# ADVANCED JAVASCRIPT

Basics, Fundamentals
and Technical Details

## What does “use strict” do?

- **Strict mode** allows you to put a program or a function into a **strict operating context**. It makes debugging easier bringing code errors to the fore instead of failing silently. Enter **“use strict”;** at the top of the JavaScript file to enable **strict mode**. (It is a string to accommodate older browsers when it first came out). You can also put **“use strict”;** at the top of a function block to only affect that function, usually in legacy JavaScript code.

- An example of a common bug is when an undefined variable is given a value, without using **strict mode** it will automatically become a **global** variable. That can make it very difficult to find a simple bug like a syntax error or typo when trying re-assign a variable that has already been declared. **use strict** also stops you from using words that are reserved for future versions of JavaScript like **“let”**.

- You cannot not delete variables, functions, or arguments to functions in use **strict mode**.

- Makes **eval()** much safer to use

## Does JavaScript pass variables by reference or by value?

- Primitive types such as **Strings**, **Numbers**, and **Booleans** are passed by value… and **Objects** are passed by reference.

## What is pass by value?

- This means that if you change the value of a **primitive variable** inside of a function the changes won’t affect the variable in the **outer scope**. Passing in by value is essentially passing in a **copy** of the variable with the **value matching**.

## What is pass by reference?

- When changing the value of a property on an **object** you are actually changing **that object**, not just a copy of it. Inside of a function these changes still take effect in the **outer scope**. Can only add a property or change a property, cannot change what the variable points to (cannot change entire object).

## What are the different types in JavaScript?

- Five primitive types: **Boolean**, **Number**, **String**, **Null**, **Undefined**. One non-primitive type: **Object**, can be object literal like: **var a = {}**, or you can instantiate it with: **new Object();**

## What is the difference between undefined and null?

- **undefined** is used by Javascript when a variable has **no value**. Uninitialized variables, missing parameters for functions, unknown variables, or an unknown property of an object. **undefined** is a core JavaScript function used by the JavaScript engine to let you know that one of the things listed is what's wrong.

- **null** is used by programmers to indicate no value. The JavaScript engine will always set an unknown variable to **undefined**, only a programmer can set it to **null**.

## What's the difference between a dynamically typed language (Javascript) and a statically typed language (Java)?

- In statically typed languages you have to specify what type the variable will hold **before** assigning the **value**. You cannot change the type once it is defined in Java, but in JavaScript you can. In JavaScript (dynamically) the type of the variable is assigned at **runtime**, in Java (static) the type is defined statically when you write the code in the file.

## What's the difference between == and ===?

- **==** is called **equality**. This checks for only the **equality** of the **values**. JavaScript tries to intelligently **convert** the types to match and then compare the values, this is called type **coercion**.

- **===** is called **strict equalit**y. This checks for both the **equality** of the **types** and the **values**.

## What is NaN and how can you check for it?

- Not a Number. **NaN** is the only JavaScript value that compared to itself **(NaN == NaN)** returns false. **isNaN(NaN)** returns true. **isNaN()** performs type coercion and will return false if a number inside of a string is passed to the function: **isNaN("5")** returns false.

- Because **NaN** is considered not equal to itself (or anything else) you can test to see if a variable is indeed **NaN** by using the not equal to strict equality comparison: **var a = NaN**; **a !== a   // returns true**

## What are the different scopes in JavaScript?

- **Scope** is the lifetime of a variable i.e. where that variable is **visible** and **available** to use in your code. Any variable declared **outside** of a function are called **global** variables and are available from **any part** of your application, even deeply nested inside of functions, as long as it wasn't redefined inside of another scope beforehand. All **global** variables are actually **properties** of the **window object** in browsers and the **global object** inside of node.

- **Local scope** variables are defined **inside** of a function. These variables **do not** exist **outside** of the function. JavaScript does not have block level scope, so for example, a variable declared **inside** of a **loop** is available in the **global** scope.

## What is variable hoisting?

- **Variable hoisting** is when JavaScript automatically **hoists** the declaration of a variable to the **top** of its **current scope**. JavaScript also performs hoisting on **functions** as well.

- Example:

```
var salary = '$1000';
(function() {
  console.log('weekly income is' + salary);
  var salary = '$2000';
})()
```

- The console prints **"weekly income is undefined"** because the variable is declared without a value at the top of the function due to **variable hoisting**.

## What is the scope chain?

- The **scope chain** is when a variable is being used **inside** a nested inner function it looks **up** the **scope chain**. It **first** looks at variables **inside** its own function scope, if it isn't found there is will go search the **outer** function's **scope**, and it will keep going **outer** and **outer** until it reached the **global scope**.

- The **scope chain** is defined lexically, meaning that the **scope chain** is defined in the **order** in which the code is **written** in the file.

## What is an IIFE and why might you use it?

- **IIFE** stands for **Immediately-Invoked Function Expression** (pronounced " **iffy** "). This is used so when **separate** JavaScript files are used for an app, **none** of the variables from the separate files will be given **global scope** over the entire **application**.

- Example:

```
(function() {
   var thing = { 'hello' : 'main' };
   console.log( thing );
})();
```

## What are function closures?

- **Function closure** is when a function returns another function. The function that is returned keeps a **reference** to any variables that it **needs** to run. It can refer to variables in **outer** functions even when **outside** of those functions. In this way it **bypasses** normal scope rules

- A very **important** fact is that when the closure is created it **does not** get a copy of the variable at the **current state**, in a loop for example, it **instead** gets the value of it in the **outer** scope, so the value would be the **global** value of the variable once the loop has completed.

- Example:

```
var foo = [ ];
for (var i = 0; i < 10; i++) {
   foo[ i ] = function() { return i };
}

console.log( foo[0]());   // 10 not 0
console.log( foo[1]());   // 10 not 1
console.log( foo[2]());   // 10 not 2
```

- To **fix** this problem you can use an **IIFE inside** of the loop and define a new variable within the **local scope**:

```
var foo = [ ];
for (var i = 0; i < 10; i++) {
   (function() {
          var y = i;
          foo[ i ] = function() { return y };
   })();
}
```

## What does the *this* keyword mean?

- In JavaScript ***this*** is defined by the calling **context**. By default JavaScript sets ***this*** as the global **window object**. When setting a property **directly inside** of an object the ***this*** keyword will refer to that **object**, while making a function for example, but make a **nested** function within that, and the ***this*** keyword refers back to the **global** window object again. This mistake can be **avoided** by using the **"use strict"** keyword at the top of the function, because any ***this*** that does not represent the **local** object will return as **undefined**.

- Another great way to **avoid** errors is to **stabilize** the ***this*** keyword into **variable** at the top of the **function**.

- Example:

```
var jacob = {
  checkThis: function() {
    var self = this;
    console.log(self);

    function checkOther() {
      console.log(self);
    }
  }
}
```

## What do the functions call, bind, and apply do?

- These are different ways in JavaScript of **locking down** what the ***this*** keyword means when **executing** different functions. It's very **important** to remember that **functions** are also **objects** in JavaScript, so you can **add properties** to a function the same way you can add properties to an object. **call**, **bind**, and **apply** are functions you can call on a function.

- **exampleFunction.call();** does the same thing as calling a function the **standard** way, but by using call you can **pass** into the function what you would like ***this*** to be. For example, when running **exampleFunction.call(1);** the value of ***this*** is **1**. You can also pass other parameters into **call()** if the function you are attaching it to **requires** them. The first parameter defines ***this***, everything after are the **parameters** your function would **normally** take: **exampleFunction.call(1, a, b, c);**

- **exampleFunction.apply();** is very **similar** to the **call();** function, the first parameter you pass into the **apply()** function is what you would like to **apply** as the ***this*** value for the function that you're **calling**. Any additional **parameters** you are passing into your function are put into an array like this: **exampleFunction.apply(1, [a, b, c]);** You would normally use **call()** instead of **apply()** unless the function took a variable that is an **array** for one of the **parameters**.

- **bind();** is used to assign a value to ***this*** at the time you **define** the function expression (not at the time you **call** the function expression like both **call** and **apply**). The **bind** keyword **only** works with function **expressions** i.e. defining a function to a **variable**. This saves the function as the variable **object** and you are adding the **bind** property to it.

- Example:

```
var a = function() {
   console.log(this);
}.bind(1);
```

## What is the prototype chain?

- Every object in JavaScript has a **prototype**. When looking for a **property** on an object JavaScript will **first** attempt to find the property on the object **itself**, if it can't find it it will **then** search on the object's **prototype** and so on. The **prototype** will point to **another** object so JavaScript will **then** search that **object** for the **property**, if that object does not have it JavaScript will **look** to see if the **prototype** of that object **points** to another new **object** and search for the **property** there. It will **keep** looking for that **property** down the **prototype chain**, and will only return **undefined** if none of the objects on the **prototype chain** have the property it is looking for.

- This works like **single-parent inheritance**, every object will **inherit** from one other **object**. You can check to see if one object is a **prototype** of another with the **isPrototypeOf()** function. Updating a property that previously only existed on the **prototype** of an object will add it to the actual object and not the **prototype** of the object.

- using **Object.create()** will create a new object and assign the parameter given as the **prototype** of the new object.

- Example:

```
var animal = {
  kind: 'human'
};
var jacob = Object.create(animal);

console.log(jacob.kind);   // prints 'human' to the console
```

## What is the difference between prototypal and classical inheritance?

- **Classical inheritance** is the methods of **object orientation** used in older languages like **Java** and **C++**. There is **class** which acts as a **blueprint** or design and then there is an **instance** of that **class**. The **class** is like an architectural **diagram** and the **instance** is a **house** built with that architectural **diagram**. You **cannot** live in the **diagram (class)** you **can** only live in the **house (instance)**.

- **Prototypal inheritance** in JavaScript creates new objects out of **existing objects**. There **isn't** an architectural **diagram** for a house, you just **build** a house **based** on an existing **house**. There is a way in JavaScript to **emulate** what looks like **classical inheritance**, but **all** inheritance in JavaScript it **prototypal**.

## What is the Constructor OO pattern?

- The **constructor pattern** is sometimes called **Pseudo Classical Inheritance**. JavaScript can mimic a class of older languages by using a **constructor function**. This can be done by using the **new** keyword. The ***this*** keyword is set to the constructor function when using the **new** keyword, much like when using the **call**, **apply**, or **bind** keywords.

- Example:

```
function Person(first_name, last_name) {
    this.firstName = first_name;
    this.lastName = last_name;
    this.fullName = function() {
        return this.firstName + ‘ ‘ + this.lastName;
    };
};

var dude = new Person(“Jacob”, “Erling”);
console.log(dude.fullName());  // prints “Jacob Erling” to the console
```

- You can also use **prototype** to add functions or properties to a pseudo class in JavaScript. If you are adding multiple instances of the pseudo class, this will add the function to the **prototype** and will be available to all those instances. Below will be an example of adding the fullName function using **prototype** instead of adding it inside the body of the **constructor function**.

- Example:

```
Person.prototype.fullName = function() {
    return this.firstName + ‘ ‘ + this.lastName;
}
```

## How do you implement inheritance in the constructor pattern?

- By using the **Object.create();** function with the object **prototype**. This will create your own **prototype chain** for your new **class**, thus creating **inheritance** between your **objects**. You can **continually** do this down the **prototype chain**.

- Example: **Professional.prototype = Object.create(Person.prototype);**

## What is the Prototype OO pattern?

- **Prototypal inheritance** is an **alternative** Object-oriented solution for JavaScript **instead** of **classical inheritance**. You have **already** worked with it and it is a lot **easier** to understand. It is just the **prototype chain**, nothing more, nothing less. You don't create a **class** or **pseudo-class** like with classical inheritance, there are no function **constructors** and you don't use the **new** keyword. You **basically** say "Hey **create** this object who's **prototype** is this other **object**, oh and also **bootstrap** this new object with some **properties**". One of the **benefits** of using the **prototype pattern** is that you're using the **tools** JavaScript has to offer **natively**, rather than attempting to **imitate** features of other languages with **fake** classes.

- Example:

```
var Person = {
    fullName: function() {
        return this. first_name + ‘ ‘ + this.last_name;
    };
};

function PersonFactory(first_name, last_name) {
    var person = Object.create(Person);
    person.first_name = first_name;
    person.last_name = last_name;
    return person;
}

var jacob = PersonFactory(“Jacob”, “Erling”);
```

## What is CORS?

- **CORS** stands for **Cross Origin Resource Sharing**. CORS allows you to **break** the same origin policy of a browser. That is when the browser **blocks** data that is coming in from another **server**. For example, you can have your main **application** on your **server** where the **browser** it getting its information from, but it your app **sends** out a **request** to an **API's server**, the browser will **block** the response from propagating into your code and **application**. The reason it exists is because it is a **security feature**. **CORS** is a mechanism that **allows** you to selectively **unblock** certain requests.

- The first way to implement **CORS** is with a simple **GET request**. The browser will add the **Origin** in the **header** of that **request** (the domain who's website the browser is currently displaying). For **CORS** to take effect the **response** has to have the header **Access-Control-Allow-Origin** with the value the **same** as the value of the **Origin** header. If the value of the **Access-Control-Allow-Origin** header is * that is **another way** for the browser to **accept** the response, it is **basically** the API's way of **saying anyone** can use it. If the **Access-Control-Allow-Origin** has anything **other** than those two **responses** the browser will **block** the response and your JavaScript code will **not** even see the data. It is **important** to remember that it is **always** the response getting **blocked** by the browser from **propagating** back into your code, it **is not** the **request** getting blocked by the **server** it is sent to.

- There is **another** method in which **CORS** handles **PUT**, **POST**, or **DELETE** requests. To handle this **CORS** sends something called a **Preflight Request**. This is basically an **HTTP** request with the **OPTIONS method**. The **OPTIONS method** sends the **additional** header **Access-Control-Request-Method** with the value being the **method** it wants to send to the **server (PUT, POST, PATCH, DELETE)**. The **server** must respond with the **same** method in the **Access-Control-Allow-Methods** header in the **response**. Only when this **response** has been **received** will the browser and the server be able to perform actual **responses** and **requests**.

## What is JSONP?

- You know that **JSON** is **JavaScript Object Notation**, what **JSONP** stands for is **JSON with Padding**. It **predates CORS** and was created as a **pseudo-standard** way to retrieve data from different domain. In this way it solves the **same** problem that **CORS** solves but it does have a few **limitations**. The main limitation is that it **only works** with **GET** requests, it does **not** work with **PUT**, **POST**, or **DELETE** requests. Even with that limitation it **does** have a lot of **uses**, for example, you can use **JSONP** to query the **Yahoo Weather Service** and it will **work just fine** on any browser with no extra **configuration**. So if you just want to use a **GET** request and the **API** allows **JSONP** than it is a much **simpler** solution. **JSONP** wraps a **typical JSON** response in a **function**, making it a **valid** piece of **JavaScript**.

**What is the difference between event capturing and event bubbling?**

- When you click on a page the **event** always starts from the **root (window)** goes to the **target**, and then back up to the **root** again. The part that goes from **the root to the target** is called the **Event Capturing Phase (phase one)**. The part that goes from the t**arget back to the root** is called the **Event Bubbling Phase (phase two)**. When you add an **Event Listener** you can **choose** which **phase** you want it to **listen** to. If you don't **specify** which event you want it to **listen** to it will by **default** listen to the **Event Bubbling Phase**.

## What is the difference between stopPropagation() and preventDefault()?

- **stopPropagation()** actually stops the event from **proceeding** down the **Event Capturing Phase** or going back up the **Event Bubbling Phase**. No **listeners** will be called after the **event** has been told to **stop** propagating. It stops the event from **moving** to the next **callback**. Called by **event.stopPropagation();**

- **preventDefault()** does not stop the event from **propagating** or from **moving** along its back from **root to target** or **vice versa**. What it does do is **prevent** the event from **performing** its **default** behavior that the event would have **triggered** in whatever element you perform the **event** on. For example, it **prevents** a checkbox from checking off or it **prevents** a form submission from **reloading** the page. Called by **event.preventDefault();**

Source: **Asim Hussain** - Advanced JavaScript | Udemy