LESSON 8
SCHEDULED CALLBACKS, CALL FORWARDING, 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 Points

1. Timers and scheduled callbacks
2. Decorators and call forwarding, call/apply
3. Function context manipulation with bind/call/apply and arrow functions

Main Point Preview: Timout callbacks

The asynchronous global methods setTimeout and setInterval take a function reference as an argument and then callback the function at a specified time. Science of Consciousness: Accepting an assignment and carrying it out at a designated time is a fundamental capability required for intelligent behavior. A clear awareness and mind promotes good memory and the ability to successfully execute tasks.

Timers

- setTimeout allows to run a function once after the interval of time.
- setInterval allows to run a function regularly with the interval between the runs.

setTimeout

let timerId = setTimeout(func, [delay], [arg1], [arg2], ...)

- Func: Function or a string of code to execute.
- Delay: delay before run, in milliseconds (1000 ms = 1 second), by default 0.
- arg1, arg2… : Arguments for the function

```
function sayHi() {
  alert('Hello');
}
setTimeout(sayHi, 1000);
```

- With arguments:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Pass a function, but don't run it

- Novice developers sometimes make a mistake by adding brackets ()

```
// wrong!
setTimeout(sayHi(), 1000);
```

- doesn't work,
  - setTimeout expects a reference to a function.
  - here sayHi() runs the function,
  - result of its execution is passed to setTimeout.
  - result of sayHi() is undefined (the function returns nothing), so nothing is scheduled

- function call versus function binding
  - sayHi() versus sayHi
  - execute the function versus reference to the function
  - fundamental concept!!

Canceling with clearTimeout

A call to setTimeout returns a “timer identifier” timerId that we can use to cancel the execution.

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

schedule the function and then cancel it

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier
clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

setInterval

The setInterval method has the same syntax as setTimeout:

```
let timerId = setInterval(func, [delay], [arg1], [arg2], ...)
```

Repeatedly calls the function after the given interval of time.

To stop further calls, we should call clearInterval(timerId).

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);
// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Zero delay setTimeout

- There’s a special use case: setTimeout(func, 0), or just setTimeout(func).

- schedules the execution of func as soon as possible.
  - after the current code is complete.

```
setTimeout(() => alert(“Hello"), 0);
alert(“World");
```

- The first line “puts the call into calendar after 0ms”
  - scheduler will only “check the calendar” after the current code is complete
  - "Hello" is first, and "World" – after it.

- There are also advanced browser-related use cases of zero-delay timeout, that we’ll discuss in the chapter Event loop: microtasks and macrotasks.

Exercises

- Output every second
- What will setTimeout show?

Main Point: Timout callbacks

The asynchronous global methods setTimeout and setInterval take a function reference as an argument and then callback the function at a specified time. Science of Consciousness: Accepting an assignment and carrying it out at a designated time is a fundamental capability required for intelligent behavior. A clear awareness and mind promotes good memory and the ability to successfully execute tasks.

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 is critical to living a successful life.

Problem with ‘this’ inside timeout

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

```
const abc = {a:1,  b:2,  add: function() { console.log("1+2 = 3?",this.a + this.b);  }}
abc.add(); //works
setTimeout(abc.add, 2000); //problem!
```

- ‘this’ represents the object calling the function
  - setTimeout is a global function, which means it is actually a method of window
  - abc.add is a reference to the add function
  - it has now been passed as an argument to the setTimeout method
  - when it is called inside setTimeout the lexical context and value of ‘this’ will be window
    - (or undefined if running in ‘strict mode’

Can be solved by setting the ‘this’ context

- several techniques to set the ‘this’ context parameter

```
const abc = {a:1,  b:2,  add: function() { console.log("1+2 = 3?",this.a + this.b);  }}
abc.add(); //works
setTimeout(abc.add, 2000); //problem!
setTimeout(abc.add.bind(abc), 2000); //works
setTimeout(function() {abc.add.call(abc)}, 2000); //works
setTimeout(function() {abc.add.apply(abc)}, 2000); //works
```

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' because setTimeout is a method on window

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`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 foo() { console.log(this); }
const bob = {
 log: function() {
   console.log(this);
 }
};

console.log(this); // this generally is window object
foo(); //foo() is called by global window object
bob.log();//log() is called by the object, bob
```

`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";
}
```

Solution 1: a wrapper

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }};
setTimeout(function() {  user.sayHi(); }, 1000);  //wrapped versus just “user.sayHi”
//Or
setTimeout(() => user.sayHi(), 1000);
```

- 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

Solution 2: bind

- Functions provide a built-in method bind that sets the value of ‘this’ for the function
  - Similar to call and apply except that bind does not execute the function
  - Returns ‘a’ new function object with the new value of ‘this’ as the context

```
let boundFunc = func.bind(context);
```

```
let user = {
  firstName: "John"
};
function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}
// bind this to user
let funcUser = func.bind(user);
funcUser("Hello"); // Hello, John (argument "Hello" is passed, and this=user)
```

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 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.

.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
    - useful in events. (ES5)
  - .call() or .apply() when you want to invoke the function immediately and modify the 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...]);
```

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

```
const me = {
 first: 'Tina',
 last: 'Xing',
 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'); // Tina Xing 180cm 70kg

log.call(me, '180cm', '70kg'); // Tina Xing 180cm 70kg
log.apply(me, ['180cm', '70kg']); // Tina Xing 180cm 70kg
log.call(me, '180cm', '70kg’)(); // Tina Xing 180cm 70kg
```

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

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

Self Pattern – Legacy Solution

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

abc.greet();
```

- 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 abc = {
  name: "",
  log: function() {
    this.name = "Hello";
    console.log(this.name); //Hello
    const setFrench = (newname => this.name = newname);  //inner function
    setFrench("Bonjour");
    console.log(this.name); //Bonjour
  }
};

a.log();
```

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
```

Exercises

- Bound function as a method
- Second bind
- Function property after bind
- Fix a function that loses “this” (do with bind, wrapper, call, and apply)
- Partial application for login

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.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

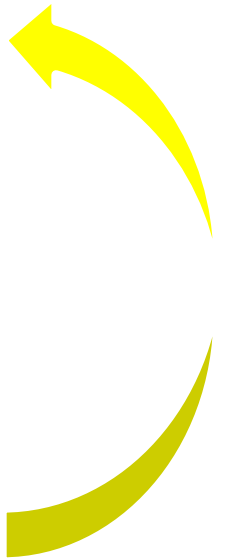
Knowledge Is Different in Different States of Consciousness

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.



Main Point Preview: Call forwarding and call context

Call forwarding passes a function and all arguments and context to another function. Call forwarding is used in the decorator pattern which adds wraps the forwarded function with extra functionality. Science of Consciousness: In this case the original functionality is enhanced by the wrapper functionality. This is like actions performed in higher states of consciousness where the same actions are performed, but they are enhanced by the influence of expanded awareness.

Decorators and call forwarding

- JavaScript gives exceptional flexibility when dealing with functions.
  - can be passed around,
  - used as objects,
  - now we'll see how to forward calls between them and decorate them.

- Transparent caching
  - caching is an example of decorator pattern
  - decorator pattern example of call forwarding
  - Imagine a function slow(x)
    - CPU-heavy,
    - for the same x it always returns the same result.
  - may want to cache the results to avoid spending extra-time on recalculations.
  - instead of adding functionality into slow() create a wrapper function that adds caching

Decorator benefits

- What cachingDecorator does (demo)
  - Only calls slow once with 1 and once with 2
  - slow not called again with same value
- cachingDecorator is reusable
  - can apply it to another function.
  - e.g., Fibonacci task in recursion section
- logic is separate, it did not increase the complexity of slow itself
- can combine multiple decorators if needed
- is there a closure?
  - free variables?

```
function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x);

    cache.set(x, result);
    return result;
  };
}
```

wrapper

around the function

Problem if use object method

- worker.slow() stops working after the decoration:
- error occurs in the line (*) that tries to access this.someMethod and fails. Can you see why?
- The reason is wrapper calls original function as func(x) in line (**)
  - the function gets this = undefined
- observe a similar symptom if we run:

```
let func = worker.slow;
func(2);
```

- wrapper passes the call to the original method, but without the context ‘this’

Fix cachingDecorator to work with methods

1. After the decoration worker.slow is now the wrapper function (x) { ... }
    1. See demo console.log(worker.slow)

2. worker.slow(2) is executed
    1. gets 2 as argument
    2. this=worker (it's the object before dot)

3. Inside the wrapper func.call(this, x) passes
    1. current this (=worker)
    2. current argument (=2) to the original method.

Extending cachingDecorator example (apply)

- we will add another extension to the cachingDecorator example
  - more practice with the concepts of object context (this) and closures
  - also illustrates where ‘apply’ might be used instead of ‘call’

- suppose want to cache a function that takes multiple arguments
  - the cachingDecorator used the single argument as key to the cache values
  - need to make a single key out of the multiple arguments
  - also need to adust func.call(this, x) for multiple arguments
    - func.call(this, arg1, arg2), or
    - func.call(this, …arguments)

```
let worker = {
  slow(min, max) {
    return min + max; // scary CPU-hogger is assumed
  }
};
// should remember same-argument calls
worker.slow = cachingDecorator(worker.slow);
```

Extending cachingDecorator example (apply) 2

- works with any number of arguments
- 2 changes
  - In the line (*) it calls hash to create a single key from the arguments array-like object
    - “3,5” for call worker.slow(3, 5)
  - (**) uses func.call(this, ...arguments) to pass both the context and all arguments
    - calls slow(this, 3, 5) where this is bound to worker
    - arguments is JS array-like object for the original worker.slow(3, 5) call
- Instead of func.call(this, ...arguments) we could use func.apply(this, arguments)
  - only syntax difference between call and apply is that call expects a list of arguments, while apply takes an array-like object with them

Call forwarding

- Passing all arguments along with the context to another function is called *call forwarding*.
  - When external code calls wrapper, it is indistinguishable from original function call

```
let wrapper = function() {
  return func.apply(this, arguments);
};

worker.slow = cachingDecorator(worker.slow);
```

Exercises

- instead of destructuring the arguments object, change the extended code (the version using ‘arguments’)
  - to use exactly 2 arguments for the min max args
  - Then change the code to use ‘apply’ instead of ‘call’
  - Do apply for both your 2 arguments version and the arguments object version
  - not as general as the arguments version, but good exercise to confirm understanding
- Spy decorator
  - (real world example sinon.spy for unit testing)
- Delaying decorator
  - interesting exercise of setTimeout

Main Point: Call forwarding and call context

Call forwarding passes a function and all arguments and context to another function. Call forwarding is used in the decorator pattern which adds wraps the forwarded function with extra functionality. Science of Consciousness: In this case the original functionality is enhanced by the wrapper functionality. This is like actions performed in higher states of consciousness where the same actions are performed, but they are enhanced by the influence of expanded awareness.

Partial functions

- We can bind not only this, but also arguments. That's rarely done, but sometimes can be handy let boundFunc = func.bind(context);

  let bound = func.bind(context, [arg1], [arg2], ...);

```
function mul(a, b) {
  return a * b;
}
let double = mul.bind(null, 2);
alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

- create a new function that passes calls to mul,
  - fixing null as the context and 2 as the first argument.
  - Further arguments are passed "as is".
- partial function application
  - create a new function by fixing some parameters of the existing one.

Why make a partial function?

- create an independent function with a readable name (double, triple).
  - omit first argument as it's fixed with bind.

- useful when have generic function and want a less universal variant
  - function send(from, to, text).
  - inside a user object we may want to use a partial variant of it: sendTo(to, text) that sends from the current user

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(){alert(this.title + ': ' + student);   //error – ‘this’ is undefined (or window)
      student => alert(this.title + ': ' + student)    //works as expected – ‘this’ comes from lexical environment, showList method
    );
  }
};
group.showList();
```

Arrow functions lexical ‘this’ 2

- 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: fix the code at right
  - Using arrow function
  - Using bind

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],
  showList() {
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student)
    });
  }
};
group.showList();
```