LECTURE 7
SCHEDULED CALLBACKS AND CALL CONTEXT

Knowledge is Different in Different States of Consciousness

Wholeness: JavaScript allows functions to be called back at scheduled times and with different contexts. All functions can access a private data structure containing context information referred to by the keyword ‘this’. Calling a function with different contexts enables code reuse and different behavior from the same code. Science of Consciousness: This is an example of knowledge being different in different contexts. Knowledge is different in different states of consciousness.

Main Point Preview: 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 promotes a successful and fulfilling life.

Problem with 'this' inside timeout

- There is a problem if you call a function using 'this' inside a timeout

```
let user = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};
user.sayHi(); //works
setTimeout(user.sayHi, 2000); //problem!
```

- 'this' represents the object calling the function
  - setTimeout is a global function, which means it is actually a method of window (or global in Node.js)
  - user.sayHi is a reference to the sayHi function, it has now been passed as an argument (callback) to the setTimeout method, when it is called inside setTimeout the lexical context and value of 'this' will be window

Function binding

- When passing object methods as callbacks, for instance to `setTimeout`, there's a known problem: losing "`this`"

- The general rule: '`this`' refers to the object that calls a function
  - since functions can be passed to different objects in JavaScript, the same '`this`' can reference different objects at different times
  - Does not happen in languages like Java where functions always belong to the same object

- `setTimeout` can have issues with '`this`'
  - sets the call context to be `window`

```
let user = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};
setTimeout(user.sayHi, 1000); // Hello, undefined
```

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`'
    - Not 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 w
indow
user.sayHi(); //sayHi() is called by the object user, this is user
```

Solution 1: a wrapper

```
let user = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};
setTimeout(function() { user.sayHi(); }, 2000); //wrapped versus just “user.sayH
i”
//Or
setTimeout(() => user.sayHi(), 2000);
```

- Works because ‘`this`’ references the calling object and now the user object is calling the function
- Closure?
  - free variable?
- This anonymous function wrapper technique can be used whenever you want to pass a function as a callback along with arguments
  - In this case we are, in effect, passing the ‘this’ argument for the function call

.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
  - http://stackoverflow.com/questions/15455009/javascript-call-apply-vs-bind

```
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 2~4: `call()` `.apply()` `.bind()`

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

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

user.sayHi(); //works
setTimeout(user.sayHi, 2000); //problem! - this refers to window object
setTimeout(user.sayHi.bind(user), 2000); //works
setTimeout(() => user.sayHi.call(user), 2000); //works
setTimeout(() => user.sayHi.apply(user), 2000); //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
```

`this` inside event handler

- When using `this` inside an event handler, it will always refer to the invoker. (`event.target`)
  - A very useful feature of '`this`' for JavaScript and DOM manipulation
  - Portable context
  - Rule: '`this`' refers to the object that called the function

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

changeMyColorButton1.onclick = changeMyColor;
changeMyColorButton2.onclick = changeMyColor;

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

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.

Main Point Preview: Bind and call context

Functions have built-in methods call, apply and bind that set the ‘this’ context of a given function. Call and apply execute the function immediately. Bind returns a new function object to be executed later with context and/or arguments set to specified values. Science of Consciousness: The same function can have different semantics depending on the ‘this’ context. Our own understanding can change depending on our level of awareness. Knowledge is different in different states of consciousness.

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 lexical 'this' [Continued]

- error occurs because `forEach` runs functions with `this=undefined`
  - Same logic as `window.setTimeout`, `window.prompt` etc
  - Language rule is '`this`' refers to calling object
  - `Array.forEach` and `window.setTimeout`
- That doesn't affect arrow functions, because they don't have '`this`' parameter
  - '`this`' comes from the parent lexical environment
- arrow function expressions are best suited for non-method functions
  - Methods need the 'this' parameter from the object

Exercise

- Exercise: fix the code
  - Using arrow function

```
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();
```

Fix: arrow function

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

    showList: function() {
        this.students.forEach(student => 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()); //3

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

Main Point: Bind and call context

Functions have built-in methods call, apply and bind that set the ‘this’ context of a given function. Call and apply execute the function immediately. Bind returns a new function object to be executed later with context and/or arguments set to specified values. Science of Consciousness: The same function can have different semantics depending on the ‘this’ context. Our own understanding can change depending on our level of awareness. Knowledge is different in different states of consciousness.

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.