`this` in JavaScript

`this` in JavaScript

- One of the most confusing concepts in JS is the ‘this’ keyword.
  - In most languages, ‘this’ is a reference to the current object instantiated by the class.
- In JavaScript, ‘this’ normally refers to the object which ‘owns’ the method, and it depends on how a function is called.
- If there’s no current object, ‘this’ refers to the global object.
  - In a web browser, that’s ‘window’

`this` inside functions

If we have a function `f()` that uses `this`, then we determine the meaning/object it refers to: *(how it's been called, not where it is sitting lexically)*

```
var x = 5; // var is used intentionally

function foo() {console.log(this.x);}
foo();

const obj = {x: 10, bar: function () {console.log(this.x);}};
obj.bar();

const bar2 = obj.bar;
bar2();

obj.foo = foo;
obj.foo();
```

`this` inside event handler

- When using `this` inside an event handler, it will always refer to the invoker.

```
var changeMyColorButton1 = document.getElementById("btn1");
var changeMyColorButton2 = document.getElementById("btn2");

changeMyColorButton1.onclick = changeMyColor;
changeMyColorButton2.onclick = changeMyColor;
function changeMyColor () {
        this.style.backgroundColor = "red";
}
```

Be careful!

```
function a() {
        this.newvariable = 'hello';
}

console.log(newvariable); // ReferenceError: newvariable is not defined(…)
a(); // this = window
console.log(newvariable); // hello
```

Self Pattern – The Problem

```
var a = {
        greet: '',
        log: function() {
                this.greet = 'Hello';
                console.log(this.greet); // "Hello "
                var changeGreet = function(greet) {
                                        this.greet = greet;
                                }
                changeGreet('Bonjour');
                console.log(this.greet); // "Hello
        }
}
a.log();
```

"this" here refer to **window** as changeGreet() is invoked without context

Self Pattern – The Solution

```
var a = {
        name: '',
        log: function() {
                var self = this; // self = a Object
                self.name = 'Hello';
                console.log(self.name); // Hello
                var changeName= function(newname) {
                                        self.name = newname;
                                }
                changeName('Bonjour');
                console.log(self.name); // Bonjour
        }
}
a.log();
```

Arrow functions (ES6)

- Arrow functions are function shorthand using => syntax.
- Syntactically similar to Java 8, lambda expressions

- Two factors influenced the introduction of arrow functions:
  - Shorter functions
  - Non-binding of this

Arrow functions (ES6)

Arrow functions can be a shorthand for an anonymous function in callbacks.

```
(arguments) => { return statement } // general syntax
  argument => { return statement } // one parameter
      argument => statement // implicit return
            () => statement // no input
```

```
function multiply (num1, num2) {
        return num1 * num2;
}
var output = multiply(5, 5);
```

```
var multiply = (num1, num2) => num1 * num2;
var output = multiply(5, 5);
```

But how do we implicitly return an object? `() => ({     })`

`this` inside arrow functions

```
Button1.addEventListener('click', function(){
        // this = Button1
        this.classList.toggle("highlight");

        setTimeOut( () => this.classList.toggle("highlight"), 1000);

        setTimeOut( function(){ return this.classList.toggle("highlight"); }, 1000);
})
```

Button1

Window

`Call`, `Apply` and `Bind`

There are many helper methods on the Function object in JavaScript

```
var func2 = func.bind(this);
func.call(this, param1, param2 ...);
func.apply(this, [param1, param2 ...]);
```

- Use `.bind()` when you want that function to later be called with a certain context, useful in events.
- Use `.call()` or `.apply()` when you want to invoke the function immediately, and modify the context.

- Call/apply call the function immediately, whereas bind returns a function that when later executed will have the correct context set for calling the original function.
  - This way you can maintain context in aysnc callbacks, and events.

Function Invocation Example

```
var me = {
        first: 'Jim',
        last: 'Carrey',
        getFullName: function() {
                return this.first + ' ' + this.last;
        }
}

var log = function(height, weight) { // 'this' refers to the invoker
        console.log(this.getFullName() + height + ' ' + weight);
}

log.call(me, '180cm', '70kg'); // Jim Carrey 180cm 70kg
log.apply(me, ['180cm', '70kg']); // Jim Carrey 180cm 70kg

var logMe = log.bind(me);
logMe('180cm'); // Jim Carrey 180cm undefined
```

Function Borrowing

```
var me = {
        first: 'Jim',
        last: 'Carrey',
        getFullName: function() {
                return this.first + ' ' + this.last;
        }
}

var you = {
        first: 'George',
        last: 'Smith'
}

console.log(me.getFullName.apply(you)); // George Smith
```

Main Point
the keyword ‘this’

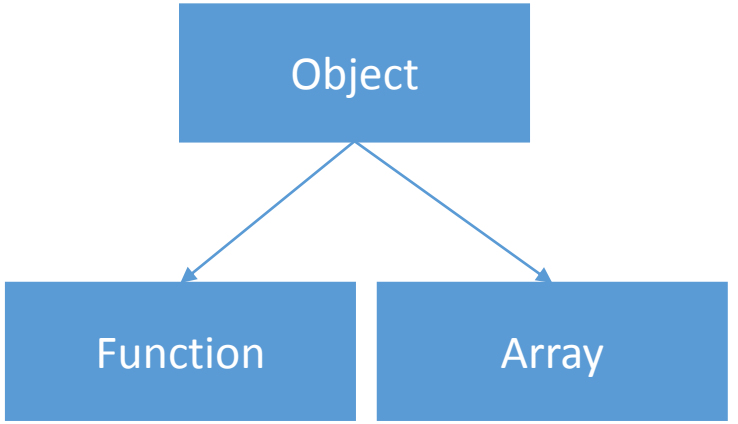
- In JavaScript, like Java, the keyword ‘this’ refers to the containing object. However, in JavaScript the same ‘this’ can refer to many different types of objects depending on the context.

- **Science of Consciousness:** The keyword ‘this’ is an important form of self-referral and understanding this self-referral is critical to writing successful JavaScript. Experiencing and understanding self-referral consciousness is critical to living a successful life.

JavaScript Closures & IIFE

Functions Review

- Functions define a new scope
- Functions are objects
- Functions are first-class citizens
  - Assign them to variables
  - Pass them around as parameters
  - Create them of the fly
- Functions can be anonymous (name property is empty)
- Functions are invokable




```
function eat(){
        console.log(eat.meal);
}
eat.meal = "pizza";
eat()
```

Notice how we added a property to the function

Recall: Calling an inner function

```
function init() { //function declaration
 const name = "Mozilla";
 function displayName() {
      console.log(name);
 }
 displayName();
}
init();
```

Returning an inner function

```
function makeFunc() {
 const name = "Mozilla"; //local to makeFunc
 function displayName() {
   console.log(name);
 }
 return displayName;
}
const myFunc = makeFunc();
myFunc();
```

- Q:is the local variable still accessible by myFunc?
- A: yes. Example of saving local state inside a JavaScript closure.

Closures

Closure

A first-class function that binds to free variables that are defined in its execution environment.

Free variable

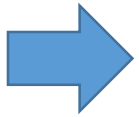
A variable referred to by a function that is not one of its parameters or local variables.

Closure Example

```
var x = 1;

function f() {
        var y = 2;
        var summ = function() {
                        var z = 3;
                        console.log(x + y + z);
                    }; //  inner function closes over free variables x, y as it is declared
        y = 10;
        return summ;
}

var g = f();
g(); // 1+10+3 is 14
```

Common Closure Bug and solution

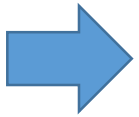
```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() {
            return i; // closure
    }
};


console.log(funcs[0]()); // 5
console.log(funcs[1]()); // 5
console.log(funcs[2]()); // 5
console.log(funcs[3]()); // 5
console.log(funcs[4]()); // 5
```



```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = helper(i);
};

function helper(m) {
     return function() { return m; }
}

console.log(funcs[0]()); // 0
console.log(funcs[1]()); // 1
console.log(funcs[2]()); // 2
console.log(funcs[3]()); // 3
console.log(funcs[4]()); // 4
```

Solving the Closure Bug with ES6

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() {
            return i; // closure
    }
};

console.log(funcs[0]()); // 5
console.log(funcs[1]()); // 5
console.log(funcs[2]()); // 5
console.log(funcs[3]()); // 5
console.log(funcs[4]()); // 5
```



```
var funcs = [];
for (let i = 0; i < 5; i++) {
    funcs[i] = function() {
            return i; // closure
    }
}; // will use separate i each time!

console.log(funcs[0]()); // 0
console.log(funcs[1]()); // 1
console.log(funcs[2]()); // 2
console.log(funcs[3]()); // 3
console.log(funcs[4]()); // 4
```

Main Point

Closures are created whenever a function binds to free variable(s) that are defined in its execution environment. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

Science of Consciousness: Transcendental consciousness provides encapsulation of thoughts, perceptions and actions by our Self, pure awareness. This experience provides a common stable positive blissful environment for all point value thoughts, perceptions and actions.

Immediately-Invoked Function Expression

```
(function(params) {
        statements;
})(params);
```

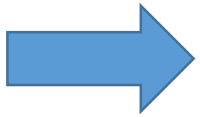
=

```
(function(params) {
        statements;
}(params));
```

- Declares and immediately calls an anonymous function
  - Parenthesis are an expression that wraps a function expression that will be immediately invoked
    - **“immediately invoked function expression (IIFE)”**
  - Used to create a new scope and closure around it
  - Can help to avoid declaring global variables/functions
  - Used by JavaScript libraries to keep global namespace clean

Solving the Closure Bug with IIFE

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() {
            return i; // closure
    }
};

console.log(funcs[0]()); // 5
console.log(funcs[1]()); // 5
console.log(funcs[2]()); // 5
console.log(funcs[3]()); // 5
console.log(funcs[4]()); // 5
```



```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = (function(n) {
        return function() { return n; }
    })(i);
};

console.log(funcs[0]()); // 0
console.log(funcs[1]()); // 1
console.log(funcs[2]()); // 2
console.log(funcs[3]()); // 3
console.log(funcs[4]()); // 4
```

Practical uses of closures

- A closure lets you associate some data (the environment) with a function—parallel to properties and methods in OOP.
- Consequently, you can use a closure anywhere you might use an object with a single method.
  - objects have properties to capture state info
  - JavaScript closures capture state info by saving references to free variables
- Situations like this are common on the web.
  - Making reusable event handlers using function factory.
    - Achieving reusable event handlers without closure is not easy.
  - Closures also very useful in JavaScript for encapsulation and namespace protection

Class work

- Part 1 (Make it work)
  - Create a page with three html buttons with value ‘size-12’, ‘sizee-14’ and ‘size-16’ such that when these buttons are clicked font size for whole page body is changed to 12px, 14px and 16px respectively. See demo code below.

```
<p>Some text in a paragraph.</p>
<button id="makeSize32">Make body font size 32px</button>

<script>
    window.onload = function () {
        document.getElementById("makeSize32").onclick = makeSize32;

        function makeSize32() {
            document.body.style.fontSize = "32px";
        }
    }
</script>
```

Class work

- Part 2 (Refactor)
  - If you have used three functions for above solution, try to refactor it to single reusable function.
  - Don’t go not next slide unless you tried this for a while

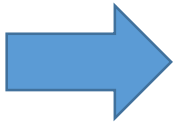
Function factory with closures

```
<p>Some paragraph text</p>
<button id="size-12">12</button>
<button id="size-14">14</button>
<button id="size-18">18</button>
```

```
function makeSizer(size) {
        return function() {
                document.body.style.fontSize = size + 'px';
        };
}

document.getElementById('size-12').onclick = makeSizer(12);
document.getElementById('size-14').onclick = makeSizer(14);
document.getElementById('size-18').onclick = makeSizer(18);
```

Namespace protection using module

```
// old: 3 globals
var count = 0;
function incr(n) {

        count += n;
}
function reset() {
        count = 0;
}
incr(4);
incr(2);
console.log(count);
```



```
// new: 0 globals!
(function() {
  var count = 0;
  function incr(n) {

          count += n;
  }
  function reset() {
          count = 0;
  }
  incr(4);
  incr(2);
  console.log(count);
})();
```

Avoids common problem with namespace/name collisions

Namespace protection using object

```
var com = {};

if (!com.example) { // make sure you are not overwriting example property of com
    com.example = {
        property1: value1,
        property2: value2,
        ...
        method1 : function(data) {...},
        method2 : function(data) {...},
        ...
    };
}
```

Revealing Module Pattern

```
/* widely used in single page web apps */
const Module = (function() {
 const privateMethod = function() {…};
 const someMethod = function() {…};
 const anotherMethod = function() {…};
 return {
   someMethod: someMethod,
   anotherMethod: anotherMethod
 };
})();
```

Accessing Private Methods

```
const myModule = (function() {
 const privateMethod = function(message) { console.log(message); };
 const publicMethod = function(text) { privateMethod(text); };
 return {
    publicMethod: publicMethod
 };
})();

// Example of passing data into a private method
// Private method will console.log() 'Hello!'
myModule.publicMethod('Hello!');
```

Access Private Variables

```
const Module = (function() {
 const privateArray = [];
 const publicMethod = function(something) {
    privateArray.push(something);
 };
 return {
    publicMethod: publicMethod
 };
})();
```

Extending Modules

```
/* very easy due to dynamic nature of JavaScript—can dynamically add
  properties to objects */
const Module = (function() {
 const privateMethod = function() {...};
 const someMethod = function() {...};
 const anotherMethod = function() {...};
 return { someMethod: someMethod, anotherMethod: anotherMethod };
})();

Module.extension = function() {
 // another method! (Q:  public or private?)
};
```

Example (revealing module pattern)

```
var counter = (function() {
        var privateCounter = 0; //private data
        function changeBy(val) { //private inner function
                privateCounter += val;
        }
        return { // three public functions are closures
                increment: function() { changeBy(1); },
                decrement: function() { changeBy(-1); },
                value: function() { return privateCounter; }
        }
})();

alert(counter.value()); /* Alerts 0 */
counter.increment();
counter.increment();
alert(counter.value()); /* Alerts 2 */
counter.decrement();
alert(counter.value()); /* Alerts 1 */
```

Main Point
Revealing Module Pattern

- The revealing module pattern is widely used to provide a public API to an underlying implementation of private methods and properties.
- **Science of Consciousness:** The Transcendental Meditation program is a sort of API to access the support of all the laws of nature through the experience of pure consciousness, the source of all the laws of nature.

Question

- How would you change the code in earlier slide if you need more than one instances of counter?

Object factory using closure

```
var makeCounter = function() {
        var privateCounter = 0; //private data
        function changeBy(val) { //private inner function
                privateCounter += val;
        }
        return { // three public functions are closures
                increment: function() { changeBy(1); },
                decrement: function() { changeBy(-1); },
                value: function() { return privateCounter; }
        }
};

var counter1 = makeCounter();
var counter2 = makeCounter();

alert(counter1.value()); /* Alerts 0 */
counter1.increment();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Life Is Found in Layers

1. JavaScript is a functional OO language that has a shared global namespace for each page and local scope within functions.
2. Closures and objects are fundamental to JavaScript best coding practices, particularly for promoting encapsulation, layering, and abstractions in code.

---

3. **Transcendental consciousness** is the experience of the most fundamental layer of all existence, pure consciousness, the experience of one's own Self.

4. **Impulses within the transcendental field:** The many layers of abstraction required for sophisticated JavaScript implementations will be most successful if they arise from a solid basis of thought that is supported by all the laws of nature.

5. **Wholeness moving within itself:** In unity consciousness, one appreciates that all complex systems are ultimately compositions of pure consciousness, one's own Self.