LAB # 8

JavaScript – The Good Parts

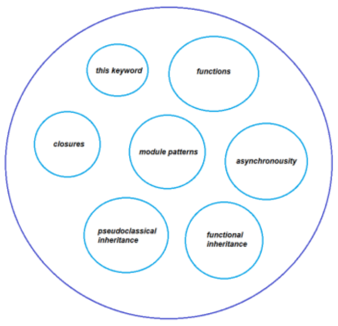
OBJECTIVE

To get familiar with different good parts available for programming which are typical of JS language.

THEORY

Javascript has some features and peculiar behavior which is typical of JS language itself. To understand different programming constructs and features available and to be more proficient and comfortable in writing and reading already written code by some professional authors found in other apps, you need to understand some interesting and good parts of Javascript language.

The good parts include *functions*, *this* keyword, *closure*, *module* pattern, *asynchronicity*, *functional inheritance.*



Pic 1 Javascript - The Good Parts

<u>Functions:</u>

In Javascript, there are a few ways a *function* can be defined. Most of the times, two syntax are used to define a function. One is called *function expression* syntax and the other one is called *function declaration* syntax. Also, unlike other languages, functions are first class objects; meaning that they have full capability of what an object has. Properties can be defined on a function object, they can be passed around, they can be updated with values etc., just like normal object. A better way to remember function in JS is to think of them being a regular object with the ability to be *invoked*.

```
var aFunction = function() { ... };
```

So, aFunction is a function object declared using function expression syntax. On the other hand, we have declarative syntax as well;

```
function bFunction() { ... }
```

So, bFunction in above example is a function declared using declarative syntax. So, if a function is defined on the right side of an equal operator (assignment operator), it is a function expression syntax.

Both can have properties like normal objects;

```
aFunction.prop1 = 1;
bFunction.prop1 = 2;
```

They can be passed around to another function;

```
function parentFunction(func){
	func();
}

parentFunction(aFunction);
parentFunction(bFunction);
```

And you can invoke (call/execute) them just like normal functions;

```
aFunction();
bFunction();
```

And more interestingly you can assign them new properties which can be functions themselves;

```
aFunction.newFunction = function() {...};
aFunction.newFunction();
```

this keyword:

this is an object or a keyword which points to an object. Any regular object can have a method defined using function expression like this;

```
var student = {
	markAttendance: function() { ... }
};
```

Now, if *markAttendance* method is called using *student* object, *this* keyword would point to *student* object and it will be available within the scope of the *markAttendance* method. For example;

```
var student = {
        noOfDays: 0,
        markAttendance: function() {
                student.noOfDays += 1;
        }
};
student.markAttendance();
```

It is easy to understand that when *markAttendance* method is called, it will increment the *noOfDays* property by 1 using its own defining parent object. But there is no need to explicitly use the *student* object from within the *markAttendance* method which is itself defined in the same object. There is a shortcut to refer to the parent object when the member property of the same object is called (in this case, *noOfDays* property) from within its another member of the same object (in this case it is the *markAttendance* method), use *this* instead.

Instead of using this code;

```
var student = {
        noOfDays: 0,
        markAttendance: function() {
                ~~student~~.noOfDays += 1;
        }
};
```

Use this code;

```
var student = {
        noOfDays: 0,
        markAttendance: function() {
                this.noOfDays += 1;
        }
};
```

Here *this* = *student* as it is being used from within the same object (again, *student* object) in the method *markAttendance*. *this* keyword is not merely a shortcut, there are other benefits associated with using *this* keyword. When you use *this* instead of the reference to the actual defining object, you are writing flexible code. In JS, there are techniques that the *markAttendance* method can be called from another object as well.

```
var teacher = {
        noOfDays: 0
};
```

There is no method called *markAttendance* defined on the *teacher* object. But still there is a technique we can use to ‘borrow’ the behavior and use that with other objects. Whichever object invokes the *markAttendance* method, *this* keyword would point to that object.

Normally, we call methods defined within an object using the reference of only that object itself. But sometimes, the behavior (method) is so general that we take that out of that object and make it available to be consumed by other objects. Let’s see how that work;

```
function markAttendance() {
        this.noOfDays += 1;
}

markAttendance.apply(teacher);
```

As the method is not being called using following syntax <object>.<method_name>(), but without any object reference, JS provides us two common methods to set the caller (object) of the method (*markAttendance* in this case), *call* and *apply*. We are using *apply* in the above example, which is available to all the function objects.

Now, *teacher* = *this* in the method *markAttendance* passed via built-in *apply* method, it will increment the *noOfDays* property on the *teacher* object.

Also,

```
markAttendance.apply(student);
```

Here, *student* = *this* in the method *markAttendance* passed via built-in *apply* method, it will increment the *noOfDays* property on the *student* object. So, using *this* keyword gives us great flexibility in reusing the behavior.

What happens when built-in *apply* method is not used available on the *markAttendance* function object and still the *markAttendance* function is called? What object will *this* keyword point to? Or will it not point to any object at all?

```
markAttendance(); //What object will this keyword point to?
```

this keyword will still point to some object and in the case above, it will point to default *window* object. So, if there is a global variable called noOfDays declared, it will be incremented, otherwise it will raise an error when accessed (NaN error – Not a Number error);

```
var noOfDays = 0;
markAttendance();
console.log(noOfDays)  // prints 1
```

Closure:

There is an important technique in JS which is called *closure*. By default, there is no concept of private property or behavior of an object in JS but using *closures* we can create one.

```
var student = {
        nicNo: ‘11111-1111111-1’,
        getNicNo: function() { return this.nicNo; }
};
```

It is easy to overwrite the property as given below;

```
student.nicNo = ‘22222-2222222-2’;
```

Using *closures*, we can protect properties of the object from being accessed. Here is the function closure as given below;

```
var student = (function() {
        var _nicNo = ‘33333-3333333-3’;
        var _getNicNo = function() {
                return _nicNo;
        };
        return {
                getNicNo: _getNicNo
        };
}());

student.getNicNo();
```

The closure effect will remain there if there is function *_getNicNo* accessing the private variable *_nicNo* of its *parent enclosing* function. You need to remember, that scope of the variables in JS is function based, so in other words, *_nicNo* variable will still be in the scope and hence will be available to the *_getNicNo* method as long as the *student.getNicNo* function is there as it is using the member of its enclosing function - *_nicNo*. The technique used above to create the closure is called IIFE (Immediately Invoked Function Expression).

Here is the step-by-step explanation on how IIFE works to create the closure:

1. **(** function(){…} () **).** First you need to put the parenthesis around the function to create an *expression*.
2. (**function(){…}** ()). Then you need to define the *function* within the parentheses.
3. (function(){ ***private members;*** } ()). Then you need to define the *private* members of the function.
4. (function(){ *private members*; ***return public members;*** } ()). Then you need to define the *public* members (API) of the function to manipulate the private members and return it.

5. (function(){…} ()). Then you need to *immediately invoke* it, again with parentheses.

One interesting use case of *closure* is to create a *counter* as given below;

```
var Counter = (function() {
        var currentValue = 0;
        var increment = function() { currentValue++;};
        var decrement = function() { currentValue--;};
        var getCurrentValue = function() { return currentValue; };

        return {
                increment,
                decrement,
                getCurrentValue
        };
}());

Counter.increment(); //increments 1 to the currentValue
Counter.increment(); //increments 1 more to the currentValue
Counter.getCurrentValue(); //returns 2
Counter.decrement(); //decrements 1 from the currentValue
Counter.getCurrentValue(); //returns 1
```

Modules:

In old JS code and many JS authors (mostly libraries authors outside the NodeJS world) still use *closure* to create isolated/private parts of the code which is called a *module*. Since, it is created using closure, it not only provides encapsulation but also helps avoiding the name conflict which occur using the same variable names used in various parts of the code within the global scope again and again. Also, code is better structured, organized, and modular using modules. Modern module patterns have borrowed a lot from this concept.

Here is an example of a *module* creation using closure as given below;

```
var SSUET = (function(){
        var IT = (function(){
                        var name = ‘it;
                        var teachProgramming = function(){ return `${name} is teaching`; };

                        return {
                                teachProgramming
                        };
                }());

        var ADMIN = (function(){
                                var name = ‘admin’;
                                var conductExams = function() { return `${name} is conducting exams`; };

                                return {
                                        conductExams
                                };
                        }());

        var FACULTY = (function(){
                                var name = ‘faculty’;
                                var prepareSyllabus = function() { return `${name} is preparing syllabus`; };

                                return {
                                        prepareSyllabus
                                };
                        }());

        var BOARD = (function(){
                                var name = ‘board’;
                                var decideStrategy = function() { return `${name} is deciding strategy`; };

                                return {
                                        decideStrategy
                                };
                        }());

        return {
                IT,
                ADMIN,
                FACULTY,
                BOARD
        };
}());

console.log(SSUET.IT.teachProgramming());
console.log(SSUET.ADMIN.conductExams());
console.log(SSUET.FACULTY.prepareSyllabus());
console.log(SSUET.BOARD.decideStrategy());
```

Creating module gives us cleaner, readable, and manageable code. Also, we achieve encapsulation and better organization of the code. There are various design patterns around module creation

Asynchronicity:

Asynchronous code means that the code is not executed in sequential order. In synchronous code, every statement is fully executed before it can move on to other statement next to it. For example, following is the synchronous code;

```
1.  function master() {
2.      var a = 1;
3.      var b = 2;
4.      var s = getStudent ();
5.      var c = 3;
6.  }

1.  function getStudent() {
2.     return {
3.        id: 1
4.     };
5.  }

master();
```

when master function is called, every statement in the method from line 2 to 6 will be executed sequentially. When the control will reach to line 4 to call *getStudent* method, it will not execute statement at line 5 until it has executed all the statements found in the method *getStudent*. In other words, it will wait for the *getStudent* method to finish and then execute the next line – line 5. This is an example of *synchronous* code.

Here is an example of asynchronous code;

```
1.  async function master() {
2.      var a = 1;
3.      var b = 2;
4.      var s = await getStudent ();
5.      var c = 3;
6.  }

1.  async function getStudent() {
2.     return {
3.        id: 1
4.     };
5.  }

master();
```

When master function is called, every statement in the method from line 2 will be executed till line 5. The code will not wait for the *getStudent* to finish at line 4, instead, it will immediately execute the next statement at line 5. This is an example of *asynchronous* code.

You may be wondering how the output of that *getStudent* method will be assigned to variable *s* if that method is not waited and next line is immediately executed. There are few patterns of writing asynchronous code, one of which is used in the code above, which provides the mechanism of how to receive the output of any asynchronous code. There are three famous asynchronous code structures.

1. Using *callback* functions
2. Using *promise* pattern
3. Using *async/await* pair of keywords

Using async/await pair of keywords is the most elegant and easiest to understand, as it is closest to the synchronous coding style. All the styles are different ways to do the asynchronous programming in JS.

async/await:

```
async getExamsResult(){
        await ExamController.getExamsResult();
}

let examsResult = await getExamsResult();
if( examsResult >= 70) {
        console.log(‘passed’);
} else {
        console.log(‘failed’);
}
```

promise:

```
let examsResult = Math.random() * 100;
let examsResultPromise = new Promise((resolve, reject) => {
        if(examsResult >= 70) {
                resolve(‘passed’);
        } else {
                reject(‘failed’);
        }
});

examsResultPromise.then((result) => {
        console.log(result);
}).catch((error) => console.log(error));
```

callback:

```
let examsResultToken = setTimeout(function(){
        console.log('exams result announced');
}, 100000);
```

Functional Inheritance:

Unlike other languages, Javascript supports functional inheritance. There is no formal concept of defining a class in classic JS but the equivalent of creating a class is to define a function and then create an object using the *new* keyword and *function name* as follows;

```
function Student(firstname, lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.getFullName = function() {
                return `Full Name: ${this.firstname},${this.lastname}`;
        };
}

let student = new Student('abc', 'xyz');

console.log(student.firstname);
console.log(student.lastname);
console.log(student.getFullName());
```

The '*class*' in JS is created using the same old friend – the function. In a sense, it is a very powerful thing that no other programming construct needs to be learnt to create a class and subsequently an object of that class. So, the definition of '*class*' in JS starts with '*function*' keyword and then any suitable name for the class name follows after that.

The '*constructor*' of the above '*class*' is actually the *parameter names* (firstname, lastname) and the *members* of the '*class*' is declared by prefixing '*this'* keyword to the property names. Not only the members of the class are created but at the same time, the parameters by which arguments are received are assigned to the newly created members.

When you need to create an object of a class (*Student*), all you need is to type '*new*' keyword and then name of the *class* and passing along any arguments needed by the *constructor* of that *class*. Here is a step-by-step explanation of what happens when a new object is created.

1. First a dummy/empty object is created and made available using *this* keyword: *this* = {} to the scope of the class (function in JS)
2. Whatever is passed as arguments to the function is received via parameters and all of them are assigned to *this* object.
3. Even though there is no explicit *return* statement, *this* object is returned.

4. All the steps defined above will only happen when *new* keyword is used before the class name, otherwise this will happen:
    a. *this* keyword will be set to the *window* object.
    b. All the properties will be assigned to *window* object.
    c. No object will be returned except undefined

```
let student = new Student('abc', 'xyz'); //An object will be created as new keyword is used
console.log(student);

let student = Student('abc', 'xyz'); //window object will get new properties and undefined will be returned as there is 'new' keyword' used and it has just become a regular function instead of a class
console.log(student);
```

Inheritance in JS works as follows;

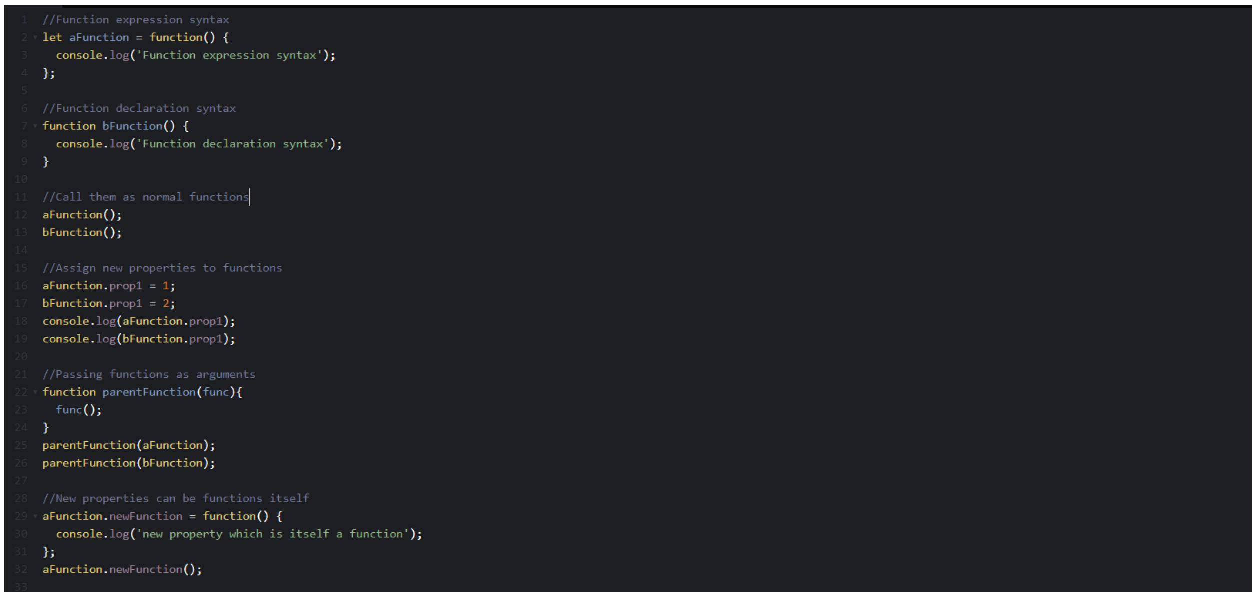
```
function StaffMember() {
	this.getFullName = function() { return `Full Name: ${this.firstname}:${this.lastname}`;};
}

function Student(firstname, lastname){
	this.firstname = firstname;
	this.lastname = lastname;
}

Student.prototype = new StaffMember();

let student = new Student('abc', 'xyz');

console.log(student.firstname);
console.log(student.lastname);
console.log(student.getFullName());
```

StaffMember is defined as *base/parent* class and *Student* is defined as *child* class. Every function has a *prototype* property. Any object you assign to that property will inherit all the members of that object (in this case base *StaffMember* class object created using *new* keyword) into the inherited class (in this case child *Student* class). *StaffMember* class has a function property called *getFullName* which is inherited in *Student* class because of those statements.

Samples:

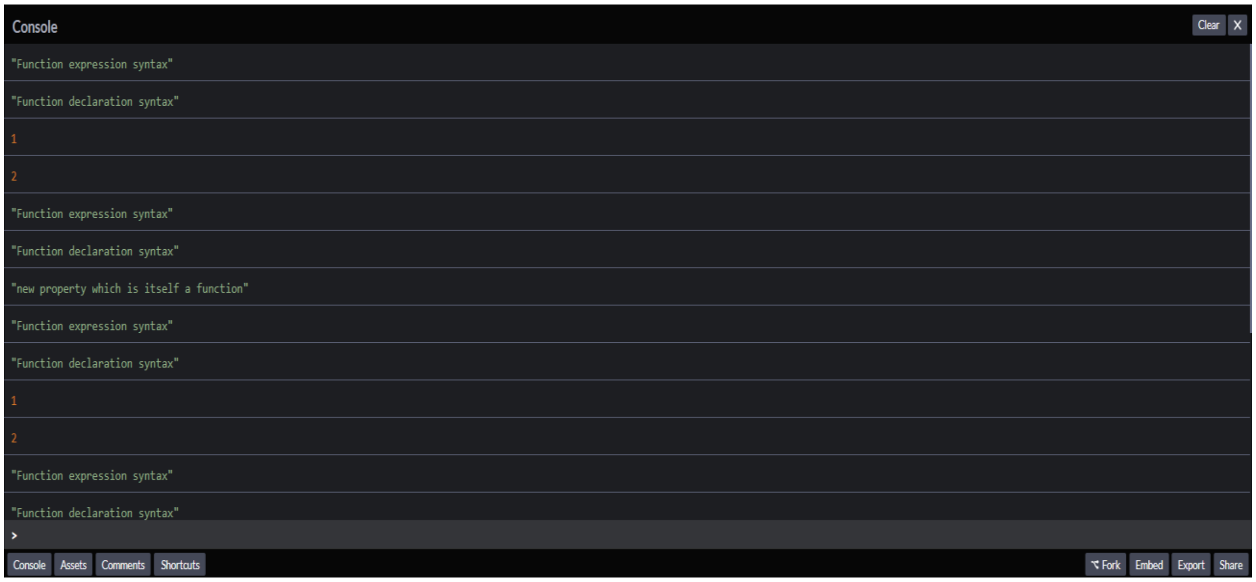
1 Functions: https://codepen.io/syed-owais-owais/pen/xxdMxLr

Code:



```
//Function expression syntax
let aFunction = function() {
  console.log('Function expression syntax');
};

//Function declaration syntax
function bFunction() {
  console.log('Function declaration syntax');
}

//Call them as normal functions
aFunction();
bFunction();

//Assign new properties to functions
aFunction.prop1 = 1;
bFunction.prop1 = 2;
console.log(aFunction.prop1);
console.log(bFunction.prop1);

//Passing functions as arguments
function parentFunction(func){
  func();
}
parentFunction(aFunction);
parentFunction(bFunction);

//New properties can be functions itself
aFunction.newFunction = function() {
  console.log('new property which is itself a function');
};
aFunction.newFunction();
```

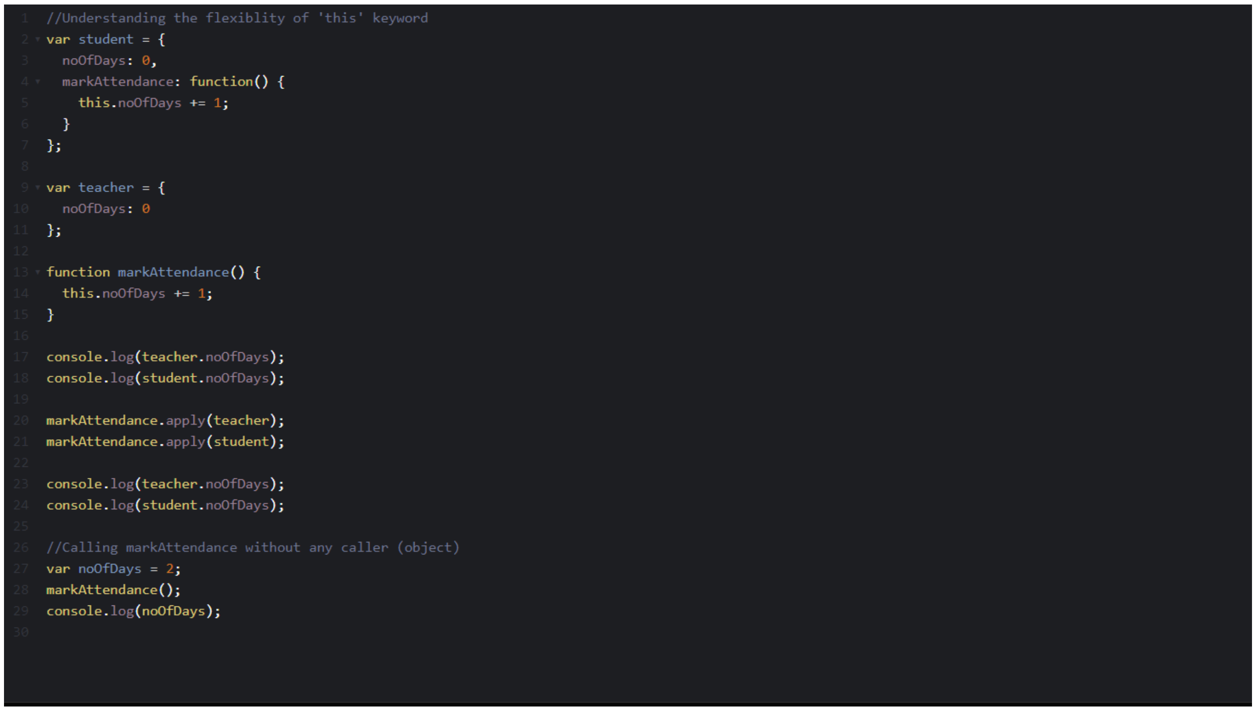
Output:



```
"Function expression syntax"
"Function declaration syntax"
1
2
"Function expression syntax"
"Function declaration syntax"
"new property which is itself a function"
"Function expression syntax"
"Function declaration syntax"
1
2
"Function expression syntax"
"Function declaration syntax"
>
```

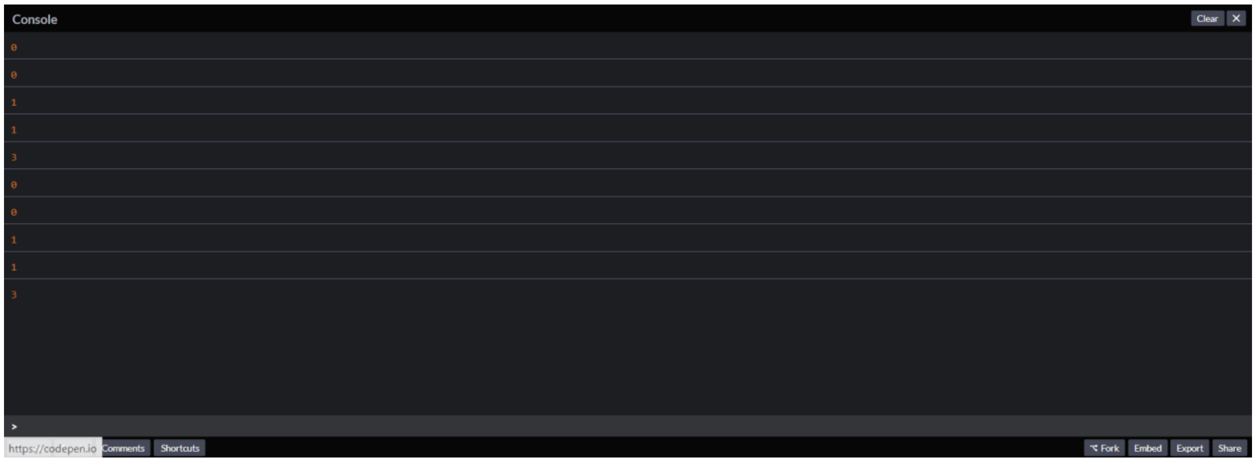
2 *this* Keyword: https://codepen.io/syed-owais-owais/pen/MWmLWGy

Code:



```
//Understanding the flexiblity of 'this' keyword
var student = {
  noOfDays: 0,
  markAttendance: function() {
    this.noOfDays += 1;
  }
};

var teacher = {
  noOfDays: 0
};

function markAttendance() {
  this.noOfDays += 1;
}

console.log(teacher.noOfDays);
console.log(student.noOfDays);

markAttendance.apply(teacher);
markAttendance.apply(student);

console.log(teacher.noOfDays);
console.log(student.noOfDays);

//Calling markAttendance without any caller (object)
var noOfDays = 2;
markAttendance();
console.log(noOfDays);
```

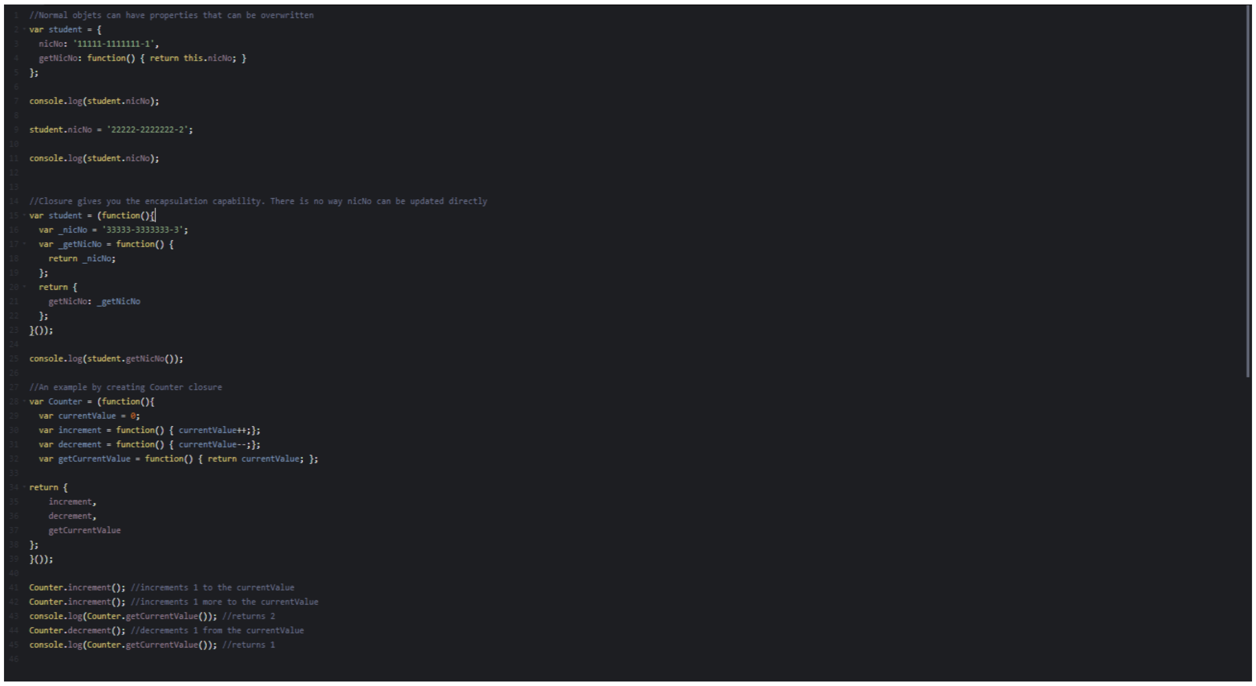
Output:



```
0
0
1
1
3
0
0
1
1
3
>
```

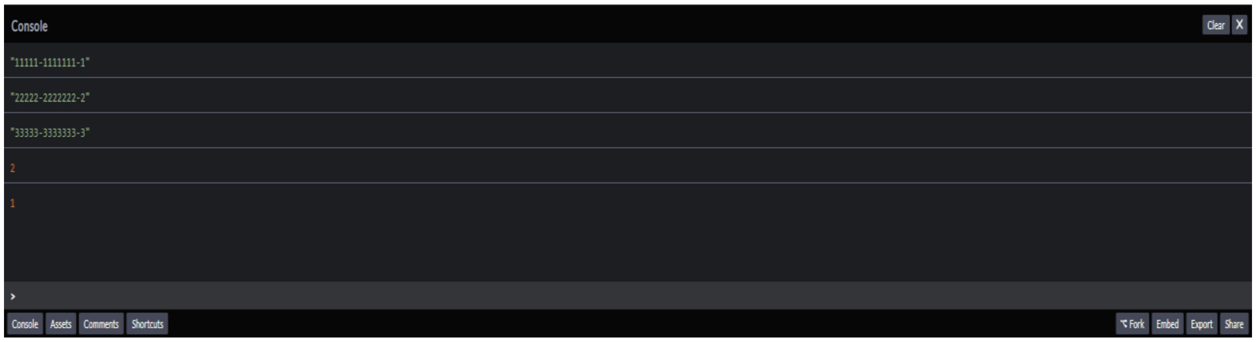
3 Closure: https://codepen.io/syed-owais-owais/pen/LYyqYJd

Code:



```
//Normal objets can have properties that can be overwritten
var student = {
  nicNo: '11111-1111111-1',
  getNicNo: function() { return this.nicNo; }
};

console.log(student.nicNo);

student.nicNo = '22222-2222222-2';

console.log(student.nicNo);


//Closure gives you the encapsulation capability. There is no way nicNo can be updated directly
var student = (function(){
  var _nicNo = '33333-3333333-3';
  var _getNicNo = function() {
    return _nicNo;
  };
  return {
    getNicNo: _getNicNo
  };
}());

console.log(student.getNicNo());

//An example by creating Counter closure
var Counter = (function(){
  var currentValue = 0;
  var increment = function() { currentValue++;};
  var decrement = function() { currentValue--;};
  var getCurrentValue = function() { return currentValue; };

return {
    increment,
    decrement,
    getCurrentValue
};
}());

Counter.increment(); //increments 1 to the currentValue
Counter.increment(); //increments 1 more to the currentValue
console.log(Counter.getCurrentValue()); //returns 2
Counter.decrement(); //decrements 1 from the currentValue
console.log(Counter.getCurrentValue()); //returns 1
```

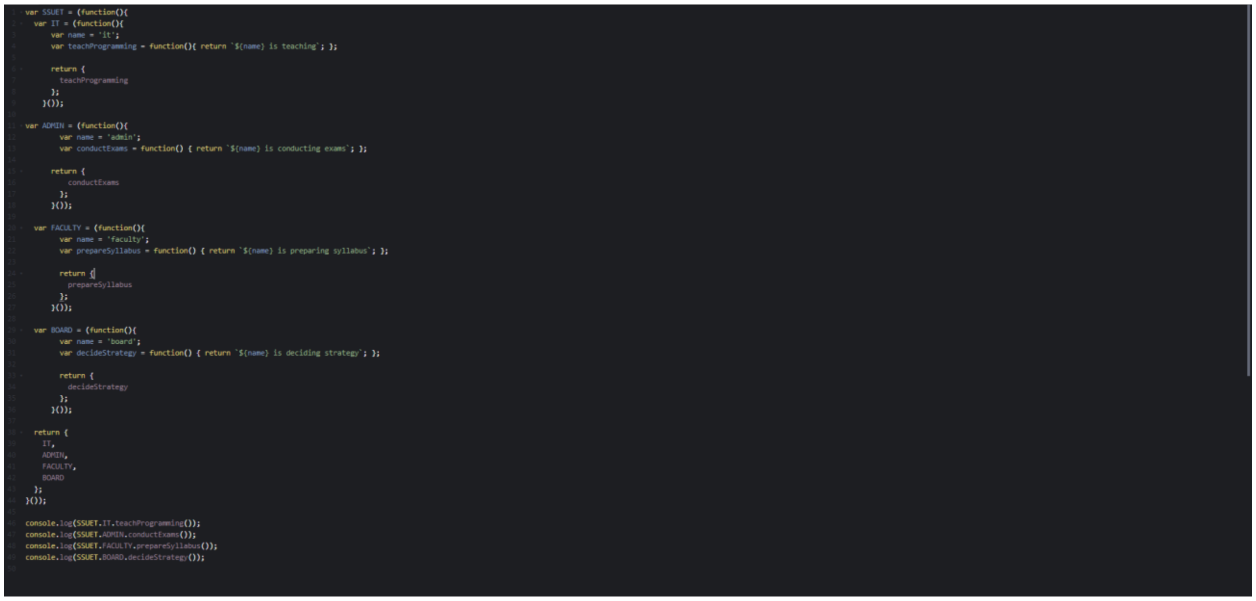
Output:



```
"11111-1111111-1"
"22222-2222222-2"
"33333-3333333-3"
2
1
```

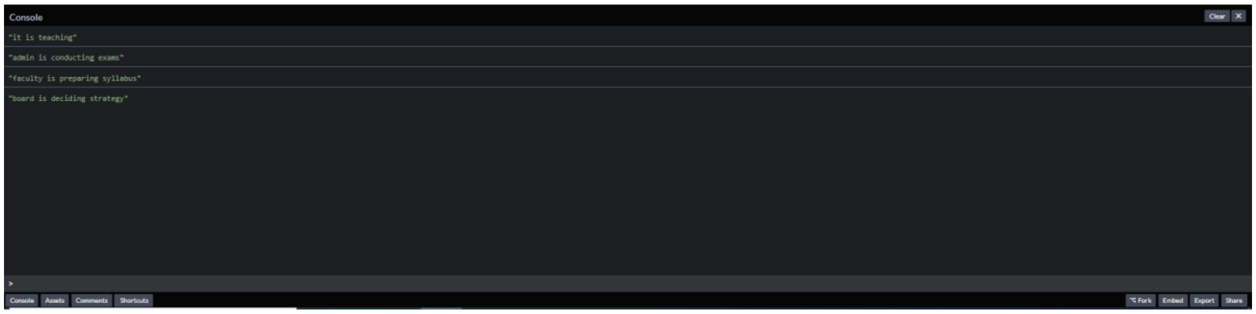
4 Module: https://codepen.io/syed-owais-owais/pen/MWmLWLb

Code:



```
var SSUET = (function(){
  var IT = (function(){
      var name = 'it';
      var teachProgramming = function(){ return `${name} is teaching`; };

      return {
        teachProgramming
      };
    }());

var ADMIN = (function(){
        var name = 'admin';
        var conductExams = function() { return `${name} is conducting exams`; };

      return {
          conductExams
        };
      }());

  var FACULTY = (function(){
        var name = 'faculty';
        var prepareSyllabus = function() { return `${name} is preparing syllabus`; };

        return {
          prepareSyllabus
        };
      }());

  var BOARD = (function(){
        var name = 'board';
        var decideStrategy = function() { return `${name} is deciding strategy`; };

        return {
          decideStrategy
        };
      }());

  return {
    IT,
    ADMIN,
    FACULTY,
    BOARD
  };
}());

console.log(SSUET.IT.teachProgramming());
console.log(SSUET.ADMIN.conductExams());
console.log(SSUET.FACULTY.prepareSyllabus());
console.log(SSUET.BOARD.decideStrategy());
```

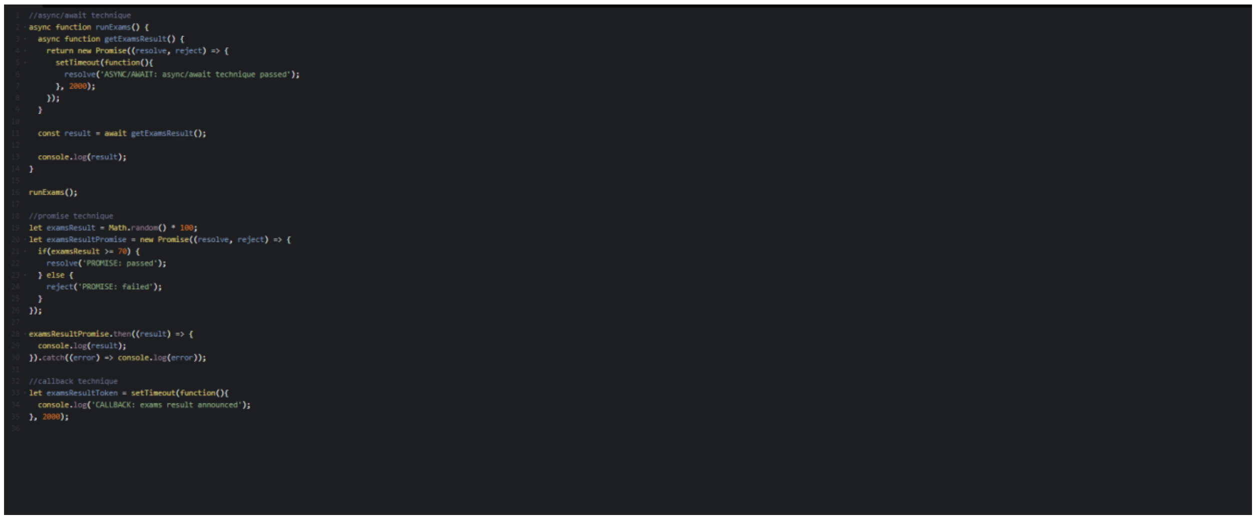
Output:



```
Console
"it is teaching"
"admin is conducting exams"
"faculty is preparing syllabus"
"board is deciding strategy"
```

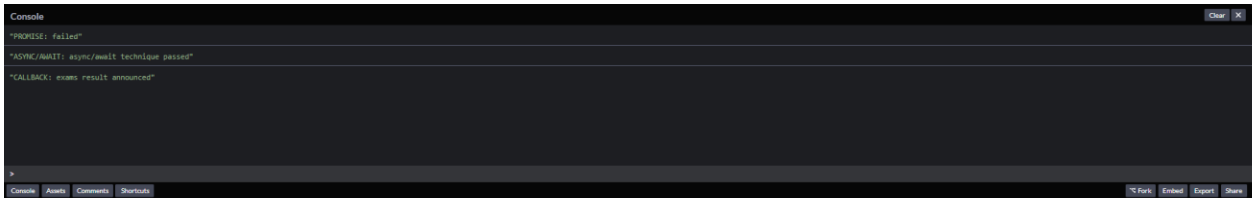
5 Asynchronicity: https://codepen.io/syed-owais-owais/pen/eYWxYoW

Code:



```
//async/await technique
async function runExams() {
  async function getExamsResult() {
    return new Promise((resolve, reject) => {
      setTimeout(function(){
        resolve('ASYNC/AWAIT: async/await technique passed');
      }, 2000);
    });
  }

  const result = await getExamsResult();

  console.log(result);
}

runExams();

//promise technique
let examsResult = Math.random() * 100;
let examsResultPromise = new Promise((resolve, reject) => {
  if(examsResult >= 70) {
    resolve('PROMISE: passed');
  } else {
    reject('PROMISE: failed');
  }
});

examsResultPromise.then((result) => {
  console.log(result);
}).catch((error) => console.log(error));

//callback technique
let examsResultToken = setTimeout(function(){
  console.log('CALLBACK: exams result announced');
}, 2000);
```

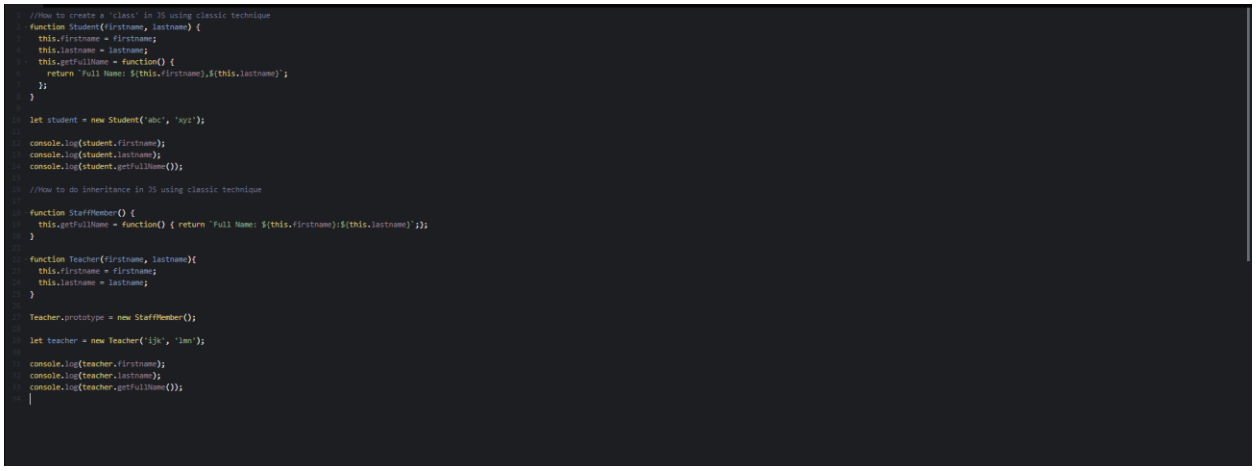
Output:



```
Console
"PROMISE: failed"
"ASYNC/AWAIT: async/await technique passed"
"CALLBACK: exams result announced"
>
```

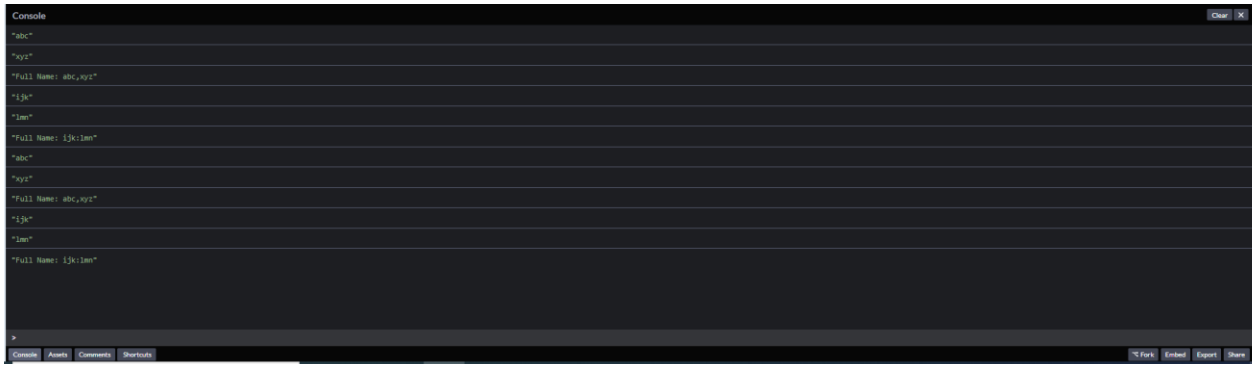
6 Functional Inheritance: https://codepen.io/syed-owais-owais/pen/Exmrxqo

Code:



```
//How to create a 'class' in JS using classic technique
function Student(firstname, lastname) {
  this.firstname = firstname;
  this.lastname = lastname;
  this.getFullName = function() {
    return `Full Name: ${this.firstname},${this.lastname}`;
  };
}

let student = new Student('abc', 'xyz');

console.log(student.firstname);
console.log(student.lastname);
console.log(student.getFullName());

//How to do inheritance in JS using classic technique

function StaffMember() {
  this.getFullName = function() { return `Full Name: ${this.firstname}:${this.lastname}`;};
}

function Teacher(firstname, lastname){
  this.firstname = firstname;
  this.lastname = lastname;
}

Teacher.prototype = new StaffMember();

let teacher = new Teacher('ijk', 'lmn');

console.log(teacher.firstname);
console.log(teacher.lastname);
console.log(teacher.getFullName());
```

Output:



```
"abc"
"xyz"
"Full Name: abc,xyz"
"ijk"
"lmn"
"Full Name: ijk:lmn"
"abc"
"xyz"
"Full Name: abc,xyz"
"ijk"
"lmn"
"Full Name: ijk:lmn"
>
```

Lab Task

1 *Functions*
   a. Create a function called *calculator* and accept three parameters; *op1*, *op1*, *operator*.
   b. *op1*, *op2* are numerical values and *operator* is a function.
   c. From within the *calculator* function call *operator* and pass it two parameters *op1*, *op2*.
   d. Save the result returned by *operator* into a variable and return that result from *calculator* function.
   e. Validate the result by logging the result.

2 *this keyword*
   a. Create a function called *getFullName.*
   b. Using *this* keyword return full name string built by concatenating firstname and lastname.
   c. Create two objects containing firstname and lastname.
   d. Call *getFullName* using function built-in apply method and passing it the created objects one by one.
   e. Validate if full name is being returned correctly using *console.log* method.

3 *Closure*
   a. Create a *Timer* closure and make it tick (console log) after every 1 second.
      i. Create a private *counter* variable
      ii. Create a private *tick* method and use *setInterval* to increment the *counter* after 1 second and console.log *counter* variable. If the *counter* exceeds *limit* stop. You may need to accept the *limit* parameter in *tick* method.
      iii. Create a *start* method and call tick method from it. Accept the *limit* parameter.
      iv. Return the *start* method by creating an object and assigning the start method as a public API on the returned object.

4 *Module*
   a. Create *Student* module using closure and two child sub-modules *Courses* and *Result*.
   b. Add properties to each module (parent module and child sub-modules)
   c. Log each property via module and sub-module properties.

5 *Asynchronicity*
   a. Create function called *getProgramResults* and create a promise called *runProgramPromise* inside it, and await its result using *async* and *await.*

6 *Functional Inheritance*
   a. Create a base class called *Program* and two child classes called *TeacherProgram* and *StudentProgram*
   b. Provide run function in *Program* base class.
   c. Provide *debug* function in child *StudentProgram* class.
   d. Provide *release* function in child *TeacherProgram*.
   e. Inherit *Program* class behavior by creating new object of it and assigning it to the *prototype* property of each child class.

- f. Validate if respective functions are available in each child class objects by logging using *console.log* method.
- g. Validate if *run* method is available in child class objects by logging using *console.log* method.

Home Task

1 *Functions and Closure*
- a. Create a closure using function called *publishSubscribeExamResults*.
- b. Create a private member array called *subscribers*
- c. Return an object containing two functions from this function
  - i. *subscribe*: to add a subscriber
  - ii. *publish*: to publish a message to all the subscribers using *subscribers'* array

2 *Module*
- a. Create a module called *Class*.
- b. Create three sub-modules called *Teacher, Notes* and *Lecture*.
- c. Add relevant members to the containing (*Class* module) and to each sub-module.
- d. Create instances of the *Class* module and log the members

3 *Asynchronicity*
- a. Create a *ClassAlarm* function.
- b. Return an *AlarmPromise* promise from that function.
- c. Call *resolve* after the class time is over with the help of *setTimeout* function after 30 minutes and pass in the string message 'Class is over' to *resolve* function.
- d. *Await* the result using *await* keyword and console.log that returned message.

4 *Functional Inheritance*
- a. Create a base class called *Gadget* and define following properties
  - i. startTime date and time property.
  - ii. salePrice numeric property.
  - iii. *Start* and *End* methods.
- b. Create two child classes called *StopWatch* and *SmartWatch*
  - i. Define their own properties common to them.
  - ii. Inherit the base *Gadget* class into *StopWatch* class using *prototype* property
  - iii. Create instance of *StopWatch* class and verify the members by logging results.
  - iv. Create two instances of *SmartWatch* class.
  - v. Create a method called *connectToInternet* and assign it to the prototype property of *SmartWatch* class.
  - vi. Verify by if it is inherited by calling method *connectToInternet* from the two instances of *SmartWatch* class.