ES 2015

Dr. Peiwei Mi

UNIT III – FUNCTIONAL PROGRAMMING

- A few ES6 ‘eye candy’:
  - Destructuring Operators, Spread Operators, Default, Rest
- Functional Programming
  - Definition
  - Arguments and Parameters
  - Types of function: Anonymous, Callback, Self-Executing Anonymous, Recursive, arrow function
  - Function that returns a function
  - Function that rewrites itself
  - Function Closure

DESTRUCTURING AND SPREAD

1-destructuring.js

DESTRUCTURING ASSIGNMENT

Split ("destructure") a composite object into individual variables

- Array destructuring
  - Assignment: let a = [1, 2, 3]
  - Normal access: let a0=a[0]; a1=a[1], a2=a[2]
  - Destructing: let [a0, a1, a2]=a;
- Object destructuring
- Parameter destructuring – a bit later

DESTRUCTING OBJECT COMPARISON

```
function register (props)
 { var { onChangeEmail, email, onChangePasswor, password, submit } = props;
  return (
            <div>
              <span>Email:</span>
              <input type='text' onChange={onChangeEmail} value={email} />
              <span>Password:</span>
              <input type='text' onChange={onChangePassword} value={password} />
              <button onClick={submit}>Submit</button>
            </div>
  ) }
```

```
function register (props)
 { return (
            <div>
              <span>Email:</span>
              <input type='text' onChange={props.onChangeEmail} value={props.email} />
              <span>Password:</span>
              <input type='text' onChange={props.onChangePassword} value={props.password} />
              <button onClick={props.submit}>Submit</button>
            </div>
  ) }
```

2-spread.js

SPREAD OPERATOR(...)

- Spreads out elements of any “iterable” object so they are treated as separate arguments to a function or elements in a literal array

```
let arr1 = [1, 2];
let arr2 = [3, 4];
arr1.push(...arr2);
console.log(arr1);                          // [1, 2, 3, 4]

let dateParts = [1961, 3, 16];
let birthday = new Date(...dateParts);
console.log(birthday.toDateString());       // Sun Apr 16, 1961

let arr1 = ['bar', 'baz'];
let arr2 = ['foo', ...arr1, 'qux'];
console.log(arr1); // ['foo', 'bar', 'baz', 'qux']
```

FUNCTIONAL PROGRAMMING

FUNCTIONAL PROGRAMMING

- Functions are first class citizens – objects/variables
  - Can do same things that objects do, and value is stored in variables
  - Can be used as code and data at the same time
- High Order Functions (HOF)
  - Functions that either has functions as parameters and/or return a function
- Programming style
  - Lots of small functions
  - Considered more Declarative (than Imperative)
- Concept introduced by Lisp
- ES6 added a lot of it into JavaScript

ARGUMENTS / PARAMETERS

- arguments is an Array-like object
  - To get values into a function, not possibly out
  - Considered local to the function
  - When a function is called, the actual values are sometimes called parameters

```
function avg(a, b) {
		return (a + b)/2;
}

avg(5, 10);				// 7.5
```

3-function-passvalue.js

PASSING BY VALUE / REFERENCE

General Concept

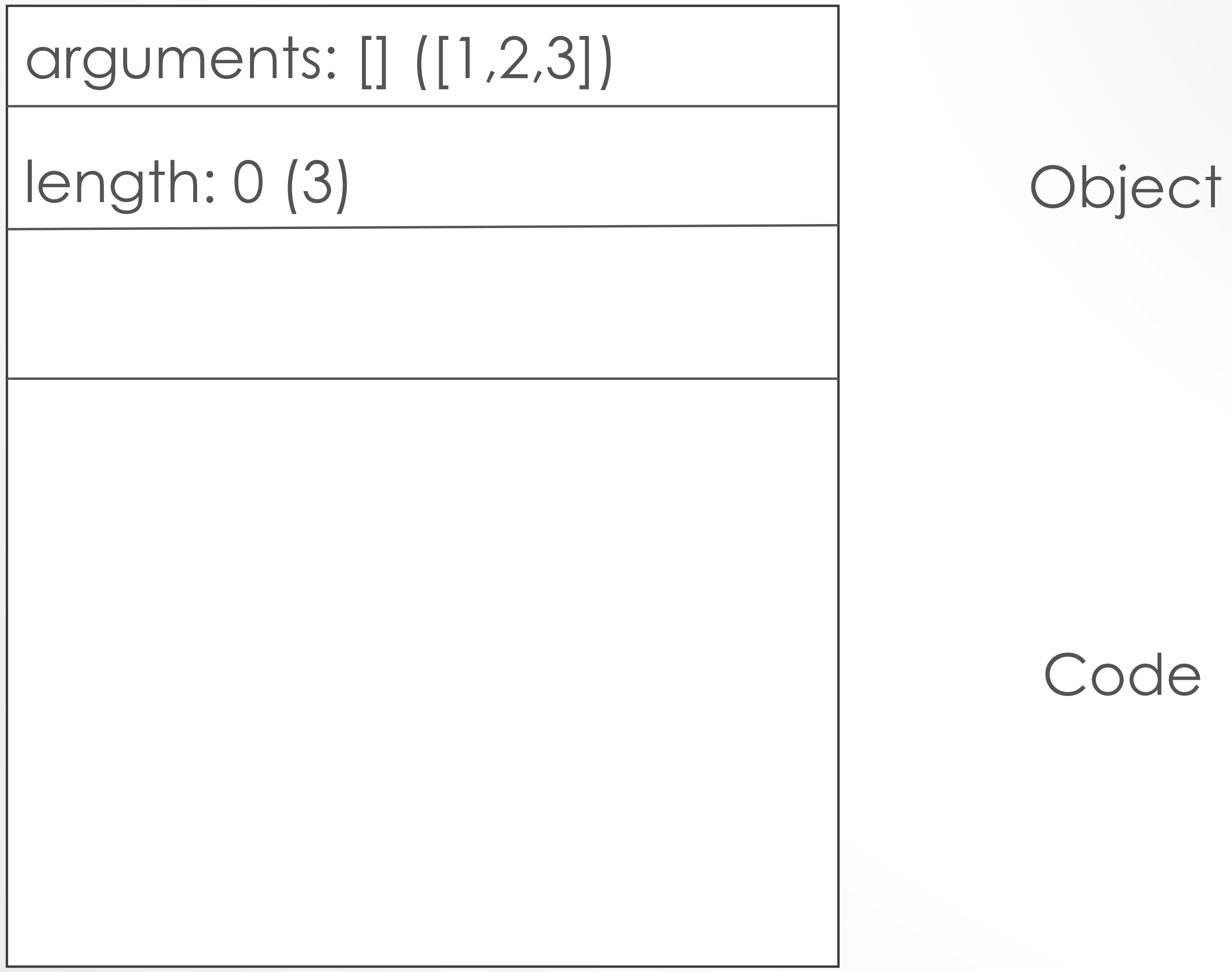
- Exactly how values of parameters are passed into arguments
  - It depends on data types
- By value for primary data types ➔ like deep copy
- By reference for advanced data types ➔ like shallow copy

4-function-param.js

MATCHING PARAMETERS

With any number of arguments, it creates a list using each argument as an item in the list

```
function sum() {
  var sum = 0;
  for (let i = 0; i < arguments.length; i ++)
        sum += parseInt(arguments[i]);
  return sum;
}

console.log(sum(1, 2, 4));
```


arguments: [] ([1,2,3])
length: 0 (3)
Object
Code

4-function-param-num.js

Also, what if actual parameters are more / less than arguments?

```
function f1(a,b,c) {
     …
}

f1(1, 2, 4)
f1(1, 2) // c=undefined
f1(1, 2, 3, 4)
```

MATCHING PARAMETERS – ANOTHER EXAMPLE

5-function-param.js

```
function myConcat(separator) {
		let vals = Array.prototype.slice.call(arguments, 1);
		return vals.join(separator);
}

console.log( myConcat(',', 'red', 'orange', 'blue') );

console.log( myConcat(',', 'sage', 'basil', 'oregano', 'pepper', 'parsley') );
```

Please try it on your own

MATCHING PARAMETERS (2)

- Destructing

  1-destructuring.js

- Default
  - Allows formal parameters to be initialized with default values if no value or undefined is passed.
- Rest
  - Allows representing an indefinite number of arguments as an array
- What's the difference between Rest and Spreads?

DEFAULT

- Default value expressions can refer to preceding parameters
- Explicitly passing **undefined** triggers use of default value

```
let today = new Date();

function makeDate(day, month=today.getMonth(), year=2021) {
		return new Date(year, month, day).toDateString();
}

console.log(makeDate(16, 3, 1961));		// Sun Apr 16 1961
console.log(makeDate(16, 3));			// Mon Apr 16 2021, run on 1/24/18
console.log(makeDate(16));			// Tue Jan 16 2021
```

REST

- Gathers variable number of arguments after named parameters into an array
- If no corresponding arguments are supplied, value is an empty array, not **undefined**
- Removes need to use **arguments** object

```
function report(firstName, lastName, ...colors) {
        let phrase = colors.length === 0 ? 'no colors' :
                colors.length === 1 ? 'the color ' + colors[0]:
                'the colors ' + colors.join(' and ');
        console.log(firstName, lastName, 'likes', phrase + '.');
}

report('John', 'Doe');                      // John Doe likes no colors.
report('Mark', 'Volkmann', 'yellow');       // Mark Volkmann likes the color yellow.
report('Tami', 'Volkmann', 'pink', 'blue'); // Tami Volkmann likes the colors pink and blue.
```

ANONYMOUS FUNCTION

- A function without a name / an identifier
  - When a function expression is used without assigning it to a variable.
- When executed, interpreter will create its memory block
  - But address is not assigned to any variable
- Can be used:
  - As a callback / function parameter - HOF
  - Self executing / IIFE
  - As function's return - HOF

```
function (a, b) {
        return a + b;
}
```

7-function-callback.js

CALLBACK (HOF)

A function variable is used as an input parameter

of a high order function

```
function invokeAdd(a, b) {
        return a() + b(); }

function one() { return 1; }

function two() { return 2; }


invokeAdd(one, two);


invokeAdd( function () {return 1; } , function () {return 2; } );
```

Functions become more flexible!

8-function-iife.js

IIFE

- IIFE: immediately-invoked function expression
- Or Self-Executing Anonymous Function

```
(function(){
        console.log('Hello World!');
})();
```

Commonly used for setup or to avoid global variables

IIFE

```
(function (name) {
	console.log('Hello ' + name + '!');
})('dude');
```

9-function-factorial.js

RECURSION

```
function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * factorial(num-1);
    }   }

var anotherFactorial = factorial;

anotherFactorial(4) // 24

factorial = null;

anotherFactorial(4); //error!

anotherFactorial(1); // ???
```

```
// solution 1
function factorial(num){
if (num <= 1){
     return 1;
} else {
     return num * arguments.callee(num-1);
} }

// solution 2
var factorial = (function f(num){
    if (num <= 1){
        return 1;
    } else {
        return num * f(num-1);    }
});
```

FUNCTION THAT RETURNS A FUNCTION (HOF)

10-function-return-function.js

```
function a() {
		console.log('A!');
		return function () {
				console.log('B!');
			};
}

var newFunc = a();
newFunc();
a()();

(function () {
		console.log('A!');
		return function () {
				console.log('B!');
			};
})()();
```

How about a()()()?

FUNCTION THAT REWRITES ITSELF

10-function-return-function.js

```
function a() {
        console.log('A!');
        a = function () {
                console.log('B!');
        };
}

a();
a();
a();
```

```
function a() {
        console.log('A!');
        var a = function () {
                console.log('B!');
        };
};

a();
a();
```

11-anonymoussummary.js

ANONYMOUS FUNCTION - SUMMARY

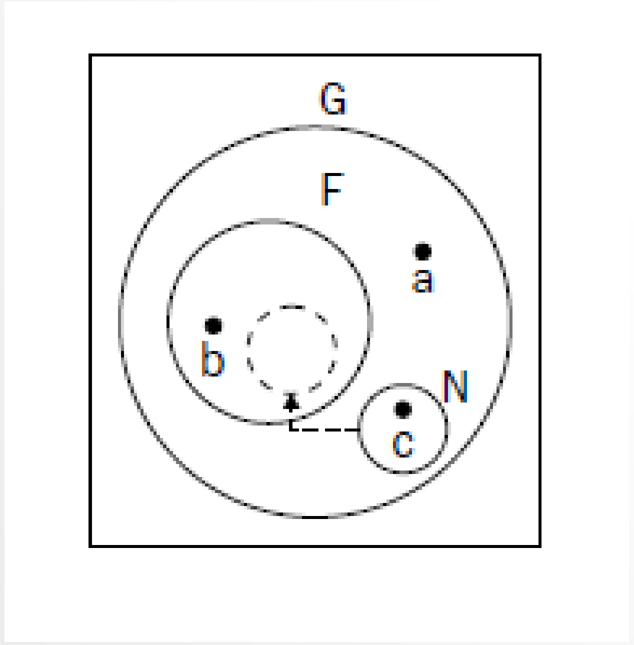
```
function createCF(propertyName) {
   return function(object1, object2){
                        var value1 = object1.propertyName;
                        var value2 = object2.propertyName;

                        if (value1 < value2){
                                    return -1;
                        } else if (value1 > value2){
                                    return 1;
                        } else {
                                    return 0;
                        }
            };
}
```

```
let o1 = {fName: “john”, lName: “lee”};
let o2 = {fName: “mary”, lName: “andersen”};
let o3 = {fName: “peter”, lName: “hong”};
```

This example is discussed in a video in the same google doc folder. Please watch it offline

FUNCTION CLOSURES

- Issue: how to protect local variable and yet give access out of its scope chain
- Breaking scope chain with a closure



GLOBAL AND LOCAL VARIABLES

```
var counter = 0;

function add() {
    counter += 1;
}

add();
add();
add();

State                .vs.
```

Keep state, no privacy

```
function add() {
    var counter = 0;
    counter += 1;
}

add();
add();
add();


Privacy
```

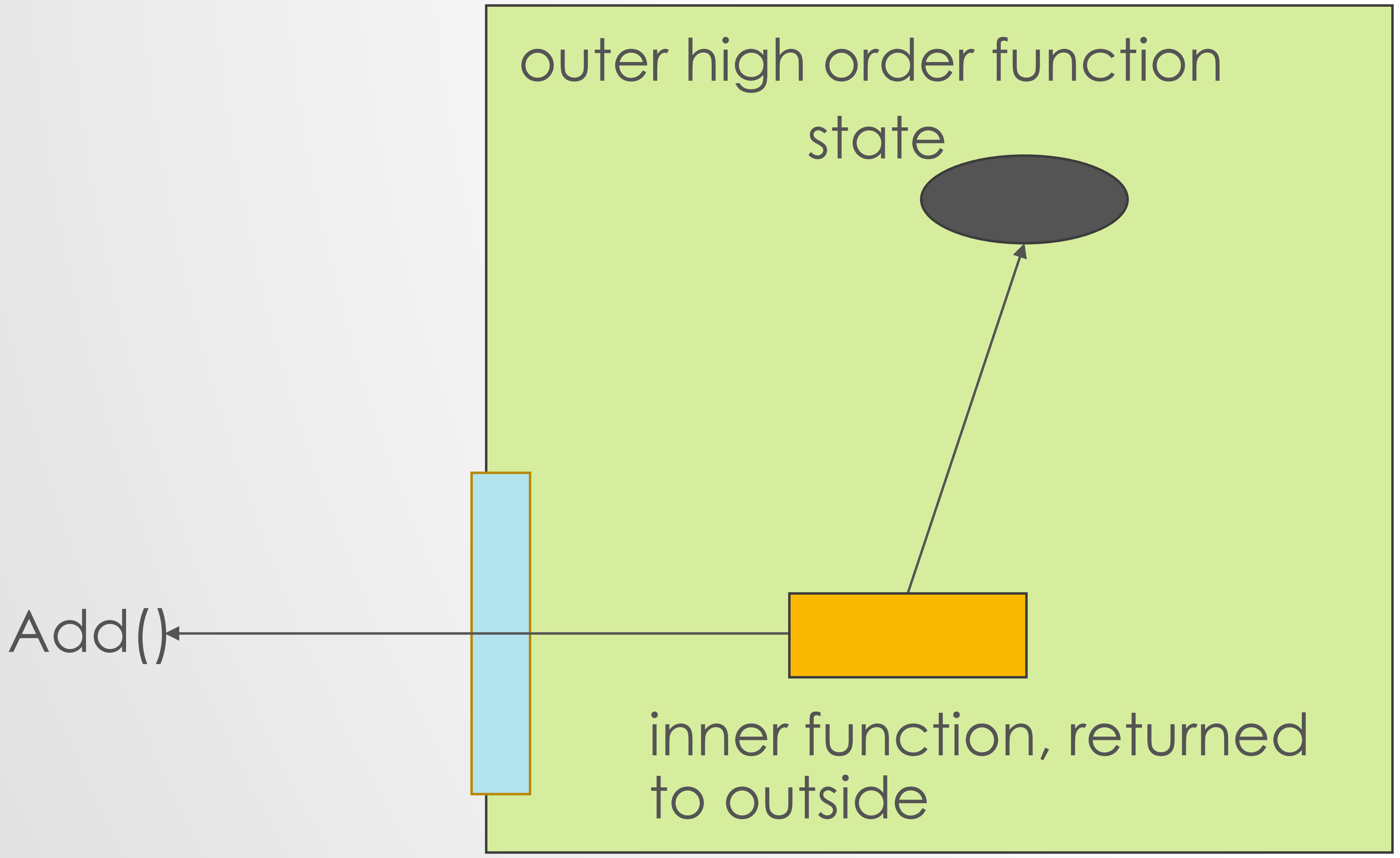
No state, with privacy

12-function-closure.js

FUNCTION CLOSURES

```
var firstf = function () {
    var counter = 0;
    return function () {
                  return counter += 1;}
}

var add = firstf();

add();
add();
add();
```

1. **Outer high order function,**
2. **State**
3. **Inner function, returned to outside**


outer high order function
state
Add()
inner function, returned to outside

FUNCTION CLOSURES

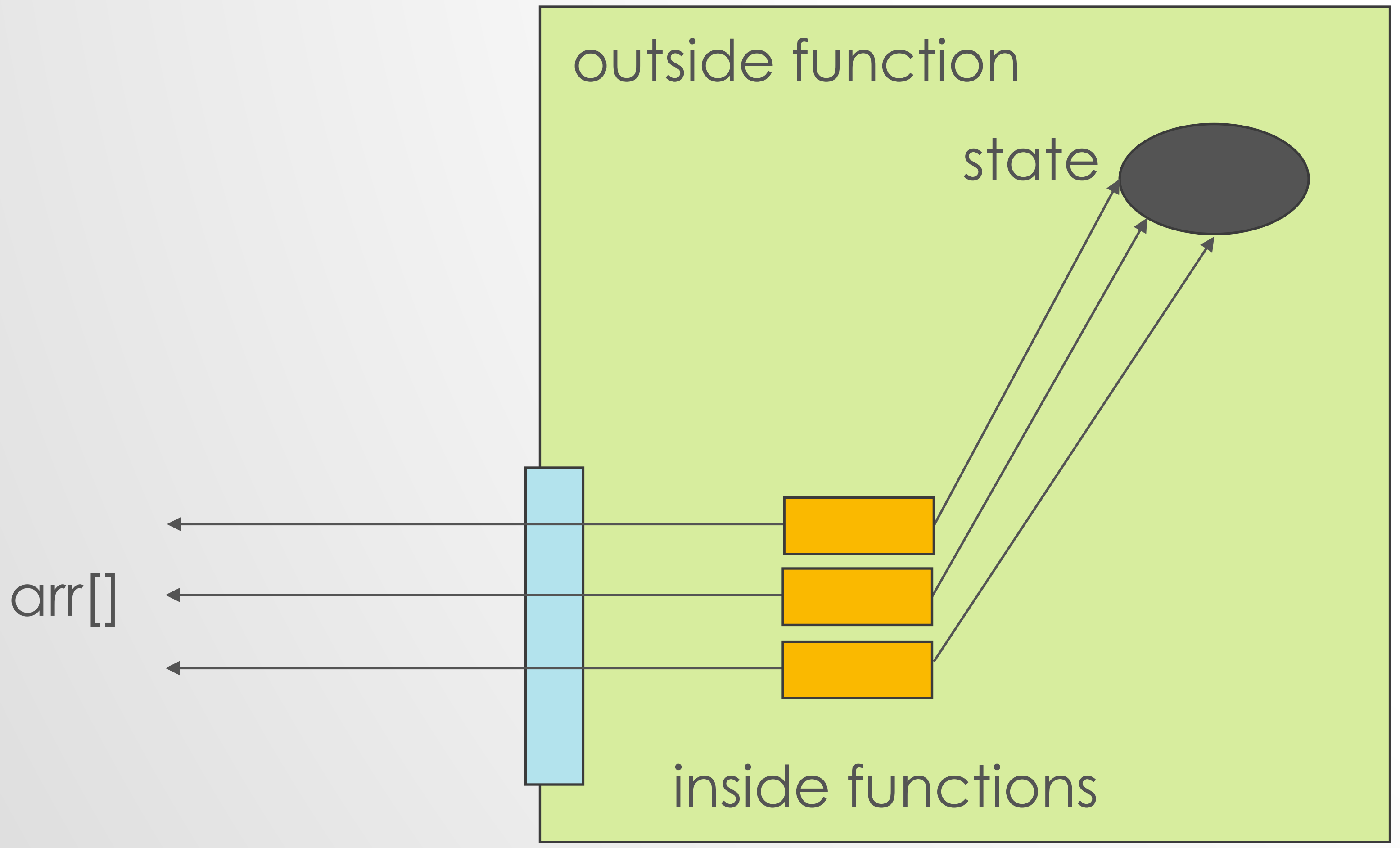
- Through the nesting of functions.
  - It grants the inner function full access to all the variables and functions defined inside the outer function.
  - The outer function does not have access to the variables and functions defined inside the inner function.
- A closure is created when the inner function is made available to any scope outside the outer function.
- Small pitfall:
  - If an enclosed function defines a variable with the same name as the name of a variable in the outer scope
    - There is no way to refer to the variable in the outer scope again!

13-function-closureloop.js

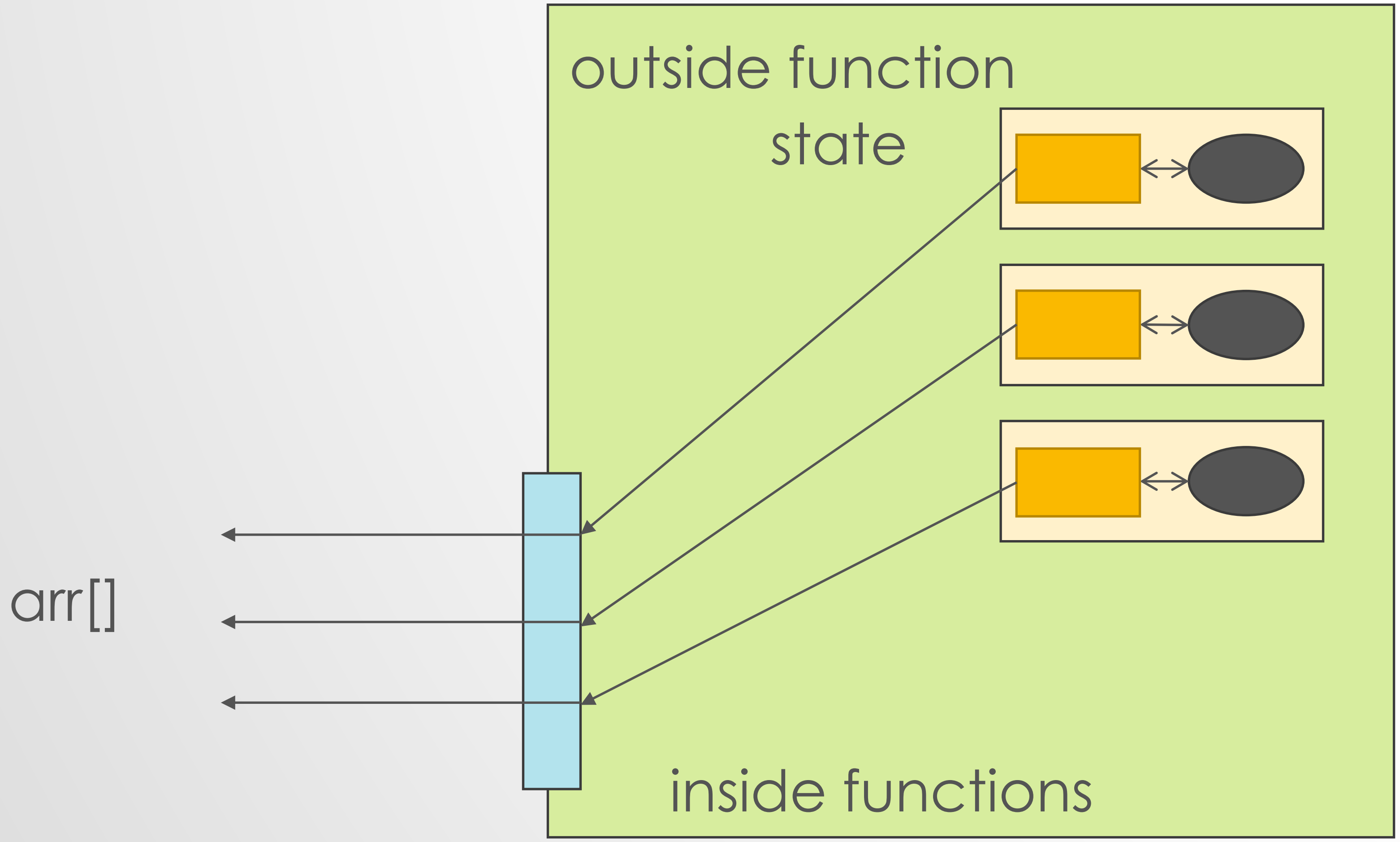
CLOSURE IN A LOOP

```
function F() {
		var arr = [], i;
		for (i = 0; i < 3; i++) {
				arr[i] = function () {
							return i;
						};
		}
		return arr;
}
var MyArr = F();
var i1 = MyArr[0]();
var i2 = MyArr[1]();
Var i3 = MyArr[2]();
```

FUNCTION F()



FUNCTION F1()



14-call-and-apply.js

CALL() AND APPLY()

- Both of them are used to invoke a function explicitly
- call() requires the parameters be listed individually
- apply() requires an argument array as the 2nd parameter

```
function f (name, job) {
    console.log("My name is " + name + " and I am a " + job + ".");
}

f("John", "fireman");
f.apply(undefined, ["Susan", "school teacher"]);
f.call(undefined, "Claude", "mathematician");
```

15-function-arrow.js

ARROW FUNCTIONS

- Introduced in ES2015, used very often nowadays
- Simplify syntax in three ways:
  - Omit the word function, and use arrow (=>) instead.
  - Omit the parentheses If the function takes a single argument
  - omit curly braces and the return statement if function is a single expression
- Always anonymous

```
let arr = [1, 2, 3, 4];
let doubled = arr.map(x => x * 2);
console.log(doubled);                    // [2, 4, 6, 8]
let product = (a, b) => a * b;
console.log(product(2, 3));              // 6
let average = numbers => {
          let sum = numbers.reduce((a, b) => a + b);
          return sum / numbers.length;
};
console.log(average(arr)); // 2.5
```

ARROW FUNCTIONS …

- **(*params*) => { *expressions* }**
  - if only one parameter and not using destructuring, can omit parems
  - if no parameters, need parems
  - cannot insert line feed between parameters and **=>**
  - if only one expression, can omit braces and its value is returned without using **return** keyword
- ***expression*** can be another arrow function that is returned
  - if expression is an object literal, wrap it in parens to distinguish it from a block of code
  - Arrow functions are typically used for anonymous functions like those passed to **map** and **reduce**.
- Functions like **product** and **average** are better defined the normal way so their names appear in stack traces.

HOMEWORK

- #1 – Destructuring
- #2 – Destructuring
- #3 – Function parameters
- #4 – Functions
- #5 – Array manipulation – Shuffle 52 cards
- #6 – Function and string manipulation – string to number
- #7 - Concepts you learned today and should memorize for interview