42.) Summary → Functions :-

- Functions can be written in inline form
- A function is a "value" that can be assigned to a variable.
- can be called by passing in arguments
- functions are objects!
- Flexible argument count.
  - ↳ If you pass more than required these will be ignored, if you ~~test~~ pass the less than they will be undefined.
- function overloading is not possible
- Default arguments :- hidden/implicit arguments
  - → arguments
  - → this

⇒ JavaScript function Declaration :-

```
function addNumbers (a,b) {
    return a+b;
}
var number = 1;
var result = addNumbers (number, number);
```

⇒ function Expression ⇒

```
var additionfn = function addNumbers (a,b) {
    return a+b;
};
result = additionfn (number, number);
```


⇒ Anonymous function Expression :-

```
var additionfn = function (a,b) {
    return a+b;
};
result = additionfn (number, number);
```

⇒ functions as object property :-

```
var mathObj = {};
mathObj.add = function (a,b) {
    return a+b;
};
result = mathObj.add (number, number);
```

45°

⇒ Array methods :- Array is an inbuilt object. like an array we have some inbuilt object in JS

Some inbuilt fn in JS :-

last position

- myArray.push(500); ⇒ add at last position
- myArray.pop(); → deletes last elements
- myArray.shift(); → remove & at front position
- myArray.unshift(42); → add at first

first position

```
var myArray = [10, 20, "Hello", {}];
myArray.push(10); // 5 → returns length
```

~~where element added.~~

myArray // Array [10, 20, "Hello", object, 10]

myArray.pop() // 10 → returns the value which is removed

myArray.pop(); // object {}

myArray // Array [10, 20, "Hello"]

myArray.shift(); // 10 → return deleted element at array. Starting position of an array.
myArray // Array ["abcd", "Hello"]

myArray.unshift("abcd"); // 3 → return total length
myArray // Array ["abcd", 20, "Hello"]

43. ⇒ Array for each method:-

Ex.1
```
var myArray = [10, 20, "Hello", 14];
myArray.forEach(function(){
    console.log("for an element");
});
```
→ output ⇒ 4 times print for an element
means this function will execute no of times this array.

Ex.2
```
var myfunction = function(item){
    console.log("element " + item);
};
myArray.forEach(myfunction);
```
element 10
" 20
" Hello
" @ [Object object]

→ along with item couple of other arguments are also passed each time when it executes.

```
myArray.forEach(function(item, index, array){
    console.log(item, index);
});
```


45. Reading Assignment (Global object methods at MDN):-
Global object Date

46. Next steps:-
- Scopes and closures
- objects and prototypes - equivalent to classes
- Asynchronous JavaScript - callbacks and promises
- Client side frameworks - Angular, React
- Server side frameworks - Node.js, Express

delete operator on array element

```
delete myArray[3]; // true
myArray // [10, 20, 30, empty],
myArray[4] // undefined
myArray[3] // undefined
```

Scopes and Closures

1. → Introduction :-

2. JavaScript functions primer :-

3. Understanding Sopes and Block Building :-

4. Function scoping in JavaScript :-

- In JS scoping is based on functions not based on blocks.

Ex-1
```
var name = "Shubham";
if (name == Shubham) {
    var department = "Engineering";
}
console.log(name); || Shubham
console.log(department); || This will be printed
                         → Engineering
```

Ex-2
```
var name = 'Shubham';
function allocateDepartment() {
    if (name == 'Shubham') {
        var department = "Engineering";
    }
}
                  ↓
            Function Scope
allocateDepartment();
Console.log(department); || Uncaught ReferenceError:
                            department is not defined
```


5. Scopes ~~and Closures~~ - Exercise :-

~~var name = "Shubham";~~
~~function~~

6. IIFE :- (Immediately invoked function Expression)

- Global variable are bad they will evil encarnate.
- Global variable available to each JS file associated with the page.

```
                                    → Global function
function myFn() {
    var a = 10;
    var b = 10;          } → Local variable
    console.log(a+b);
}
myFn();
```

same as

To avoid this we can write in this way →

```
( function () {
    var a = 10;
    var b = 10;
    console.log(a+b);
} )();
```

→ anonymous function Expression which automatically ~~invoked~~ to be Immediately invoked function expression. (IIFE)

→ Now a and b and function are not Global.

8. Read & write operations :-

Ex-1
```
var foo;
console.log(foo); || undefined
```

Ex-2
```
console.log(foo); || SyntaxError: ---------
```
means declaration is required before using a variable.

Ex: foo = 10; → This variable is created in Global scope.
console.log(foo); || 10

→ without declaration we can do write operation but not reading operation.

9. The Window object:- (Global object):-

var a = 10; → Global → This will be stored in window object as property.
fuction foo() { → This also a property in Global object.
var b = 20; → local
}

→ There is a global object window every time you execute javascript browser.
→ Every Global variable is a property on Global object (window).

→ window.a; we can use with window as well as simple a =
→ In case of browser Global object will be window.
→ node.js Global object is Global.
→ because the window is starting point for the page to render script to execute.
→ different Runtime have different Global objects.

10. Compilation & Interpretation:-

→ In language like C, Java we execute intermediate file which is generated by compiler. First compiler compile source code and create bytecode (in case of Java) and this bytecode we run.

* → But in case of javascript we actually run direct source code not any intermediate code.
* → Runtime (Browser) runs the code directly.
→ When browser get the Javascript it does two thing
(i) Compilation Step:- In this browser looks up at our source-code and identifying the particular set of things which it needs in order to execute it. So it does not execute very first time.
→ In compilation step point is not to generate inter-mediatry file but look for signs and to make a note of certain things which it needs to execute.

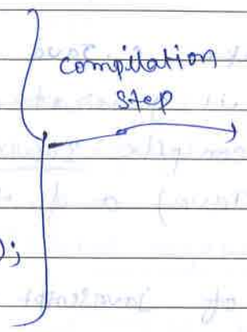
(ii) Interpretation Step:- Here actual code will run.

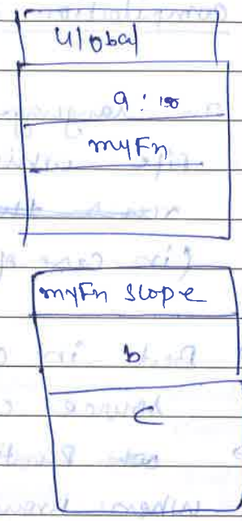
→ So Javascript is both Compiled as well as interpreted lang.
→ compilation is like accounting or bookkeping step where the compiler makes reference of all the variable, where they being used all this information is ready for execution step.

→ These two steps Happen very quickly. When browser get Javascript file it nerns the compiler compilation step a fraction of second before it runs the interpretation step. These are very closly linked.

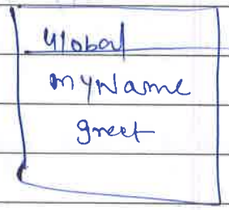
11. Understanding the compilation phase :-

Ex. var a = 10;
function myfn() {
var b = 20;
var c = b;
console.log(a+b);
}
myfn();





Ex. 2 var myName = "Shubham";
function greet(name) {
console.log("Hello" + name);
}
greet(myName);



12. ~~Understanding the interpreter step~~:-

Execution is a two-phase thing.

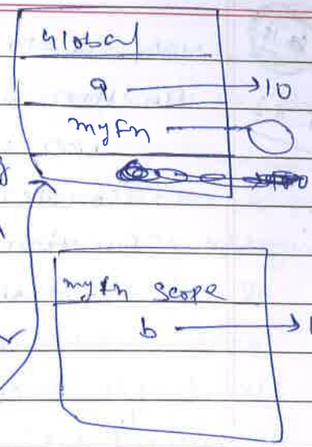
→ First step is compilation step where Scope chains are created, so there is variable being declared and each variable goes inside into one of the Scope of Scope chain.

→ In interpretation step it gets those variables by referring to the scope chain and ~~execute~~ then it uses those variables depending upon what's available in Scope chain.

13. Global Scope problem :-

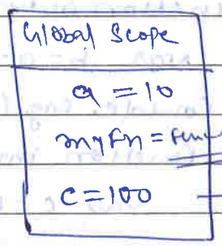
var a = 10;
function myfn() {
var b = 9;
console.log(b);
console.log(c); // → Throw error
}
myfn();

here we are trying to read but this variable is not available thereby

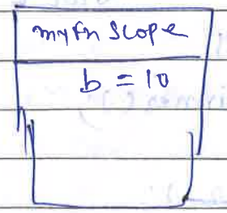


If we are writing some value to undeclared variable then interpreter first checks it in function scope & if it's not present then interpreter checks in Global Scope. If there also also this is not present then ~~interpreter~~ it will be defined there. (In Global Scope)

Ex.
var a = 10;
function myfn() {
var b = a;
console.log(b);
c = 100;
}
myfn();

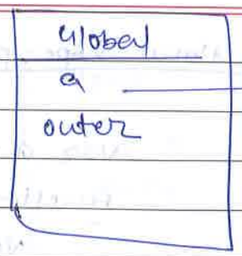


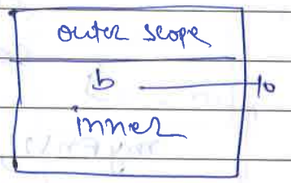
added in Global Scope at the time of interpretation or execution

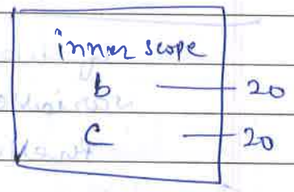


Ex-1

```
var a = 10;
function outer() {
    var b = a;
    console.log(b); //10
    function inner() {
        var b = 20;
        var c = b;
        console.log(c); //20
    }
    inner();
}
outer();
```

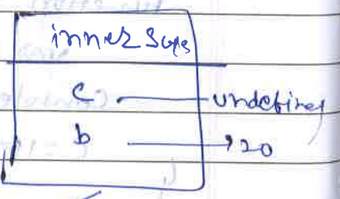






Ex-2

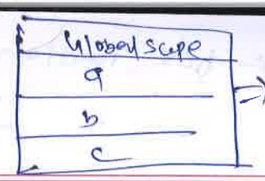
```
var a = 10;
function outer() {
    var b = a;
    console.log(b);
    function inner() {
        var c = b;
        console.log(c); //undefined
        var b = 20;
    }
    inner();
}
outer();
```



Ex-3

```
console.log(a); //undefined
var a = 10;
```

compilation



Interpretion

Hoisting :-

Jd doesn't matter in JavaScript where we declare variable they will be lift/Hoisted to top.

Actually in compilation step all the variables are recognized as per the scope chain so during interpretation step interpreter knows about all the variable declarations.

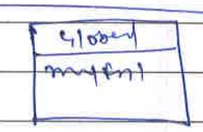
```
a = 10;
console.log(b);
c++;
console.log(c);

var a;
var b;
var c;
```

undefined

NaN

Hoisting

they will be moved to top



Ex-2

```
myFn1();

function myFn1() {
    ...
}
```

fn is hoisted to top so no error it will be executed.

⇒ Recursive function :-

```
function fnA() {
    fnB();
}
function fnB() {
    fnA();
}
```

Declaration order does not matter

↳ Here Hoisting Helps, in compilation step only we go to know about on declaration so now we can call any function and it will execute.

Hoisting does not work for function ~~not~~ expression

fnA(); //error

~~In this case~~

~~function~~ var fnA = function() {

};

Here fnA is not assigned any value in compilation step and in interepation step we first calling it so it is undefined. we can not call a function for undefined variable.

→ Hoisting will work for function declaration but not for function expression.

16. Strict mode :-

var myName = " ";

---

by mistake wrote small 'm'

myname = "Shubham";

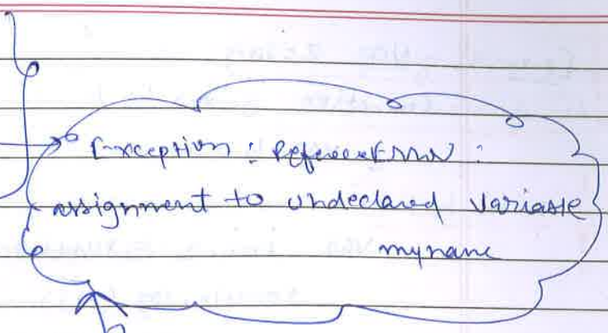
↳ but here JavaScript creates one more variable with name 'myname' in Global Scope that is very bad

→ To prevent these type of behaviour we can use Strict mode.

"use strict"

var myName = " ";

myname = "Shubham";



→ function myCode() {

"use strict"

var myName = " ";

myname = "koushik";

}

~~myName = "Shubham";~~

→ only this function will be executed in strict mode.

→ ~~Strict~~ Strict mode avoid crazy behaviour of JavaScript.

→ This can be used for whole code or where ever we want (for specific code also like function)

17. Closures :-

var a = 10;

function outer() {

var b = 20;

function inner() {

console.log(a); //10

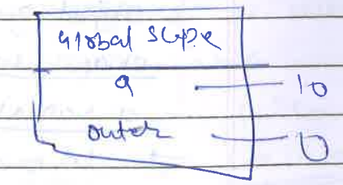
console.log(b); //20

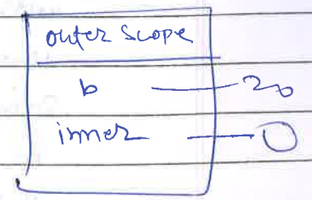
}

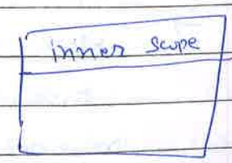
inner();

}

outer();

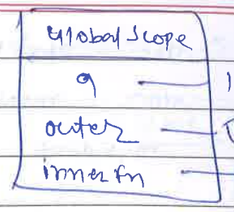


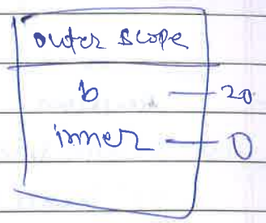




Ex-2.

```
var a = 10;
function outer () {
    var b = 20;

    var inner = function () {
        console.log (a);
        console.log ( b);
    };
    return inner;
}

var innerFn = outer ();
innerFn();
```





→ Here scope of outer is over but still values of a and b will be printed due to closures.

Reason:- when Javascript creates a function no matter where it is (may be function declaration or function expression) along with function object creation it also remembers the scope chain when the function was declared or assigned to a variable.

→ In our case 'inner' variable is assigned with function object that contains not only the function itself it also contains scope information. It remembers everything in this scope at this point of time when this function object being created.

→ Here we have two variables a and b, so it knows about these two variables and it remembers it, so no matter where you pass this object. because it has a snapshot or pointer to those valle

Imp

→ so we can give this object to third party library also because this object knows where each variable is that it needs to points.

→ It don't creates copy of those variable, instead it has pointer to the actual variable.

⇒ Definition of closures:- A function which remembers its scope.

→ function remembers scopes during the time of declaration.

18. Closures in detail:-

```
Ex.  var a = 10;
     function outer () {
         var b = 20;

         var inner = function () {
             a++;
             b++;
             console.log(a);
             console.log(b);
         };
         return inner;
     }

     var innerFn = outer ();
     innerFn(); || [11 21] a=11 b=21
     var innerFn2 = outer ();
     innerFn2(); || [12 21] a=12 b=21
```

Scope of local variable inside function is local & but 'b' in this case innerFn holds the pointer to variable 'b' So its no clear by garbage collector even after execution of outer fn is over because 'b' still has a reference with 'innerFn' so we not delete it like in java

→ i 'b' did not changes means every time fn call a new copy of local variable 'b' will be created.

→ but global variable 'a' will be done for all fn calls.

→ Practical use of closures

19. Closures in Callbacks:-

Ex.
```
var a = 10;
var fn = function() {
    console.log(a);
};
setTimeout(fn, 6000); → This will execute after 6 secs
console.log("Done") - This will execute first
```

output = Done
10 → after 6 sec setTimeout will run this function

Here we are sending fn (function object) to setTimeout method which is written in a file somewhere in Javascript engine & when it runs after 6 seconds this function so it don't know about 'a' because we are executing this function completely different place and file.

⇒ So here concept of closures come into picture. because this fn variable has the scope chain it remembers the scopes so it has pointer to 'a' So while the execution it goes to that location and gets the value and print it

→ help us to create private data & public APIs.

20. The module pattern:- (* Javascript don't have concept of public and private.

Ex:
```
var person = {
    "FN" : "Shubham",
    "LN" : "Kumar",
    "getFirstName" : function() {
        return this.FN;
    },
    "getLastName" : function() {
        return this.LN;
    }
};
person.getFirstName(); // "Shubham"
person.FN; // "Shubham"
```

Here through getters we are able to get FN but property fn is also available.

** → To avoid these variables not accessible through object we need to use closures.
→ Bcoz public and private is not available.

~~Example with closures so "FN" and "LN" will not be accessible:~~

Ex:2
```
function createPerson() {
var returnObj = {
    "FN": "Shubham",
    "LN": "Kumar",
    "getFirstName": function() {
        return this.FN;
    },
    "getLastName": function() {
        return this.LN;
    }
};
```

```
var person =
    createPerson();
person.getFirstName()
→ Shubham
person.FN = Shubham
```

This is not our solution

→ Here we are hiding variable and accessing these variable through ~~not~~ getters.

Ex-3

```
function createPerson () {
    var FN = "Shubham";
    var LN = "kumar";
    var returnObj = {
        "getFirstName" : function() {
            return FN;
        },
        "getLastName" : function(){
            return LN;
        }
    };
    return returnObj;
}

var person = createPerson();
Console.log( person.getFirstName()); || Shubham
    person.FN; || Undefined
```

→ due to closures ~~FN~~ is getFirstName has the scope chain or remembers the variable FN.

without accessing variables

Ex-4. Getting and setting ~~the variable~~ ~~object prop~~ with hiding the variables

```
function createPerson () {
    var FN = "Shubham";
    var LN = "kumar";

    var returnObj = {
        "getFN" : function () {
            return FN;
        },
        "getLN" : function () {
            return LN;
        },
        "setFirstName" : function(name) {
            FN = name;
        },
        "setLastName" : function(name) {
            LN = name;
        }
    };
    return returnObj;
}

var person = createPerson();
person.getFN(); || Shubham
person.setFirstName("Foo");
person.getFirstName(); || Foo
```

* Any variables declared in a function get created everytime the function is called.

2). Closures in asyne callbacks:-

Ex:1
```
var i;
for(i=0; i<10; i++){
    console.log(i);    // 0 1 2 3 4 5 6 7 8 9
}
```

```
var print = fuction(){ console.log(i); };
```

Ex 2.
```
for(i=0; i<10; i++){
    setTimeout(print, 1000);
}
```

output :- 10 10 10 10 10 10 10 10 10 10

bcoz setTime will execute after 1 sec till the time loop continue to run and 'i' reached to 10 and when SetTimeOut fiboll start printing it will print 10, 10.....;

@ 2. Solving asyne with closures :-

Step-1
```
var i;

var print = function(){ console.log(i); };

for(i=0; i<10; i++)
{
    (function(){
        setTimeOut(print, 1000);
    })();
}
```
→ IIFE

output :- 10, 10 .....

Step-2
```
var i;

for(i=0; i<10; i++)
{
    (function(){
        setTimeOut(function(){
            console.log(i);
        }, 1000);
    })();
}
```

output => 10, 10, 10 .....

Solution

```
var i;

for(i=0; i<10; i++){
    (function(){
        var currentValueOfI = i;
        setTimeOut(function(){
            console.log(currentValueOfI);
        }, 1000);
    })();
}
```
→ Imp

output :- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

alternate soln with function parameter

Step-4

```
var i;
for(i=0; i<10; i++){

  (function(currentValueOfI){

    setTimeOut(function(){
      console.log(currentValueOfI);
    }, 1000);
  })(i);
}

output :- 0,1,2,3 --- 9.
```

better soln than (3)

Objects and prototypes :-

→ The ES6 version has a class keyword that gives class-like syntax. But you still don't get all the typical class features.

2. **Objects Basis :-**

An object essentialy collection of multiple values.

Object creation :-

(i) **Inline :**

```
var myObj = {};
console.log(myObj); // object {}
myObj.foo = "value";
myObj.foo // "value"
myObj.foo = 100;
myObj.foo // 100
```

(ii) **Object literals**

```
var myObj = {
  "name" : "Shubham",
  "age" : 20,
  "address" : {
      "street" : "123 JS",
      "city" : "JS",
      "pincode" : 12345
  }
};

myObj.name // Shubham
myObj["name"] // Shubham
```

3) JS is oops Language. ~~bec~~ in sense we can create Objects to solve our problems.

Creating Objects:-

Ex.1:

```
var empl = { };
empl.FN = "Shubham"
empl.LN = "kumar"
empl.gender = "M";
```

if we have 100 employee we need to write ~~the~~ above code multiple time. so it is tedious.

Soln for Ex 1:

Ex.2 =

```
function createEmployeeObject () {
    var newObject = { };
    newObject.FN = "Shubham";
    newObject.LN = "kumar";
    newObject.gender = "M";
    return newObject;
}
```

① ④ ⑤

also we can pass argument to make it dynamic

```
emp2 empn = createEmployee obj ( );
emp3 = createEmployeeObj ( );
```


4) Javascript Constructor:-

In Ex.2 ~~we~~ ~~the~~ ~~code~~ a/c to example-2 if we will create multiple different object then line ① and ⑤ will be common so we can skip their creation object and return by using constructor method. (A shortcut way)

Ex. =

```
function createEmpObj (FN, LN, gender) {
    // var this = { };
    this.FN = FN;
    this.LN = LN;
    this.gender = gender; // return this;
}
```

JS interpreter do automatically (internally)

```
var emp3 = new createEmpObj ("Shubh", "Kum", "M");
```

we have to use this keyword

5) Diff b/w normal and custom calling method

```
function createBicycle (cadance, speed, gear) {
    var newBicycle = { };
    newBicycle.cadance = cadance;
    newBicycle.speed = speed;
    newBicycle.gear = gear;
    return newBicycle;
}
```

normal way

```
var bicycle1 = createBicycle (50, 20, 4);
```

constructor way

```
function Bicycle (cadance, gear) {
    this.cadance = cadance;
    this.gear = gear;
}
var bicycle3 = new Bicycle (50, 4);
```

for convention we use Pascal Casing for constructor function

6. Switching functions Types and Calls:-

var bicycle1 = new CreateBicycle(50, 20, 4)

function CreateBicycle(Cadace, Speed, gear){

// var this

var newBicycle = {};

newBicycle.cadace = Cadace;

newBicycle.speed = speed;

newBicycle.gear = gear;

return newBicycle;

// return this;

}

Here we are using 'this' obj so it is useless to call createBicycle method in constructor mode.

→ when we call a fn with new (constructor mode) beyond then interpreter is automatically adds these two lines of code.

* Calling a regular function in constructor mode using the new keyword still works! but it's useless.

→ Calling Constructor fn as regular fn:-

function Bicycle(Cadace, gSpeed, gear){

this.cadace = Cadace;

this.speed = Speed;

this.gear = gear;

}

var bicycle2 = Bicycle(50, 20, 4);

→ Here interpreter runs this function as regular method

'this' is not available in function scope. So it will go for global scope and in global scope 'this' refers to window object So Cadace, Speed and gear added to window object

window.a // 10 window.cadace; // 50

this.cadace; // 20

bicycle2 // undefinie

↓

because Bicycle() not return anything

→ if we return 'this' in Bicycle() fn

'bicycle2' will be hold 'window' object.

7. Function expression Types:-

(i) function foo(){

console.log("Hello");

}

foo(); // method 1 for calling functions in Javascript

(ii) var obj = {};

obj.foo = function(){

console.log("Hello");

};

obj.foo(); // method-2

* method-1 and method-2 seems same but they are different.

(iii) new foo(); // method-3 (constructor mode)

(iv) foo.call({}) // method-4

can pass an object that will be the this object for that function

→ (Execution context in JS)

8. The 'this' argument values:-

* There are two default arguments to every function call: 'arguments' and "this."

(i)
```
function foo() {
    console.log("Hello");   console.log(this) → window reference
}
foo(); // method #1
```
→ In this case value for this parameter is the global object — (window)

(ii)
```
var obj = { };
obj.foo = function() {
    console.log("Hello"); // Hello
    console.log(this); // object obj { foo: obj.foo() }
};
obj.foo(); // method
```

method 2:- → Calling functions as property of an object reference

this reference :- The object reference.
↳ in above example — obj

(iii) new foo();

method 3:- Calling standalone functions using 'new' keyword

'this' reference :- The newly created object


```
function foo() {
    // var this = { };
    console.log("Hello"); // Hello
    console.log(this); // object { } - empty object
    // return this;
}
```

new foo();

9. working on objects with 'this' reference :-

Ex. 2

```
function Bicycle(cadence, speed, gear, tirepressure) {
    this.cadence = cadence;
    this.speed = speed;
    this.gear = gear;
    this.tirepressure = tirepressure;
    this.inflateTires = function() {
        this.tirepressure += 3; }
}

var bicycle1 = new Bicycle(50, 20, 4, 25); // tirep = 25
bicycle1.inflateTires(); → Tirepressure = 28

var bicycle2 = new Bicycle(50, 20, 4, 30); → tp = 30
bicycle2.inflateTires(); → tp = 33
```

bicycle1.tirepressure = 28;

bicycle2.tirepress = 33

'this' reference for above two function are different even though one of them inside another.