

# Week 3 - ECMAScript

Dr Kiki Adhinugraha

# Outline

- What is ECMAScript?
- Variable Scoping
- Array
- Objects
- Callbacks
- Promises
- Additional Resources
  - Please visit the links provided in this lecture note

# What is ECMAScript?

- ECMA stands for European Computer Manufacturer Association.
- ECMA is the association that puts out the guidelines for JavaScript in all browsers. So, what you know as Javascript is also called ECMAScript.
- ECMAScript 7, also known as ECMAScript 2016, is the version of the ECMAScript standard. Before ES7 was finalized, ES6 was a significant update to the language, and the first update to the language since ES5 was standardized in 2009.
- JavaScript runs code sequentially in top-down order

# Javascript IS NOT Java

- Javascript does not really have anything to do with Java.
- Some of the syntax of Javascript is the same as Java but that is true of almost any other language.

# JS for web development

- Client-Side Interactivity
- DOM Manipulation
- Event Handling
- Asynchronous Operations
- Cross-Browser Compatibility
- Frameworks and Libraries
- Server-Side Development

# Where to put Javascript Code

- Javascriptcode is placed between `<script>` and `</script>` for HTML5

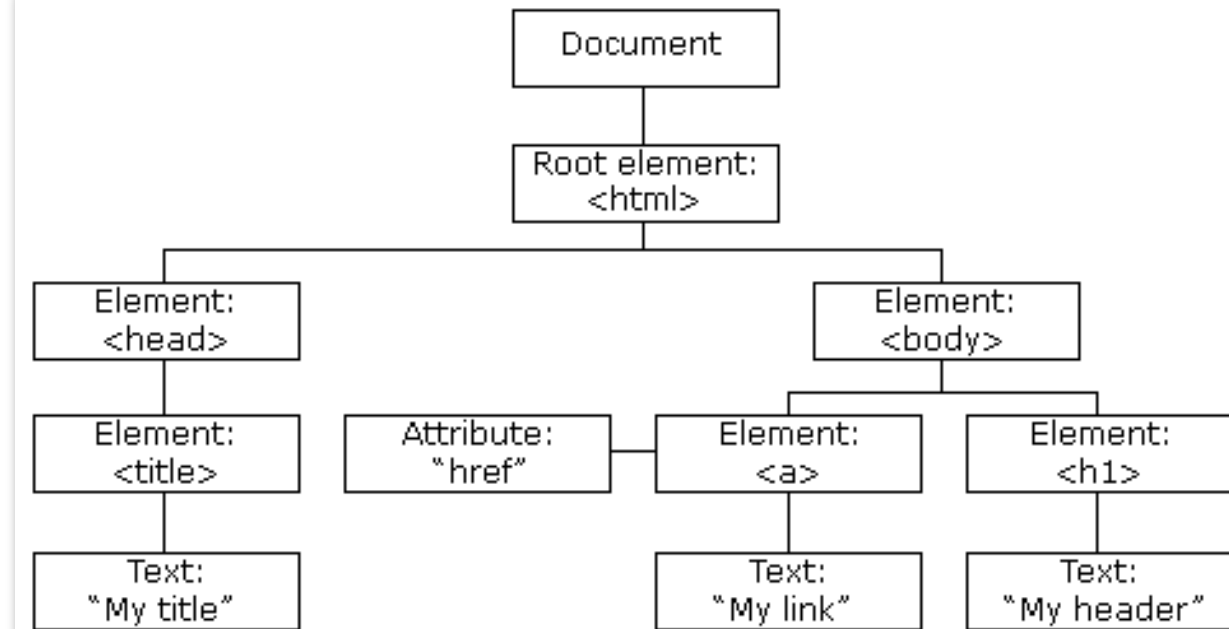
```
<script>  
    ... JavaScript statements ...  
</script>
```

- For HTML 4.x or XHTML the following is used

```
<script type="text/javascript">  
    ... JavaScript statements ...  
</script>
```

# The DOM

- Refer to Week 03A – HTML CSS lecture notes.
- The Document Object Model (DOM) is a representation of the whole document as nodes and attributes of a tree
- Each element in the HTML document is represented by a node.
- You can traverse the tree to access each of the nodes and attributes.
- You can add, remove, change any of the nodes or attributes in the tree.



# The DOM

- Each time your browser is asked to load and display a page, it needs to parse the source code contained in the HTML file comprising the page.
- As part of this parsing process the browser creates the DOM which it uses as its internal representation of the web page.
- The browser uses the DOM to render the web page.
- Using Javascript you can edit the DOM



# Finding HTML Elements

- We can use the following methods of the document DOM object to find HTML Elements.

- The following example finds all elements with ID of demo

```
document.getElementById("demo");
```

- The following example finds all elements with tag <h1>

```
document.getElementsByTagName("h1");
```

- The following example finds all elements with class name of cities.

```
document.getElementsByClassName("cities");
```

# Changing HTML Elements

- `element.innerHTML=new html content`
  - Change the inner HTML of an element
  - E.g. `document.getElementById("demo").innerHTML= "Hello";`
- `element.attribute= new value`
  - Change the attribute value of an HTML element
  - E.g. `document.getElementById("homepageLink").href= "http:home.com";`
- `element.setAttribute(attribute, value)`
  - Change the attribute value of an HTML element
  - E.g. `document.getElementById("demo").setAttribute("class", "democlass");`
- `element.style.property= new style`
  - Change the style of an HTML element
  - E.g. `document.getElementById("demo").style.color= "blue";`

## Changing an HTML element via ID

- myFunction is triggered when the button is clicked
- <p> element with id "demo" changed by the javascript function myFunction().
- Example in file1.html

Initial state:

Click the button to change the text in this paragraph.

Try it

After clicking Try it:

Hello World

Try it

```
<!DOCTYPE html>
<html>
<body>

<p id="demo">Click the button to change the text in this paragraph.</p>
<button onclick="myFunction()">Try it</button>

<script>
function myFunction() {
    document.getElementById("demo").innerHTML = "Hello World";
}
</script>

</body>
</html>
```

## Changing an HTML elements via ClassName

- In this example all three <p> elements with class test are changed by the Javascript script to the words Good Job!
- Notice that the `getElementsByClassName` function returns an array of elements with the specified class name

Good Job!

Good Job!

Good Job!



```
<!DOCTYPE html>
<html>
<body>
<p class="test">Use the DOM to change my text!</p>
<p class="test">Use the DOM to change my text2!</p>
<p class="test">Use the DOM to change my text3!</p>
</body>

<script>
var elemArray = document.getElementsByClassName("test");
for (var i = 0; i < 3; i++){
    elemArray[i].innerHTML = "Good Job!";
}
</script>

</html>
```

# Adding and Deleting HTML Elements

- `document.createElement(element)`
  - Create an HTML element
  - E.g `document.createElement("p");`

- `document.removeChild(element)`
  - Remove an HTML element
  - E.g.

```
var parent = document.getElementById("div1");  
var child = document.getElementById("p1");  
parent.removeChild(child);
```

- E.g: File4.html

# Adding and Deleting HTML Elements

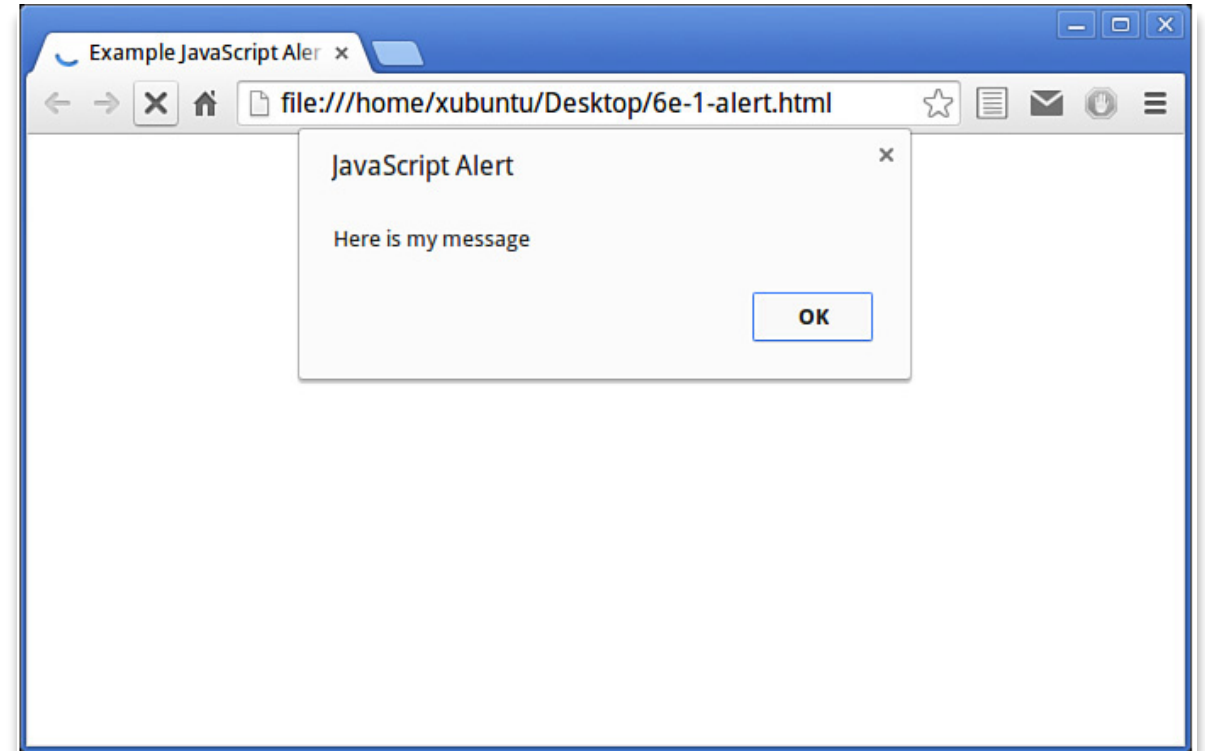
- `document.appendChild(element)`
  - Add an HTML element
- `document.replaceChild(element)`
  - Replace an HTML element
  - E.g.

```
var para = document.createElement("p");  
var node = document.createTextNode("This is new.");  
para.appendChild(node);
```

```
var parent = document.getElementById("div1");  
var child = document.getElementById("p1");  
parent.replaceChild(para, child);
```

# Talking to the User

- Popup a window
  - `window.alert("Here is a message")` or just `alert("Here is a message")` since window is at the top of the hierarchy.



# Event Handlers

- Javascript can be executed when an event occurs, like when a user clicks on an HTML element.
- Code

```
<!DOCTYPE html>  
<html>  
<body>  
  
<h1 onclick="this.innerHTML='Ooops!'">Click on this text!</h1>  
  
</body>  
</html>
```

**Click on this text!**  **Ooops!**



# Common HTML Events

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
Onload	The browser has finished loading the page

[https://www.w3schools.com/tags/ref\\_eventattributes.asp](https://www.w3schools.com/tags/ref_eventattributes.asp)

# Event Listeners

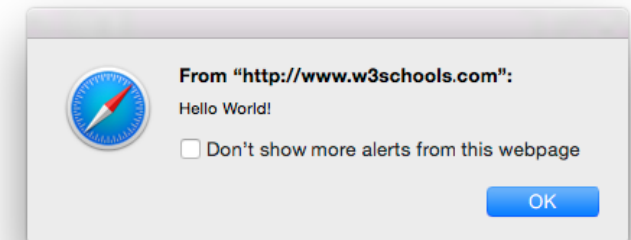
- Using event listeners, an event can trigger more than one event.
- The following code pops up two windows when the button is clicked.

```
<!DOCTYPE html>
<html>
<body>
<p>This example uses the addEventListener() method to add two click events to the same button.</p>
<button id="myBtn">Try it</button>
<script>
var x = document.getElementById("myBtn");
x.addEventListener("click", myFunction);
x.addEventListener("click", someOtherFunction);
function myFunction() {
    alert ("Hello World!");
}
function someOtherFunction() {
    alert ("This function was also executed!");
}
</script>
</body>
</html>
```

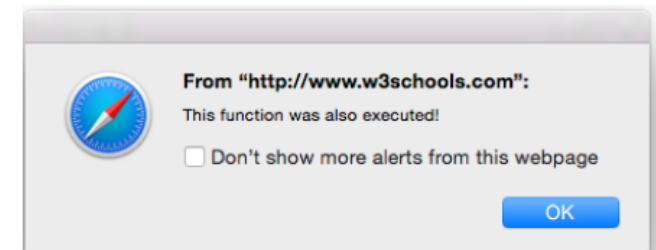
After clicking button:

This example uses the addEventListener() method to add two click events to the same button.

Try it



After clicking OK on 1<sup>st</sup> alert:



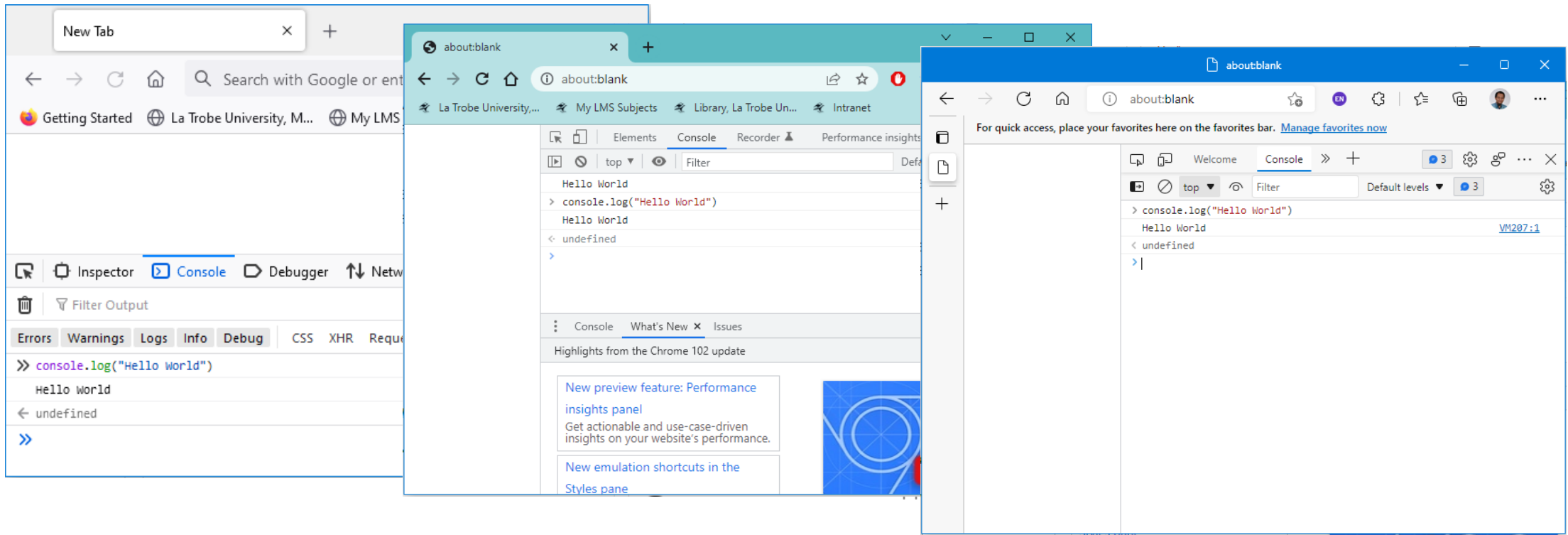
# Javascript Foundations

- Variables
- Data Types
- Type Conversion
- Loops
- Functions
- Scope
- Arrays

[https://www.w3schools.com/whatis/whatis\\_js.asp](https://www.w3schools.com/whatis/whatis_js.asp)

# Web Browser Devtools

- Press F12 will bring the Devtools that can be used to test JS functionality without using any programming IDE



# Variables

- JavaScript variables are containers for storing data values.

```
var x = 5;
```

```
var y = 6;
```

```
var z = x + y;
```

# Data Types

- Javascript is a dynamically typed language. This means the same variable can have many different types

- E.g.

```
var x ;  
var x = 5;  
var x = "John";
```

- When adding a number and a string, Javascript will treat the number as a string

- E.g.

```
var x= 16 + "Volvo";
```

- x will have the value "16Volvo"

# Data Types

- Javascript Strings

- E.g.

- ```
var carName= "Volvo XC60";
```

- Numbers can be written with decimal or without decimal or with scientific notation.

- E.g.

- ```
var x1 = 34.00;
```

- ```
var x2 = 34;
```

- ```
var y = 123e5;
```

- ```
var z = 123e-5;
```

- Javascript Booleans

- E.g.

- ```
var x = true;
```

- ```
var y = false;
```

# Type Conversion

- **Converting Numbers to String.** The global method `String()` can be used to convert numbers to strings.
  - `String(x)`
  - `String(123)`
  - `String(100 + 23)`
- **Another way to do the same is the following**
  - `x.toString()`
  - `(123).toString()`
  - `(100 + 23).toString()`
- **Converting Strings to Numbers**
  - `Number("3.15")`
  - `Number(" ")` // returns 0
  - `Number("99 88")` // return NaN



# Loops

- Very similar to Java
- Example of a for loop

```
for (i = 0; i < cars.length; i++) {  
    text += cars[i] + "<br>";  
}
```

- Example of a while loop

```
while (i < 10) {  
    text += "The number is " + i;  
    i++;  
}
```

# Operators

| Operator | Description    |
|----------|----------------|
| +        | Addition       |
| -        | Subtraction    |
| *        | Multiplication |
| /        | Division       |
| %        | Modulus        |
| ++       | Increment      |
| --       | Decrement      |

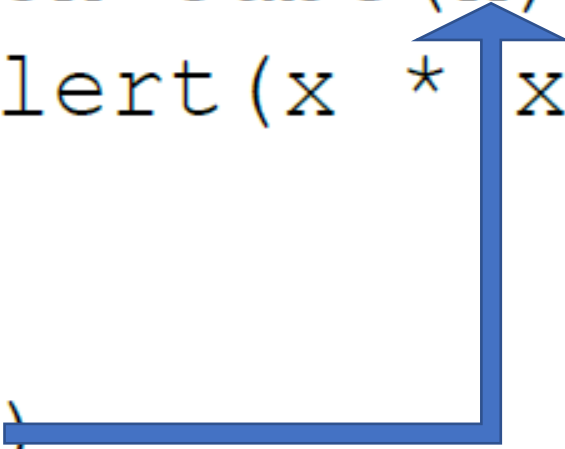
# Functions

- Using functions allows us to write the Javascript code inside the head of the document.
- In this example when the button is clicked the sayHello() function is called.

```
<!DOCTYPE html>
<html>
<head>
  <title>Calling Functions</title>
  <script>
    function sayHello() {
      alert("Hello");
    }
  </script>
</head>
<body>
  <input type="button" value="Say Hello" onclick="sayHello()" />
</body>
</html>
```

# Passing Argument into Functions

```
function cube(x) {  
    alert(x * x * x);  
}  
  
cube(3)
```



A blue arrow originates from the argument '3' in the function call 'cube(3)' and points upwards to the parameter 'x' in the function definition 'function cube(x)'. This illustrates the flow of data from the caller to the function's local scope.

# Function Return

- When Javascript reaches a return statement the function stops executing.
- When a function contains a return value, the return value is returned to the caller.

```
var x = myFunction(4, 3);    // after the function call x will be assigned the value 12
```

```
function myFunction(a, b) {  
    return a * b;           // Function returns the product of a and b  
}
```

# ES6: Arrow Functions

- A new feature of ES 6 is the arrow function.
- Arrow functions allows you to write shorter functions
  - Arrow function (ES6):

```
const addTogether= (a, b) => a + b;
```

- Normal function (old way):

```
const addTogetherOld= function(a, b) {  
  return a + b;  
};
```

- More arrow functions

```
const func= x => x * x;
```

- Concise syntax, implied “return”

```
const func= (x, y) => {return x + y};
```

- Syntax with block body, explicit “return” needed
- We can then use the function like:
- `func(2, 4)`

# Scope

## Local Variable

```
// code here can not use carName

function myFunction() {
    var carName = "Volvo";

    // code here can use carName
}
```

## Global Variable

```
var carName = "Volvo";

// code here can use carName

function myFunction() {

    // code here can use carName
}
```

# Automatic Global Variable

- If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable.
- This code will declare carName as a global variable

```
// code here can use carName

function myFunction() {
    carName = "Volvo";

    // code here can use carName
}
```



# Function Scope not Block Scope

- In Javascript when you declare a variable inside a function using var it has function scope.
- For example in the example below the second declared x is the same variable as the first:

```
function varTest() {  
    var x = 31;  
    if (true) {  
        var x = 71; // same variable!  
        console.log(x); // 71  
    }  
    console.log(x); // 71  
}
```

- This is not nice since you may accidentally change the value of a variable that you did not intend to change.

# ES6:Let Statement used to Define Block Scope

- In JavascriptES6 (latest version). There is a new type of variable called let.
- Using let we can define a variable that has block scope. In the example below the second declaration of x is a new variable.

```
function letTest() {  
    let x= 31;  
    if (true) {  
        let x = 71; // different variable  
        console.log(x); // 71  
    }  
    console.log(x); // 31  
}
```

# ES6:Good place to use Let

- A good place to use Let is in for loops

```
for( let i=0 ; i < 10 ; i++ ) {  
    console.log(i);  
}
```

- In the above example let makes the variable i have local scope within the for loop

# ES6: const

- In ES 6 the const variable is declared.
- A const variable must be initialized to a value and any subsequent reassignment of its value will be ignored.
- The example below will print 7 since the second assignment to 20 is not allowed.

```
const My_FAV= 7;  
My_FAV= 20;  
document.write(My_FAV)
```

# Variable Scoping


- Var
  - ES5
  - Global if declared outside a function
  - Local if declared in a function
  - Should be restricted when writing JS in HTML
- Let
  - ES6
  - Limited to blocked-scope
  - Used for temporary variables
- Const
  - ES6
  - Readonly Constant
  - Do not use in mutable object

# Objects

- A car has properties weight and color, and methods like start and stop
- This code assigns many values (Fiat, 500, white) to a variable named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

- The values are written as name:value pairs
- The name:value pairs are called properties

Object	Properties	Methods
	<code>car.name = Fiat</code> <code>car.model = 500</code> <code>car.weight = 850kg</code> <code>car.color = white</code>	<code>car.start()</code> <code>car.drive()</code> <code>car.brake()</code> <code>car.stop()</code>

# Object Methods

- Methods are actions that can be performed on objects.
- Methods are stored in properties as function definitions.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	<code>function() {return this.firstName + " " + this.lastName;}</code>

# Object Methods

- We can first declare a function like the following method :

```
function fullName() {  
    return this.firstName+ " " + this.lastName;  
}
```

- Note the function uses the variable `this` to refer to the object that the function is attached to.
- Then we can declare an object that uses the function like the following

```
const person1 = {  
    firstName: "Peter",  
    lastName: "Smith",  
    completeName: fullName  
};
```

- Then you can call the function `completeName` as follows:

```
person1.completeName();
```



# Accessing Properties

- Two ways to access properties
  - `person.lastName`
  - `person["lastname"]`
- Calling a method
- `name = person.fullName()`

# Updating Properties

```
var stooge = {  
    "first-name" : "Jerome",  
    "last-name"  : "Howard"  
};
```

- Given the above the following updates the “first-name” property to “Peter”
  - `stooge["first-name"] = "Peter"`
- The following will add a new property called “middle-name” into stooge
  - `stooge["middle-name"] = "Lester"`

# Declaring Objects inside Objects

- In this example the properties departure and arrival are objects themselves.

```
var flight = {  
  airline: "Oceanic",  
  number: 815,  
  departure: {  
    IATA: "SYD",  
    time: "2004-09-22 14:55",  
    city: "Sydney"  
  },  
  arrival: {  
    IATA: "LAX",  
    time: "2004-09-23 10:42",  
    city: "Los Angeles"  
  }  
};
```

# Pass By Reference

- Objects are passed around by reference. They are never copied:

```
var stooge = {  
    "first-name" : "Jerome",  
    "last-name" : "Howard"  
};  
var a = { };  
a = stooge;
```

- In the above example object a points to the stooge object.
- So when we change a we also change the stooge object

# Pass by Value

- Simple data type are passed around by the value. They are copied

```
var x = 5;
```

```
var y = x;
```

```
y = 0;
```

- In this example, at line 2, y contains the same value as x
- when y = 0, x still has the original value 5

# Reflection

- The typeof operator can be used to find the type of a property:
  - `typeof flight.status// "string"`
  - `typeof flight.arrival// "object"`

- You can iterate through the all the properties

```
var name
for (name in another_stooge) {
    document.writeln(name + ": " +
another_stooge[name]);
}
```

# Remove Property

- You can remove a property by using the delete operator  
delete another\_stooge.nickname

# Arrays

- Creating an array
  - `var cars = ["Saab", "Volvo", "BMW"];`
- The first element of an array have index 0
  - `var name = cars[0];`
  - `cars[0] = "Opel";`
- In Javascript you have data of different types in the same array
  - `myArray[0] = "peter";`
  - `myArray[1] = 2.2;`
  - `myArray[2] = true;`



# Some Array Methods

- You can find all the JavascriptArray functions with great examples from the following web page:
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

Function	Description
concat()	Concatenates two strings
find()	Return index of array element searched
forEach()	Iterates through an array calling a function on each object
length()	Returns the length of an array
map()	Calls a function on each element of an array
push()	Pushes an element to the end of the array.
sort()	Sorts the elements of an array

# Switch and Break

- You can also use break and continue to exit (break) or skip an iteration of a loop (continue)

```
switch(expression) {  
    case n:  
        code block  
        break;  
    case n:  
        code block  
        break;  
    default:  
        default code block  
}
```

```
for (i = 0; i < 10; i++) {  
    if (i === 3) { break; }  
    text += "The number is " + i + "<br>";  
}  
  
for (i = 0; i < 10; i++) {  
    if (i === 3) { continue; }  
    text += "The number is " + i + "<br>";  
}
```

# Lodash **Lo**

A modern JavaScript utility library delivering modularity, performance & extras.

## Normal JS

```
var gases = [ "Hydrogen", "Helium",  
  "Lithium", "Beryllium"];  
  
var output = []  
for (let i= 0; i< gases.length; i++) {  
  output[i] = gases[i].length;  
}
```

[8, 6, 7, 10]

## lodash

```
var gases = [ "Hydrogen", "Helium",  
  "Lithium", "Beryllium"];  
var output = _.map(gases, x => x.length);  
[8, 6, 7, 10]
```

<https://lodash.com/docs/4.17.15>

# Lodash `_.map`

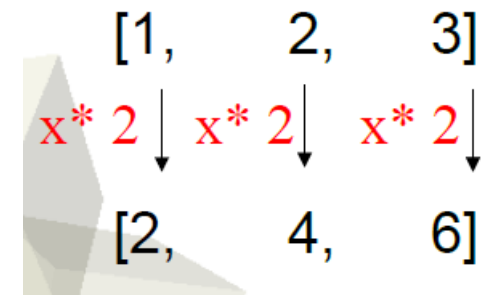
- Map takes an array as input and then outputs a new array with a transformation applied to each element of the input array.
- For example

```
_.map([1, 2, 3], x => x * 2)
```

outputs the following array:

```
[2, 4, 6]
```

So what actually happens is the following



# Lodash `_.map`

- Another example

```
var names = ["Peter", "James", "John"];  
_.map(names, x => "Hello " + x)
```

- outputs the following array:

```
["Hello Peter", "Hello James", "Hello John"]
```

# Lodash `_`.reduce

- Often we want to combine values together, such as when we want to sum an array of values together. Using reduce we can do it very easily.
- Suppose you want to do the following:

```
var input = [1, 2, 3, 4]
var total = 0;
for (let i= 0; i< input.length; i++) {
    total += input[i];
}
```

- At the end of the above program total will equal: 10
- Using `_`.reduce we can write the following

```
_.reduce(input, (accum, x) => accum+ x, 0);
```

<https://lodash.com/docs/4.17.15#reduce>

# Lodash `_reduce`

- What does the following code actually do?

```
var input = [1, 2, 3, 4]
```

```
_reduce(input, (accum, x) => accum+ x, 0);
```

- The output is: 10
- It will iterate through the input array from left to right applying the function

Set accum to 0

Set accum= accum+ 1

Set accum= accum+ 2

Set accum= accum+ 3

Set accum= accum+ 4

[1, 2, 3, 4] outcome: accum= 0

[1, 2, 3, 4] outcome: accum= 1

[1, 2, 3, 4] outcome: accum= 3

[1, 2, 3, 4] outcome: accum= 6

[1, 2, 3, 4] outcome: accum= 10

# Lodash `_`.reduce

- Another example

```
var input = ["a", "b", "c", "d"]  
_.reduce(input, (accum, x) => accum+ x, "");
```

- The output is: "abcd"
- It will iterate through the input array from left to right applying the function

Set accum to ""

["a", "b", "c", "d"] outcome: accum= ""

Set accum= accum+ "a"

["a", "b", "c", "d"] outcome: accum= "a"

Set accum= accum+ "b"

["a", "b", "c", "d"] outcome: accum= "ab"

Set accum= accum+ "c"

["a", "b", "c", "d"] outcome: accum= "abc"

Set accum= accum+ "d"

["a", "b", "c", "d"] outcome: accum= "abcd"



# Searching using `_.find`

- `_.find(fruits, fruit => fruit.price <= 1)`
  - This function returns object:  
`{name: 'apple', price: 0.99, onSale: true}`
  - So the function returns the first object in the array which satisfies the predicate specified in the 2<sup>nd</sup> argument.
- `_.find(fruits, 'onSale')`
  - This function returns object:  
`{name: 'apple', price: 0.99, onSale: true}`
- `_.find(fruits, {name: 'orange', onSale: false})`
  - This function returns object:  
`{name: 'orange', price: 1.99, onSale: false}`

```
var fruits = [  
  {  
    name: 'apple',  
    price: 0.99,  
    onSale: true  
  },  
  {  
    name: 'orange',  
    price: 1.99,  
    onSale: false  
  },  
  {  
    name: 'passion fruit',  
    price: 4.99,  
    onSale: false  
  }  
];
```

# Sorting

- Sorts in ascending order

```
_.sortBy([1, 3, 2]);
```

```
output: [1, 2, 3]
```

- Sorts in descending order

```
_.sortBy([1, 3, 2]).reverse()
```

# Sorting Array of Objects

```
var users = [  
  { 'user': 'fred',    'age': 48 },  
  { 'user': 'barney',  'age': 36 },  
  { 'user': 'fred',    'age': 40 },  
  { 'user': 'barney',  'age': 34 }  
];  
  
_.sortBy(users, function(o) { return o.user; });  
// → objects for [['barney', 36], ['barney', 34], ['fred', 48], ['fred', 40]]  
  
_.sortBy(users, ['user', 'age']);  
// → objects for [['barney', 34], ['barney', 36], ['fred', 40], ['fred', 48]]
```

# Use `_.filter` to pick out the items you want to keep

```
var objects = [  
  { 'name': 'zhen', 'age': 5 },  
  { 'name': 'peter', 'age': 10 },  
  { 'name': 'john', 'age': 12 }  
];  
  
_.filter(objects, { 'age': 5 } );  
// → [{ 'name': 'zhen', 'age': 5 }]  
  
_.filter(objects, x => x.age > 6);  
// → [ { 'name': 'peter', 'age': 10 },  
//      { 'name': 'john', 'age': 12 } ]  
  
_.filter([1,2,3,4,5], x => x > 3);  
// → [4, 5]
```

<https://lodash.com/docs/4.17.15#filter>

# Use `_.assign` to return new extended object

- Suppose you have the following object:

```
var originalObject = {  
  'name': 'zhen',  
  'age' : 80  
}
```

- You now want to add another field like height with value of 1.8. Then you can use the following code:

```
_.assign({}, originalObject, {'height': 1.8});  
// → {name: "zhen", age: 80, height: 1.8}
```

- As you can see the `_.assign` method returns a new object which has the additional height attribute.
- Note the first argument is used for the destination object. But I propose we don't use it and so just set it to `{}`

<https://lodash.com/docs/4.17.15#assign>

# Use `_.assign` to return new object with modified property value

- Suppose you have the following object:

```
var originalObject = {  
  'name': 'zhen',  
  'age' : 80  
}
```

- You now want to return a new object that is the same as the originalObject but with the age changed to 5. The code is the following:

```
_.assign({}, originalObject, {'age': 5});  
// → {name: "zhen", age: 5}
```

# Use `_.assign` to return new extended array object

- We can also use `_.assign` with array objects. In the example below the `_.assign` function call returns a new array object that has an element 6 attached to the end.

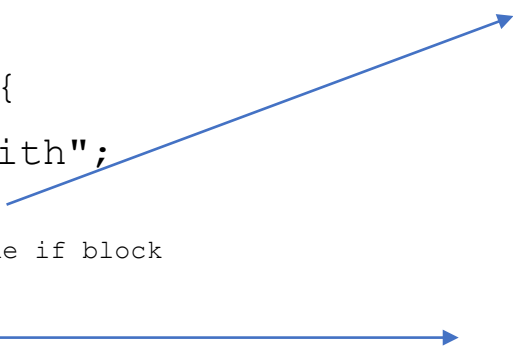
```
var originalArray = [1,2,3,4,5];  
  
_.assign([], originalArray, {[5]: 6})  
// → [1,2,3,4,5,6]
```

# Additional Examples and Features



# Let Example - 1

```
function example() {  
  let name = "Jane Doe";  
  if(name === "Jane Doe"){  
    let name = "Jane Smith";  
    console.log(name);  
    //outputs Jane Smith: reference inside the if block  
  }  
  console.log(name);  
  //outputs Jane Doe: reference at the start of the function  
}
```



- The name variable inside the **if** block is a new variable and it shadows the name variable declared at the top of the script. Therefore, the value of the name in the console is Jane Smith.
- When the JavaScript engine completes executing the **if** block, the name variable inside the if block is out of scope. Therefore, the value of the name variable that follows the if block is Jane Doe.

# Let Example-2

```
function numericExample() {  
  let x = 3;  
  for(let i=0; i < 3; i++){  
    let x = 10;  
    console.log("Inner Scope");  
    console.log(i);  
    console.log(x);  
  }  
  console.log("Outer Scope");  
  console.log(x);  
}
```

> numericExample()		
Inner Scope		<a href="#">VM2203:5</a>
0		<a href="#">VM2203:6</a>
10		<a href="#">VM2203:7</a>
Inner Scope		<a href="#">VM2203:5</a>
1		<a href="#">VM2203:6</a>
10		<a href="#">VM2203:7</a>
Inner Scope		<a href="#">VM2203:5</a>
2		<a href="#">VM2203:6</a>
10		<a href="#">VM2203:7</a>
Outer Scope		<a href="#">VM2203:9</a>
3		<a href="#">VM2203:10</a>

# Temporal Death Zone (TDZ)

```
{ // enter new scope, TDZ starts  
  let log = function () {  
    console.log(message); // message declared later  
  };
```

```
  // This is the TDZ and accessing foo  
  // would cause a ReferenceError
```

```
  let message= 'Hello'; // TDZ ends  
  log(); // called outside TDZ
```

```
}
```

```
> { // enter new scope, TDZ starts  
  let log = function () {  
    console.log(message); // message declared later  
  };  
  
  // This is the TDZ and accessing foo  
  // would cause a ReferenceError  
  
  let message= 'Hello'; // TDZ ends  
  log(); // called outside TDZ  
}
```

Hello

VM1213:3

# TDZ (2)

```
{ // enter new scope, TDZ starts
```

```
  let log = function () {
```

```
    console.log(message); // message declared later
```

```
  };
```

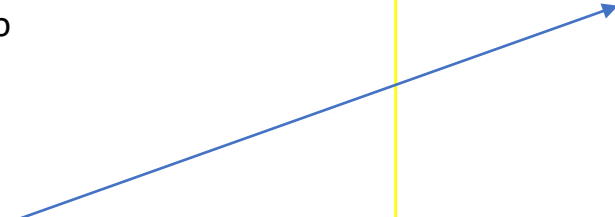
```
  // This is the TDZ and accessing foo
```

```
  // would cause a ReferenceError
```

```
  log(); // called inside TDZ
```

```
  let message= 'Hello'; // TDZ ends
```

```
}
```




```
✖ ▶ Uncaught ReferenceError: Cannot access  
  'message' before initialization      VM3121:3  
    at log (<anonymous>:3:21)  
    at <anonymous>:9:5
```

# Const

- The const keyword creates a read-only reference to a value.
- By convention, the constant identifiers are in uppercase.
  - `const PI=3.14159`

```
function testMutability(){  
  let savings = 1000;  
  const RATE_OF_INTEREST = 0.02; //2% rate of interest  
  let amountFromInterest = savings * RATE_OF_INTEREST * 5;  
  console.log(amountFromInterest); //prints 100  
  savings = 2000; // is allowed as let variables are mutable  
  RATE_OF_INTEREST = 0.03; // throws Uncaught TypeError}
```



✖ ▶ Uncaught TypeError: Assignment to constant variable. [VM3361:7](#)  
 at testMutability (<anonymous>:7:22)  
 at <anonymous>:1:1

# Array

- The JavaScript Array class is a global object that is used in the construction of arrays; which are high-level, list-like objects.
- High order array functions
  - A function that accepts functions as parameters and/or returns a function
  - A callback function is a function that is passed into another function as an argument
  - In JavaScript, functions are objects, functions can be passed as arguments
  - Functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well

# Array Prototypes

## Array.prototype.forEach

- Executes a callback function on each of the elements in an array in order.

```
const array = [1,2,3,4,5];  
array.forEach((el)=> console.log(el*2));
```

```
> const array = [1,2,3,4,5];  
array.forEach((el)=> console.log(el*2));  
2  
4  
6  
8  
10
```

## Array.prototype.map

- Executes a callback function on each element in an array
- Returns a new array made up of the return values from the callback function

```
const array = [1,2,3,4,5];  
x = array.map((el)=> el*2);
```

```
> const array = [1,2,3,4,5];  
x=array.map((el)=> el*2);  
< ▶ (5) [2, 4, 6, 8, 10]  
  
> x  
< ▶ (5) [2, 4, 6, 8, 10]  
  
> array  
< ▶ (5) [1, 2, 3, 4, 5]
```

# Array Prototypes

## Array.prototype.filter

- Executes a callback function on each element in an array
- The callback function for each of the elements must return either true or false
- The returned array is a new array with any elements for which the callback function returns true

```
const array = [1,2,3,4,5,6];  
/*Return all even numbers from array */  
x=array.filter((el)=> el%2 === 0);
```

```
> x  
< ▶ (3) [2, 4, 6]  
  
> array  
< ▶ (6) [1, 2, 3, 4, 5, 6]  
  
>
```

## Array.prototype.reduce

- Takes a callback function with two parameters (accumulator, currentValue) as arguments
- On each iteration, the accumulator is the value returned by the last iteration, and the currentValue is the current element
- Optionally, a second argument can be passed which acts as the initial value of the accumulator

```
const array = [1,2,3,4,5,6];  
/*Return sum of all numbers from array */  
x=array.reduce((acc, currvalue) => acc+currvalue);
```

```
> x  
< 21
```

i	acc	curr
1	0	1
2	1	2
3	3	3
4	6	4
5	10	5
6	11	6
x	21	



# Array Prototypes

## Array.prototype.find

- Returns the value of the first element in the provided array that satisfies the provided testing function.
- If no values satisfy the testing function, undefined is returned.

```
const array = ["Brian", "Jack", "Jim",  
"Olivia", "Elen"];  
array.find((el) => el === "Olivia");
```

```
> const array = ["Brian", "Jack", "Jim", "Olivia", "Elen"];  
  
    array.find((el) => el === "Olivia");  
◀ 'Olivia'  
↘
```

## Array.prototype.findIndex

- Returns the index of the first element in the array that satisfies the provided testing function.
- Otherwise, it returns -1, indicating that no element passed the test

```
const array = ["Brian", "Jack", "Jim",  
"Olivia", "Elen"];  
array.findIndex((el) => el === "Olivia");
```

```
> const array = ["Brian", "Jack", "Jim", "Olivia", "Elen"];  
    array.findIndex((el) => el === "Olivia");  
◀ 3  
  
> array.findIndex((el) => el === "Bob");  
◀ -1
```

# Array Prototypes

## Array.prototype.includes

- Determines whether an array includes a certain value among its entries, returning true or false as appropriate.

```
const array = ["Brian", "Jack", "Jim",  
"Olivia", "Elen"];  
  
array.includes("Jim");  
  
array.includes("Omar")
```

```
> const array = ["Brian", "Jack", "Jim", "Olivia", "Elen"];  
< undefined  
  
> array.includes("Jim");  
< true  
  
> array.includes("Omar")  
< false
```

## Array.prototype.slice

- Returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included)
- The start and end represent the index of items in that array
- The original array will not be modified

```
const array = ["Brian", "Jack", "Jim",  
"Olivia", "Elen"];  
  
x=array.slice(2,4);
```

```
> const array = ["Brian", "Jack", "Jim", "Olivia", "Elen"];  
< undefined  
  
> x=array.slice(2,4);  
< ▶ (2) ['Jim', 'Olivia']  
  
> x  
< ▶ (2) ['Jim', 'Olivia']
```

# Array Prototypes

## Array.prototype.splice

- Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.
- To access part of an array without modifying it, use `.slice`

```
const array = ["Brian", "Jack", "Jim", "Olivia", "Elen"];

array.splice(4,1, "Elliot");
/*replace 1 element and index 4 with Elliot.*/
```

## More prototypes in

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

```
> const array = ["Brian", "Jack", "Jim", "Olivia", "Elen"];
  array.splice(4,1, "Elliot");
< ▶ ['Elen']

> array
< ▶ (5) ['Brian', 'Jack', 'Jim', 'Olivia', 'Elliot']
```

# Objects

- Objects are used to store keyed collections of various data and more complex entities.
- An object can be created with figure brackets {...} with an optional list of properties.
  - A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

```
let cat = new Object();  
cat.name = "Mikasa";  
cat.age = 2;  
cat.bread = "Short-hair";  
console.log(cat);
```

```
let cat = {  
  name: "Mikasa",  
  age: 2,  
  bread: "Short-hair"  
};  
console.log(cat);
```

```
> let cat = {  
  name: "Mikasa",  
  age: 2,  
  bread: "Short-hair"  
};  
console.log(cat);  
  
▶ {name: 'Mikasa', age: 2, bread: 'Short-hair'}
```

```
> let cat = new Object();  
cat.name = "Mikasa";  
cat.age = 2;  
cat.bread = "Short-hair";  
console.log(cat);  
  
▶ {name: 'Mikasa', age: 2, bread: 'Short-hair'}
```

# Accessing object properties and values

- How to access

```
let cat = {  
  name: "Mikasa",  
  age: 2,  
  bread: "Short-hair"  
};  
x=cat['age']  
console.log(x);
```

- How to assign (Change the value of object's properties)

```
cat['address'] = '123, Meow Rd, VIC'  
  
console.log(cat);
```

# Object.assign

- Copies all enumerable own properties from one or more source objects to a target object.
- Returns the modified target object

`Object.assign(target, ...sources)`

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

```
let cat = {
  name: "Mikasa",
  age: 2,
  bread: "Short-hair"};
newCat = Object.assign({}, cat)
Object.assign(newCat, {address: "123, Meow Rd, VIC"});
console.log(cat);
console.log(newCat);
```

```
> let cat = {
  name: "Mikasa",
  age: 2,
  bread: "Short-hair"
};
< undefined
> newCat = Object.assign({}, cat)
< ▶ {name: 'Mikasa', age: 2, bread: 'Short-hair'}
> Object.assign(newCat, {address: "123, Meow Rd, VIC"})
< ▶ {name: 'Mikasa', age: 2, bread: 'Short-hair', address: '123, Meow Rd, VIC'}
> cat
< ▶ {name: 'Mikasa', age: 2, bread: 'Short-hair'}
> newCat
< ▶ {name: 'Mikasa', age: 2, bread: 'Short-hair', address: '123, Meow Rd, VIC'}
```

# Object.entries

- Returns an array of a given object's own enumerable string-keyed property [key, value] pairs, in the same order as that provided by a for...in loop.
- (The only important difference is that a for...in loop enumerates properties in the prototype chain as well).

```
> let people={
  justin:{
    id: 1,
    lastname: "Timberlake",
    age: 30
  },
  jay:{
    id: 2,
    lastname: "Z",
    age: 34
  },
  ice:{
    id: 3,
    lastname: "Cube",
    age: 50
  }
}
< undefined
> for(const[key,value] of Object.entries(people)){
  console.log(`${key} has lastname ${value.lastname} and
  is ${value.age} yo `)
}
justin has lastname Timberlake and is 30 yo      VM54:2
jay has lastname Z and is 34 yo                  VM54:2
ice has lastname Cube and is 50 yo               VM54:2
```

# Object.keys

- Returns an array of a given object's own enumerable property names, iterated in the same order that a normal loop would.

```
> let people={
  justin:{
    id: 1,
    lastname: "Timberlake",
    age: 30
  },
  jay:{
    id: 2,
    lastname: "Z",
    age: 34
  },
  ice:{
    id: 3,
    lastname: "Cube",
    age: 50
  }
}
< undefined
> console.log(Object.keys(people))
▼ (3) ['justin', 'jay', 'ice'] VM1073:1
  0: "justin"
  1: "jay"
  2: "ice"
  length: 3
  ► [[Prototype]]: Array(0)
```



# Object.values

- Returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop.
- The only difference is that a for...in loop enumerates properties in the prototype chain as well.

```
> let people={
  justin:{
    id: 1,
    lastname: "Timberlake",
    age: 30
  },
  jay:{
    id: 2,
    lastname: "Z",
    age: 34
  },
  ice:{
    id: 3,
    lastname: "Cube",
    age: 50
  }
}
```

< undefined

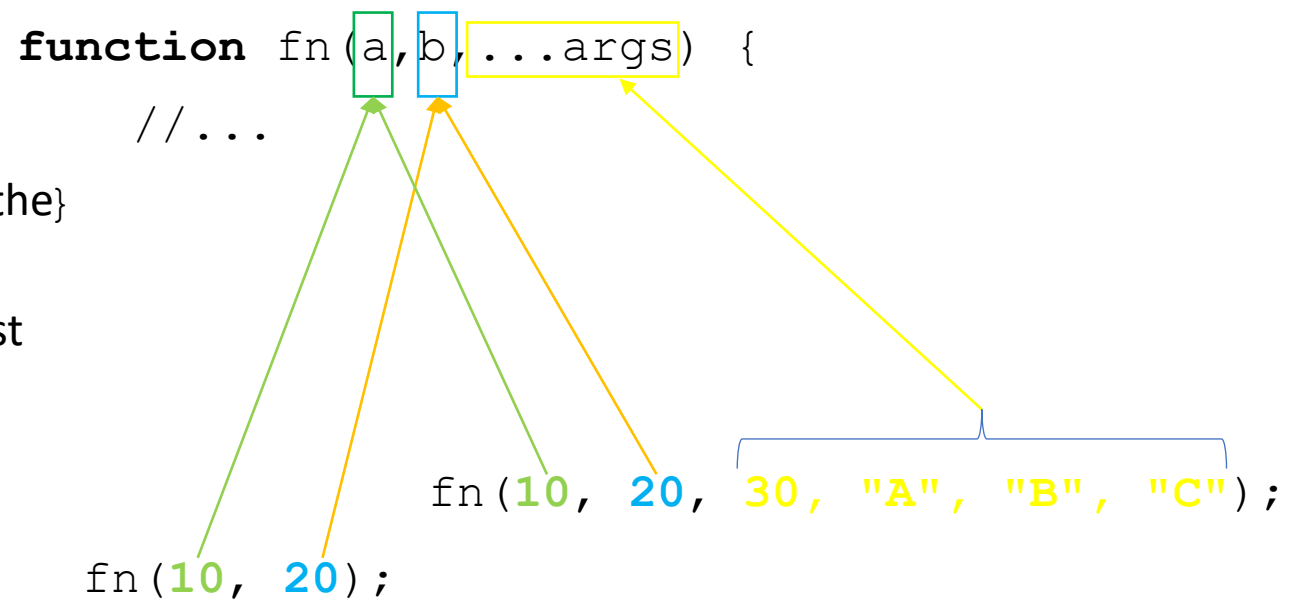
```
> console.log(Object.values(people))
```

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ► 0: {id: 1, lastname: 'Timberlake', age: 30}
  ► 1: {id: 2, lastname: 'Z', age: 34}
  ► 2: {id: 3, lastname: 'Cube', age: 50}
  length: 3
  ► [[Prototype]]: Array(0)
```

VM1216:1

# REST Parameter

- new kind of parameter so-called rest parameter that has a prefix of three dots (...)
- A rest parameter allows you to represent an indefinite number of arguments as an array.
- The rest parameters must appear at the end of the argument list
- If you pass only the first two parameters, the rest parameter will be an empty array []



# SPREAD Operator

- New operator called spread operator that consists of three dots (...)
- Allows you to spread out elements of an iterable object such as an array, a map, or a set
- The spread operator should be used to create a copy of an array. This is to avoid mutating existing array

```
const odd = [1,3,5];  
const numbers = [2,4,6, ...odd];  
console.log(numbers);
```

```
const odd = [1,3,5];  
const numbers = [2,4,6, ...odd];  
console.log(numbers);  
▶ (6) [2, 4, 6, 1, 3, 5]
```

# Object structuring and destructuring

- Destructuring is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
- Allows extracting data from arrays and objects and assign them to variables.
- Destructuring the object allows you to extract its properties more efficiently.

- Consider the following object with multiple properties

```
const person = {  
  firstname: 'James',  
  lastname: 'Coles',  
  age: '30',  
  address: {  
    street: 'Birdeye st',  
    unit: '1'  },  
  email: 'jamescoles@email.com'  
}
```

# Object structuring and destructuring: Accessing the Properties

```
let age = person.age;  
let firstname = person.firstname;  
let lastname = person.lastname;
```

```
let { age, firstname, lastname } = person;
```

```
> let { age, firstname, lastname } = person;  
< undefined  
-----  
> age  
< '30'  
-----  
> firstname  
< 'James'  
-----  
> lastname  
< 'Coles'
```

```
let { key: aliasName} = object;
```

```
> let { address: homeAddress } = person;  
< undefined  
-----  
> homeAddress  
< ▶ {street: 'Birdeye st', unit: '1'}
```

# Callbacks

- Asynchronous programming
  - When code runs (or must run) after something else happens and also not sequentially.
  - For example: when an application wants to get/ send data to/from the server and continue the rest of the operation
  - If the application has to wait till the server responds with the data, it will result in a long delay and a bad user experience
  - To avoid this, we can use callbacks
- How to create callback
- Examine doChores function below.
  - Takes one variable
  - Incomplete explanation

# Promises

- A Promise is a proxy for a value not necessarily known when the promise is created.
- It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
- This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

# Promise State

- Pending: initial state, neither fulfilled nor rejected
  - Can either be fulfilled with a value or rejected with a reason (error).
  - The associated handlers queued up by a promise's then method are called
- Fulfilled: meaning that the operation was completed successfully.
- Rejected: meaning that the operation failed.
- If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.



# Example

```
let promiseExample = new
Promise((resolve, reject) => {

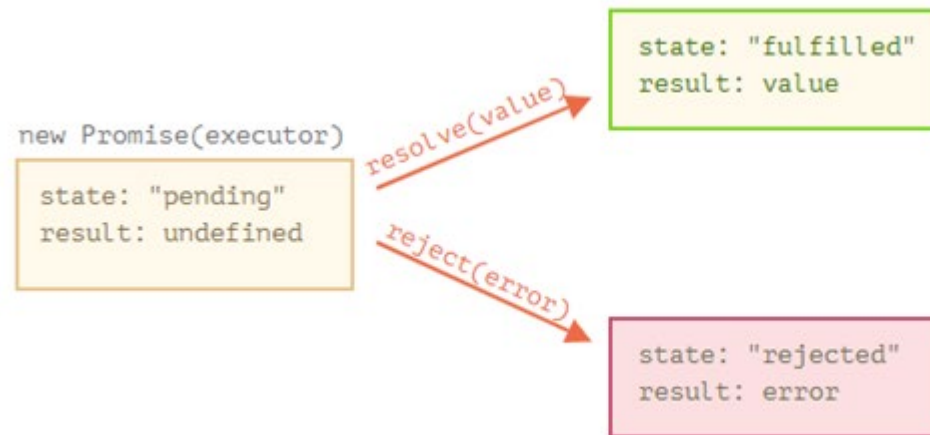
    // do some logic. If it is
    // executed successfully

    resolve("Done
    successfully"); // you can send
    any value. In this example we
    will send a string - Done
    successfully

    // on error, or
    // unsuccessful execution

    reject("Unsuccessful
    execution"); // can you send a
    value.
})
```

- The Promise takes two arguments, resolve and reject and executes as a callback.
- A Promise will either resolve or reject. The executor should call only one resolve or one reject. Any state change is final.



# Promise Consumer

```
1. /** 3 functions, executing asynchronously using callbacks */
2. firstRequest(function(response) {
3.     secondRequest(response, function(nextResponse) {
4.         thirdRequest(nextResponse, function(finalResponse) {
5.             console.log('Final response: ' + finalResponse);
6.         }, failureCallback);
7.     }, failureCallback);
8. }, failureCallback);
```

- The once the firstRequest resolves, it will return secondRequest and pass response as args.
- On successful execution of secondRequest, it will call thirdRequest and once thirdRequest resolves successfully, it will

```
console.log('Final Response: ' + <response>)
```

# Data and Error Handling

- Instance method of the Promise objects that return a separate promise object:
  - then()
  - catch()
  - finally()
- We are creating a dummy api function.
  - This function returns a promise.
  - The if condition will always evaluate to true, so we will send a resolve after waiting for 3 seconds.
  - If there is an error, we will send an Error object.

```
let dummyApi = new Promise((resolve, reject) => {  
  if (true) {  
    // this condition will always evaluate to true  
    setTimeout(() => {  
      resolve('Sending response from api');  
    }, 3000);  
    // after 3sec timeout, it will resolve to  
  }  
  reject(new Error("api error"));  
});
```

# Promise Objects

- `Promise.prototype.then()`
  - Returns a Promise. It takes up to two arguments: callback functions for the success and failure cases of the Promise.
  - `promiseObject.then(onFulfilled, onRejected);`
- `Promise.prototype.catch()`
  - Returns a Promise and deals with rejected cases only.
  - It behaves the same as calling `Promise.prototype.then(undefined, onRejected)`
  - (in fact, calling `obj.catch(onRejected)` internally calls `obj.then(undefined, onRejected)`).
  - Must provide an `onRejected` function even if you want to fall back to an undefined result value
    - `obj.catch(() => {})`

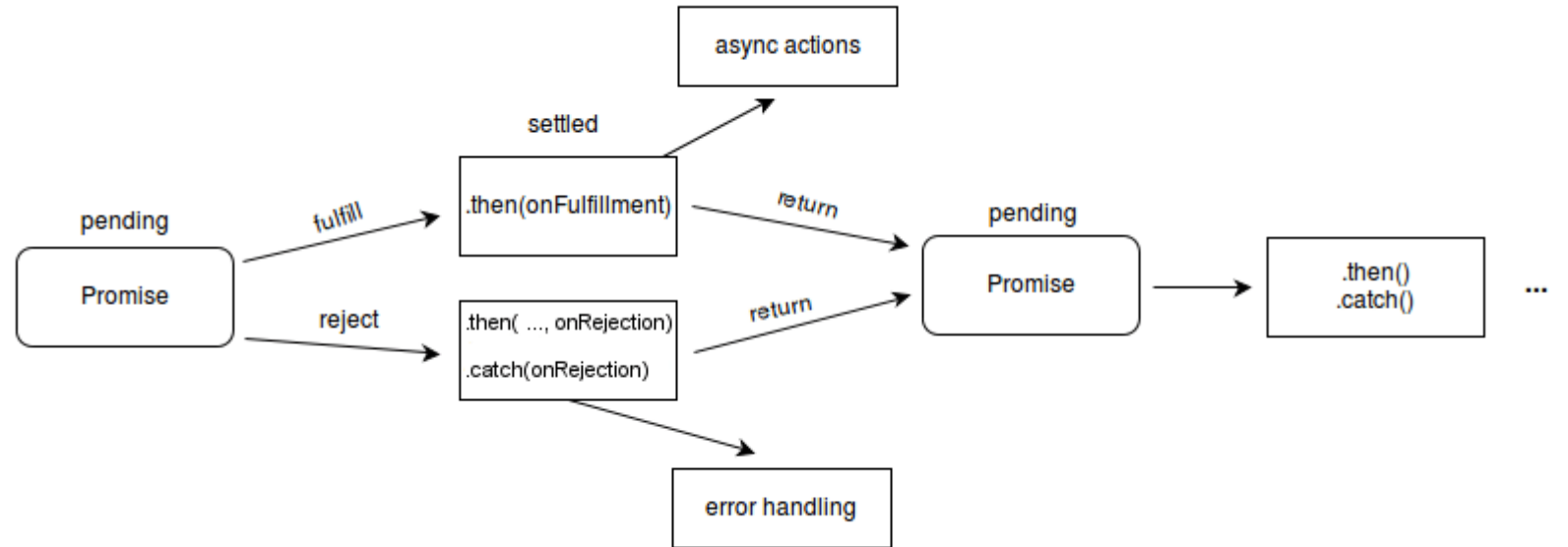
# Promise Objects

- `Promise.prototype.finally()`
  - Returns a Promise.
  - When the promise is settled, i.e either fulfilled or rejected, the specified callback function is executed.
  - This provides a way for code to be run whether the promise was fulfilled successfully or rejected once the Promise has been dealt with.

```
dummyApi.then((response) => {  
    console.log(response);  
}).catch((err) => {  
    console.error(err);  
}).finally((data) => {  
    console.log(data); //always executes no matter resolve or  
reject  
})
```

# Promise Chain

- Promises are useful when you have to handle more than one asynchronous task, one after another.
- Continuous processes in handling multiple situations



# Promise Chain

```
dummyApi.then((response)=>{  
    console.log(response);  
}).catch((err)=>{  
    console.error(err);  
}).finally((data)=>{  
    console.log(data);  
//always executes no matter resolve or reject  
})
```

```
api()  
  .then(function (result) {  
    return api2();  
  })  
  .then(function (result2) {  
    return api3();  
  })  
  .then(function (result3) {  
    // do work  
  })  
  .catch(function (error) {  
    //handle any error that may occur before this point  
  });
```

# Promise Methods

Method	Description
<code>all(iterable)</code>	Waits for all promises to be resolved or anyone to be rejected
<code>allSettled(iterable)</code>	Waits until all promises are either resolved or rejected
<code>any(iterable)</code>	Returns the promise value as soon as any one of the promises is fulfilled
<code>race(iterable)</code>	Wait until any of the promises is resolved or rejected
<code>reject(reason)</code>	Returns a new Promise object that is rejected for the given reason
<code>resolve(value)</code>	Returns a new Promise object that is resolved with the given value
<code>catch()</code>	Appends the rejection handler callback
<code>then()</code>	Appends the resolved handler callback
<code>finally()</code>	the



# Thank You

