


Advanced JavaScript

Eng. Niveen Nasr El-Den
iTi



Day 1



*These are the
Golden Days of
JavaScript*



JavaScript is designed on a
simple **object-based**
paradigm



Objects are the
Fundamental unit of
JavaScript



An **object** is a collection
of **properties**.

A **property** is an **association**
between a **name**
(or **key**) and a **value**.

JavaScript Objects

According to Client-side

- JavaScript Objects fall into **4** categories:
 - ▷ Custom Objects
 - ▷ Built – in Objects
 - ▷ **BOM** Objects “**B**rowser **O**bject **M**odel” (Host)
 - ▷ **DOM** Objects “**D**ocument **O**bject **M**odel”.
- In addition to objects that are predefined in the browser
 - we can define our own objects.

JavaScript Built-in Objects

- **String**

- **Number**

- **Array**

- **Date**

- **Math**

- **Boolean**

- **RegExp**

- **Error**

- **Function**

- **Object**

Creating variables

Literal “Short-Hand”	Constructor
<code>var str = “abc”;</code>	<code>var str = new String();</code>
<code>var arr = [];</code>	<code>var arr = new Array();</code>
<code>var reg = /[a-z]/gmi;</code>	<code>var reg = new RegExp('[a-z]', 'gmi');</code>
<code>var obj = {};</code>	<code>var obj = new Object();</code>
<code>var fn = function(a, b){ return a + b; }</code>	<code>var fn = new Function('a', 'b', 'return a+b');</code>
	<code>var fn = new Function('a, b' , 'return a+b');</code>

Error Object

Error Object Creation

- Whenever an error occurs, an instance of **error** object is created to describe the error.
- Error objects are created either by the environment (the browser) or by your code.
- Developer can create Error objects by 2 ways:
 - ▷ **Explicitly:**
 - `var newErrorObj = new Error();`
 - thrown using the throw statement
 - ▷ **Implicitly:**

Error Object Construction

- Error constructor

- `var e = new Error();`

- More than **Six** additional Error constructor ones exist and they all inherit Error:

EvalError	Raised by eval when used incorrectly
RangeError	Numeric value exceeds its range
ReferenceError	Invalid reference is used
SyntaxError	Used with invalid syntax
TypeError	Raised when variable is not the type expected
URIError	Raised when <code>encodeURIComponent()</code> or <code>decodeURIComponent()</code> are used incorrectly

- Using ***instanceOf*** when catching the error lets you know if the error is one of these built-in types.

Error Object Properties

Property	Description
description	Plain-language description of error (IE only)
fileName	URI of the file containing the script throwing the error
lineNumber	Source code line number of error
message	Plain-language description of error (ECMA)
name	Error type (ECMA)
number	Microsoft proprietary error number

Error Object Standard Properties

- **name** → The name of the error constructor used to create the object

▷ Example:

- `var e = new EvalError('Oops');`
- `e.name;`

→ "EvalError"

- **Message** → Additional error information:

▷ Example:

- `var e = new Error('jaavcsritp is _not_ how you spell it');`
- `e.message`

→ "jaavcsritp is _not_ how you spell it"

Example!

throw Statement

- The throw statement allows you to create an exception.
- Using throw statement with the try...catch, you can control program flow and generate accurate error messages.

- Syntax

`throw(exception)`

- The exception can be a **string**, **integer**, **Boolean** or an **object**

throw Example

```
if(x<100)
    throw "less100"
else if(x>200)
    throw "more200"
```

```
var e= new Error("more200")
if(x<100)
    throw new Error("less100" )
else if(x>200)
    throw e
```

Example!

Object Object

Object Object

- **Object** object is the **parent** of all JavaScript objects, which means that every object you create inherits from it
 - ▷ **Reminder** : the **Global** object is **window** object
- To create an object
 - ▷ `var obj = { }; → preferable way`
 - ▷ `var obj = new Object();`
- **Object** object has **constructor** property that used to return the constructor function of the created Object.
- Objects are considered **Associative Arrays** also called a **hash** (the keys are strings)



JavaScript uses **Arrays** to
represent **indexed** arrays
&
Objects to represent
associative arrays.

Object Object

```
//old way of creating an object  
var obj = new Object();  
//new way of creating an object (Literal notation)  
//var obj={ };  
// adding property to object obj  
obj.name = "JavaScript";//dot notation → preferable approach  
//obj["name"] = "JavaScript";// subscript notation
```

```
var obj = {  
    // adding property to object obj  
    name : "JavaScript",  
    // "name" : "JavaScript",  
};
```

Example!

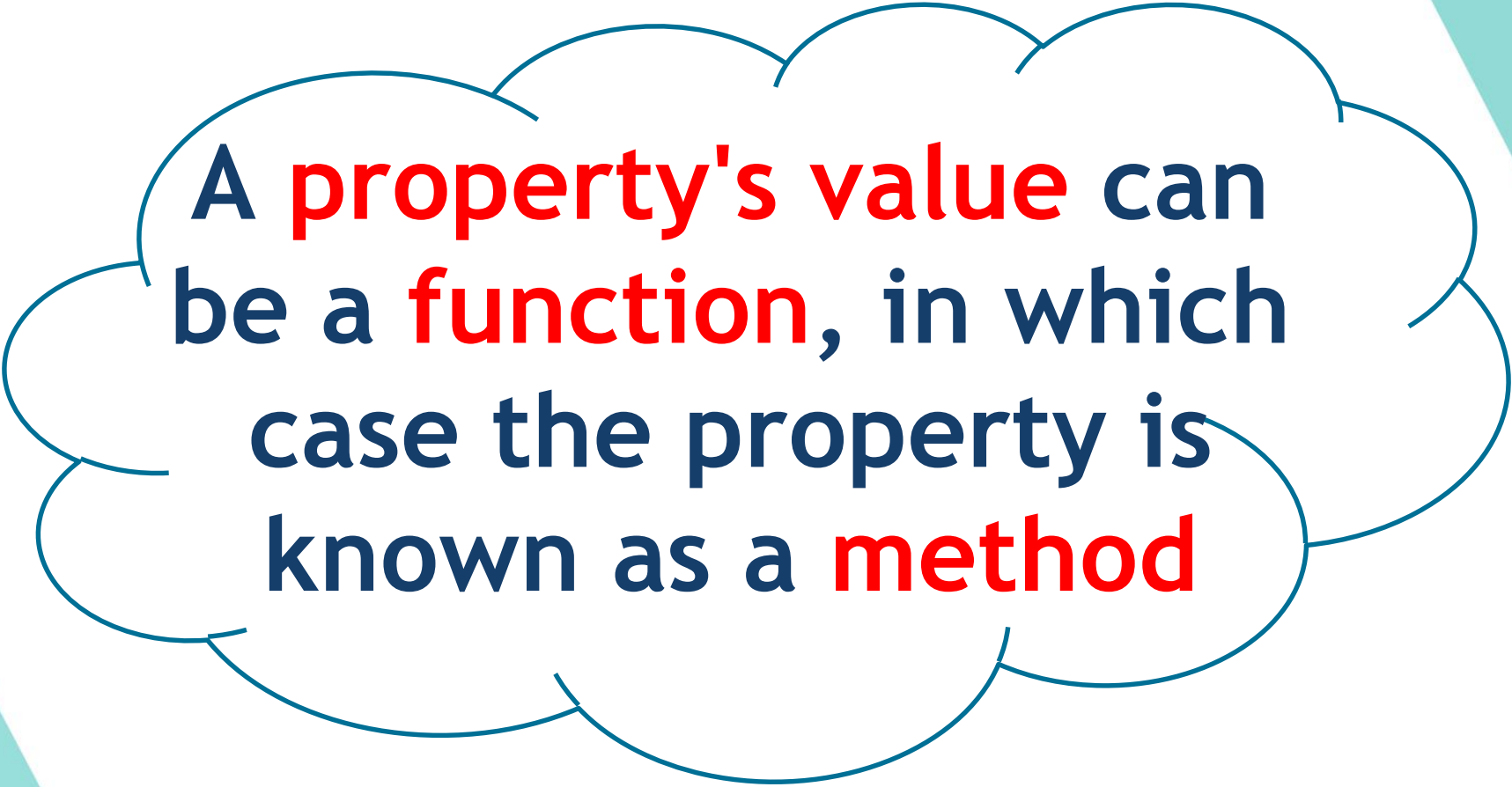
Object Object Properties & Methods

- `hasOwnProperty("prop")`
- `.valueOf()`
- `.toString()`
- `Object.keys(obj)`
- `Object.entries(obj)`
- `Object.values(obj)`
- `Object.defineProperty(obj,"prop",{...})`
- `Object.defineProperties(obj,{...})`
- `Object.create(obj [, {...}])`
- ...

Facts #1 About JavaScript Function

- Every thing you can do with other data types can be done with functions
 - ▷ Function can be
 - assigned to
 - a variable,
 - an array element,
 - passed as an argument to another function
 - a value returned from a method call
 - created on the fly
- This makes using *functions* a very handy and flexible, but also a confusing one.

DEMO!!!



A **property's value** can
be a **function**, in which
case the property is
known as a **method**

Adding Method to Object

- Method is an action performed by executing a function
- Method is added to an object by assigning a function object as a **value of** an object **property**



Adding Method to Object

- Method is an action performed by executing a function
- Method is added to an object by assigning a function object as a **value of** an object **property**

```
// adding method to object obj  
obj.say = function myFun() { //literal function  
    alert( "hello" );  
};  
// “say” is a property that contains a function object
```

Adding Method to Object

- Method is an action performed by executing a function
- Method is added to an object by assigning a function object as a **value of** an object **property**

```
// adding method to object obj  
obj.say = function () { //literal function  
    alert( "hello" );  
};  
  
// “say” is a property that contains a function object
```

Adding Method to Object

```
var obj = new Object(); //old way of creating an object  
// var obj={ }; //new way of creating an object (Literal)
```

```
// adding property to object obj
```

```
obj.name = "JavaScript"; // dot notation
```

```
//obj["name"] = "JavaScript"; // subscript notation
```

```
// adding method to object obj
```

```
obj.say = function() { //literal function  
    alert( "hello" );  
};
```

```
// “say” is a property that contains a function object
```

Adding Method to Object


```
var obj ={  
    // adding property to object obj  
    name : "JavaScript",  
    // "name" : "JavaScript",  
  
    // adding method to object obj  
    say : function() { //literal function  
        alert( "hello" );  
    }  
};
```

Example!

Adding Method to Object

```
var obj ={  
    // adding property to object obj  
    name : "JavaScript",  
    // "name" : "JavaScript",  
  
    // adding method to object obj  
    say : say  
};  
  
function say() {  
    alert( "hello" );  
}
```

Example!



In JavaScript we have
property that contains
Function Object



**A function always
returns a value**

Facts #2 About JavaScript Function

- JavaScript is a first-class function.
 - *Functions* are treated as **first class citizens** since it can be considered as *values* in JavaScript and treated like any variable. i.e. being passed as an argument, returned from a function, modified, and assigned to a variable
- Functions in JavaScript are first-class **objects**
 - Functions are a **special data type**.
 - Function objects have **properties** and **functions**
- Functions are actually objects that are **invokable**
- There is a built-in constructor function called **Function()** which allows an alternative (but not recommended) way to create a function.

Function Object

Function Object

- JavaScript functions are objects. They can be defined using the Function constructor
(Dynamic / Anonymous / **Function Constructor**)
 - `var sum = new Function('a', 'b', 'return a + b;');`
 - `alert(sum (2 , 3));`
- JavaScript functions using the function literal, it is also known as Factory Function
(Literal / Anonymous / **Function expression**)
 - `var sum = function(a, b){return a + b;};`
 - `alert(sum (2 , 3));`
- The more common traditional way:
(Declarative / Static / **Function Statement**)
 - `function sum(a, b){return a + b;}`
 - `alert(sum (2 , 3));`

Declarative/ Static Function

- The most common traditional type of function uses the declarative/static format.
- This approach begins with
 - ▷ function keyword,
 - ▷ followed by function name,
 - ▷ parentheses containing zero or more arguments,
 - ▷ and then the function body
 - ▷ **Parsed once when the page is loaded**
 - The parsed result is used each time the function is called
 - ▷ **Hoisted** (useful for **mutual recursion**)
- Simple to read and understand
- Its a **function statement** that does some work

```
function funNm (par1, par2,... , parn){  
    //fun body;  
}
```

Dynamic/Anonymous Function

```
var variable = new Function("param1", "param2", .. , "paramn", "function body");
```

- The Dynamic/Anonymous Function:
 - *Anonymous*: because the function itself isn't directly declared or named.
 - *Dynamic*: The JavaScript engine creates the anonymous function dynamically,
 - **each time** it's invoked, the function is dynamically **reconstructed**.
 - Uses Function object constructor
 - Do not create **closures** to their creation contexts; they are always created in the global scope

- Example:

```
var sayHi = new Function("toWhom","alert('Hi ' + toWhom);");  
sayHi("World!");
```

Literal Declaration

```
var func = function [fun_nm] (params) { statements; }
```

- Also known as **function expressions** because the function is created as part of an expression, rather than as a distinct statement type.
- They resemble **anonymous** functions in that they don't have a specific function name.
- They resemble **declarative** functions, in that function literals are **parsed only once**.
- Example:

```
var func = function (x, y) { return x * y; }  
alert(func(3,3));
```

Anonymous Function

- functions are like any other variable so they can also be used without being assigned a name.
- Anonymous functions are functions that are passed as arguments or declared inline and have no name
- Example:
 - ▷ 1;
 - ▷ [1,2,"str"];
 - ▷ "Hello!!"
 - ▷ `function(a){return a;}`

Anonymous Function

- Advantage:
 - ▷ Used as callback function
 - ▷ Used as **IIFEs**
 - ▷ Hide Variables from the Global Scope
- Disadvantage:
 - ▷ Cant execute twice unless it is put inside loop or another function

Calling anonymous Functions

- You can pass an anonymous function as a parameter to another function. The receiving function can do something useful with the function that you pass.
- **Self-invoking Functions**
 - You can define an anonymous function and execute it right away. By calling this function right after it was defined. This is called **IIFE “Immediate Invoke Function Expression”**

➤ Example

```
(  
    function(){  
        alert('hellooooo');  
    }  
)()
```

```
(  
    function(){  
        alert('hellooooo');  
    }  
())
```

IIFE

- IIFE stands for Immediately Invoked Function Expression
- It is a function expression that is invoked immediately
- A common often extra ordinary used **pattern**
- Can be invoked on the fly at the point it is created
- Function expression is wrapped within **()** operator

IIFE

- Prefix operators may be used

```
void function () {  
  console.log("inside IIFE") }();
```

```
!function () {  
  console.log("inside IIFE") }();
```

- Trailing semicolon is required between two IIFEs

```
(  
  function(){  
    alert('hello IIFE1');  
  }  
());  
(  
  function(){  
    alert('hello IIFE2');  
  }  
())
```

Function Object Properties

- **prototype**

- This is another way to add more functionality to already created objects

- **length**

- Specifies the number of parameters the function expects as set in its signature

- **caller** (obsolete)

- This returns a reference to the function that called our function

- **name**

- This returns the name of the function if it has a name after function keyword.

arguments Object


- When a function receives parameter values from a **caller**, those parameter values are implicitly assigned to the **arguments** property of the function object.
 - i.e. arguments, contains values of all parameters passed to the function.
 - Missing parameters are treated as **undefined** values
 - We can pass more arguments than expected.
- It looks like an array, but its not an array although it contains indexed elements and has **length** property that returns number of arguments **passed to** the function
- It has **callee** property. (**deprecated**)
 - This contains a reference to the function being called.
 - **arguments.callee** allows **Self-invoking anonymous** functions to call themselves **recursively**.
- It is used **inside** function body implementation.

Example!

arguments.callee Example

```
function accumulateSum (n){  
  
    if(n!=0)  
        return n + accumulateSum (--n);  
  
    else  
        return n;  
  
}
```

```
function accumulateSum (n){  
    if(n!=0)  
        return n + arguments.callee(--n);  
    else  
        return n;  
}
```



Function Default arguments

```
function myFun(){  
  var x = arguments[0] || 10;  
  var y = arguments[1] == undefined ? 11 : arguments[1]  
  
  return x+y;  
}
```

```
myFun(); //21  
myFun(1); //12  
myFun(1,2); //3
```

```
function myFun(x=10,y=11){ /*ES6*/  
  return x+y;  
}
```

Function Object Methods

- Function borrowing:

- ▷ `apply(this_obj, params_array)`

- Allows you to call another function while overwriting its `this` value.
 - The first parameter that `apply()` accepts is the object to be bound to `this` inside the function and the second is an array of parameters to be passed to the function being called.

- ▷ `call(this_obj,p1, p2, p3, ...)`

- Same as `apply()` but accepts parameters one by one, as opposed to as one array.

- ▷ `bind(obj)`

- Allows you to call a function into another object.
 - We can achieve function `currying` via `bind`

<http://javascriptissexy.com/javascript-apply-call-and-bind-methods-are-essential-for-javascript-professionals/>

Function Object Methods

```
var myStr = "this is an example of using Function Methods";  
var arr = [];
```

```
//borrowing using apply  
arr.join.apply(myStr, ["*"]);
```

```
//borrowing using apply  
arr.join.call(myStr, "*");
```

```
//currying using bind  
var newBind = arr.join.bind(myStr);  
newBind("*");
```

```
var newBind = arr.join.bind(myStr, "*");  
newBind();
```

Note: arr can be replaced by [] or (new Array)

Result: "t*h*i*s* *i*s* *a*n*
*e*x*a*m*p*l*e* *o*f* *u*s*i*n*g*
*F*u*n*c*t*i'o*n* *M*e*t'h'o*d*s"

call() and apply() Example

```
var myObj={  
  name:"myObj Object",  
  myFunc:function(){  
    alert(this.name)  
  },  
  myFuncArgs:function(x,y){  
    alert(this.name+" " + x +" "+y)  
  }  
};  
  
var obj1={name:"obj1 Object"};
```

```
myObj.myFuncArgs(1,2);//myObj Object 1 2
```

```
myObj.myFuncArgs.apply(obj1,[1,2]);//obj1 Object 1 2
```

```
myObj.myFuncArgs.call(obj1,1,2);//obj1 Object 1 2
```

Assignments