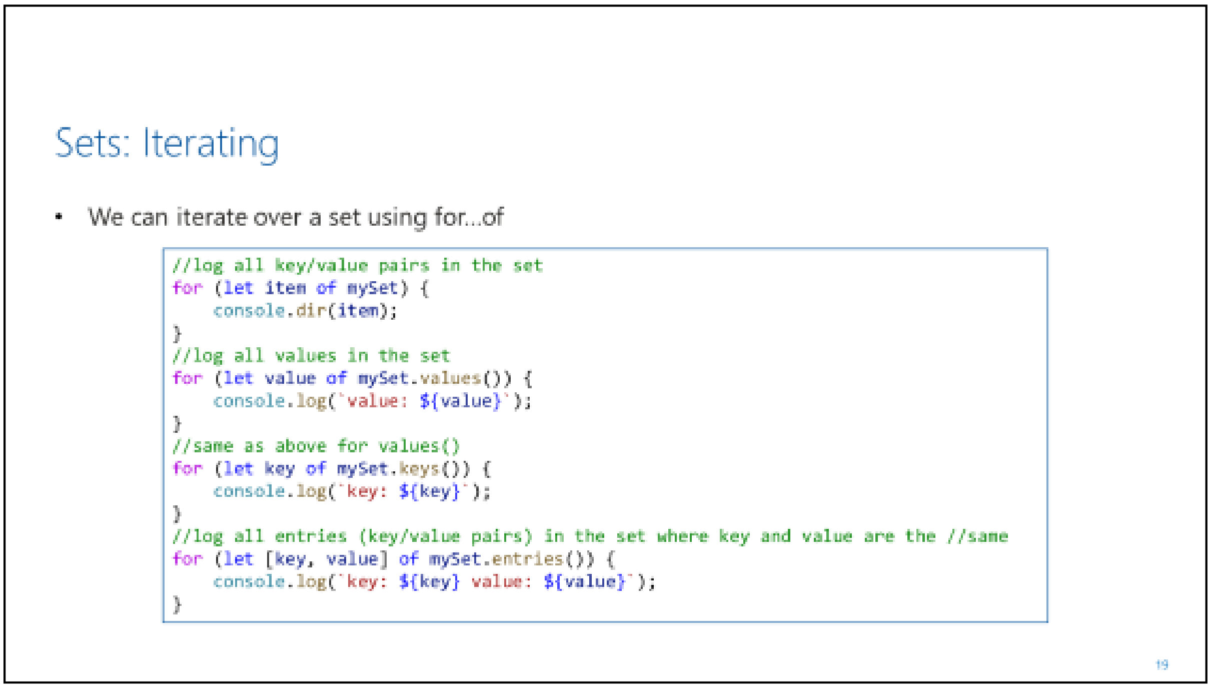
- Sets allow you to store unique values of any type

```
let mySet = new Set();
```

- With some helpful methods

```
mySet.add("apples")
mySet.add("bananas")
console.log(mySet.size) //2

mySet.add("apples")
console.log(mySet.size) //2 (the 2nd apples is not
unique)

console.log(mySet.has("apples")); //true

mySet.delete("apples");
console.log(mySet.size); //1

mySet.clear();
console.log(myMap.size); //0
```

18

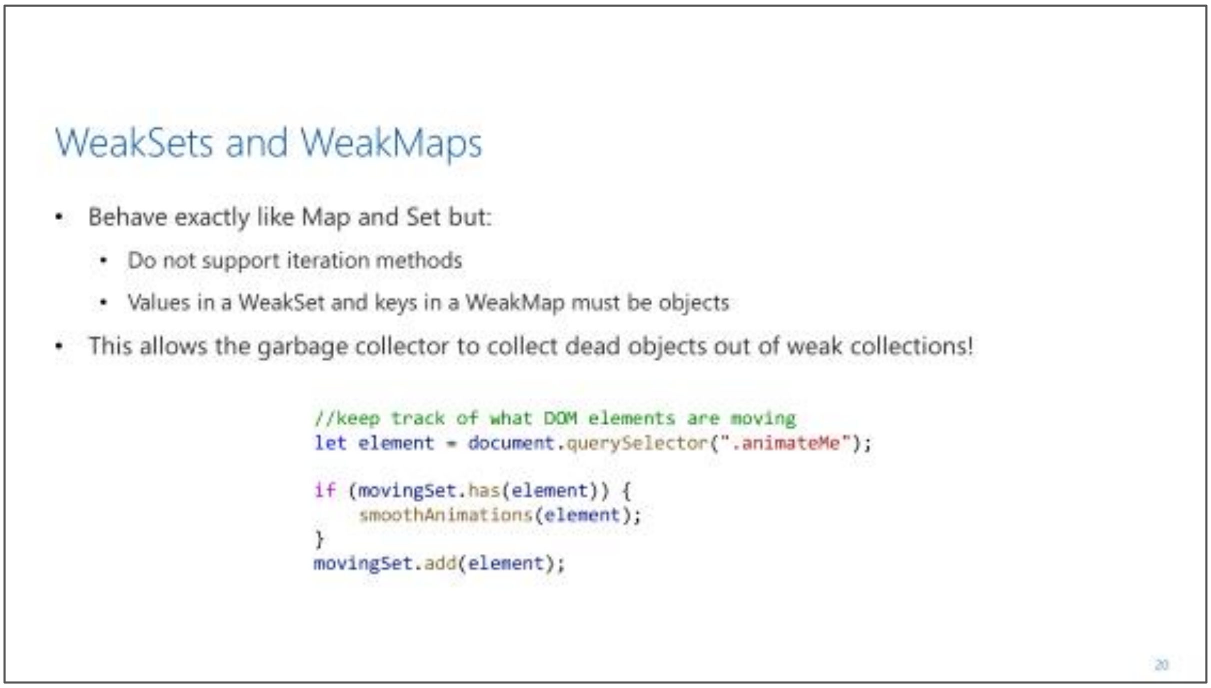


Sets: Iterating

- We can iterate over a set using for...of

```
//log all key/value pairs in the set
for (let item of mySet) {
    console.dir(item);
}
//log all values in the set
for (let value of mySet.values()) {
    console.log(`value: ${value}`);
}
//same as above for values()
for (let key of mySet.keys()) {
    console.log(`key: ${key}`);
}
//log all entries (key/value pairs) in the set where key and value are the //same
for (let [key, value] of mySet.entries()) {
    console.log(`key: ${key} value: ${value}`);
}
```

19

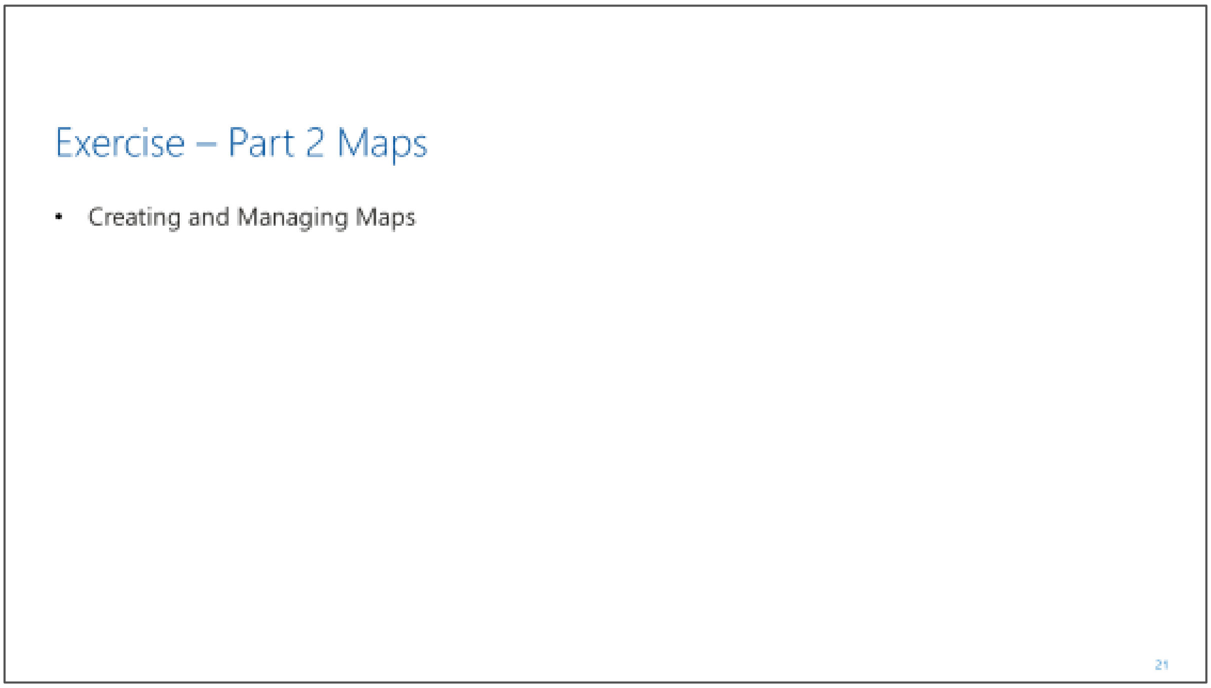


WeakSets and WeakMaps

- Behave exactly like Map and Set but:
  - Do not support iteration methods
  - Values in a WeakSet and keys in a WeakMap must be objects
- This allows the garbage collector to collect dead objects out of weak collections!

```
//keep track of what DOM elements are moving
let element = document.querySelector(".animateMe");

if (movingSet.has(element)) {
    smoothAnimations(element);
}
movingSet.add(element);
```

20



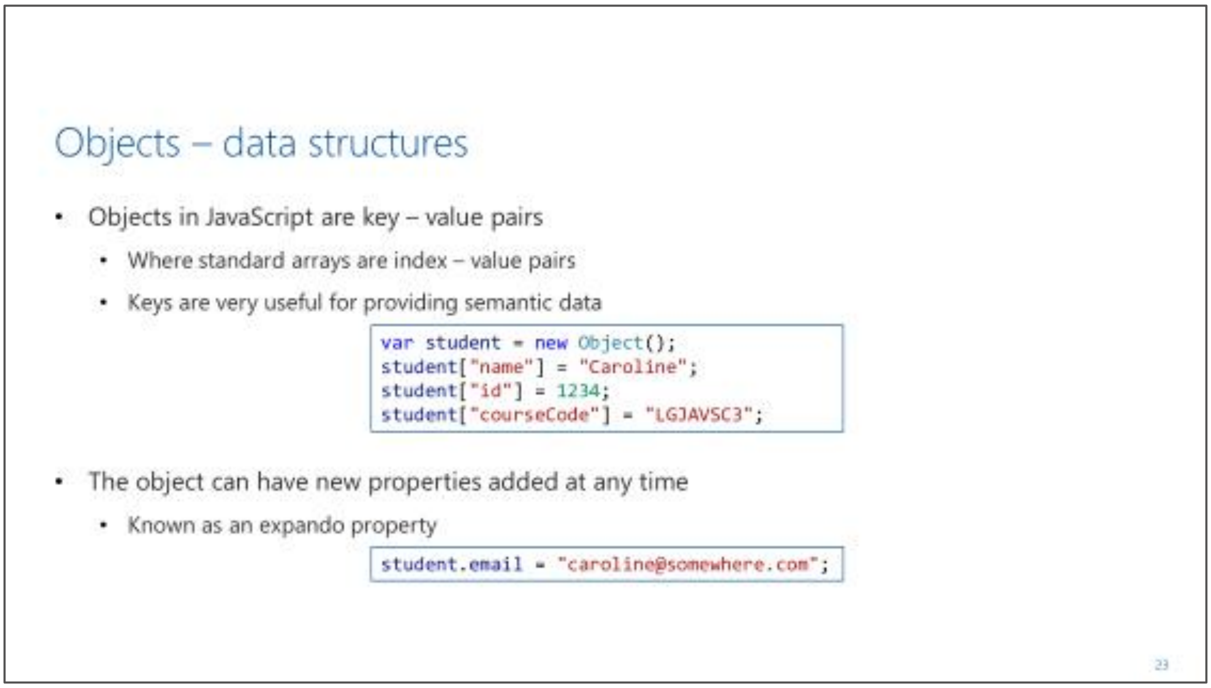
Exercise – Part 2 Maps

- Creating and Managing Maps

21

Objects

PROGRAMMING WITH JAVASCRIPT



Objects – data structures

- Objects in JavaScript are key – value pairs
  - Where standard arrays are index – value pairs
  - Keys are very useful for providing semantic data

```
var student = new Object();
student["name"] = "Caroline";
student["id"] = 1234;
student["courseCode"] = "LGJAVSC3";
```

- The object can have new properties added at any time
  - Known as an expando property

```
student.email = "caroline@somewhere.com";
```

23

Many programming paradigms can be used within JavaScript, one of which is object oriented programming. It provides a series of input objects we will shortly be examining that include window and document and their numerous offspring, which are very important – but they are defined by the browser, not by the programmer. We can define our own objects.

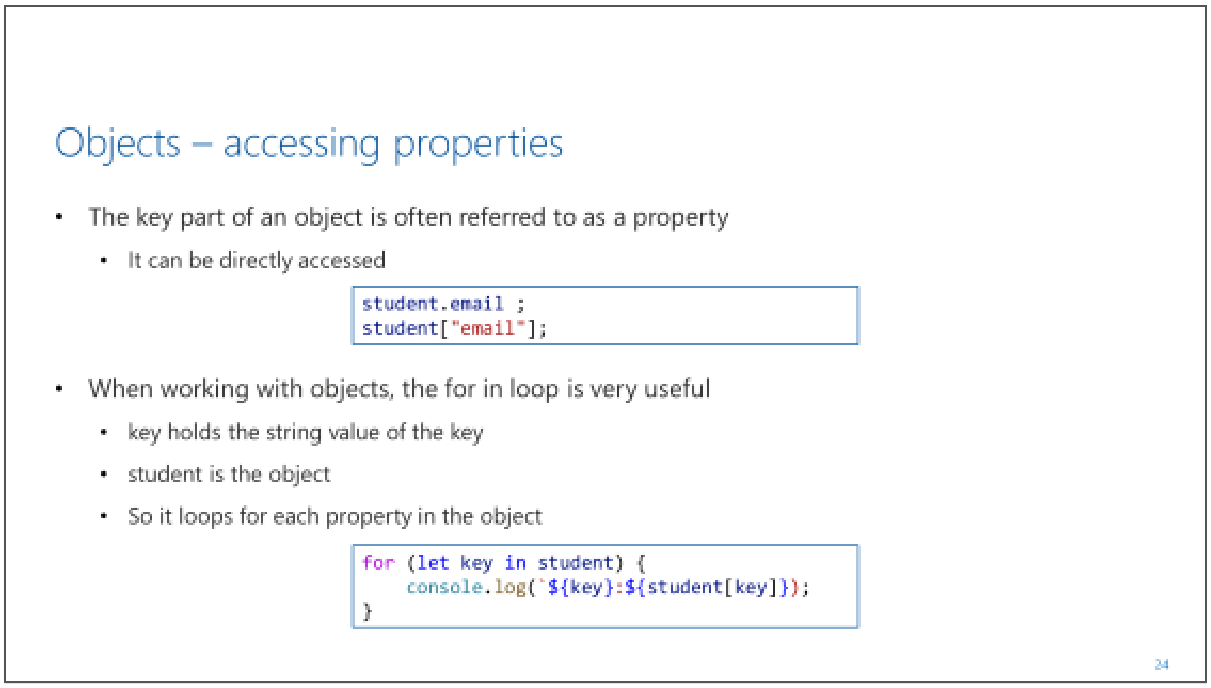
Objects are principal objects in JavaScript. If the variable does not refer to a primitive type, it is an object of some kind. In principal, they are very similar to the concepts of arrays but, instead of an indexed identifier, a string-based key is used (in fact – Arrays in JavaScript are nothing more than special objects).

the property…

```
student["name"]
```

can also be read or written by calling:

```
student.name
```



Objects – accessing properties

- The key part of an object is often referred to as a property
  - It can be directly accessed

```
student.email ;
student["email"];
```

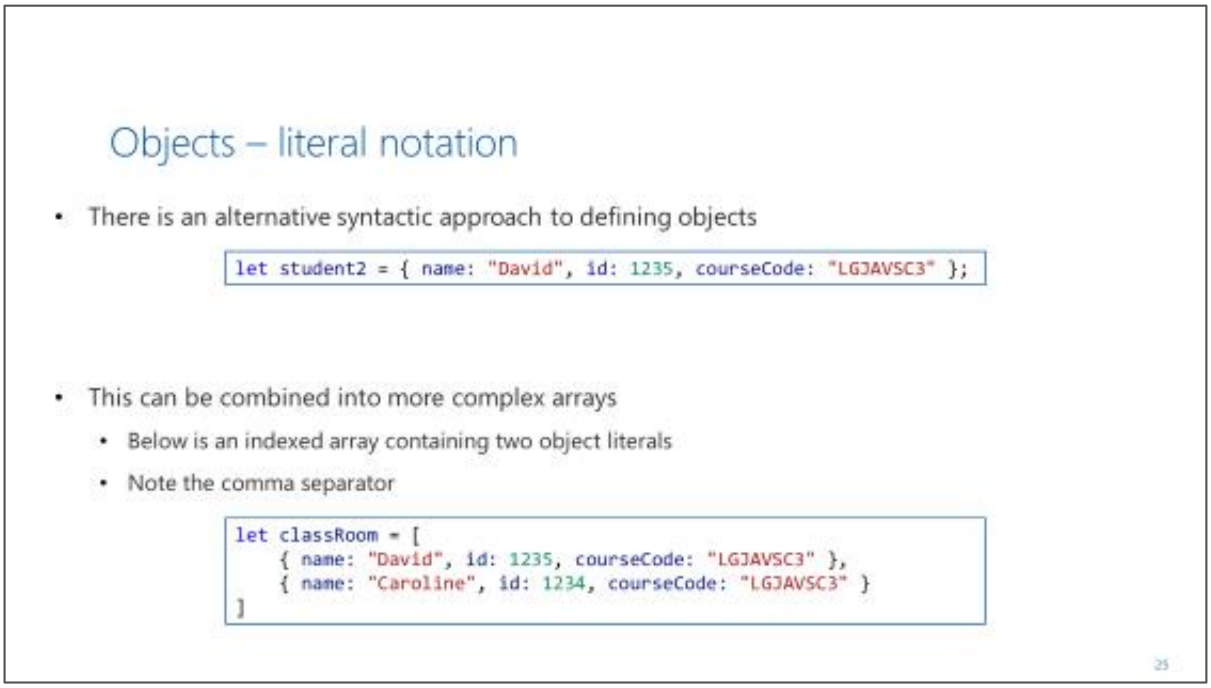
- When working with objects, the for in loop is very useful
  - key holds the string value of the key
  - student is the object
  - So it loops for each property in the object

```
for (let key in student) {
    console.log(`${key}:${student[key]}`);
}
```

24

Objects are collections of properties and every property gets its own standard set of internal properties. (We can think of these as abstract properties – they are used by the JavaScript engine but aren't directly accessible to the user. ECMAScript uses the [[*property*]] format to denote internal properties).

One of these properties is [[Enumerable]]. The for-in statement will iterate over every property for which the value of [[Enumerable]] is true. This includes enumerable properties inherited via the prototype chain. Properties with an [[Enumerable]] value of false, as well as *shadowed* properties – i.e. properties that are overridden by same-name properties of descendant objects – will not be iterated.



Objects – literal notation

- There is an alternative syntactic approach to defining objects

```
let student2 = { name: "David", id: 1235, courseCode: "LGJAVSC3" };
```

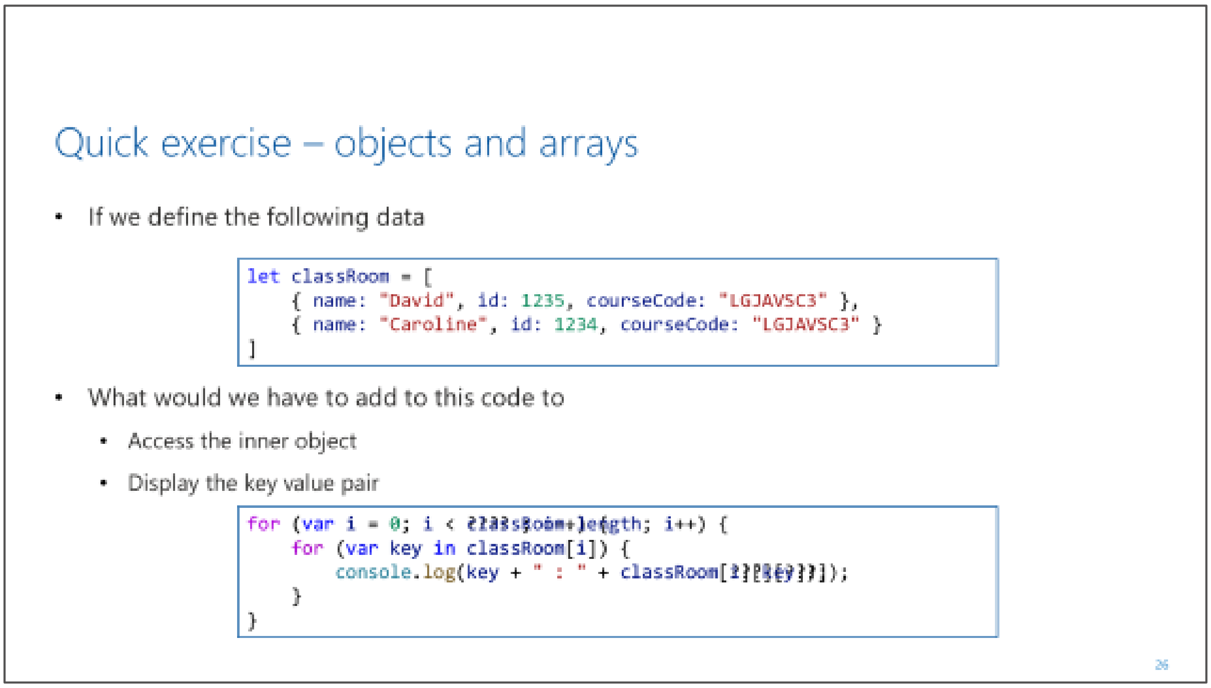
- This can be combined into more complex arrays
  - Below is an indexed array containing two object literals
  - Note the comma separator

```
let classRoom = [
    { name: "David", id: 1235, courseCode: "LGJAVSC3" },
    { name: "Caroline", id: 1234, courseCode: "LGJAVSC3" }
]
```

25

The above code is a quick implementation for JavaScript object. It initialises the object and sets three properties.

The second example creates an indexed array of object literals



Quick exercise – objects and arrays

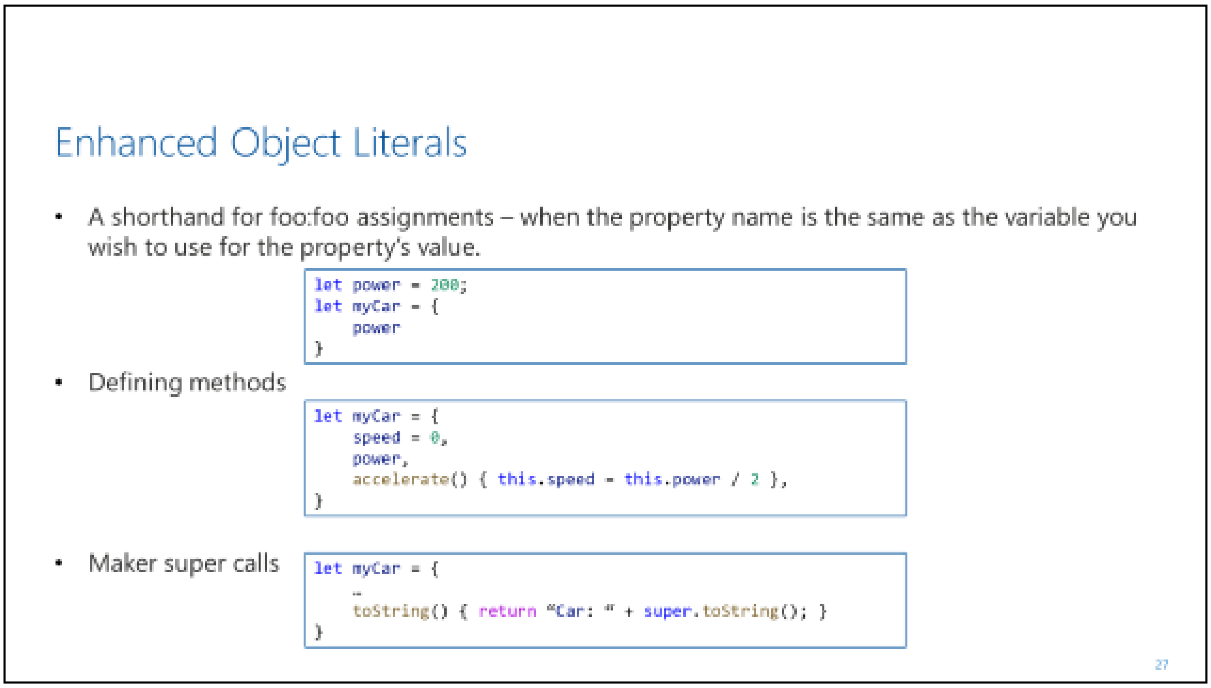
- If we define the following data

```
let classRoom = [
    { name: "David", id: 1235, courseCode: "LGJAVSC3" },
    { name: "Caroline", id: 1234, courseCode: "LGJAVSC3" }
]
```

- What would we have to add to this code to
  - Access the inner object
  - Display the key value pair

```
for (var i = 0; i < classRoom.length; i++) {
    for (var key in classRoom[i]) {
        console.log(key + " : " + classRoom[i][key]);
    }
}
```

26



Enhanced Object Literals

- A shorthand for foo:foo assignments – when the property name is the same as the variable you wish to use for the property's value.

```
let power = 200;
let myCar = {
    power
}
```

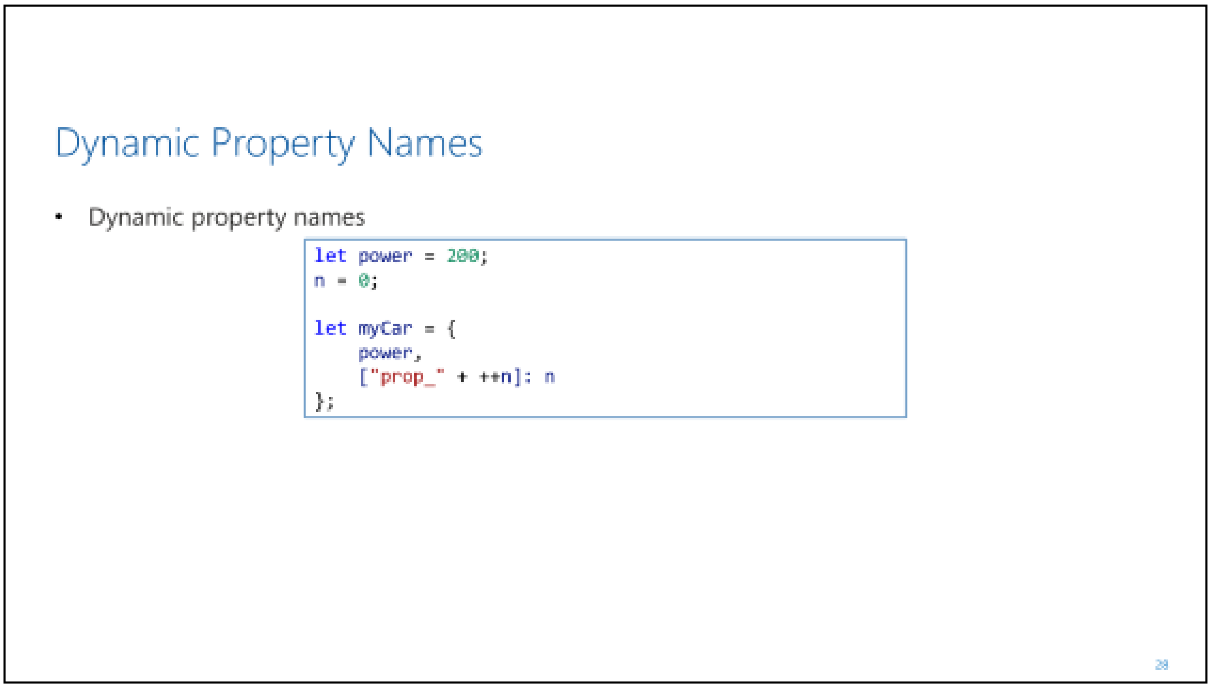
- Defining methods

```
let myCar = {
    speed = 0,
    power,
    accelerate() { this.speed = this.power / 2 },
}
```

- Maker super calls

```
let myCar = {
    …
    toString() { return "Car: " + super.toString(); }
}
```

27

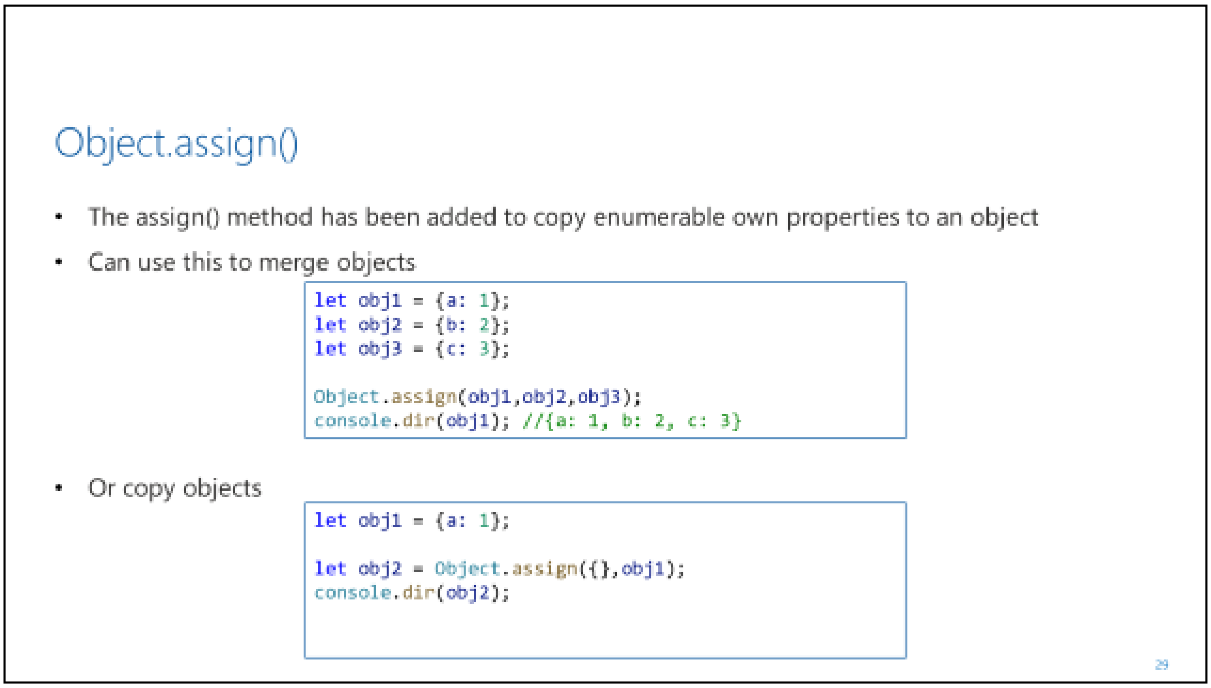


Dynamic Property Names

- Dynamic property names

```
let power = 200;
n = 0;

let myCar = {
    power,
    ["prop_" + ++n]: n
};
```

28



Object.assign()

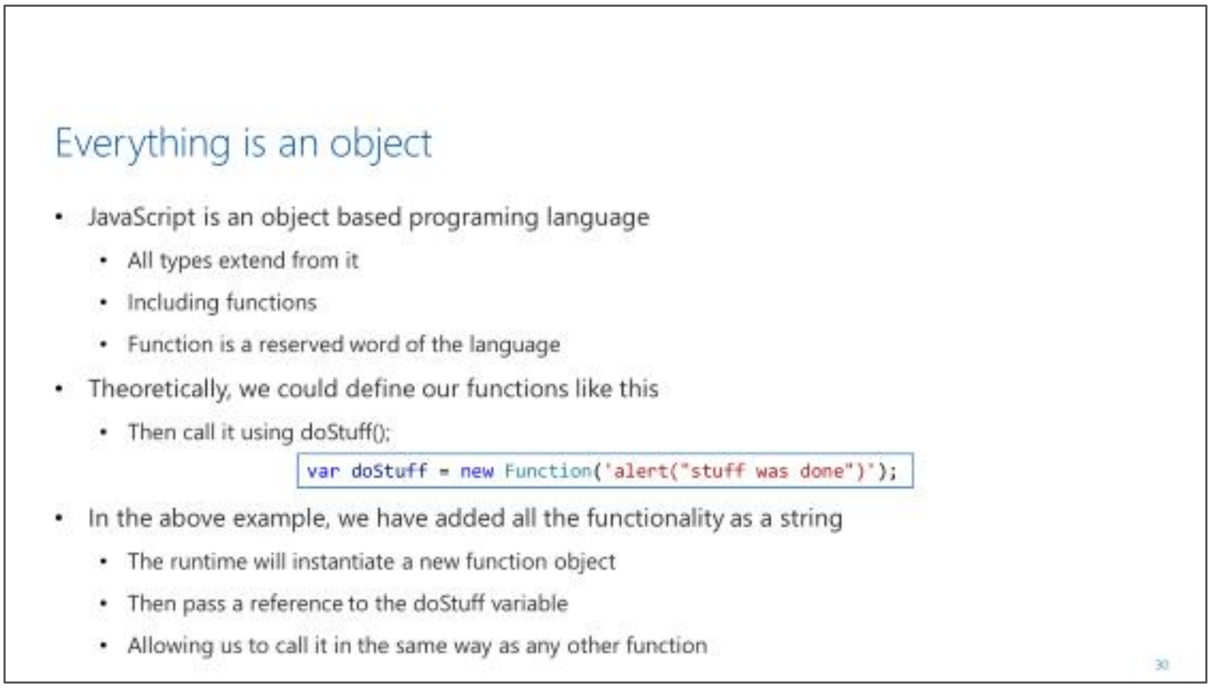
- The assign() method has been added to copy enumerable own properties to an object
- Can use this to merge objects

```
let obj1 = {a: 1};
let obj2 = {b: 2};
let obj3 = {c: 3};

Object.assign(obj1,obj2,obj3);
console.dir(obj1); //{a: 1, b: 2, c: 3}
```

- Or copy objects

```
let obj1 = {a: 1};

let obj2 = Object.assign({},obj1);
console.dir(obj2);
```

29



Everything is an object

- JavaScript is an object based programing language
  - All types extend from it
  - Including functions
  - Function is a reserved word of the language
- Theoretically, we could define our functions like this
  - Then call it using doStuff();

```
var doStuff = new Function('alert("stuff was done")');
```

- In the above example, we have added all the functionality as a string
  - The runtime will instantiate a new function object
  - Then pass a reference to the doStuff variable
  - Allowing us to call it in the same way as any other function

30

Objects are the building blocks of the JavaScript language. In fact, it is defined as an *object based programming language*. Absolutely everything we work with in JavaScript has an object at its core. Consider the following code:

```
var x = 5;

var y = new Number(5);
```

Both code implementations create an object of type number and the variable then receives a memory reference to that object. It is important to remember that JavaScript variables are simply pointers to this object.

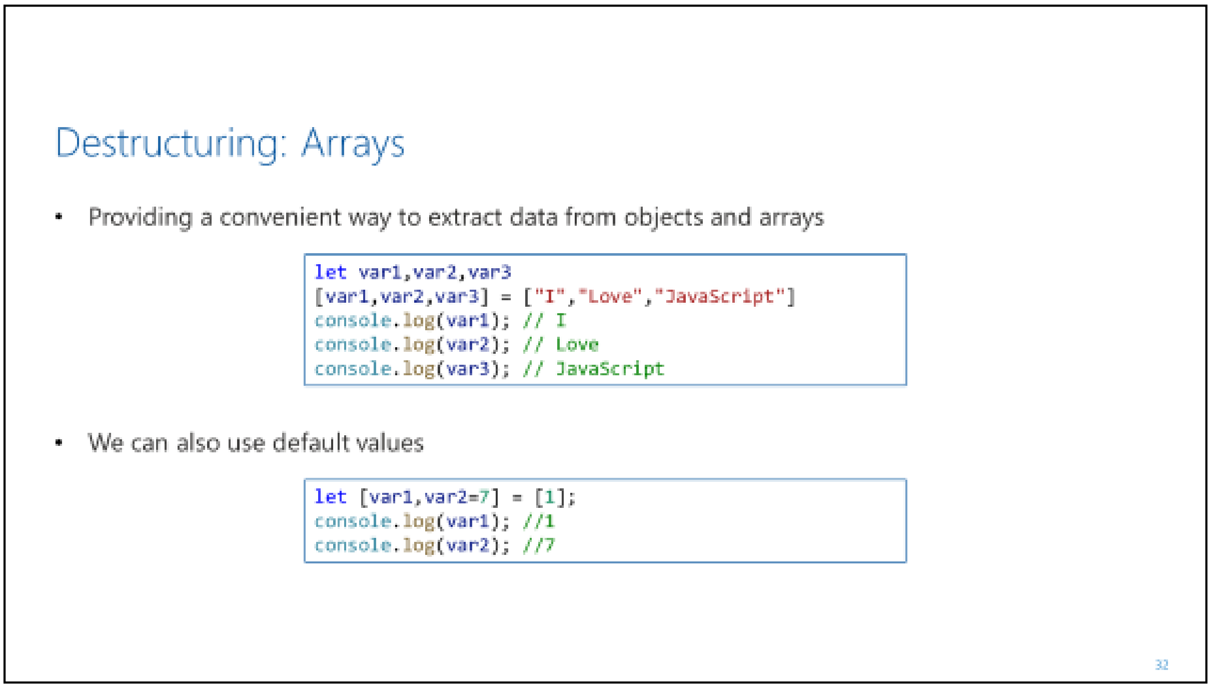
This concept extends to functions. In the code block above, we see another way of creating a function. The **new** keyword instantiates a function, with the logic passed in as a string. **doStuff** receives a reference to the function. As long as the variable **doStuff** remains in scope, the function remains available.

(N.B. THIS IS A THEORETICAL APPROACH! Never implement functions like this! Later in the course your instructor will show you a pattern called self-executing functions that create a security vulnerability of incredible risk.

Destructuring

NEXT GENERATION JAVASCRIPT: ECMASCRIPT 2015

QA



Destructuring: Arrays

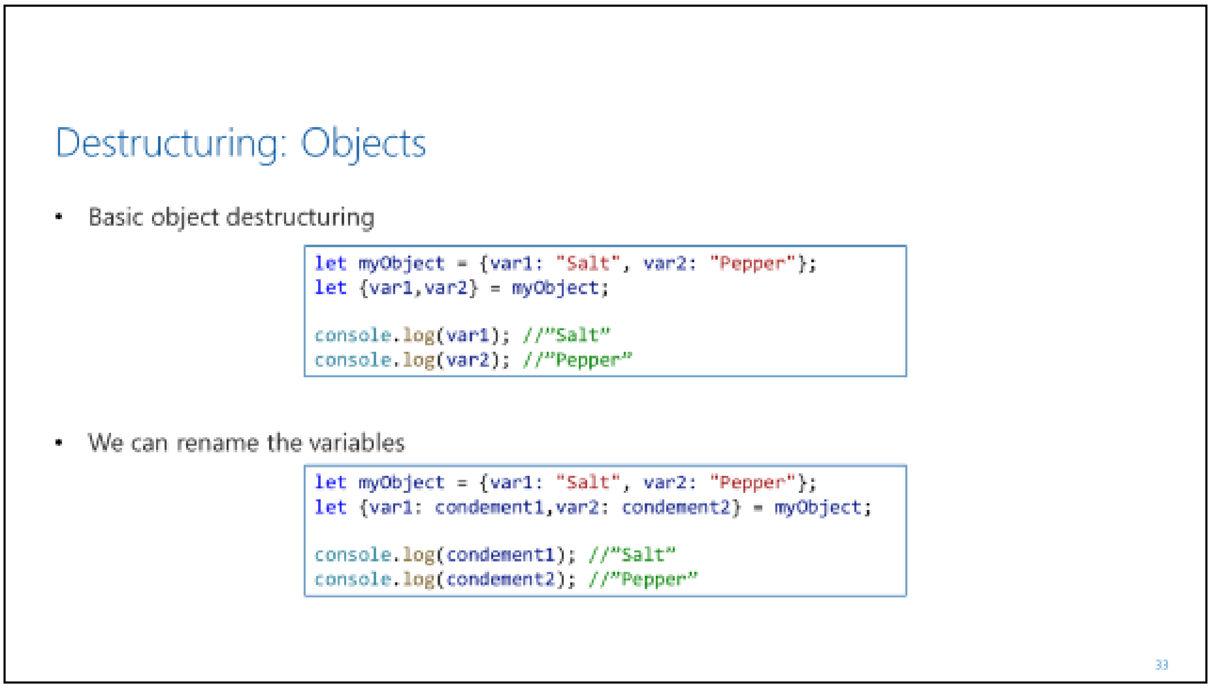
- Providing a convenient way to extract data from objects and arrays

```
let var1,var2,var3
[var1,var2,var3] = ["I","Love","JavaScript"]
console.log(var1); // I
console.log(var2); // Love
console.log(var3); // JavaScript
```

- We can also use default values

```
let [var1,var2=7] = [1];
console.log(var1); //1
console.log(var2); //7
```

32



Destructuring: Objects

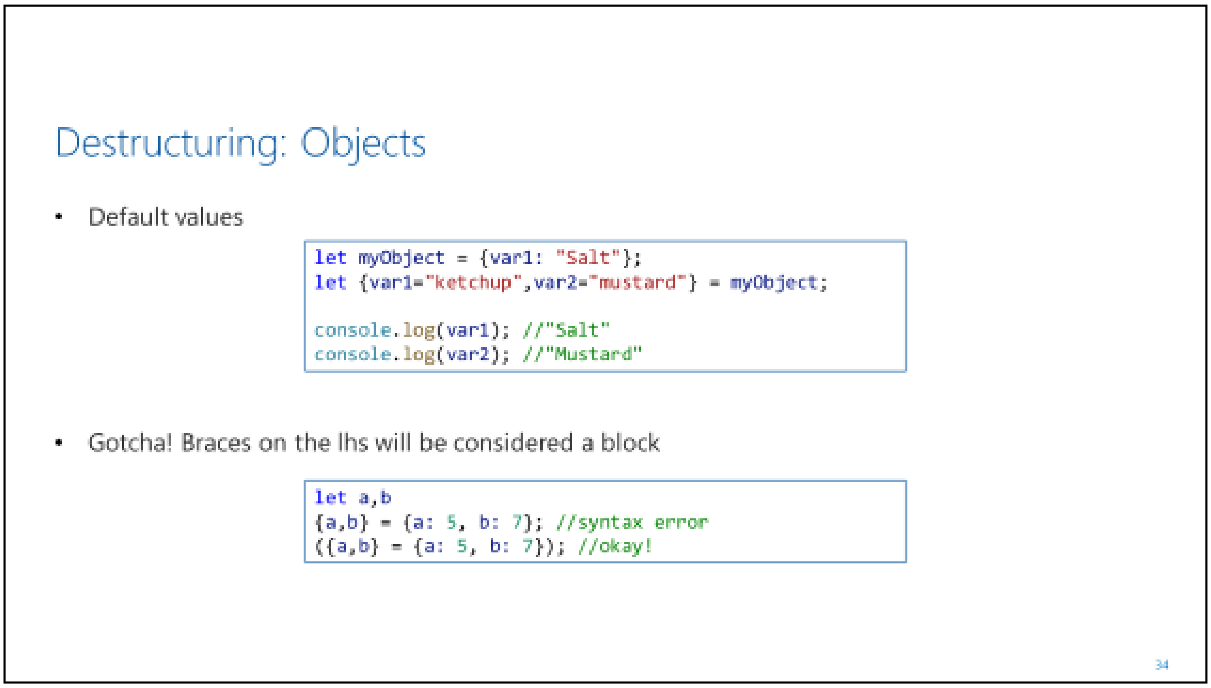
- Basic object destructuring

```
let myObject = {var1: "Salt", var2: "Pepper"};
let {var1,var2} = myObject;

console.log(var1); //"Salt"
console.log(var2); //"Pepper"
```

- We can rename the variables

```
let myObject = {var1: "Salt", var2: "Pepper"};
let {var1: condement1,var2: condement2} = myObject;

console.log(condement1); //"Salt"
console.log(condement2); //"Pepper"
```

33



Destructuring: Objects

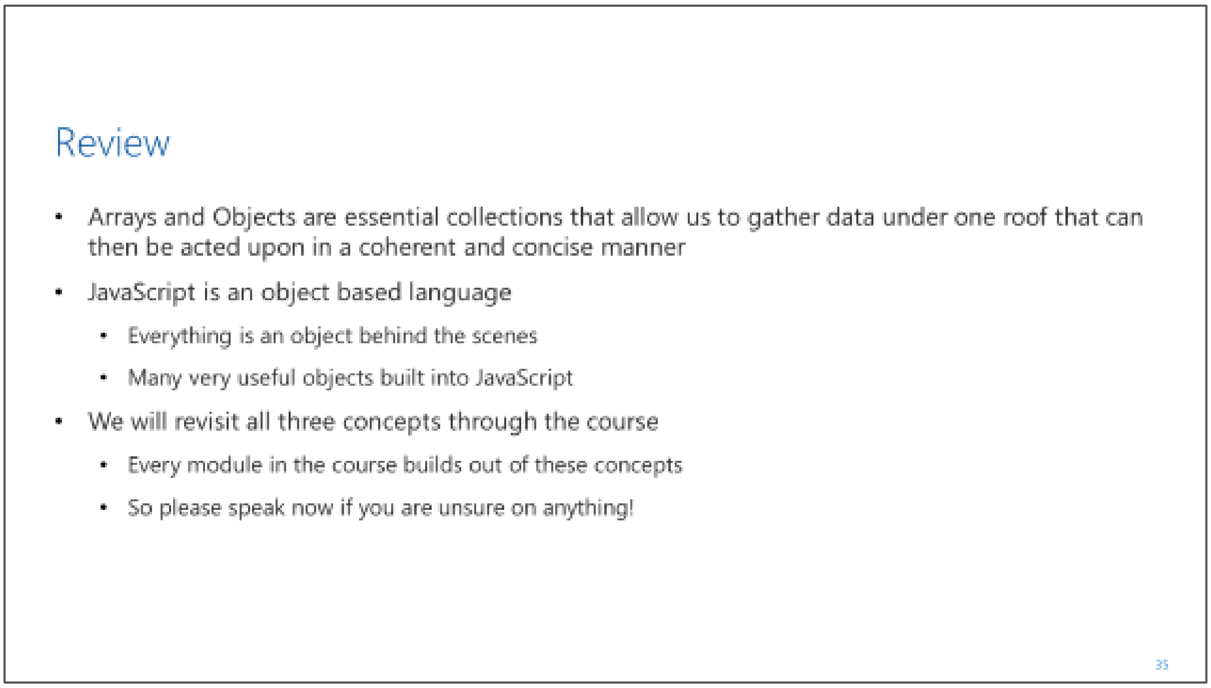
- Default values

```
let myObject = {var1: "Salt"};
let {var1="ketchup",var2="mustard"} = myObject;

console.log(var1); //"Salt"
console.log(var2); //"Mustard"
```

- Gotcha! Braces on the lhs will be considered a block

```
let a,b
{a,b} = {a: 5, b: 7}; //syntax error
({a,b} = {a: 5, b: 7}); //okay!
```

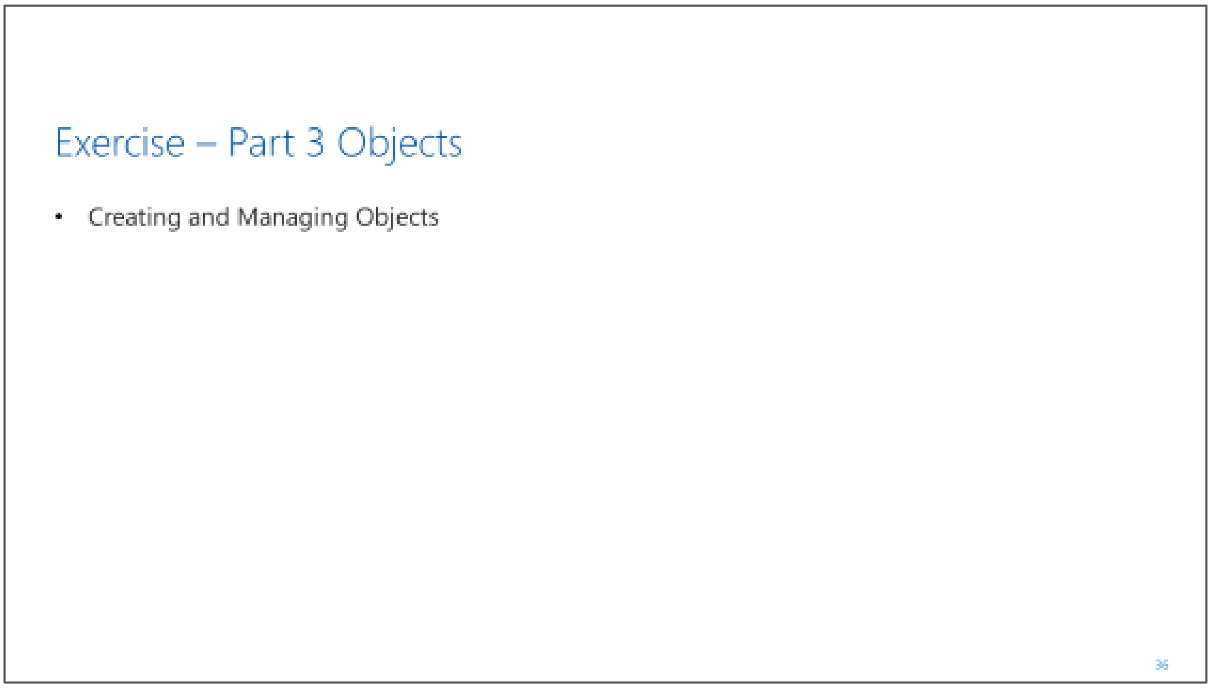
34



Review

- Arrays and Objects are essential collections that allow us to gather data under one roof that can then be acted upon in a coherent and concise manner
- JavaScript is an object based language
  - Everything is an object behind the scenes
  - Many very useful objects built into JavaScript
- We will revisit all three concepts through the course
  - Every module in the course builds out of these concepts
  - So please speak now if you are unsure on anything!

35



Exercise – Part 3 Objects

- Creating and Managing Objects

36