# Javascript

phuong.vu@htklabs.com

# Agenda

- Javascript Concepts
- Objects
- Asynchronous Nature of Javascript
- Hoisting
- Scope & Closures
- Context & This

# Javascript Concepts

- Object-Oriented
  - JavaScript is an object-oriented language.
  - Functions are objects, too. They can have properties and methods.
  - Two main types of objects:
    - Native: further be categorized as built-in (for example, Array, Date) or user-defined (var o = {};)
    - Host: Defined by the host environment, for example, window and all the DOM objects
- No classes: There are no classes in JavaScript
- Prototype: is an object (not a class or anything special) and every function has a prototype property . Prototypes are used for inheritance.

# Objects

# Objects

- In JavaScript, objects are king. If you understand objects, you understand JavaScript.
- In JavaScript, almost "everything" is an object.
- Only five primitive types are not objects: *number, string, boolean, null,* and *undefined*, and the first three have corresponding object representation in the form of primitive wrappers.
- Number, string, and boolean primitive values are easily converted to objects.
- Dates, Maths, Regular expressions, Arrays, Functions are always objects.
- Objects are objects.
- Objects are Variables Containing Variables

# Objects are Variables Containing Variables

- JavaScript variables can contain single values:
  - Ex: `var person = "John Doe";`
- Objects are variables too. But objects can contain many values. A JavaScript object is an unordered collection of variables called named values.
  - Ex: `var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};`

# Object Properties

- The named values, in JavaScript objects, are called properties.
- Properties can usually be changed, added, and deleted, but some are read only.
- The syntax for accessing the property of an object are:
  - *objectName.property  // person.age*
  - *objectName["property"]  // person["age"]*
  - *objectName[expression]  // x = "age"; person[x]*

# Object Methods

- Methods are actions that can be performed on objects.
- Object properties can be both primitive values, other objects, and functions.
- An object method is an object property containing a function definition.
- Create an object method with the following syntax:
  - *methodName : function() { code lines }*
  - *objectName.methodName()*

# Object Prototypes

- Every JavaScript object has a prototype. The prototype is also an object.
- All JavaScript objects inherit their properties and methods from their prototype.
- The standard way to create an object prototype is to use an object constructor function
  - Ex:

```
function person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
```

# Creating Object - 3 ways

- Define and create a single object, using an object literal.
  - Ex: `var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};`
- Define and create a single object, with the keyword new.
  - Ex: `var person = new Object();`
- Define an object constructor, and then create objects of the constructed type.
  - Ex:

```
function person(first, last, age, eye) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eye;
    }
    var myFather = new person("John", "Doe", 50, "blue");
```

# Built-in JavaScript Constructors

- new Object();  // A new Object object
- new String();  // A new String object
- new Number();  // A new Number object
- new Boolean()  // A new Boolean object
- new Array();  // A new Array object
- new RegExp();  // A new RegExp object
- new Function();  // A new Function object
- new Date();  // A new Date object

| Built-in constructors (avoid) | Literals and primitives (prefer) |
| --- | --- |
| `var o = new Object();` | `var o = {};` |
| `var a = new Array();` | `var a = [];` |
| `var re = new RegExp(`<br>`  "[a-z]",`<br>`  "g"`<br>`);` | `var re = /[a-z]/g;` |
| `var s = new String();` | `var s = "";` |
| `var n = new Number();` | `var n = 0;` |
| `var b = new Boolean();` | `var b = false;` |
| `throw new Error("uh-oh");` | `throw {`<br>`  name: "Error",`<br>`  message: "uh-oh"`<br>`};`<br><br>... or<br><br>`throw Error("uh-oh");` |

# Asynchronous Nature of Javascript

# Asynchronous? What's Asynchronous?

- General concepts of programming languages introduce two distinct mechanisms of concurrency, namely synchronous and asynchronous control flow.
- **Synchronous** control flow means that a process runs only as a result of some other process being completed or handing off operation.
- **Asynchronous** control flow means that a process operates independently of other processes.
- JavaScript runs in the browser which is itself a single process on the operation system.
- JavaScript runs in a single thread within the browser process.

# Synchronous Code

- We could explain synchronous control flow by explaining how to read a book:
  - take the book
  - only when you took the book: read page 1
  - only when you read page 1: read page 2
  - only when you read page 2: read page 3
  - ...
  - only when you read the last page: burn the book
- In synchronous, each step is executed only when the previous step has completed.

# Asynchronous Code

- If these actions are asynchronous, we can’t ensure that the first step is finished firstly, the second step secondly etc:
- However, since JavaScript is single-threaded, it’s a little different and asynchronous control flow doesn’t exactly mean that - a process operates independently of other processes.
- Instead, the script is always run completely first, while all asynchronous actions are queued up by the browser, and only when the script is run completely (the single script thread), the asynchronous actions are executed. ***Always remember:*** the whole script is executed before any asynchronous action is performed!

# Hoisting

# Understanding JavaScript hoisting

- Is JavaScript's default behavior of moving declarations to the top.
- Is when JavaScript moves variable and function declarations to the top of their scope before any code is executed.

# Declarations vs Expressions

## Function Declarations

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

## Function Expressions

```
var hello = function() {
    alert('hello');
};
```

## Work as expected

```
x();
function x() {
    alert('I am x');
}
```

## Error

```
x();
var x = function() {
    alert('I am x');
}
```

# Declarations vs Initializations

- `var x = 7;` : both a declaration and an initialization
- `var x;` : is a declaration
- `x = 7;` : is initialization

# Scope & Closures

# Scope

- Refers to where variables and functions are accessible, and in what context it is being executed.
- A variable or function can be defined in a global or local scope.
- Variables have so-called function scope, and functions have the same scope as variables.

# Scope

- **Global Scope:** it is accessible from anywhere in your code. Ex:

```
var monkey = "Gorilla";
function greetVisitor () {return alert("Hello dear blog reader!");}
```

- **Local Scope:** As opposed to the global scope, the local scope is when something is just defined and accessible in a certain part of the code, like a function. Ex:

```
var saying = “Welcome”
function talkDirty () {
        var saying = "Oh, you little VB lover, you";
        return alert(saying);
}
alert(saying); // Throws an error
```

# Closures

- Closures are expressions, usually functions, which can work with variables set within a certain context
- Try and make it easier: inner functions referring to local variables of its outer function create closures.
- A closure is a function having access to the parent scope, even after the parent function has closed.

# 1. Closures have access to the outer function's variable even after the outer function returns

```
function celebrityName (firstName) {
var nameIntro = "This celebrity is ";
// this inner function has access to the outer function's variables, including the parameter
function lastName (theLastName) {
return nameIntro + firstName + " " + theLastName;
}
return lastName;
}

var mjName = celebrityName ("Michael"); // At this juncture, the celebrityName outer function has
returned.

// The closure (lastName) is called here after the outer function has returned above
// Yet, the closure still has access to the outer function's variables and parameter
mjName ("Jackson"); // This celebrity is Michael Jackson
```

# 2. Closures store references to the outer function's variables

```
function celebrityID () {
var celebrityID = 999;
// We are returning an object with some inner functions
// All the inner functions have access to the outer function's variables
return {
getID: function () {
// This inner function will return the UPDATED celebrityID variable
// It will return the current value of celebrityID, even after the changeTheID function changes it
return celebrityID;
},
setID: function (theNewID) {
// This inner function will change the outer function's variable anytime
celebrityID = theNewID;
}
}

}

var mjID = celebrityID (); // At this juncture, the celebrityID outer function has returned.
mjID.getID(); // 999
mjID.setID(567); // Changes the outer function's variable
mjID.getID(); // 567: It returns the updated celebrityId variable
```

# 3. Closures Gone Awry

```
// This example is explained in detail below (just after this code box).
function celebrityIDCreator (theCelebrities) {
var i;
var uniqueID = 100;
for (i = 0; i < theCelebrities.length; i++) {
theCelebrities[i]["id"] = function () {
uniqueID = uniqueID + 1;
return uniqueID + i;
}
}
return theCelebrities;
}

var actionCelebs = [{name:"Stallone", id:0}, {name:"Cruise", id:0}, {name:"Willis", id:0}];

var createIdForActionCelebs = celebrityIDCreator (actionCelebs);

var stalloneID = createIdForActionCelebs [0];
console.log(stalloneID.id()); // 103
```

# Q&A

# Thanks for your listening!