

JavaScript OOP

IMY 220 • Lecture 6

Indexing object properties

Everything in the browser is an **object**.

When you want to access something in the HTML with JavaScript, you need to access the object.

```
<form name="theForm">  
  <input type="text" name="theText" value="Your Name" />  
</form>
```

To access the form in JavaScript, you use:

```
document.theForm
```

To access the text box within the form, you use:

```
document.theForm.theText.value
```

The “theText” object is the text box object.

This object has various functions, event handlers and properties linked to it.

Indexing object properties

You can also access elements using their id value

```
<div id="names"> Name: </div>
```

```
document.getElementById("names").innerHTML += "Jake the dog";
```

(Appends the value to the div's existing value)

Objects as associative arrays

The dot operator isn't the only way to access an object's properties and methods:

```
document.write("Hello World!");
```

they can also be accessed using the **subscript notation**:

```
document["write"]("Hello World!");
```

In short, you can treat an object as an **associative array** that maps a string to a value in the same way that a typical array maps a number to a value.

Object categories in JavaScript

Native Objects

Objects supplied by JavaScript.

Examples: String, Number, Array, Image, Date, Math, etc.

User-Defined Objects

Objects defined by the programmer

Host Objects

Objects that are supplied to JavaScript by the browser environment.

Examples: window, document, forms, etc.

Native Object Creation

You can create an object in JS using the following syntax:

```
theObject = new Object();  
ratings = new Array(6,9,8,4,5,7,8,10);  
var home = new String("Residence");  
var futdate = new Date();  
var num1 = new Number();
```

The **new** operator creates an instance of any built-in or user-defined object.

You can create a string object like this

```
var theString = "my strings!";
```

Object literal

Data-types that can be **declared literally**, such as arrays and strings, can also be declared as objects.

This is mainly so that object methods can be applied to literal values when needed. E.g. the `String.length` method.

JavaScript does this by temporarily converting your literal into an object.

```
var strlen = "hello".length;  
// strlen == 5
```

User-defined objects

Using a constructor function

Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.

```
function Car(make, model, year) {  
    this.make = make;  
    this.model = model;  
    this.year = year;  
}
```

Create an instance of the object with new.

```
var myCar = new Car("Toyota", "Corolla", 1993);
```


Defining new properties for objects

Prototype property

Prototype property can be used to create **new properties** for user or system defined objects.

This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object.

The following code adds a color property to all objects of type car, and then assigns a value to the color property of the object car1.

```
Car.prototype.color = null;  
myCar.color = "black";
```

User-defined objects

You can add member functions to your object by using the following syntax:

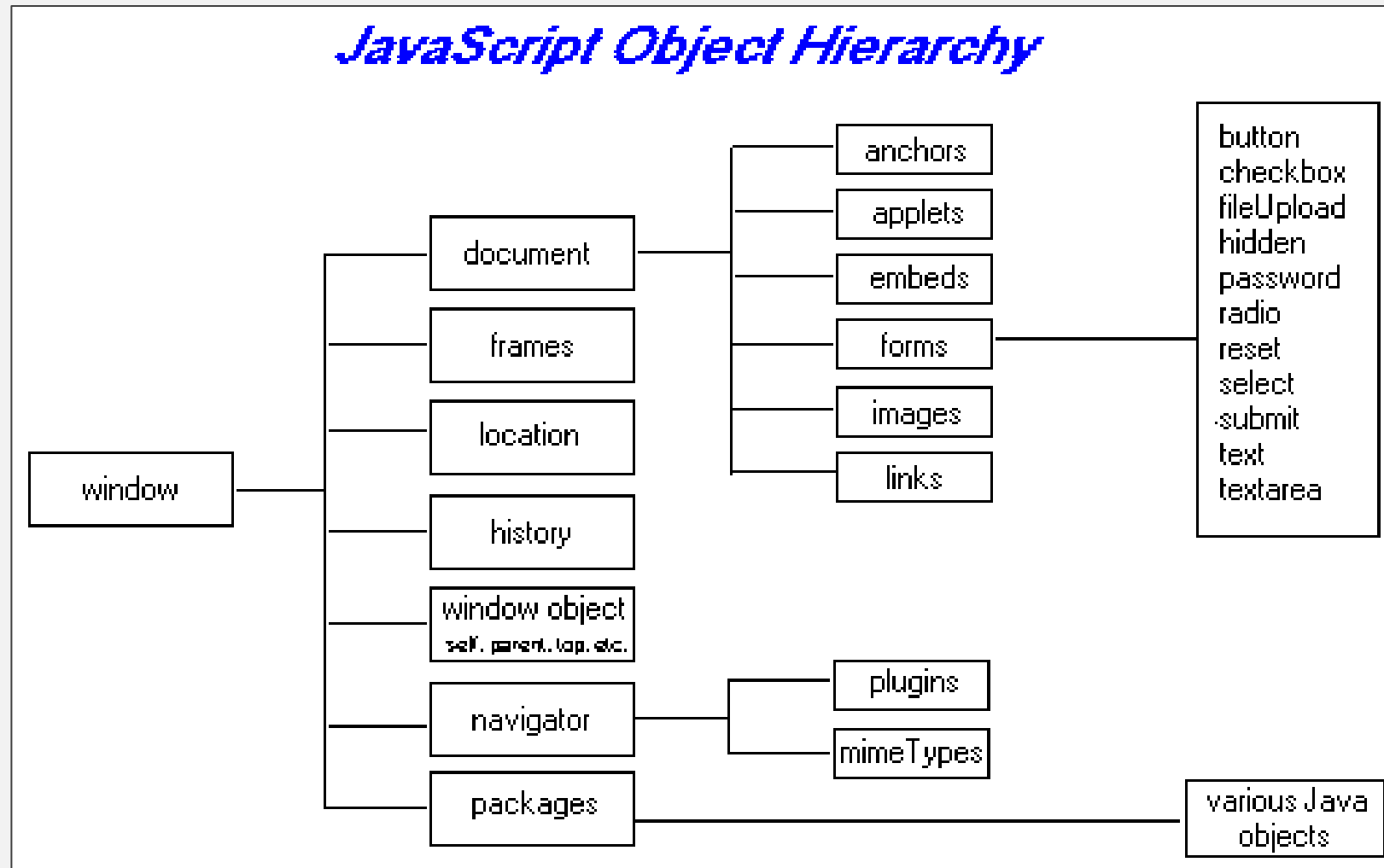
```
function Car(make, model, year) {  
    this.make = make;  
    this.model = model;  
    this.year = year;  
    this.noise = function() {  
        alert("Tooot!");  
    }  
}  
  
var newCar = new Car("Toyota", "Corolla", 2014);  
  
newCar.noise();
```

User-defined objects

You can also add member functions to your object by using the prototype property:

```
function Car(make, model, year) {  
    this.make = make;  
    this.model = model;  
    this.year = year;  
}  
  
Car.prototype.noise = function(){  
    alert("Toot!");  
}  
  
var newCar = new Car("Toyota", "Corolla", 2014);  
  
newCar.noise();
```

Host Objects



Host Objects

- Window
- Navigator
- Document

Window Object

Window Objects in JavaScript

These objects have **functions**

Example:

```
alert("This is an alert!");
```

This object can be called like this as well:

```
window.alert("This is an alert!");
```

- This is because the window object is the highest level object

```
this.alert("This is an alert!");
```

- The “this” object declaration works as well

Window Object

`window.history`

Provides access to session history of the current tab

```
window.history.back();  
// go back one page
```

Navigator

The `navigator` interface represents the state and the identity of the user agent.

It allows scripts to query it and to register themselves to carry on some activities.

A `navigator object` can be retrieved using the read-only `window.navigator` property.

Navigator Properties

`navigator.cookieEnabled`

Returns a Boolean value indicating whether cookies are enabled or not (read-only).

```
if (!navigator.cookieEnabled) {  
    // let the user know that enabling cookies  
    // makes the web page more useful  
}
```

Navigator Properties

navigatorLanguage.language

The [navigatorLanguage.language](#) read-only property returns a string representing the preferred language of the user, usually the language of the browser UI.

```
if (window.navigator.language != "en") {  
    doLangSelect(window.navigator.language);  
}
```

Document Object

Each [web page](#) loaded in the browser has its own document object.

The Document interface serves as an entry point into the web page's content (the DOM tree, including elements such as `<body>` and `<table>`) and provides functionality which is global to the document (such as obtaining the page's URL and creating new elements in the document).

Document Object

Access it using [window.document](#)

```
<!DOCTYPE html>
<html>
<head>
  <title> Hello World! </title>
</head>
<body>
  <script type="text/javascript">
    var doc = window.document;
    alert(doc.title); // alerts: Hello World!
  </script>
</body>
</html>
```

Document Object

Document Properties

Document Methods

Document Properties

`document.activeElement`

Returns the currently focused element.

`document.anchors`

Returns a list of all of the anchors in the document.

`document.body`

Returns the `<body>` element of the current document.

`document.cookie`

Returns a semicolon-separated list of the cookies for that document or sets a single cookie.

Document Properties

document.embeds

Returns a list of the embedded <embed> elements within the current document.

document.forms

Returns a list of the <form> elements within the current document.

document.head

Returns the <head> element of the current document.

document.images

Returns a list of the images in the current document.

Document Functions

`document.write()`

`document.write(markup);`

`markup` is a string containing the text to be written to the document.

Document Functions

document.getElementById()

Returns a reference to the element by its ID.

```
<!DOCTYPE html>
<html>
<head>
  <title> getElementById example </title>
  <script type="text/javascript">
    function changeColor(newColor){
      var elem = document.getElementById("para1");
      elem.style.color = newColor;
    }
  </script>
</head>
<body>
  <p id="para1"> Some text here </p>
  <button onclick="changeColor('blue');">blue</button>
  <button onclick="changeColor('red');">red</button>
</body>
</html>
```

Document Functions

document.getElementsByClassName()

Returns an array-like object of all child elements which have all of the given class names.

When called on the document object, the complete document is searched, including the root node.

```
// Get all elements that have a class of 'test'  
document.getElementsByClassName("test");
```

Document Functions

document.writeln()

Writes a string of text followed by a newline character to a document.

```
document.writeln("<p> enter password: </p>");
```

HTMLElement

The HTMLElement interface represents any HTML element. Some elements directly implement this interface, others implement it via an interface that inherits it.

HTMLElement Methods

HTMLElement.blur()

Removes keyboard focus from the currently focused element.

HTMLElement.click()

Sends a mouse click event to the element.

HTMLElement.focus()

Makes the element the current keyboard focus.

HTMLElement.forceSpellCheck()

Makes the spell checker runs on the element.

GlobalEventHandlers

`onMouseOver` – when the cursor goes over the text field.

```
<input type="text" name="theText" onMouseOver="theFunction()" />
```

`onMouseDown` – when you click down on the text field

`onMouseUp` – when you click and lift your finger

`onClick` – when you click

`onBlur` – when the text field loses focus by using the tab key (or clicking somewhere else)

`onFocus` – if the text field gets focus

WindowTimers - methods

WindowTimers.setTimeout()

Sets a delay for executing a function

WindowTimers.setInterval()

Schedules the execution of a function each X milliseconds

WindowTimers.clearTimeout()

Cancels the delayed execution set using WindowTimers.setTimeout()

WindowTimers.clearInterval()

Cancels the repeated execution set using WindowTimers.setInterval()

setTimeout function

The `setTimeout` function is useful for when you want a function to execute after a specific amount of time.

The function takes 2 parameters

```
setTimeout(the_function, 1000);
```

The name of your function (**NB:** don't include the “()” at the end)

The number of milliseconds JS needs to wait before executing the code.

Hoisting

JavaScript puts variable and function declarations into memory during the compile phase. This is called hoisting.

This can lead to some unexpected behaviour with regards to how variables are declared and used, for example:

```
console.log(num) ;  
var num;
```

This “should” give an error, since the variable is only declared after it is used, but instead it gives **undefined**

Hoisting

This is because the variable declaration is hoisted, i.e. it is put into memory before executing any code that may use that declaration. For that reason, the following piece of code...

```
console.log(num); // Output: undefined  
var num = 5;
```

...is implicitly understood as

```
var num;  
console.log(num);  
num = 5;
```

This example also illustrates that hoisting only applies to declarations, not initialisations

Hoisting

The same applies to function *declarations*, which allows us to use functions before they are declared

```
console.log(sayHello("Diffie"));

function sayHello(name) {
    return "Hello " + name;
}
```

Function declarations are thus hoisted before the code executes

Hoisting

The same, however, does not apply to function *expressions*, i.e. functions assigned to variables using the **var** keyword

```
console.log(sayHello("Diffie"));

var sayHello = function(name) {
    return "Hello " + name;
}

// This code gives an error
```

Function expressions only load when the line of code which defines the function is reached

Arrays

Create new Array

```
var names = [];
```

Access the array elements

```
document.write(names[0]);
```

- Prints the first element in the array

Add the names to the array

```
names[0] = "Jake";  
names[1] = "Amy";  
names[2] = "Rosa";  
names[3] = "Gina";  
names[4] = "Terry";
```

Arrays

Output the array

```
for(var x = 0; x < 5; x++) {  
    alert(names[x]);  
}
```

Assign values to array

```
for(var y = 0; y < 5; y++) {  
    names[y] = prompt('Enter a Name!', ' ');  
}
```

The [Window.prompt\(\)](#) displays a dialog with an optional message prompting the user to input some text.

Arrays

Associative arrays

```
names["one"] = "Raymond";  
names["two"] = "Jake";  
names["three"] = "Charles";  
names["four"] = "Gina";  
names["five"] = "Terry";
```

Quick aside: Functions

Functions can also be passed as parameters (we will deal more with this in the next lecture)

```
function doFiveTimes(event) {  
    for(var i = 0; i < 5; i++){  
        event();  
    }  
}  
  
doFiveTimes(function() {alert("Heeeey!");});
```


Arrays

JavaScript has a bunch of built-in functions that simplify dealing with arrays:

- `concat()`
- `forEach()`
- `map()`
- `filter()`
- `every()`
- `splice()`
- `slice()`
- `reduce()`
- `etc.`

Arrays

“The `concat()` method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.”

(The code below creates a new array from two existing arrays)

```
var nums1 = [1, 2, 3];  
var nums2 = [4, 5, 6];  
var arr = nums1.concat(nums2);  
  
console.log(arr.toString());  
// Output: 1, 2, 3, 4, 5, 6
```

Arrays

“The `forEach()` method executes a provided function once for each array element.”

(The code below adds all elements from `arr` to string, separated by a space)

```
var arr = [1, 2, 3];
var str = "";

arr.forEach(function(element) {
    str += element + " ";
});

console.log(str);
// Output: 1 2 3
```

Arrays

Most array functions also provide the index as a second function parameter

```
var names = ["Jake", "Amy", "Rosa"];  
var str = "";  
  
names.forEach(function(element, index){  
    str += index + ": " + element + ", ";  
});  
  
console.log(str);  
// Output: 0: Jake, 1: Amy, 2:Rosa,
```

Arrays

“The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.”

(The code below creates a new array from all elements from `arr` that are larger than 3)

```
var arr = [1, 2, 3, 4, 5];
var arr2;

arr2 = arr.filter(function(element) {
    return element > 3;
});

console.log(arr2.toString());
// Output: 4, 5
```

Arrays

“The `every()` method tests whether all elements in the array pass the test implemented by the provided function” (and returns true/false)

(The code below tests whether all elements in an array are less than 4)

```
var arr = [1, 2, 3];
var smaller;

smaller = arr.every(function(element) {
    return element < 4;
});

console.log(smaller);
// Output: true
```

```
var arr = [1, 2, 3, 6];
var smaller;

smaller = arr.every(function(element) {
    return element < 4;
});

console.log(smaller);
// Output: false
```

Arrays

“The `splice()` method changes the contents of an array by removing existing elements and/or adding new elements.”

The function is called on the array to be modified and does not return a new array

The function parameters are as follows:

```
arrayName.splice(start, deleteCount, item1, item2, ...);
```

Arrays

The following code starts at index 2, removes 1 item and inserts another element at the start index

```
var names = ["Gina", "Jake", "Amy", "Rosa"];  
names.splice(2, 1, "Charles");  
console.log(names.toString());  
// Output: Gina,Jake,Charles,Rosa
```

However, you can provide any number of elements to add at the start index

```
var names = ["Gina", "Jake", "Amy", "Rosa"];  
names.splice(2, 1, "Charles", "Terry", "Raymond");  
console.log(names.toString());  
// Output: Gina,Jake,Charles,Terry,Raymond,Rosa
```


Arrays

“The `slice()` method returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included). The original array will not be modified.”

The function parameters are as follows:

```
slicedArrayName = arrayName.slice(start, end);
```

If `end` is omitted, the array is copied from `start` to the last element of the array

Arrays

The following code copies the values from names, starting at index 1 up until before index 3, into a new array

```
var names = ["Gina", "Jake", "Amy", "Rosa", "Charles"];  
var names2 = names.slice(1, 3);  
console.log(names2.toString());  
// Output: Jake,Amy
```

If the second parameter is omitted, the rest of the array, starting at the start index, is copied

```
var names = ["Gina", "Jake", "Amy", "Rosa", "Charles"];  
var names2 = names.slice(1);  
console.log(names2.toString());  
// Output: Jake,Amy,Rosa,Charles
```

Arrays

While the `slice()` function copies values, like strings, numbers, etc., directly, it only copies references to objects (i.e. shallow copy)

```
var person1 = {name: "Jake"};
var person2 = {name: "Amy"};
var names = [person1, person2];
var names2 = names.slice(1);
console.log(names2[0].name);
// Output: Amy
```

```
person2.name = "Charles";
console.log(names2[0].name);
// Output: Charles
```

Arrays

The `map()` function loops through each array element, performs some change on it, and returns a new array with the altered elements

```
var arr = [1, 2, 3];  
var arr2;  
  
arr2 = arr.map(function(element, index) {  
    return element + 1;  
});  
  
console.log(arr2.toString());  
// Output: 2, 3, 4
```

Arrays

“The `reduce()` method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.”

(The code below returns the sum of all values in an array)

```
var nums = [1, 2, 3, 4];
var sum;

sum = nums.reduce(function(accumulator, currElement){
    return accumulator + currElement;
});
console.log(sum);
// Output: 10
```

Arrays

In the first iteration, accumulator refers to the first element of the array and currElement to the second. So, using the previous example:

Iteration: 1	accumulator: arr[0] -> 1	currElement: arr[1] -> 2	return-value: 3
--------------	-----------------------------	-----------------------------	-----------------

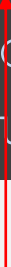
After that accumulator holds on to the value returned from the value returned from the previous iteration while the value of currElement always refers to the next element of the array:

Iteration: 2	accumulator: 3	currElement: arr[2] -> 3	return-value: 6
Iteration: 3	accumulator: 6	currElement: arr[3] -> 4	return-value: 10
End of array reached. return-value: 10			

Arrays

Apart from the function parameter, you can also provide an initial value as a second parameter to the function.

```
var nums = [1, 2, 3, 4];  
var sum;  
  
sum = nums.reduce(function(accumulator, currElement){  
    return accumulator + currElement;  
}, 10);  
console.log(sum);  
// Output: 20
```



Arrays

When an initial value is provided, the function executes the same way, except that an extra iteration is added in which **accumulator** refers to the initial value and **currValue** refers to **arr[0]**

Iteration: 1	accumulator: initial value -> 10	currElement: arr[0] -> 1	return value: 11
Iteration: 2	accumulator: 11	currElement: arr[1] -> 2	return value: 13
Iteration: 3	accumulator: 13	currElement: arr[2] -> 3	return-value: 16
Iteration: 4	accumulator: 16	currElement: arr[3] -> 4	return-value: 20
End of array reached. return-value: 20			

Arrays

Many more useful Array functions

You can read more about JS Arrays and find details about all the built-in Array functions here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Arrays

Since many functions return arrays, you can chain Array functions to perform more complex functions

```
var names1 = ["Jake", "Amy", "Charles"];
var names2 = ["Scully", "Hitchcock", "Gina"];

var arr = names1
    .concat(names2)
    .filter(function(element) {
        return element.length < 5;
    }).map(function(element) {
        return "Yass " + element + "!";
    });

console.log(arr.toString());
Output: ???
```

Extra reading

- Extra reading: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
- Useful tool for debugging JS: <https://validatejavascript.com/>

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions>
- <http://nefariousdesigns.co.uk/object-oriented-javascript.html>
- http://www.w3.org/wiki/Objects_in_JavaScript
- <http://www.sitepoint.com/oriented-programming-1-4/>
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
- <https://developer.mozilla.org/en-US/docs/Glossary/Hoisting>
- <https://developer.mozilla.org/en-US/docs/web/JavaScript/Reference/Operators/function>