LESSONS 6 - 10
KEY CONCEPTS

Recursion
Recursive Structures
Closures
Scheduled Callbacks
Call Context

The Base Case and Reduction Step

- To stop recursion going into an infinite recursion
  (and exceed the max recursion depth)
  - Reduction step: We need to make sure that each recursive call moves us closer to a base case
  - Base case: returns without calling itself

- Recursion creates stack frames on the call stack until the base case
  - Then it comes back down through the frames

execution context and stack

- information about execution of a running function is stored in its execution context.
  - internal data structure with details about execution of a function:
    - where the control flow is now,
    - current variables,
    - value of this and few other internal details.
  - One function call has exactly one execution context associated with it.

- When a function makes a nested call, the following happens:
  - current function is paused.
  - execution context associated with it is remembered in execution context stack.
  - The nested call executes.
  - After it ends, old execution context is retrieved from the stack,
    - outer function is resumed from where it stopped.

execution context and stack for pow(2, 3)

```
1. function pow(x, n) {
2.   if (n == 1) {
3.     return x;
4.   } else {
5.     return x * pow(x, n - 1);
6.   }
7. }
```

{ x: 2, n: 1, at line 1 }

Function call	Exec context	Recursive call	return
Pow(2,1)	{ x: 2, n: 1, at line 1 }		2
Pow(2,2)	{ x: 2, n: 2, at line 5 }	Pow(2,1)	2 * 2
Pow(2,3)	{ x: 2, n: 3, at line 5 }	Pow(2,2)	2 * 4

When recursive calls are appropriate

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);  }}
```

```
function pow(x, n) {
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;}
```

- Contexts take memory.
  - A loop-based algorithm uses less memory
  - Clarity is generally more important than efficiency
  - Any recursion can be rewritten as a loop.

- When recursive calls are appropriate
  - When problems have a natural recursive structure and solution
  - E.g., Tree and list data structures.

Recursive traversals

- want a function to get the sum of all salaries.
  - departments may have subdepartments which may have subsubdepartments, …
  - looping: would need loops within loops … (could be arbitrary depth)

- recursive algorithm
  - “simple” department with an array of people
    - sum the salaries in a simple loop.
  - object with N subdepartments
    - N recursive calls to get the sum for each of the subdeps and combine the results

```
let company = {
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 600 }],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],  //subdepartments
    internals: [{name: 'Jack', salary: 1300}]
  }
};
```

Recursive traversals 2

```
function sumSalaries(department) {
  if (Array.isArray(department)) { // case (1)
    return department.reduce((prev, current) => prev + current.salary, 0); // sum the array
  } else { // case (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // recursively call for subdepartments, sum the results
    }
    return sum;  }}
console.log(sumSalaries(company)); // 6700
```

```
{
  sales: [{
    name: 'John',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 600
  }],

  development: {
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],

    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```

- The code is short and easy to understand (hopefully?).
  - power of recursion. It also works for any level of subdepartment nesting

- easily see the principle:
  - for an object {...} subcalls are made,
  - while arrays [...] are the “leaves” of the recursion tree, they give immediate result.

- Note that the code uses smart features that we’ve covered before:
  - Method arr.reduce explained in the chapter Array methods to get the sum of the array.
  - Loop for(val of Object.values(obj)) to iterate over object values:
    - Object.values returns an array of them

Linked list

- "Linked list" recursive structure might be better than array in some cases
  - problem with arrays.
    - "delete element" and "insert element" operations are expensive.
    - , arr.unshift(obj) operation must renumber all elements to make room for a new obj
      - if the array is big, it takes time.
    - Same with arr.shift().

  - The only structural modifications that do not require mass-renumbering are those that operate with the end of array:
    - arr.push/pop.

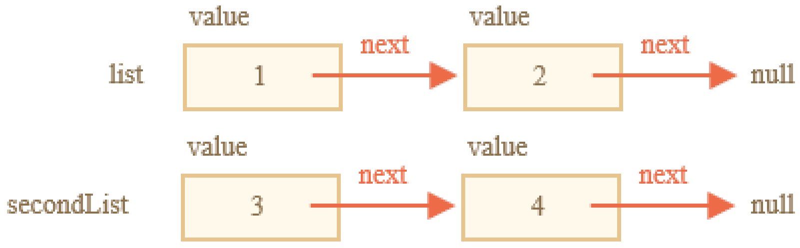
  - an array can be slow for big queues, when we must work with the beginning.
    - choose a linked list.

  - if need fast insertion/deletion,
    - choose a linked list.

Linked list operations

- easily split into multiple parts

```
let secondList = list.next.next;
list.next.next = null;
```



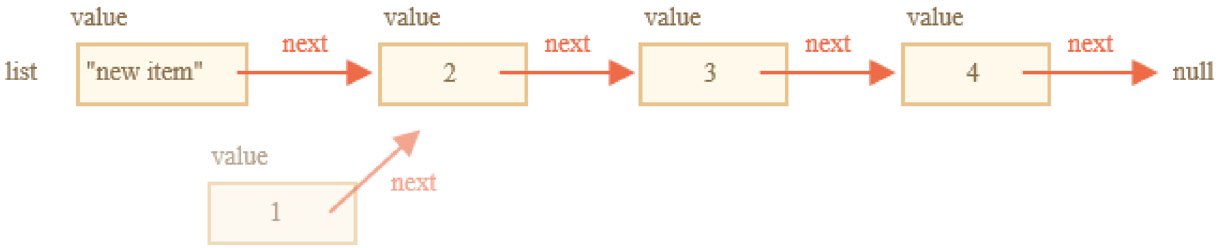
- How would you rejoin it?

- to prepend a new value, we need to update the head

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
// prepend the new value to the list
list = { value: "new item", next: list };
```

- To remove a value from the middle, change next of the previous one

```
list.next = list.next.next;
```



Rest parameters (ES6)

- rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function
- ES6 compatible code, then rest parameters should be preferred.

```
function sum(x, y, ...more) {
 //"more" is array of all extra passed params
 let total = x + y;
 if (more.length > 0) {
   for (let i = 0; i < more.length; i++) {
   total += more[i];
   }
 }
 console.log("Total: " + total);
 return total;
}
sum(5, 5, 5);
sum(6, 6, 6, 6, 6);
```

Spread operator (ES6)

- The same … notation can be used to unpack iterable elements (array, string, object) rather than pack extra arguments into a function parameter.

```
let a, b, c, d, e;
a = [1,2,3];
b = "dog";
c = [42, "cat"];

// Using the concat method.
d = a.concat(b, c); // [1, 2, 3, “dog”, 42, “cat”]

// Using the spread operator.
e = [...a, b, ...c]; // [1, 2, 3, “dog”, 42, “cat”]
copyOfA = [...a]  //[1, 2, 3]

let str = "Hello";
alert( [...str] ); // H,e,l,l,o
```

Spread operator 2 (ES6)

- make a (shallow) clone of an object

```
let a, b, c, d, e;
a = {a:1, b:2, c:3, d: 44}
b = { ...a }
console.log(b) // {a:1, b:2, c:3, d: 44}
b.a = 100;
console.log(a) // {a:1, b:2, c:3, d: 44} -- clone
```

Summary

*

- When we see "..."
  - can be rest parameters or spread operator
  - spread syntax "expands" an array into its elements
  - rest syntax collects multiple elements and "condenses" them into a single element

- ... In an assignment context then “rest parameters”
  - end of function definition parameters,
  - end of destructure assignment
  - gathers the rest of the list of arguments into an array.

- ... occurs in an evaluation or expression context then is “spread operator”
  - function call
  - array literal
  - expands an array into a sequence of elements

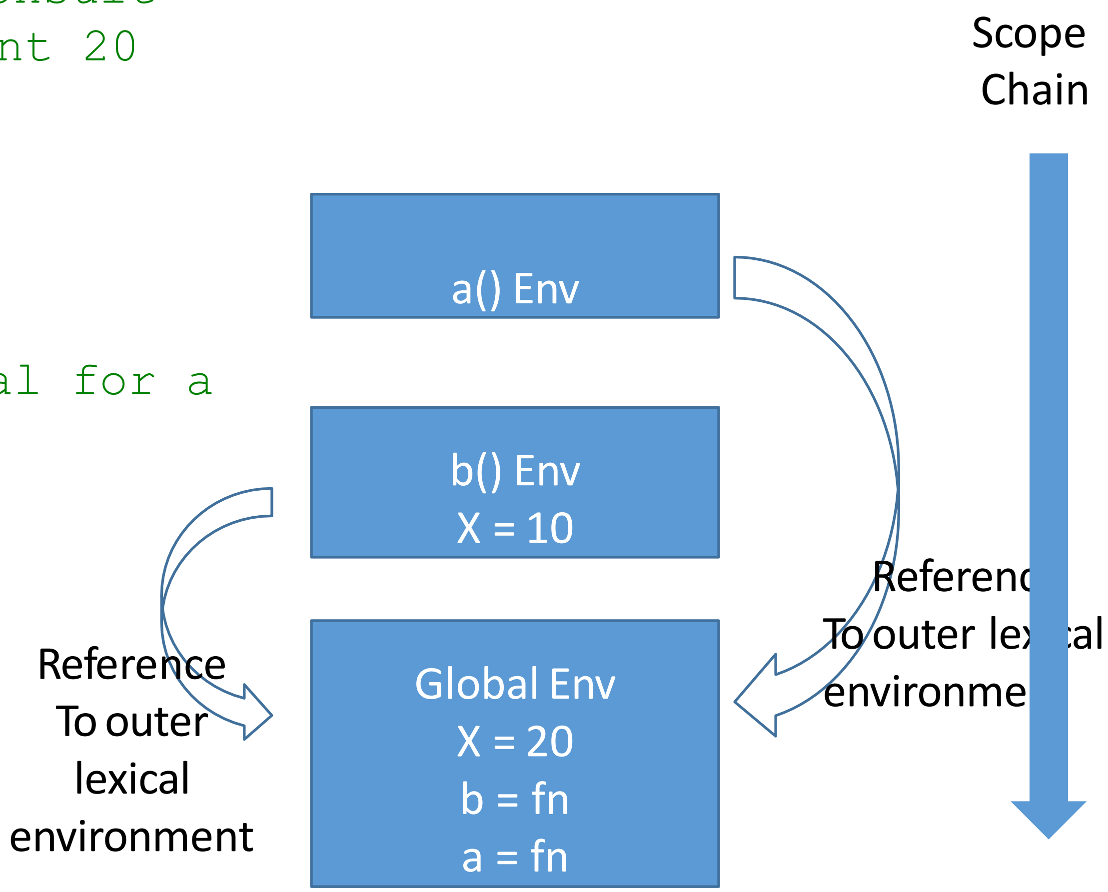
- Use patterns:
  - Rest parameters create functions that accept any number of arguments.
  - spread operator
    - spread array elements individually into another array – like concat
    - clone an array or object (shallow clone)
    - pass an array to functions that require multiple individual arguments

Closure

- JavaScript is function-oriented language.
  - A function can be created dynamically,
  - copied to another variable or
  - passed as an argument to another function and called from a totally different place later.
- a function can access variables outside of it, this feature is used quite often.
  - what happens when an outer variable changes?
    - Does a function get the most recent value or the one that existed when the function was created?
- what happens when a function travels to another place in the code and is called from there
  - does it get access to the outer variables of the new place?

Scope Example1

```
function a() {
 console.log(x); // consult
 Global for x and print 20
 from Global
}
function b() {
 const x = 10;
 a(); // consult Global for a
}
const x = 20;
b();
```

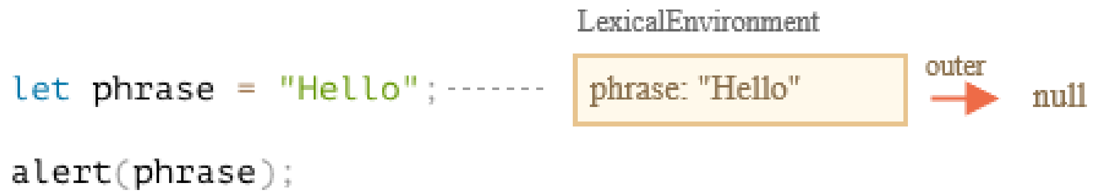


Lexical environment

- How variables work in the compiler

- every running function, code block {...}, and script have internal object
  - Lexical Environment., which has two parts
    - Environment Record –stores all local variables as its properties (and information like value of this).
    - reference to the outer lexical environment

- A “variable” is a property of Environment Record
  - To get or change a variable” means “to get or change a property of that object”.



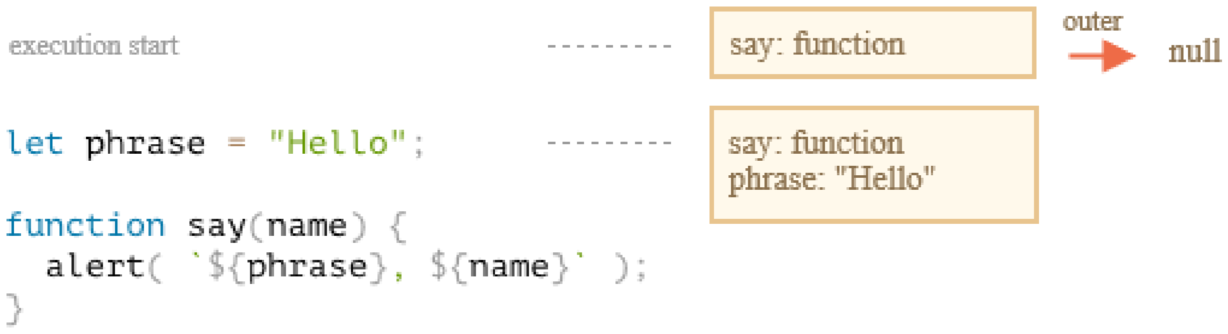
- rectangle shows Environment Record (variable store)
  - arrow means the outer reference.
  - global Lexical Environment has no outer reference, so it points to null

Global lexical environment

- To summarize:
  - A variable is a property of a special internal object,
    - associated with the currently executing block/function/script. (execution context stack)
  - Working with variables is working with the properties of that object

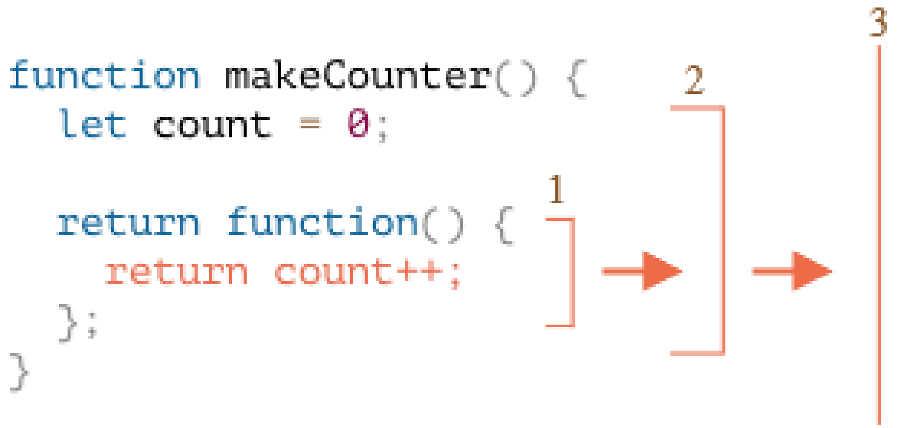
- Function Declaration
  - fully initialized when a Lexical Environment is created.
  - For top-level functions, it is the moment when the script is started.
  - why we can call a function declaration before it is defined.

  - Lexical Environment is non-empty from the beginning.
    - It has say, because that’s a Function Declaration.
    - later it gets phrase
    - “2 pass compiler”



Nested functions 2

- When inner function runs,
  - variable in count++ is searched from inside out.
    1. The locals of the nested function…
    2. The variables of the outer function…
    3. And so on until it reaches global variables.



- two questions:
  - Can we somehow reset the counter count from the code that doesn't belong to makeCounter? E.g. after alert calls
  - If we call makeCounter() multiple times
    - returns multiple counter functions.
    - independent or share the same count?

```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}
let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

What is a Closure?

- (def) A **closure** is the combination of
  - function bundled together (enclosed) with
  - references to its surrounding state (the **lexical environment**).

- a closure gives you access to an outer function’s scope from an inner function
  - closures are created every time a function is created

- Inner functions are created when the outer function is called
  - Whenever a function is called a new execution context is created and added to the call stack
  - Every execution context has a lexical environment associated with it that tracks the variable bindings and values

Closures

- general programming term “closure”, that developers should know

- Closure: function that remembers its outer variables and can access them
  - Not possible in all languages
  - in JavaScript, all functions are naturally closures
    - automatically remember where they were created
    - using hidden [[Environment]] property

- Common front end job interview question “what’s a closure?”,
  - a valid answer would be the definition
  - explanation that all functions in JavaScript are closures,
  - few more words about : the [[Environment]] property and how Lexical Environments work
  - Some define closures to be (only) when there is an inner function with free outer variable
    - “free” variables (not defined in the local function)
    - Implies an inner function

- To use a closure, define a function inside another function and expose it.
  - To expose a function, return it or pass it to another function.
  - inner function will have access to variables in the outer function scope,
    - even after outer function has returned.

Code blocks and scope

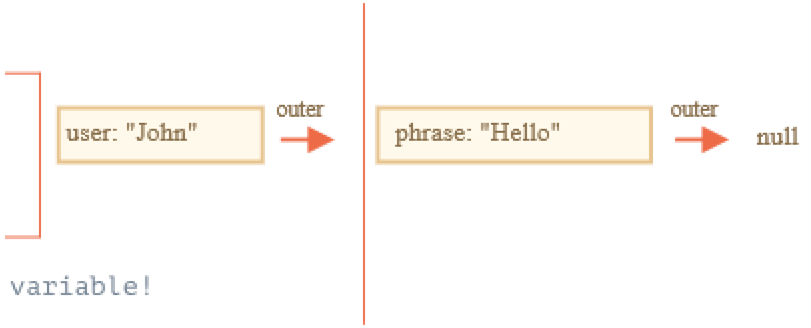
- a Lexical Environment exists for any code block {...}
  - created when a code block runs and contains block-local variables.

```
let phrase = "Hello";

if (true) {
  let user = "John";

  alert(`${phrase}, ${user}`);
}

alert(user); // Error, no such variable!
```



- When execution gets to the if block,
  - new “if-only” Lexical Environment is created for it
  - has the reference to the outer one, so phrase can be found.
  - all variables and Function Expressions declared inside if reside in that Lexical Environment
    - can’t be seen from the outside
    - after if finishes, the alert below won’t see the user, hence the error.

Global object

- window in browsers
- Window contains all the global functions
  - alert
  - prompt
  - setInterval
  - setTimer
  - console.log
  - Array
  - String
  - screenX, screenY, …
  - And hundreds of other global properties and methods
- Can view in the browser console
- Every global variable declaration or function declaration gets added to the global object
  - Bad practice to do this

*

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)
```

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'
  - sets the call context to be 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`

- in a method, this refers to the object that contains the method

- in a function, the containing object is ‘window’
  - Not in “use strict” mode→ undefined

‘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
```

.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.bind(me, '180cm', '70kg’)(); // Tina Xing 180cm 70kg
```

`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
```

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