# CS472 WAP
# Call Context

# Maharishi International University - Fairfield, Iowa

**Wholeness:** 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 promotes a successful and fulfilling life.

# ‘*this*’ keyword

- The ‘*this*’ keyword is basically a special variable that is created for every execution context and therefore any function.
- that it is one of the components of any execution context along with the variable object and scope chain.
- The value of ‘*this* ’ keyword is not static. So, its not always the same. It depends on how the function is called. The value of ‘*this*’ variable is only assigned when the function is actually called at runtime.
- The value of this can be determined depending on how the function is called. In this lecture, lets understand how value of *this* variable is determined.

# '*this*' keyword

- **Global scope**: the '*this*' keyword points to window object.
- **Regular function**: value of '*this*' is *undefined* in **strict mode** else it points to window object.
- **Method call**: the '*this*' variable points to object on which that method is called.
- **Arrow function**: arrow function does not have its own '*this*' keyword. It uses the '*this*' keyword of its parent scope (lexical scope).
- **Event handler**: the '*this*' keyword points to the DOM element on which the handler function is attached.

# 'this'

- In Java, every method has an implicit variable '`this`' which is a reference to the object that contains the method.
- Java, in contrast to JavaScript, has no functions, only methods
- So, in Java, it is always obvious what '`this`' is referring to

- In JavaScript, '`this`', usually follows the same principle
  - Refers to the containing object
  - If in a method, refers to the object that contains the method, just like Java
  - If in a function, then the containing object is '`window`'
    - in "`use strict`" mode ➔ `undefined`
  - Methods and functions can be passed to other objects!!
    - **'`this`' is then a portable reference to an arbitrary object**

# ‘this’ inside vs outside object

```
function greeting() {
    console.log(this);
}

let  user  = {
    firstName:"John",
    sayHi() {
        console.log(this);
    }
};

console.log(this); // this is window object
greeting(); // greeting() is called by global window object, this is window
user.sayHi(); //sayHi() is called by the object user, this is user
```

# .call() .apply() .bind()

- There are many helper methods on the Function object in JavaScript
  - `.bind()` when you want a function to be called back later with a certain context
  - `.call()` or `.apply()` when you want to invoke the function immediately and modify the context.
  - Can be used to manually change '`this`' context

```
var func2 = func.bind(anObject , arg1, arg2, …)// creates a copy of
  func using anObject as 'this' and its first 2 arguments bound to arg1
  and arg2 values
func.call(anObject, arg1, arg2...);
func.apply(anObject, [arg1, arg2...]);
```

# Solution: `.call()` `.apply()` `.bind()`

- several techniques to set the '`this`' context parameter

```
let  user  = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};

user.sayHi(); //works

let hi = user.sayHi();
hi()                //problem! - this refers to window object
hi.bind(user)();  //works
hi.call(user);    //works
hi.apply(user);   //works
```

# ‘Borrow’ a method that uses ‘this’ via call/apply/bind

```
const me = {
    first: 'John',
    last: 'Smith', getFullName: function()
    {
        return this.first + ' ' + this.last;
    }
}
const log = function(height, weight) { // ‘this’ refers to the invoker
    console.log(this.getFullName() + height + ' ' + weight);
}

const logMe = log.bind(me);
logMe('180cm', '70kg'); // John Smith 180cm 70kg

log.call(me, '180cm', '70kg'); // John Smith 180cm 70kg
log.apply(me, ['180cm', '70kg']); // John Smith 180cm 70kg
log.bind(me, '180cm', '70kg')(); // John Smith 180cm 70kg
```

# Self Pattern – problem with inner functions

```
const user = {
    salute: "",
    greet: function() {
        this.salute = "Hello";
        console.log(this.salute); //Hello
        const setFrench = function(newSalute) { //inner function
            this.salute = newSalute;
        };
        setFrench("Bonjour");
        console.log(this.salute); //Bonjour??
    }
};

user.greet(); //Hello  Hello  ??
```

# Self Pattern – Legacy Solution

```
const user = {
    salute: "",
    greet: function() {
        const self = this;
        self.salute = "Hello";
        console.log(this.salute); //Hello
        const setFrench = function(newSalute) { //inner function
            self.salute = newSalute;
        };
        setFrench("Bonjour");
        console.log(this.salute); //Bonjour
    }
};

user.greet(); //Hello Bonjour
```

- Self Pattern: Inside objects, always create a "`self`" variable and assign "`this`" to it. Use "`self`" anywhere else
- JavaScript functions (versus methods) use '`window`' as '`this`'
  - even inner functions in methods
  - Unless in strict mode, then '`this`' = `undefined`

# `this` inside arrow function (ES6)

- Also solves the Self Pattern problem
- 'this' will refer to surrounding lexical scope inside arrow function

```
const user = {
    salute: "",
    greet: function() {
        this.salute = "Hello";
        console.log(this.salute); //Hello
        setFrench = newSalute => this.salute = newSalute;
        setFrench("Bonjour");
        console.log(this.salute); //Bonjour
    }
};

user.greet(); //Hello   Bonjour
```

# Arrow functions inherit ‘this’ from lexical environment

- Arrow functions are not just a “shorthand” for writing small stuff. They have some very specific and useful features.
- JavaScript is full of situations where we need to write a small function, that’s executed somewhere else.

- `arr.forEach(func)` – `func` is executed by `forEach` for every array item.
- `setTimeout(func)` – `func` is executed by the built-in scheduler.

- spirit of JavaScript to create a function and pass it somewhere.
- in such functions we usually don’t want to leave the current context.
- That’s where arrow functions come in handy.

```
let group = {
    title: "Our Group",
    students: ["John", "Pete", "Alice"],

    showList: function() {
        this.students.forEach(function(student) {
            // error – ‘this’ is undefined (or window)
            console.log(this.title + ": " + student);
        });
    }
};
group.showList();
```

# Arrow functions best suited for non-method functions

- best practice to avoid arrow functions as object methods
  - Do not have their own ‘this’ parameter like function declarations/expressions
  - However, it is best practice to use them for inner functions in methods
    - Then inherit ‘this’ from the containing method and avoid the ‘Self Pattern’ problem

```
"use strict";
const x = { a: 1, b: 2, add() { return this.a + this.b } }
console.log(x.add());

const y = { a: 1, b: 2, add: () => { return this.a + this.b } }
console.log(y.add());
```

# CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

## Knowledge Is Different in Different States of Consciousness

1. Functions can be passed and called back from different objects.
2. Built-in function methods call/apply/bind can all set the ‘this’ context for function calls.

---

3. **Transcendental consciousness**. Is the experience of nonchanging pure consciousness.
4. **Impulses within the transcendental field:** Thoughts at the deepest levels of consciousness are most powerful and successful.
5. **Wholeness moving within itself:** In unity consciousness all knowledge is experienced in terms of its nonchanging basis in pure consciousness.