JavaScript (2)

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda

- Objects
- Creating Objects
- Accessing Properties
- Setting Properties
- Methods
- Executing Methods
- this keyword
- JavaScript HTML DOM Elements
- JavaScript HTML DOM EventListener
- Event Bubbling or Event Capturing?
- The removeEventListener() method
- JavaScript Timing Events
- The setTimeout() Method
- The setInterval() Method
- ECMAScript

Objects

- An object represents a real-world entity.
- Objects have properties and methods.
- Objects are great way to group property and behavior in to one container.

Creating Objects – Literal notation

- You can use the **literal** notation to define an object.

- Use curly braces to contain the key/value pairs.

```
var Person = {
  property: value,
  property: value,
  ...
};
```

Creating Objects – Constructor notation

- You can use the **constructor** notation to define an object.

- Use the new keyword with the object's constructor function.

```
var Person = new Object();
```

Accessing Properties

- You can access an object's properties value by using the **dot-notation**.

- Use a period after the object name followed by the name of the property.

  ex. Person.name

- You can access an object's properties value by using the **array-notation**.

- Access the property similar to how you would an array.

  ex. Person["name"]

Setting Properties

You can set properties of an object by using the **dot-notation** or **array-notation** followed by the assignment.

```
Person.name = "Adam";

Person["name"] = "Adam";
```

Methods

- A method is a function that is assigned as a property of an object.

```
var Person = {
        givenName: function {
                // ...
        }
}
```

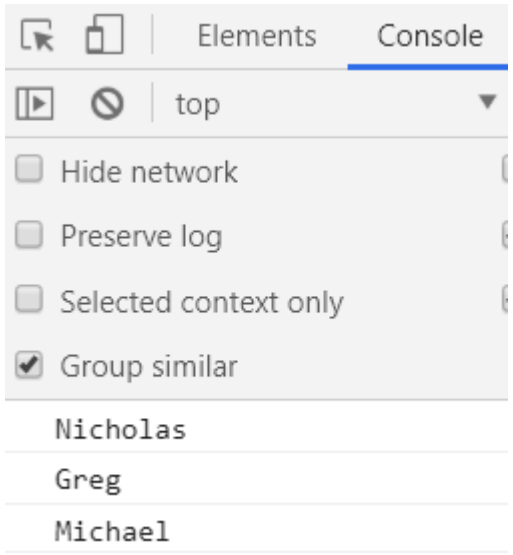
Executing Methods

- You can call a method similar to as you would a function, except that you use the dot-notation.

```
person.givenName();

person.methodTwo(param1, param2);
```

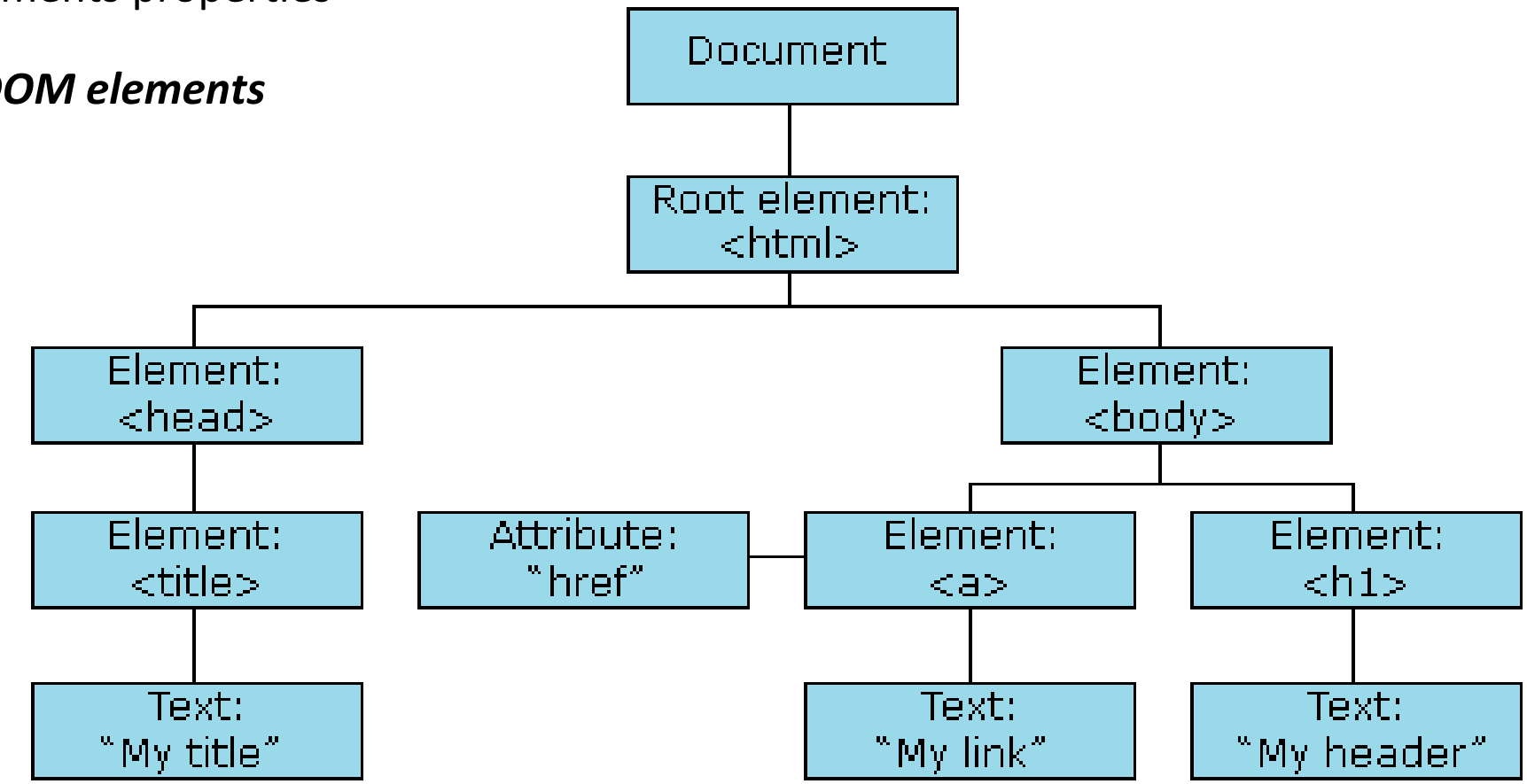
this keyword Example

```
function sayNameForAll(){
  console.log(this.name);
}

var person1 = {
  name: "Nicholas",
  sayName: sayNameForAll
};

var person2 = {
  name: "Greg",
  sayName: sayNameForAll
};

var name = "Michael";

person1.sayName();
person2.sayName();

sayNameForAll();
```



JavaScript HTML DOM Elements

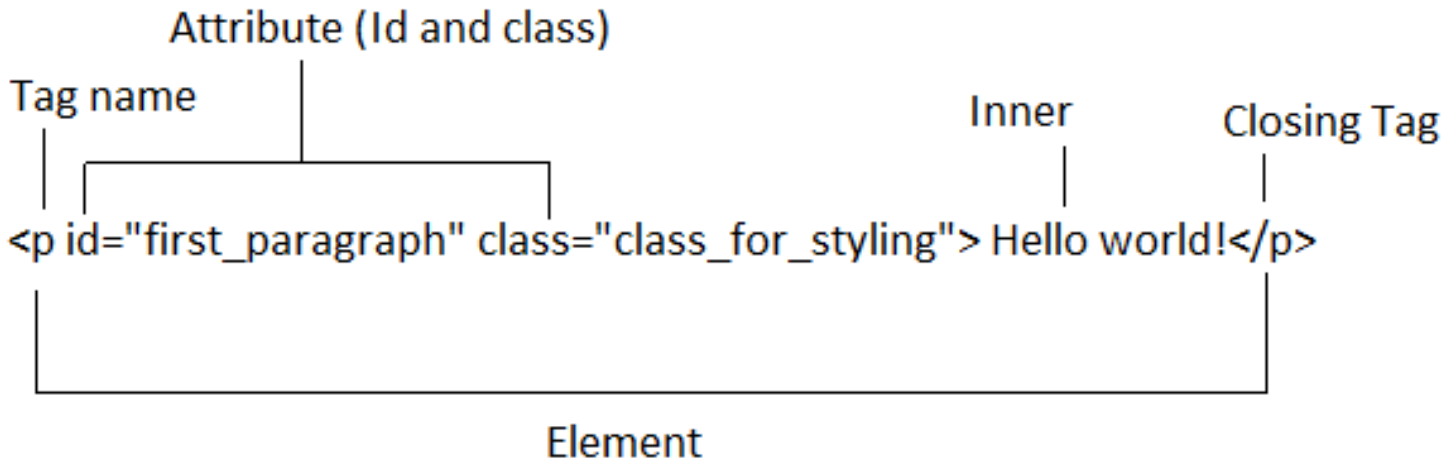
- We can access ***DOM elements***
- Edit the ***DOM*** elements properties
- Create/remove ***DOM elements***



JavaScript HTML DOM Elements

Finding HTML elements
We need to find the elements for DOM manipulation. There are several ways to do this:

- Finding HTML elements by id
- Finding HTML elements by tag name
- Finding HTML elements by class name



JavaScript HTML DOM Elements

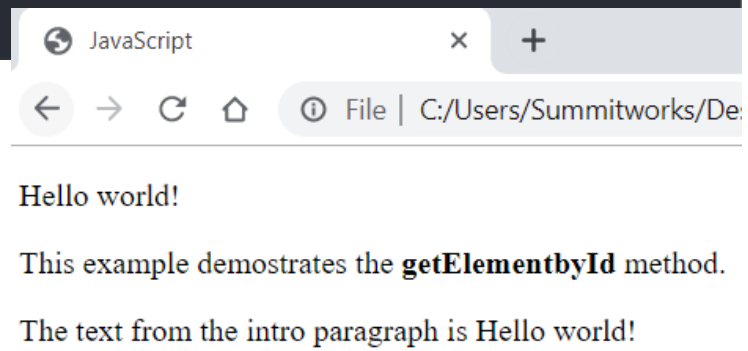
Finding HTML elements by id
The easiest way to find an HTML element in the DOM, is by using the element id.

This example finds the element with id="intro":

If the element is found, the method will return the element as an object (in myElement).

If the element is not found, myElement will contain null.

```
<body>
  <p id="intro">Hello world!</p>

  <p>
    This example demostrates the
    <b>getElementbyId</b> method.
  </p>

  <p id="demo"></p>

  <script type="text/javascript">
    var myElement = document.getElementById("intro");
    document.getElementById("demo").innerHTML =
    "The text from the intro paragraph is " + myElement.innerHTML;
  </script>
</body>
```



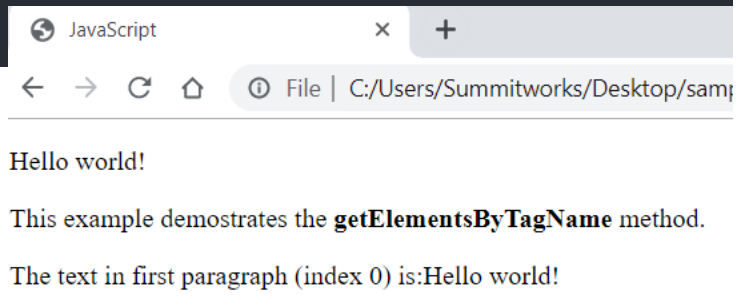
JavaScript HTML DOM Elements

Finding HTML Elements by Tag Name

This example finds all <p> elements:

This example finds the element with id="main", and then finds all <p> elements inside "main":

```
<body>
  <p>Hello world!</p>

  <p>
    This example demostrates the
    <b>getElementsByTagName</b> method.
  </p>

  <p id="demo"></p>

  <script type="text/javascript">
    var x = document.getElementsByTagName("p");
    document.getElementById("demo").innerHTML =
    "The text in first paragraph (index 0) is:" + x[0].innerHTML;
  </script>
</body>
```



JavaScript HTML DOM Elements

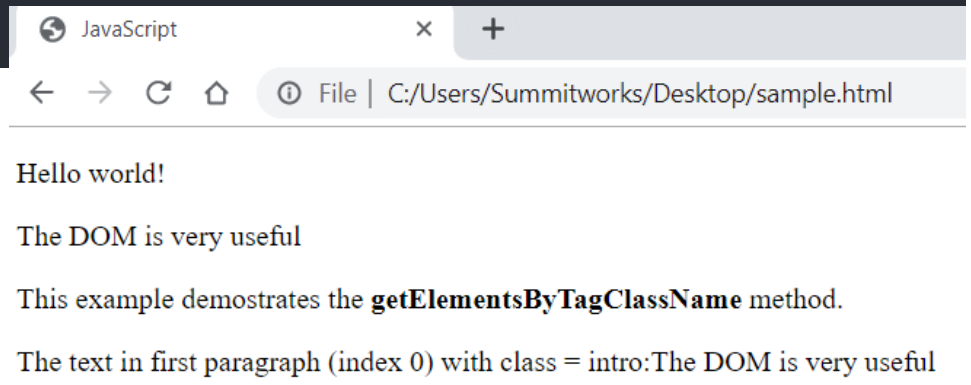
Finding HTML Elements by Class Name

If you want to find all HTML elements with the same class name, use

getElementsByClassName()

This example returns a list of all elements with class="intro".

```
<body>
  <p>Hello world!</p>

  <p class="intro">The DOM is very useful</p>
  <p class="intro">
    This example demostrates the
    <b>getElementsByTagClassName</b> method.
  </p>

  <p id="demo"></p>

  <script type="text/javascript">
    var x = document.getElementsByClassName("intro");
    document.getElementById("demo").innerHTML =
    "The text in first paragraph (index 0) with class = intro:" + x[0].innerHTML;
  </script>
</body>
```



JavaScript HTML DOM Elements

Changing DOM Elements properties

Property	Description
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element
Method	**Description**
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element

JavaScript HTML DOM Elements

Create/remove DOM elements

Method	Description
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*new*, *old*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

JavaScript HTML DOM EventListener

Changing DOM Elements properties

The addEventListener() method attaches an event handler to the specified element.

You can easily remove an event listener by using the removeEventListener() method.

Syntax

```
element.addEventListener(event, function, useCapture);
```

The first parameter is the type of the event (like "click" or "mousedown").

The second parameter is the function we want to call when the event occurs.

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

Event Bubbling or Event Capturing?

There are two ways of event propagation in the HTML DOM, bubbling and capturing.

Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In bubbling the inner most element's event is handled first and then the outer:

i.e. the <p> element's click event is handled first, then the element's click event.

In capturing the outer most element's event is handled first and then the inner:

i.e. the element's click event will be handled first, then the <p> element's click event.

With the addEventListener() method you can specify the propagation type by using the "useCapture" parameter:

```
addEventListener(event, function, useCapture);
```

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

The removeEventListener() method

The removeEventListener() method removes event handlers that have been attached with the *addEventListener()* method:

```
element.removeEventListener("mousemove", myFunction);
```

JavaScript Timing Events

Window object represents an open window in a browser.

The window object allows execution of code at specified time intervals. These time intervals are called timing events.

The two key methods to use with JavaScript are:

setTimeout(function, milliseconds)

Executes a function, after waiting a specified number of milliseconds.

setInterval(function, milliseconds)

Same as setTimeout(), but repeats the execution of the function continuously.

The setTimeout() Method

The setTimeout() method executes one time after the given time interval.

The method takes 2 parameters

- The first parameter is a function to be executed.
- The second parameter indicates the number of milliseconds before execution.

The setInterval() Method

The setInterval() method repeats a given function at every given time-interval.

The method takes 2 parameters

- The first parameter is the function to be executed.
- The second parameter indicates the length of the time-interval between each execution.

ECMAScript

The JavaScript core language features are defined in the ECMA-262 standard.

The language defined in this standard is called ECMAScript.

What you know as JavaScript in browsers and in Node.js is actually a superset of ECMAScript.

var Declarations and Hoisting

Variable declarations using var are treated as if they're at the top of the function (or in the global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called hoisting.

```
function getValue(condition){

  if(condition){
    var value = "blue";

    //other code

    return value;
  } else {

      // value exists here with a value of undefined

      return null;

  }

  //value exists here with a value of undefined
}
```

```
function getValue(condition){

  var value;

  if(condition){
    value = "blue";

    //other code

    return value;
  } else {

      return null;
  }
}
```

Block-Level Declarations

Block-level declarations declare bindings that are inaccessible outside a given block scope. Block scopes, also called lexical scopes, are created in the following places:

- Inside a function
- Inside a block (indicated by the *{* and *}* characters)

Block-Level Declarations: let Declarations

The let declaration syntax is the same as the syntax for var.

You can basically replace var with let to declare a variable but limit the variable's scope to only the current code block .

Because let declarations are not hoisted to the top of the enclosing block, it's best to place let declarations first in the block so they're available to the entire block.

```
function getValue(condition){

  if(condition){
    let value = "blue";

    //other code

    return value;
  } else {

      // value doesn't exists here

      return null;

  }

  // value doesn't exists here
}
```

Block-Level Declarations: let Declarations

No Redeclaration

If an identifier has already been defined in a scope, using the identifier in a let declaration inside that scope causes an error to be thrown.

```
var count = 30;

//throws an console.error
let count 40;
```

Conversely, no error is thrown if a let declaration creates a new variable with the same name as a variable in its containing scope

```
var count =30;

if(condition){

  //doesn't throw an error
  let count = 40;

  //more code
}
```

This let declaration doesn’t throw an error because it creates a new variable called count within the if statement instead of creating count in the surrounding block. Inside the if block, this new variable shadows the global count, preventing access to it until execution exits the block.

Block-Level Declarations: const Declarations

const Declarations

Bindings declared using ***const*** are considered constants, meaning their values cannot be changed once set.

For this reason, every ***const*** binding must be initialized on declaration

```
//valid constant
const maxItems = 30;

//syntax error: missing initialization
const name;
```

Constants vs. let Declarations

Constants, like ***let*** declarations, are block-level declarations. That means constants are no longer accessible once execution flows out of the block in which they were declared, and declarations are not hoisted.

In another similarity to ***let***, a ***const*** declaration throws an error when made with an identifier for an already defined variable in the same scope. It doesn't matter whether that variable was declared using ***var*** (for global or function scope) or ***let*** (for block scope).

Despite those similarities, there is one significant difference between ***let*** and ***const***. Attempting to assign a ***const*** to a previously defined constant will throw an error in both strict and non-strict modes.

However, unlike constants in other languages, the value a constant holds can be modified if it is an object.

```
if(condition) {
  const maxItems = 5;

  //more code
}


var message = "Hello";
let age = 25;

//each of these throws an error
const message = "Goodbye!";
const age = 30;


const maxItems = 5;

//throws an error
maxItems = 6;
```

var vs constants vs let Declarations

Var	Let	Const
Hoisted	Not hoisted	Not hoisted
Not block level scope	Block level scope	Block level scope
Redeclaration allowed	Redeclaration not allowed	Redeclaration not allowed
Same variable with var and let not allowed in same scope	Same variable with var and let not allowed in same scope	Same variable with var, let and const not allowed in same scope
Same variable with var and let allowed in different scope	Same variable with var and let allowed in different scope	Same variable with var, let and const allowed in different scope
		Considered as constant, cannot be changed
Not necessary	Not necessary	Must be initialized on declaration

Arrow Function

One of the most interesting new parts of ECMAScript 6 is the arrow function. Arrow functions are, as the name suggests, functions defined with a new syntax that uses an arrow (**=>**)

```
ONE ARGUMENT
let reflect = value => value;

//effectively equivalent to:

let reflect = function(value){
  return value;
};

NO ARGUMENT
let getName = () => "Nicholas";

//effectively equivalent to:

let getName = function() {
  return "Nicholas";
}
```

```
MORE THAN ONE ARGUMENT
let sum = (num1, num2) => num1 + num2;

//effectively equivalent to:

let sum = function(num1, num2){
  return num1 + num2;
}

MORE STATEMENTS INSIDE THE FUNCTION
let sum = (num1, num2) => {
  return num1 + num2;
}

//effectively equivalent to:

let sum = function(num1, num2){
  return num1 + num2;
}
```

Classes

Class declarations begin with the class keyword followed by the name of the class. The rest of the syntax looks similar to concise methods in object literals but doesn't require commas between the elements of the class.

```
class PersonClass {

  //equivalent of the PersonType constructor
  constructor(name) {
    this.name = name;
  }

  //equivalent of PersonType.prototype.sayName
  sayName() {

    console.log(this.name);
  }
}

let person = new PersonClass("Nicholas");
person.sayName();   //ouputs "Nicholas"
```