Var,Let and Const in details:

```
//what is variable var in javascript
//Creating a variable in JavaScript is called "declaring" a variable:
var myname;
console.log(myname);//if we will call a variable with value assignement, it will be "undefined"
//you can also assign a value to the variable when you declare it:
myname="Rajeev";
console.log(myname);//output:Rajeev

//so first we need to declare the variable then we need to assign the value
```

Variable Scope

Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are local and global:

- Global variables are those declared outside of a block
- Local variables are those declared inside of a block

In the example below, we will create a global variable.

```
// Initialize a global variable
var myName = "Rajeev";
```

We learned that variables can be reassigned. Using local scope, we can actually create new variables with the same name as a variable in an outer scope without changing or reassigning the original value.

In the example below, we will create a global "myName" variable. Within the function is a local variable with the same name. By sending them to the console, we can see how the variable’s value is different depending on the scope, and the original value is not changed.

```
// Initialize a global variable
var myName = "Rajeev";

function disaplay() {
  // Initialize a local, function-scoped variable
  var myName = "Sanjeev";
  console.log(myName);
}

// Log the global and local variable
console.log(myName);
disaplay();
console.log(myName);
```

Output:

```
Rajeev==>global value
Sanjeev==>local value
Rajeev==>global value
```

In this example, the local variable is function-scoped. Variables declared with the var keyword are always function-scoped, meaning they recognize functions as having a separate scope. This locally-scoped variable is therefore not accessible from the global scope.

```
var showname = true;

// Initialize a global variable
let myName = "Rajeev";

if (showname) {
  // Initialize a block-scoped variable
  let myName = "Sanjeev";
  console.log(`My name inside is ${myName}.`);
}

console.log(`Outside my name is ${myName}.`);
```

o/p:

```
My name inside is Sanjeev.

Outside my name is Rajeev.
```

Same example with var keyword:

```
var showname = true;


// Initialize a global variable
var myName = "Rajeev";

if (showname) {
// Initialize a block-scoped variable
var myName = "Sanjeev";
console.log(`My name inside is ${myName}.`);
}

console.log(`Outside my name is ${myName}.`);
```

O/P:

```
My name inside is Sanjeev.

Outside my name is Sanjeev.
```

In the result of this example, both the global variable and the block-scoped variable end up with the same value, “Sanjeev”. This is because instead of creating a new local variable with var, you are reassigning the same variable in the same scope. var does not recognize if to be part of a different, new scope. It is generally recommended that you declare variables that are block-scoped, as they produce code that is less likely to unintentionally override variable values.

Constants

Many programming languages feature *constants*, which are values that cannot be modified or changed. In JavaScript, the const identifier is modelled after constants, and the values assigned to a const cannot be reassigned.

It is common convention to write all const identifiers in uppercase. This marks them as readily distinguishable from other variable values.

In the example below, we initialize the variable MYNAME as a constant with the const keyword. Trying to reassign the variable will result in an error.

```
// Assign value to const
const MYNAME = "Rajeev";

// Attempt to reassign value
MYNAME = "Sanjeev";

console.log(MYNAME);
```

Output
```
error: Error: "MYNAME" is read-only
```

Since const values cannot be reassigned, they need to be declared and initialized at the same time, or will also throw an error.

```
// Declare but do not initialize a const
const TODO;

console.log(TODO);
```

Output

Uncaught SyntaxError: Missing initializer in const declaration

Values that cannot change in programming are known as *immutable*, while values that can be changed are *mutable*. Although const values cannot be reassigned, they are mutable as it is possible to modify the properties of objects declared with const.

```
// Create a CAR object with two properties
const CAR = {
   color: "blue",
   price: 15000
}

// Modify a property of CAR
CAR.price = 20000;

console.log(CAR);
```

Output

```
{ color: 'blue', price: 20000 }
```

NOTE:

We can change const value by declaring it as an object.

Functions in javascript:

1. JavaScript functions are defined with the function keyword.(i.e named function)

2. You can use a function declaration or a function expression.(i.e anonymous function [a function without a name])

named function:

```
function functionName(parameters) {
        // code to be executed
}
```

```
function myFunction(a, b) {
        return a * b;
}
```

```
console.log(myFunction(2,3));//o/p:6
```

Anonymous function:

```
var x = function (a, b) {return a * b};

console.log(x(2,3));//o/p:6
```

The Function() Constructor

As you have seen in the previous examples, JavaScript functions are defined with the function keyword.

Functions can also be defined with a built-in JavaScript function constructor called Function().

```
var myFunction = new Function("a", "b", "return a * b");
console.log(myFunction(2,3));//o/p:6
```

NOTE:

Most of the time, you can avoid using the new keyword in JavaScript. So don’t try to use above example with new keyword. In place of that use older function:

```
var myFunction = function (a, b) {return a * b};

console.log(myFunction(2,3));//o/p:6
```

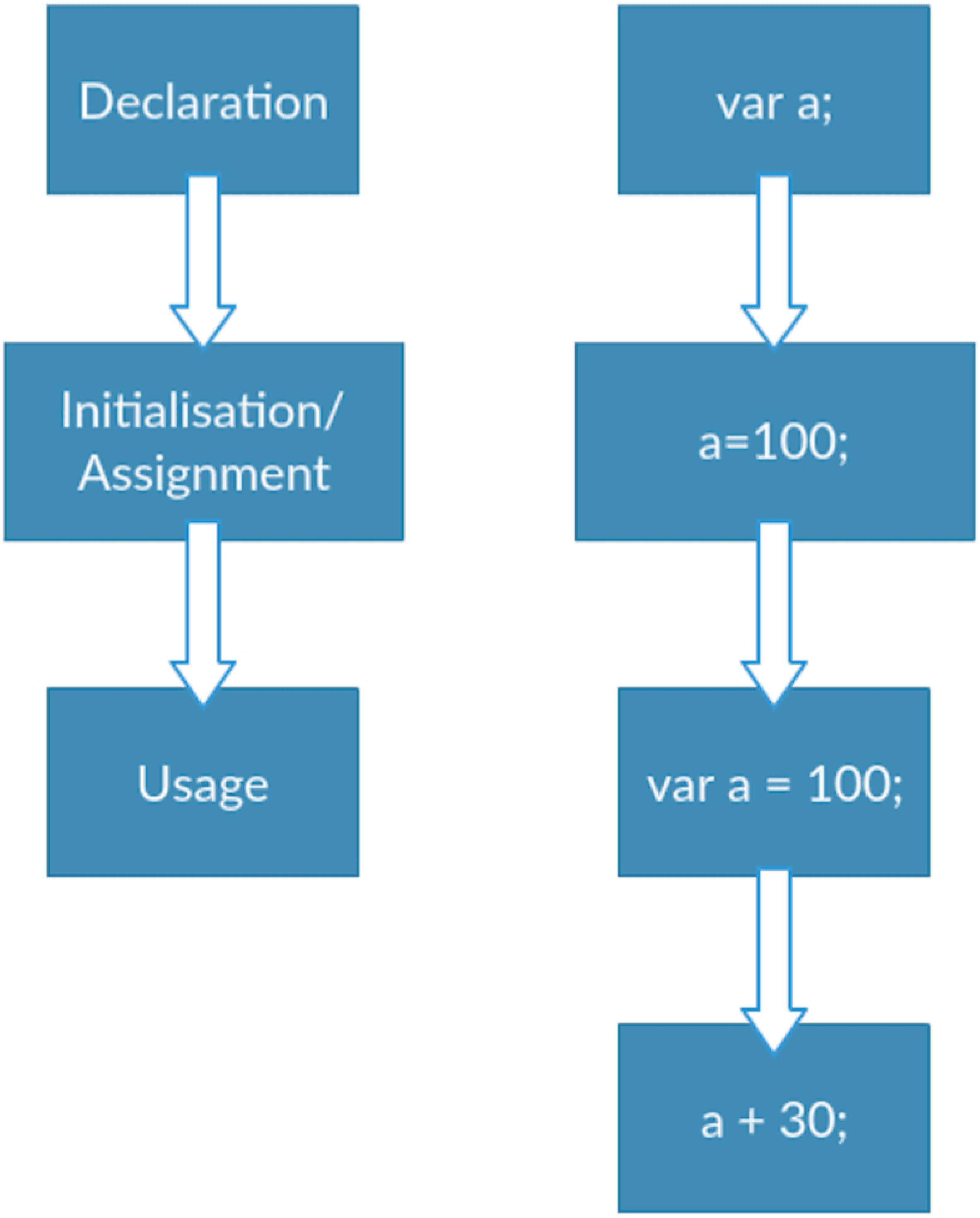
Hoisting in javascript:

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Of note however, is the fact that the hoisting mechanism only moves the declaration. The assignments are left in place.

Variable hoisting:

The following is the JavaScript lifecycle and indicative of the sequence in which variable declaration and initialisation occurs.



However, since JavaScript allows us to both declare and initialise our variables simultaneously, this is the most used pattern:

```
var a = 100;
```

It is however important to remember that in the background, JavaScript is religiously declaring then initialising our variables.

As we mentioned before, all variable and function declarations are hoisted to the **top** of their scope. I should also add that variable *declarations* are processed before any code is executed.

However, in contrast, *undeclared* variables do not exist until code assigning them is executed. Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed. This means that, **all undeclared variables are global variables.**

In most of the examples so far, we've used var to *declare* a variable, and we have *initialized* it with a value. After declaring and initializing, we can access or reassign the variable.

If we attempt to use a variable before it has been declared and initialized, it will return undefined.

```
// Attempt to use a variable before declaring it
console.log(x);

// Variable assignment
var x = 100;
```

```
Output:
undefined
```

However, if we omit the var keyword, we are no longer declaring the variable, only initializing it. It will return a ReferenceError and halt the execution of the script.

```
// Attempt to use a variable before declaring it
console.log(x);

// Variable assignment without var
x = 100;
```

```
Output
error: ReferenceError: x is not defined
```

The reason for this is due to *hoisting*, a behavior of JavaScript in which variable and function declarations are moved to the top of their scope. Since only the actual declaration is hoisted, not the initialization, the value in the first example returns undefined.

To demonstrate this concept more clearly, below is the code we wrote and how JavaScript actually interpreted it.

```
// The code we wrote
console.log(x);
var x = 100;
```

```
// How JavaScript interpreted it
var x;
console.log(x);
x = 100;
```

JavaScript saved x to memory as a variable before the execution of the script. Since it was still called before it was defined, the result is undefined and not 100. However, it does not cause a ReferenceError and halt the script. Although the var keyword did not actually change location of the var, this is a helpful representation of how hoisting works. This behavior can cause issues, though, because the programmer who wrote this code likely expects the output of x to be true, when it is instead undefined.

We can also see how hoisting can lead to unpredictable results in the next example:

```
// Initialize x in the global scope
var x = 100;

function hoist() {
  // A condition that should not affect the outcome of the code
  if (false) {
    var x = 200;
  }
  console.log(x);
}

hoist();
```

```
Output:
undefined
```

In this example, we declared x to be 100 globally. Depending on an if statement, x could change to 200, but since the condition was false it should not have affected the value of x. Instead, x was hoisted to the top of the hoist() function, and the value became undefined.

Above example javascript internally changed like this :

```
function hoist() {
        var x;
        if (false) {
                x = 200;
        }
        console.log(x);
}
```

if condition is true then x value will get changed to 200:

```
function hoist() {
        var x;
        if (true) {
                x = 200;
        }
        console.log(x);
}
```

This type of unpredictable behavior can potentially cause bugs in a program. Since let and const are block-scoped, they will not hoist in this manner, as seen below.

```
// Initialize x in the global scope
let x = true;

function hoist() {
  // Initialize x in the function scope
  if (true) {
    let x = false;
  }
  console.log(x);
}

hoist();

Output:
true

// Initialize x in the global scope
let x = true;

function hoist() {
  // Initialize x in the function scope
  if (false) {
    let x = false;
  }
  console.log(x);
}

hoist();

Output:
true
```

In the above two example we are not able to change the x value to false.

Duplicate declaration of variables, which is possible with var, will throw an error with let and const.

```
// Attempt to overwrite a variable declared with var
var x = 1;
var x = 2;

console.log(x);
```

```
Output
2
```

```
// Attempt to overwrite a variable declared with let
let y = 1;
let y = 2;

console.log(y);
```

```
Output
Uncaught SyntaxError: Identifier 'y' has already been declared
```

To summarize, variables introduced with var have the potential of being affected by hoisting, a mechanism in JavaScript in which variable declarations are saved to memory. This may result in undefined variables in one’s code. The introduction of let and const resolves this issue by throwing an error when attempting to use a variable before declaring it or attempting to declare a variable more than once.

So hoisting takes place only for var keyword not for let and const.

global variables

```
console.log(hoist); // Output: undefined

var hoist = 'The variable has been hoisted.';
```

We expected the result of the log to be: ReferenceError: hoist is not defined, but instead, its output is undefined.

Why has this happened?

This discovery brings us closer to wrangling our prey.

JavaScript has hoisted the variable declaration. This is what the code above looks like to the interpreter:

```
var hoist;

console.log(hoist); // Output: undefined
hoist = 'The variable has been hoisted.';
```

Because of this, we can use variables before we declare them. However, we have to be careful because the hoisted variable is initialised with a value of undefined. The best option would be to declare and initialise our variable before use.

Function hoisting:

JavaScript functions can be loosely classified as the following:

1. Function declarations
2. Function expressions

We'll investigate how hoisting is affected by both function types.

Function declarations

These are of the following form and are hoisted completely to the top. Now, we can understand why JavaScript enable us to invoke a function seemingly before declaring it.

```
hoisted(); // Output: "This function has been hoisted."

function hoisted() {
  console.log('This function has been hoisted.');
};
```

Function expressions

Function expressions, however are not hoisted.

```
expression(); //Output: "TypeError: expression is not a function

var expression = function() {
  console.log('Will this work?');
};
```

Let's try the combination of a function declaration and expression.

```
expression(); // Ouput: TypeError: expression is not a function

var expression = function hoisting() {
  console.log('Will this work?');
};
```

As we can see above, the variable declaration var expression is hoisted but it's assignment to a function is not. Therefore, the intepreter throws a TypeError since it sees expression as a *variable* and not a *function*.

Order of precedence

It's important to keep a few things in mind when declaring JavaScript functions and variables.

1. Variable assignment takes precedence over function declaration
2. Function declarations take precedence over variable declarations

> Function declarations are hoisted over variable declarations but not over variable assignments.

Let's take a look at what implications this behaviour has.

Variable assignment over function declaration

```
var double = 22;

function double(num) {
  return (num*2);
}

console.log(typeof double); // Output: number
```

Function declarations over variable declarations

```
var double;

function double(num) {
  return (num*2);
}

console.log(typeof double); // Output: function
```

Even if we reversed the position of the declarations, the JavaScript interpreter would still consider double a function.

Hoisting Classes

JavaScript classes too can be loosely classified either as:

1. Class declarations
2. Class expressions

Class declarations

Much like their function counterparts, JavaScript class declarations are hoisted. However, they remain uninitialised until evaluation. This effectively means that you have to declare a class before you can use it.

```
var Frodo = new Hobbit();
Frodo.height = 100;
Frodo.weight = 300;
console.log(Frodo); // Output: ReferenceError: Hobbit is not defined

class Hobbit {
  constructor(height, weight) {
    this.height = height;
    this.weight = weight;
  }
}
```

I'm sure you've noticed that instead of getting an undefined we get a Reference error. That evidence lends claim to our position that class declarations are hoisted.

If you're paying attention to your linter, it supplies us with a handy tip.

Hobbit was used before it is declared, which is illegal for class variables

So, as far as class declarations go, to access the class declaration, you have to declare first.

```
class Hobbit {
  constructor(height, weight) {
    this.height = height;
    this.weight = weight;
  }
}

var Frodo = new Hobbit();
Frodo.height = 100;
Frodo.weight = 300;
console.log(Frodo); // Output: { height: 100, weight: 300 }
```

Class expressions

Much like their function counterparts, class expressions are not hoisted.

Here's an example with the un-named or anonymous variant of the class expression.

```
var Square = new Polygon();
Square.height = 10;
Square.width = 10;
console.log(Square); // Output: TypeError: Polygon is not a constructor

var Polygon = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Here's an example with a named class expression.

```
var Square = new Polygon();
Square.height = 10;
Square.width = 10;
console.log(Square); // Output: TypeError: Polygon is not a constructor
```

```
var Polygon = class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

The correct way to do it is like this:

```
var Polygon = class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

var Square = new Polygon();
Square.height = 10;
Square.width = 10;
console.log(Square);
```

Understanding Promises

A Promise in short:

"Imagine you are a **kid**. Your mom **promises** you that she'll get you a **new phone** next week."

You *don't know* if you will get that phone until next week. Your mom can either *really buy* you a brand new phone, or *stand you up* and withhold the phone if she is not happy :(.

That is a **promise**. A promise has 3 states. They are:

1. Pending: You *don't know* if you will get that phone
2. Fulfilled: Mom is happy, she buys you a brand new phone
3. Rejected: Your mom is happy, she withholds the phone

Creating Promise:

Let's convert this to JavaScript.

```
var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);
```

The code is quite expressive in itself.

```
// promise syntax look like this
new Promise(/_ executor_/ function (resolve, reject) { ... } );
```

Consuming Promise:

Now that we have the promise, let's consume it.

```
// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
         // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};

askMom();
```

Chaining Promise:

Promises are chainable.

Let's say, you, the kid, **promises** your friend that you will **show them** the new phone when your mom buy you one.

That is another promise. Let's write it!

```
// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};
```

Notes:

```
// shorten it
...

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};
```

Let's chain the promises. You, the kid can only start the showOff promise after the willIGetNewPhone promise.

```
...

// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
            console.log(fulfilled);
         // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};
```

That's how easy to chain the promise.

Promises are Asynchronous

Promises are asynchronous. Let's log a message before and after we call the promise.

```
// call our promise
var askMom = function () {
    console.log('before asking Mom'); // log before
    willIGetNewPhone
        .then(showOff)
        .then(function (fulfilled) {
            console.log(fulfilled);
        })
        .catch(function (error) {
            console.log(error.message);
        });
    console.log('after asking mom'); // log after
}
```

What is the sequence of expected output? Probably you expect:

1. before asking Mom
2. Hey friend, I have a new black Samsung phone.
3. after asking mom

However, the actual output sequence is:

1. before asking Mom
2. after asking mom
3. Hey friend, I have a new black Samsung phone.

Why? Because life (or JS) waits for no man.

You, the kid, wouldn't stop playing while waiting for your mom promise (the new phone). Don't you? That's something we call **asynchronous**, the code will run without blocking or waiting for the result. Anything that need to wait for promise to proceed, you put that in .then.

Async Await:

Async

Async functions enable us to write promise based code as if it were synchronous, but without blocking the execution thread. It operates asynchronously via the event-loop. Async functions will always return a value. Using async simply implies that a promise will be returned, and if a promise is not returned, JavaScript automatically wraps it in a resolved promise with its value.

```
async function firstAsync() {
  return 27;
}
firstAsync().then(alert); // 27
```

Running the above code gives the alert output as 27, it means that a promise was returned, otherwise the .then() method simply would not be possible.

Await

The await operator is used to wait for a Promise. It can be used inside an Async block only. The keyword Await makes JavaScript wait until the promise returns a result. It has to be noted that it only makes the async function block wait and not the whole program execution.

The code block below shows the use of Async Await together.

```
async function firstAsync() {
    let promise = new Promise((res, rej) => {
        setTimeout(() => res("Now it's done!"), 1000)
    });
    // wait until the promise returns us a value
    let result = await promise;

    // "Now it's done!"
    alert(result);
    };firstAsync();
```

Things to remember when using Async Await

We can’t use the await keyword inside of regular functions.

```
function firstAsync() {
  let promise = Promise.resolve(10);
  let result = await promise; // Syntax error
}
```

To make the above function work properly, we need to add async before the function firstAsync();

Async Await makes execution sequential

Not necessarily a bad thing, but having paralleled execution is much much faster.

For example:

```
function promise1(){
        return new Promise(function(resolve,reject){
                resolve(1);
        })

}
function promise2(){
        return new Promise(function(resolve,reject){
                resolve(2);
        })

}
async function sequence() {
        await promise1(50); // Wait 50ms…
        await promise2(50); // …then wait another 50ms.
        return "done!";
}

        const a=sequence();
        a.then(function(resp){
                console.log(resp);
        }).catch(function(err){
        console.log("The error is: ",err);
})
```

The above takes 100ms to complete, not a huge amount of time but still slow.

This is because it is happening in sequence. Two promises are returned, both of which takes 50ms to complete. The second promise executes only after the first promise is resolved. This is not a good practice, as large requests can be very time consuming. We have to make the execution parallel.

That can be achieved by using Promise.all() .

Promise.all()

```
async function sequence() {
        await Promise.all([promise1(), promise2()]);
        return "done!";
}

const a=sequence();
a.then(function(resp){
        console.log(resp);
}).catch(function(err){
        console.log("The error is: ",err);
})
```

The promise.all() function resolves when all the promises inside the iterable have been resolved and then returns the result.

Another method:

```
async function sequence() {
	// Start a 500ms timer asynchronously…
	// …meaning this timer happens in parallel.
	const wait1=promise1(50);
	const wait2=promise2(50);
	// Wait 50ms for the first timer…
	await wait1;
	// …by which time this timer has already finished.
	await wait2;
	return "done!";
}
```

Async Await is very powerful but they come with caveats. But if we use them properly, they help to make our code very readable and efficient.

Callback in javascript:

Callbacks are a great way to handle something after something else has been completed. By something here we mean a function execution. If we want to execute a function right after the return of some other function, then callbacks can be used.

JavaScript functions have the type of Objects. So, much like any other objects (String, Arrays etc.), They can be passed as an argument to any other function while calling.

See below example:

```
// add() function is called with arguments a, b
// and callback, callback will be executed just
// after ending of add() function
function add(a, b , callback){
	console.log(`The sum of ${a} and ${b} is ${a+b}.` +"<br>");
	callback();
}

//disp() function is called just after the ending of add() function
 function disp(){
   console.log('This must be printed after addition');
 }
// Calling add() function
 add(5,6,disp);
```

output:

```
‘The sum of 5 and 6 is 11.'
'This must be printed after addition'
```

An alternate way to implement above code is shown below with anonymous functions being passed.

```
add(5,6,function disp(){
	console.log('This must be printed after addition.');
});
```

Callbacks are primarily used while handling asynchronous operations like – making an API request to the Google Maps, fetching/writing some data from/into a file, registering event listeners and related stuff. All the operations mentioned uses callbacks. This way once the data/error from the asynchronous operation is returned, the callbacks are used to do something with that inside our code.

Javascript Pass by Value & Pass by Reference

Pass by Value:

In Pass by Value, Function is called by directly passing the value of the variable as the argument. Changing the argument inside the function doesn’t affect the variable passed from outside the function.

Javascript always pass by value so changing the value of the variable never changes the underlying primitive (String or number).

```
function callByValue(varOne, varTwo) {
	console.log("Inside Call by Value Method");
	varOne = 100;
	varTwo = 200;
	console.log("varOne =" + varOne +"varTwo =" +varTwo);
}
let varOne = 10;
let varTwo = 20;
console.log("Before Call by Value Method");
console.log("varOne =" + varOne +"varTwo =" +varTwo);
callByValue(varOne, varTwo);
console.log("After Call by Value Method");
console.log("varOne =" + varOne +"varTwo =" +varTwo);
```

output will be :

```
---------------
Before Call by Value Method
varOne =10 varTwo =20
Inside Call by Value Method
varOne =100 varTwo =200
After Call by Value Method
varOne =10 varTwo =20
```

Pass by Reference:

In Pass by Reference, Function is called by directly passing the reference/address of the variable as the argument. Changing the argument inside the function affect the variable passed from outside the function. In Javascript objects and arrays follows pass by reference.

```
function callByReference(varObj) {
        console.log("Inside Call by Reference Method");
        varObj.a = 100;
        console.log(varObj);
}
let varObj = {a:1};
console.log("Before Call by Reference Method");
console.log(varObj);callByReference(varObj);
console.log("After Call by Reference Method");
console.log(varObj);
```

output will be :

```
--------------- Before Call by Reference Method
{a: 1}
Inside Call by Reference Method
{a: 100}
After Call by Reference Method
{a: 100}
```

so if we are passing object or array as an argument to the method, then there is a possibility that value of the object can change.

What is “this” keyword in JavaScript

this

In other words, every javascript function while executing has a reference to its current execution context, called ***this***. Execution context means here is how the function is called.

To understand *this* keyword, only we need to know how, when and from where the function is called, does not matter how and where function is declared or defined.

```
function bike() {
  console.log(this.name);
}
var name = "Ninja";
var obj1 = { name: "Pulsar", bike: bike };
var obj2 = { name: "Gixxer", bike: bike };
bike();           // "Ninja"
obj1.bike();      // "Pulsar"
```

```
obj2.bike();      // "Gixxer"
```

In the above code snippet, the job of bike() function is printing the this.name which means it’s trying to print the value of name property of the current execution context(i.e.*this* object).

In the above code snippet, when function bike() gets called it prints “Ninja” because the context of execution is not specified so by default its global context and there is a variable name is present at global context whose value is “Ninja”.

In case of obj1().bike() call, “Pulsar” gets printed and the reason behind this is function bike() gets called with the execution context as obj1 so this.name became obj1.name . Same with obj2.bike()call where the execution context of function bike() is obj2.

Default and Implicit binding of “this”

- If we are in strict mode then the default value of *this* keyword is undefined otherwise *this* keyword act as global object, it’s called default binding of *this* keyword. (default is window object in case of browser).

The “new” keyword in JavaScript

The ***new*** keyword in front of any function turns the function call into constructor call and below things occurred when *new* keyword put in front of function

- A brand new empty object gets created
- new empty object gets linked to prototype property of that function
- same new empty object gets bound as ***this*** keyword for execution context of that function call
- if that function does not return anything then it implicit returns ***this*** object.

```
function bike() {
  var name = "Ninja";
  this.maker = "Kawasaki";
  console.log(this.name + " " + maker);  // undefined Bajaj
}
var name = "Pulsar";
var maker = "Bajaj";
obj = new bike();
console.log(obj.maker);                  // "Kawasaki"
```

In the above code snippet, bike function is get called with *new* keyword in front of it. So, it creates a new object then that new object gets linked to prototype chain of function bike, after that the created new object bound to *this* object and function returns *this* object. That’s how the returned this object assigned to obj andconsole.log(obj.maker) prints “Kawasaki” .

In the above code snippet, this.name inside function bike() does not print “Ninja” or “Pulsar” instead it prints undefined because the name variable declared inside the function bike() and this.name are totally 2 different things. Same way this.maker and maker are different inside function bike() .

Precedence of “this” keyword bindings

- First it checks whether the function is called with ***new*** keyword.
- Second it checks whether the function is called with call or apply method means explicit binding.
- Third it checks if the function called via context object (implicit binding).
- Default global object (undefined in case of strict mode).

Call(),Apply(), Bind():

JavaScript is a dynamic language, and is flexible enough to let you do things like multiple inheritance. That’s when an object or a class can inherit characteristics from more than one parent. This can be done using one of these 3 methods: call / apply / bind.

Call():
```
var pokemon = {
        firstname: 'Pika',
        lastname: 'Chu ',
        getPokeName: function() {
                var fullname = this.firstname + ' ' + this.lastname;
                return fullname;
        }
};

var pokemonName = function(snack, hobby) {
        console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
};

pokemonName.call(pokemon,'sushi', 'algorithms');
 // o/p: Pika Chu loves sushi and algorithms
```

call() and apply() serve the **exact same purpose.** The ***only difference between how they work is that*** call() expects all parameters to be passed in individually, whereas apply() expects an array of all of our parameters.

Apply():

```
var pokemon = {
        firstname: 'Pika',
        lastname: 'Chu ',
        getPokeName: function() {
                var fullname = this.firstname + ' ' + this.lastname;
                return fullname;
        }
};

var pokemonName = function(snack, hobby) {
        console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
};

pokemonName.apply(pokemon,['sushi', 'algorithms']);
 // o/p: Pika Chu loves sushi and algorithms
```

Bind():
The bind() method creates a new function that, when called, has its this keyword set to the provided value.

1st Example:

```
let obj = {things: 3};
let addThings = function(a, b, c){
return this.things + a + b + c;
};
var bindedobj=addThings.bind(obj);//first bind to function
console.log( bindedobj(4,5,6) );//then pass function parameters to binded object
//o/p:18
```

2nd Example:

```
var pokemon = {
        firstname: 'Pika',
        lastname: 'Chu ',
        getPokeName: function() {
                var fullname = this.firstname + ' ' + this.lastname;
                return fullname;
        }
};

var pokemonName = function(snack, hobby) {
        console.log(this.getPokeName() + 'I choose you!');
        console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
        //both the function is getting called from  pokemon object.
};
```

```
var logPokemon = pokemonName.bind(pokemon); //first bind object to the function
// creates new object and binds pokemon. 'this' of pokemon === pokemon now

logPokemon('sushi', 'algorithms'); // now pass function parameters to bind object
```

o/p:

```
Pika Chu I choose you!
Pika Chu loves sushi and algorithms
```

Closure in JavaScript

a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

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.

The inner function will have access to the variables in the outer function scope, even after the outer function has returned

```
// Explaination of closure
/* 1 */ function foo()
/* 2 */ {
/* 3 */    var b = 1;
/* 4 */   function inner(){
/* 5 */      return b;
/* 6 */   }
```

Explanation: Interesting thing to note here is from **line number 9** to **line number 12** . At line number 9 we are done with the execution of **function foo()** and the entire body of **function inner()** is returned and stored in **var get_func_inner**, due to the line 7 **return inner**.
[The return statement does not execute the inner function – function is executed only when followed by () , but rather the return statement returns the entire body of the function.]

We can access the variable **b** which is defined in **function foo()** through **function inner()** as the later preserves the scope chain of enclosing function at the time of execution of enclosing function i.e. the inner function knows the value of **b** through it's scope chain.
This is closure in action that is inner function can have access to the outer function variables as well as all the global variables.
Output of the above code:

1

1

```
1
```

2nd Example:

```
function foo(outer_arg) {
        function inner(inner_arg) {
                return outer_arg + inner_arg;
        }
        return inner;
}
var get_func_inner = foo(5);

console.log(get_func_inner(4));
console.log(get_func_inner(3));
```

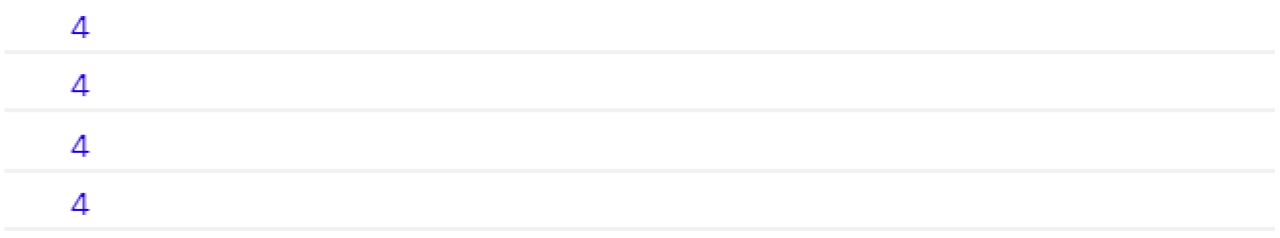
Explanation: In the above example we used a parameter function rather than a default one. Note even when we are done with the execution of **foo(5)** we can access the **outer_arg** variable from the inner function. And on execution of inner function produce the summation of **outer_arg** and **inner_arg** as desired.

Output:

```
9
8
```

```
function outer()
{
  var arr = [];
  var i;
```

Output:



```
4
4
4
4
```

Explanation: Did you guess the right answer? In the above code we have created four closure which point to the variable i which is local variable to the function outer. **Closure don't remember the value** of the variable it only **points** to the variable or stores the **reference** of the variable and hence, returns the current value. In the above code when we try to update the value of it gets reflected to all because the closure stores the reference.

Lets see an correct way to write the above code so as to get different values of i at different index.

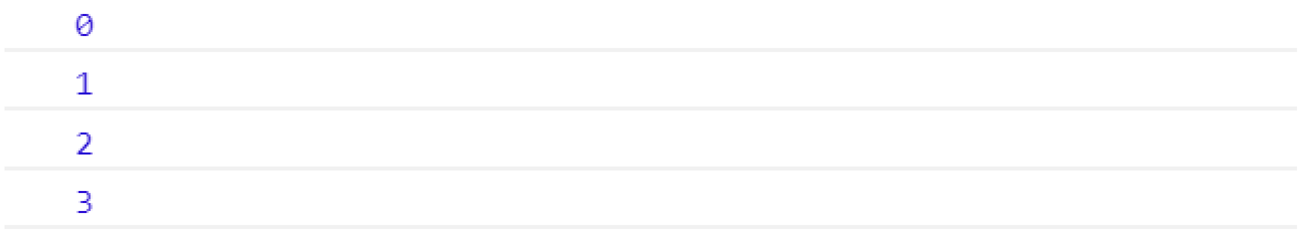
Example 4:

filter_none

brightness_4

```
// Outer function
function outer()
{
    function create_Closure(val)
    {
     return function()
       {
        return val;
      }
   }
   var arr = [];
   var i;
   for (i = 0; i < 4; i++)
    {
     arr[i] = create_Closure(i);
   }
   return arr;
}
var get_arr = outer();
console.log(get_arr[0]());
console.log(get_arr[1]());
```

Output:



```
0
1
2
3
```

Explanation: In the above code we are updating the argument of the function create_Closure with every call. Hence, we get different values of i at different index.

What is Pure function:

Given the same input, will always return the same output.

Consider this example:

```
Math.random(); // => 0.4011148700956255
Math.random(); // => 0.8533405303023756
Math.random(); // => 0.3550692005082965
```

Even though we didn’t pass any arguments into any of the function calls, they all produced different output, meaning that *`Math.random()`* is **not pure**.

the following function **is pure**:

```
const highpass = (cutoff, value) => value >= cutoff;
```

The same input values will always map to the same output value:

```
highpass(5, 5); // => true
highpass(5, 5); // => true
highpass(5, 5); // => true
```

Many input values may map to the same output value:

```
highpass(5, 123); // true
highpass(5, 6);   // true
highpass(5, 18);  // true
highpass(5, 1);   // false
highpass(5, 3);   // false
highpass(5, 4);   // false
```

Creating objects in JavaScript

Before we proceed it is important to note that JavaScript is a class-less language and the functions are used in a way so that they simulate a class.

1.**Using functions as class:**

We define a classical JavaScript function and create an object of the function using *new* keyword. The properties and methods of function are created using the *this* keyword.

```
<script>
// Function acting as a Class.
function copyClass(name, age) {
    this.name = name;
    this.age = age;
    this.printInfo = function() {
        console.log(this.name);
        console.log(this.age);
    }
}
// Creating the object of copyClass
// and initializing the parameters.
var obj = new copyClass("Vineet", 20);
// Calling the method of copyClass.
obj.printInfo();
</script>
```

Output:

```
Vineet
20
```

Explanation: A class in OOPs have two major components, certain parameters and few member functions. In this method we declare a function similar to a class, there are two parameters, name and age (the *this* keyword is used to differentiate the name and age of the class to the name and age of the arguments that are being supplied.) and a method *printInfo* that prints the value of these parameters. We then simple create an object obj of the copyClass, initialize it and call it’s method.

2.**Using object literals:**

Literals are smaller and simpler ways to define objects. Below we instantiate an object exactly same as the previous one just with the object literal.

```
<script>
// Creating Object.
var obj = {
    name : "",
    age : "",
    printInfo : function() {
         console.log(this.name);
         console.log(this.age);
     }
}
// Initializing the parameters.
obj.name = "Vineet";
obj.age = 20;

// Using method of the object.
obj.printInfo();
</script>
```

Output:

```
Vineet
20
```

Explanation: This method works same along the line of the previous one but instead of bundling the parameters (name and age) and the method (printInfo) inside of a function, we bundle them in the object itself, initialize the object and simply use the methods.

3. Singleton using a function:

The third way presented is a combination of the other two that we already saw. We can use a function to define a singleton object.

```
<script>
// Creating singleton object.
    var obj = new function() {
        this.name = "";
        this.age = "";
        this.printInfo = function() {
            console.log(this.name);
            console.log(this.age);
        };
    }
// Initializing object.
obj.name = "Vineet";
obj.age = 20;
// Calling method of the object.
obj.printInfo();
</script>
```

Output:

```
Vineet
20
```

Explanation: This is a combination of the previous two methods, we bundle the methods and parameters inside a function but don’t declare a separate function for it (Like copyClass in method 1). Instead we simple use the function structure to declare a object.

Prototype in JavaScript:

Why we need prototype:

Problem with creating objects with the constructor function:

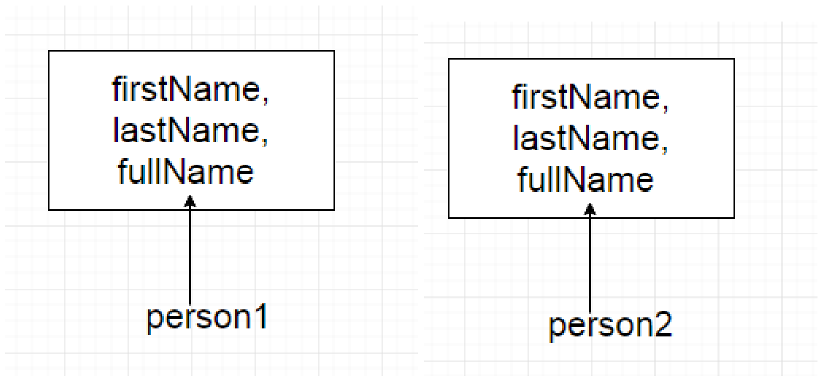
Consider the constructor function below:

```
function Human(firstName, lastName) {
this.firstName = firstName,
this.lastName = lastName,
this.fullName = function() {
return this.firstName + " " + this.lastName;
}
}

var person1 = new Human("Virat", "Kohli");

console.log(person1)
```

Let’s create objects person1 and person2 using the Human constructor function:

```
var person1 = new Human("Virat", "Kohli");
var person2 = new Human("Sachin", "Tendulkar");
```

On executing the above code, the JavaScript engine will create two copies of the constructor function, each for person1 and person2.



i.e. every object created using the constructor function will have its own copy of properties and methods. It doesn't make sense to have two instances of function fullName that do the same thing. Storing separate instances of function for each object results into wastage of memory. We will see as we move forward, how we can solve this issue.

Prototype in JavaScript

JavaScript is a dynamic language. You can attach new properties to an object at any time as shown below.

Example: Attach property to object

```
function Student() {
   this.name = 'John';
   this.gender = 'Male';
}

var studObj1 = new Student();
studObj1.age = 15;
alert(studObj1.age); // 15

var studObj2 = new Student();
alert(studObj2.age); // undefined
```

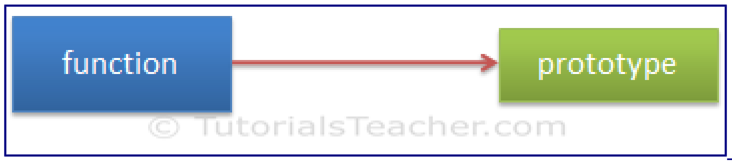
As you can see in the above example, age property is attached to studObj1 instance. However, studObj2 instance will not have age property because it is defined only on studObj1 instance.

So what to do if we want to add new properties at later stage to a function which will be shared across all the instances?

The answer is **Prototype**.

The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.

Every function includes prototype object by default.



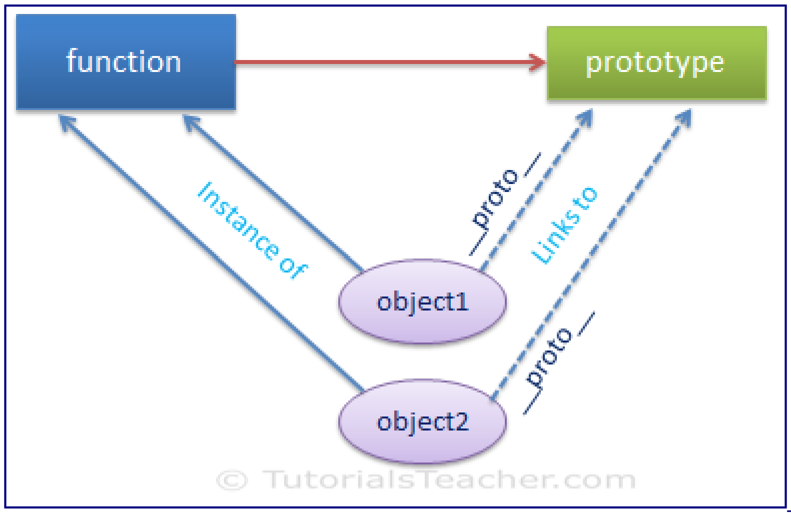
The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.

So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.

Example: prototype

```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

Student.prototype.age = 15;

var studObj1 = new Student();
alert(studObj1.age); // 15

var studObj2 = new Student();
alert(studObj2.age); // 15
```

Every object which is created using literal syntax or constructor syntax with the new keyword, includes __proto__ property that points to prototype object of a function that created this object.



You can debug and see object's or function's prototype property in chrome or firefox's developer tool. Consider the following example.

Example: prototype

```
function Student() {
   this.name = 'John';
   this.gender = 'M';
}

var studObj = new Student();

console.log(Student.prototype); // object
console.log(studObj.prototype); // undefined
console.log(studObj.__proto__); // object

console.log(typeof Student.prototype); // object
console.log(typeof studObj.__proto__); // object

console.log(Student.prototype === studObj.__proto__ ); // true
```

As you can see in the above example, Function's prototype property can be accessed using <function-name>.prototype. However, an object (instance) does not expose prototype property, instead you can access it using __proto__.

Note:
The prototype property is special type of enumerable object which cannot be iterate using for..in or foreach loop.

Object's Prototype

As mentioned before, object's prototype property is invisible. Use Object.getPrototypeOf(obj) method instead of __proto__ to access prototype object.

Example: Object's prototype

```
function Student() {
   this.name = 'John';
   this.gender = 'M';
}

var studObj = new Student();

Student.prototype.sayHi= function(){
   alert("Hi");
};

var studObj1 = new Student();
var proto = Object.getPrototypeOf(studObj1);  // returns Student's prototype object
```

```
alert(proto.constructor); // returns Student function
```

The prototype object includes following properties and methods.

Property	Description
constructor	Returns a function that created instance.
__proto__	This is invisible property of an object. It returns prototype object of a function to which it links to.

Method	Description
hasOwnProperty()	Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain.
isPrototypeOf()	Returns a boolean indication whether the specified object is in the prototype chain of the object this method is called upon.
propertyIsEnumerable()	Returns a boolean that indicates whether the specified property is enumerable or not.
toLocaleString()	Returns string in local format.
toString()	Returns string.
valueOf	Returns the primitive value of the specified object.

Chrome and Firefox denotes object's prototype as __proto__ which is public link whereas internally it reference as [[Prototype]]. Internet Explorer does not include __proto__. Only IE 11 includes it.

The getPrototypeOf() method is standardize since ECMAScript 5 and is available since IE 9.

Changing Prototype

As mentioned above, each object's prototype is linked to function's prototype object. If you change function's prototype then only new objects will be linked to changed prototype. All other existing objects will still link to old prototype of function. The following example demonstrates this scenario.

Example: Changing Prototype

```
function Student() {
   this.name = 'John';
   this.gender = 'M';
}

Student.prototype.age = 15;

var studObj1 = new Student();
alert('studObj1.age = ' + studObj1.age); // 15

var studObj2 = new Student();
alert('studObj2.age = ' + studObj2.age); // 15

Student.prototype = { age : 20 };
```

```
var studObj3 = new Student();
alert('studObj3.age = ' + studObj3.age); // 20

alert('studObj1.age = ' + studObj1.age); // 15
alert('studObj2.age = ' + studObj2.age); // 15
```

Use of Prototype

The prototype object is being used by JavaScript engine in two things, 1) to find properties and methods of an object 2) to implement inheritance in JavaScript.

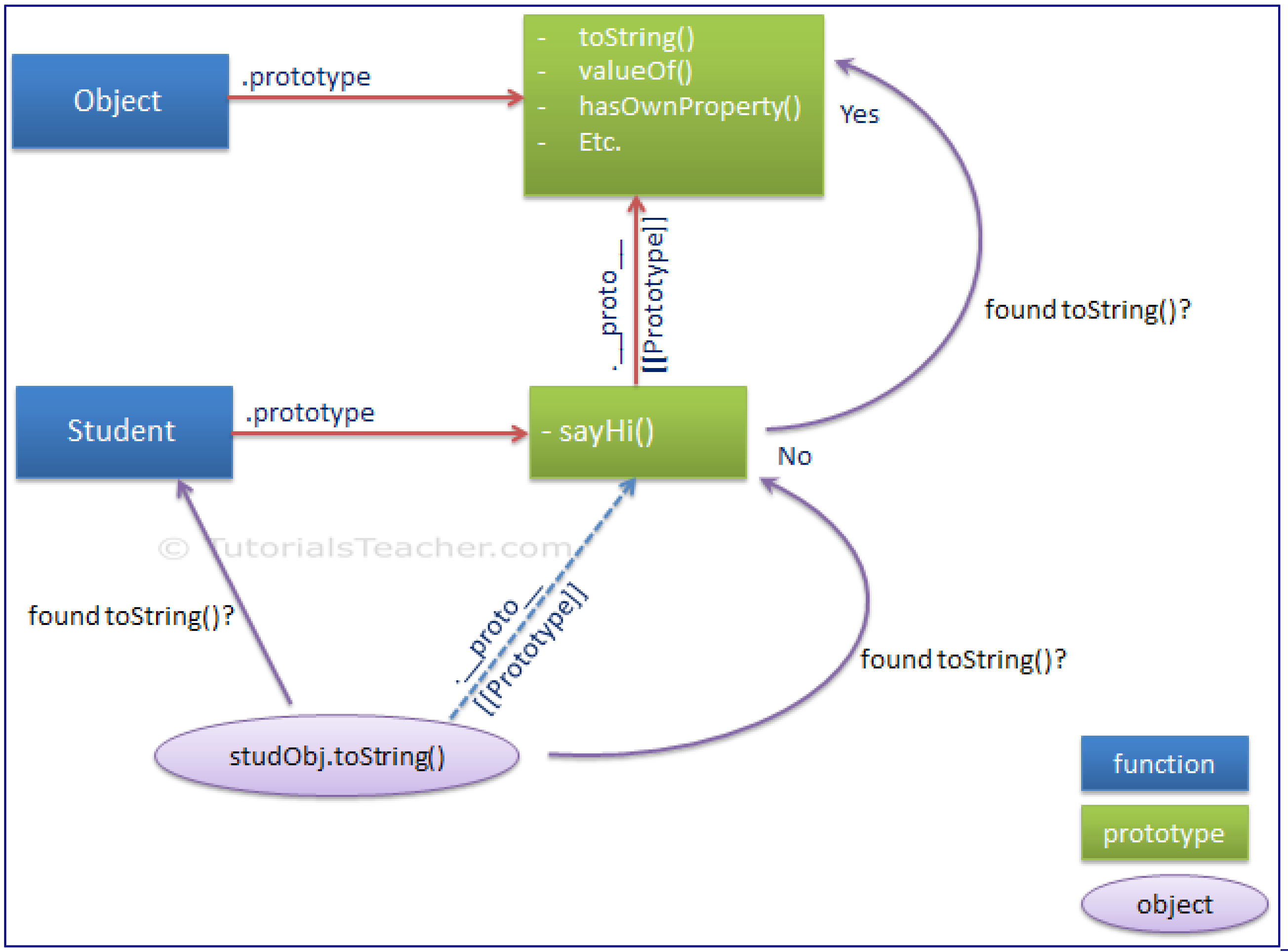
```
function Student() {
    this.name = 'John';
    this.gender = 'M';
}

Student.prototype.sayHi = function(){
    alert("Hi");
};

var studObj = new Student();
studObj.toString();
```

In the above example, toString() method is not defined in Student, so how and from where it finds toString()?

Here, prototype comes into picture. First of all, JavaScript engine checks whether toString() method is attached to studObj? (It is possible to attach a new function to a instance in JavaScript). If it does not find there then it uses studObj's **__proto__** link which points to the prototype object of Student function. If it still cannot find it there then it goes up in the heirarchy and check prototype object of Object function because all the objects are derived from Object in JavaScript, and look for toString() method. Thus, it finds toString() method in the prototype object of Object function and so we can call studObj.toString().

This way, prototype is useful in keeping only one copy of functions for all the objects (instances).

The following figure illustrates the above scenario.



Overriding

t is true that JavaScript supports overriding, not overloading. When you define multiple functions which have the same name, the last one defined will override all the previously defined ones and every time when you invoke a function, the last defined one will get executed.

```
<script type="text/javascript">
function multiplyNum(x,y,z){
    return x* y * z;
}

function multiplyNum(x, y) {
    return x * y;
}
var result = multiplyNum(1, 2, 3);
document.write(result);
```

Output

```
2
```

Looking at the above example, the value of multiplication will be equal to 2 instead of 6.

In this case, the only function available is multiplyNum(x, y). So if we think we are making a call to multiplyNum(x, y, z), it's actually calling multiplyNum(x, y). The remaining parameter(s) will be ignored.

We can also override Javascript built-in functions. The following example overrides the built-in JavaScript alert() function.

```
<script type="text/javascript">
      var alert = function(message) {
         document.write(message);
      }
      // The following calls will invoke the overridden alert() function
      alert("Learn ");
      alert("JavaScript");
</script>
```

Output

```
Learn JavaScript
```

By default, alert() function displays the message in the alert box. But here we have overridden it. Now it is displaying the message in the document.

```
function getEmployeeDetails(){
employee.prototype.getName=function(){
return “Mr. ”+this.name.toUpperCase();
}

var e1=new employee("Rajeev");
var e2=new employee("Rishi");

console.log("e1 name= "+e1.getName());
console.log("e2 name= "+e2.getName());
}

getEmployeeDetails();
```

```
function employee(name){
this.name=name;
}

employee.prototype.getName=function(){
return this.name;
}

// var e11=new employee("Rishi");
// console.log(e11.getName());

// var e11=new employee("Rajeev");
// console.log(e11.getName());
```

Prototypal Inheritance

Unlike most other languages, JavaScript's object system is based on **prototypes, not classes**.

What's the Difference Between Class & Prototypal Inheritance?

Class Inheritance: ***A class is like a blueprint — a description of the object to be created.*** Classes inherit from classes and **create subclass relationships**: hierarchical class taxonomies.

Instances are typically instantiated via constructor functions with the *`new`* keyword. Class inheritance may or may not use the *`class`* keyword from ES6. Classes as you may know them from languages like Java don't technically exist in JavaScript. Constructor functions are used, instead. The ES6 *`class`* keyword desugars to a constructor function:

```
class Foo {}
typeof Foo // 'function'
```

In JavaScript, class inheritance is implemented on top of prototypal inheritance, but *that does not mean that it does the same thing:*

JavaScript's class inheritance uses the prototype chain to wire the child *`Constructor.prototype`* to the parent *`Constructor.prototype`* for delegation. Usually, the *`super()`* constructor is also called. Those steps form **single-ancestor parent/child hierarchies** and ***create the tightest coupling available in OO design.***

> "Classes inherit from classes and **create subclass relationships**: hierarchical class taxonomies."

Prototypal Inheritance:***A prototype is a working object instance.***

Instances may be composed from many different source objects, allowing for easy selective inheritance and a flat [[Prototype]] delegation hierarchy. In other words, **class taxonomies are not an automatic side-effect of prototypal OO**: *a critical distinction.*

Instances are typically instantiated via factory functions, object literals, or *`Object.create()`*.

> ***"A prototype is a working object instance.*** Objects inherit directly from other objects."

Inheritance is an important concept in object oriented programming. In the classical inheritance, methods from base class get copied into derived class.

In JavaScript, inheritance is supported by using prototype object. Some people call it "Prototypal Inheriatance" and some people call it "Behaviour Delegation".

Let's see how we can achieve inheritance like functionality in JavaScript using prototype object.

Let's start with the Person class which includes FirstName & LastName property as shown below.

```
function Person(firstName, lastName) {
    this.FirstName = firstName || "unknown";
    this.LastName = lastName || "unknown";
};

Person.prototype.getFullName = function () {
    return this.FirstName + " " + this.LastName;
}
```

In the above example, we have defined Person class (function) with FirstName & LastName properties and also added getFullName method to its prototype object.

Now, we want to create Student class that inherits from Person class so that we don't have to redefine FirstName, LastName and getFullName() method in Student class. The following is a Student class that inherits Person class.

Example: Inheritance

```
function Student(firstName, lastName, schoolName, grade)
{
    Person.call(this, firstName, lastName);

    this.SchoolName = schoolName || "unknown";
    this.Grade = grade || 0;
}
//Student.prototype = Person.prototype;
Student.prototype = new Person();
Student.prototype.constructor = Student;
```

Please note that we have set Student.prototype to newly created person object. The new keyword creates an object of Person class and also assigns Person.prototype to new object's prototype object and then finally assigns newly created object to Student.prototype object. Optionally, you can also assign Person.prototype to Student.prototype object.

Now, we can create an object of Student that uses properties and methods of the Person as shown below.

Example: Inheritance

```
function Person(firstName, lastName) {
    this.FirstName = firstName || "unknown";
    this.LastName = lastName || "unknown";
}

Person.prototype.getFullName = function () {
    return this.FirstName + " " + this.LastName;
}
```

```
function Student(firstName, lastName, schoolName, grade)
{
    Person.call(this, firstName, lastName);

    this.SchoolName = schoolName || "unknown";
    this.Grade = grade || 0;
}
//Student.prototype = Person.prototype;
Student.prototype = new Person();
Student.prototype.constructor = Student;

var std = new Student("James","Bond", "XYZ", 10);

alert(std.getFullName()); // James Bond
alert(std instanceof Student); // true
alert(std instanceof Person); // true
```

Immediately Invoked Function Expression - IIFE

As you know that a function in JavaScript creates the local scope. So, you can define variables and function inside a function which cannot be access outside of that function. However, sometime you accidently pollute the global variables or functions by unknowingly giving same name to variables & functions as global variable & function names. For example, there are multiple .js files in your application written by multiple developers over a period of time. Single JavaScript file includes many functions and so these multiple .js files will result in large number of functions. There is a good chance of having same name of function exists in different .js files written by multiple developer and if these files included in a single web page then it will pollute the global scope by having two or more function or variables with the same name. Consider following example of two different JavaScript file included in single page.

Consider the following example of MyScript1.js and MyScript2.js with same variable & function name.

Example: MyScript1.js

```
var userName = "Bill";

function display(name)
{
    alert("MyScript1.js: " + name);
}

display(userName);
```

Example: MyScript2.js

```
var userName = "Steve";

function display(name)
{
    alert("MyScript2.js: " + name);
}

display(userName);
```

Now, if you include these JS files in your web page then guess what will happen?

Example: Script tag into <head> tag::

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>JavaScript Demo</title>
    <script src="/MyScript1.js"></<script>
    <script src="/MyScript2.js"></<script>
</head>
<body>
    <h1> IIFE Demo</h1>
</body>
</html>
```

If you run above example, you will find that every time it call display() function in MyScript2.js because MyScript2.js included after MyScript1.js in a web page. So JavaScript considers last definition of a function if two functions have the same name.

IEFE solves this problem by having its own scope and restricting functions and variables to become global. The functions and variables declare inside IIFE will not pollute global scope even they have same name as global variables & functions. So let's see what is an IIFE is.

What is an IIFE?

As name suggest, IIFE is a function expression that automatically invokes after completion of the definition. The parenthesis () plays important role in IIFE pattern. In JavaScript, parenthesis cannot contain statements; it can only contain an expression.

Example: Parenthesis ()

```
(var foo = 10 > 9); // syntax error
(var foo = "foo", bar = "bar"); // syntax error
(10 > 9); // valid
(alert("Hi")); // valid
```

First of all, define a function expression.

Example: IIFE

```
var myIIFE = function () {
        //write your js code here
    };
```

Now, wrap it with parenthesis. However, parenthesis does not allow declaration. So just remove declaration part and just write anonymous function as below.

Example: IIFE

```
(function () {
        //write your js code here
    });
```

Now, use () operator to call this anonymous function immediately after completion of its definition.

Example: IIFE

```
(function () {
    //write your js code here
})();
```

So, the above is called IIFE. You can write all the functions and variables inside IIFE without worrying about polluting the global scope or conflict with other's JavaScript code which have functions or variables with same name.

To solve the our above problem, wrap all the code in MyScript1.js & MyScript2.js file in IIFE as shown below.

Example: IIFE

```
(function () {
    var userName = "Steve";

    function display(name)
    {
        alert("MyScript2.js: " + name);
    }

    display(userName);
  })();
```

So, even if MyScript1.js & MyScript2.js file includes functions and variables with the same name, they won't conflict with each other and pollute the global scope.

Also, you can pass arguments in IIFE as shown below.

Example: IIFE

```
var userName = "Bill";

(function (name) {

    function display(name)
    {
        alert("MyScript2.js: " + name);
    }

    display(name);
})(userName);
```

Advantages of IIFE:

1. Do not create unnecessary global variables and functions
2. Functions and variables defined in IIFE do not conflict with other functions & variables even if they have same name.
3. Organize JavaScript code.
4. Make JavaScript code maintainable.

Error handling in JavaScript

JavaScript is a loosely-typed language. It does not give compile time. So some times you will get a runtime error for accessing an undefined variable or calling undefined function etc.

Note: try catch block does not handle syntax errors.

JavaScript provides error-handling mechanism to catch runtime errors using try-catch-finally block

Syntax:

```
try
{
    // code that may throw an error
}
```

```
catch(ex)
{
   // code to be executed if an error occurs
}
finally{
   // code to be executed regardless of an error occurs or not
}
```

- **try:** wrap suspicious code that may throw an error in try block.
- **catch:** write code to do something in catch block when an error occurs. The catch block can have parameters that will give you error information. Generally catch block is used to log an error or display specific messages to the user.
- **finally:** code in the finally block will always be executed regardless of the occurrence of an error. The finally block can be used to complete the remaining task or reset variables that might have changed before error occurred in try block.

Let's look at simple error handling examples.

Example: Error Handling in JS

```
try
{
   var result  =  Sum(10, 20); // Sum is not defined yet
}
catch(ex)
{
   document.getElementById("errorMessage").innerHTML = ex;
}
```

In the above example, we are calling function Sum, which is not defined yet. So, try block will throw an error which will be handled by catch block. Ex includes error message that can be displayed.

The finally block executes regardless of whatever happens.

Example: finally Block

```
try
{
    var result  =  Sum(10, 20); // Sum is not defined yet
}
catch(ex)
{
   document.getElementById("errorMessage").innerHTML = ex;
}
finally{
   document.getElementById("message").innerHTML = "finally block executed";
}
```

Use throw keyword to raise a custom error.

Example: throw Error

```
try
{
    throw "Error occurred";
}
catch(ex)
{
    alert(ex);
}
```

You can use JavaScript object for more information about an error.

Example: throw error with error info

```
try
{
    throw {
        number: 101,
        message: "Error occurred"
    };
}
catch (ex) {
    alert(ex.number + "- " + ex.message);
}
```

JavaScript array methods

1. forEach()

This method can help you to loop over array's items.

```
  const arr = [1, 2, 3, 4, 5, 6];

  arr.forEach(item => {

    console.log(item); // output: 1 2 3 4 5 6

  });
```

2. includes()

This method check if array includes the item passed in the method.

```
  const arr = [1, 2, 3, 4, 5, 6];

  arr.includes(2); // output: true
  arr.includes(7); // output: false
```

3. filter()

This method create new array with only elements passed condition inside the provided function.

```
  const arr = [1, 2, 3, 4, 5, 6];

  // item(s) greater than 3
  const filtered = arr.filter(num => num > 3);
  console.log(filtered); // output: [4, 5, 6]

  console.log(arr); // output: [1, 2, 3, 4, 5, 6]
```

4. map()

This method create new array by calling the provided function in every element.

```
  const arr = [1, 2, 3, 4, 5, 6];

  // add one to every element
  const oneAdded = arr.map(num => num + 1);
  console.log(oneAdded); // output [2, 3, 4, 5, 6, 7]

  console.log(arr); // output: [1, 2, 3, 4, 5, 6]
```

5. reduce()

The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value

```
  const arr = [1, 2, 3, 4, 5, 6];

  const sum = arr.reduce((total, value) => total + value, 0);
  console.log(sum); // 21
```

6. some()

This method check if at least one of array's item passed the condition. If passed, it return 'true' otherwise 'false'.

```
  const arr = [1, 2, 3, 4, 5, 6];
  // at least one element is greater than 4?

  const largeNum = arr.some(num => num > 4);

  console.log(largeNum); // output: true


  // at least one element is less than or equal to 0?

  const smallNum = arr.some(num => num <= 0);

  console.log(smallNum); // output: false
```

7. every()

This method check if all array's item passed the condition. If passed, it return 'true' otherwise 'false'.

```
  const arr = [1, 2, 3, 4, 5, 6];


  // all elements are greater than 4

  const greaterFour = arr.every(num => num > 4);

  console.log(greaterFour); // output: false


  // all elements are less than 10

  const lessTen = arr.every(num => num < 10);

  console.log(lessTen); // output: true
```

8. Sort():

We can sort the strings in JavaScript by following methods described below:

- **Using sort() method**
- **Using loop**

Using sort() method: In this method, we use predefined sort() method of JavaScript to sort the array of string. This method is used only when the string is alphabetic. It will produce wrong results if we store numbers in an array and apply this method.

> **Original String**
> Suraj, Sanjeev, Rajnish, Yash, Ravi
> **After sorting**
> Rajnish, Ravi, Sanjeev, Suraj, Yash
>
> **Original String**
> 40, 100, 1, 5, 25, 10
> **After sorting**
> 1, 10, 100, 25, 40, 5

Below program illustrates the above approach:

```
<script>

// JavaScript code to sort strings

// Original string

var string = ["Suraj", "Sanjeev", "Rajnish", "Yash", "Ravi"];

// Print original string array

document.write("Original String</br>");

document.write(string);

document.write("</br>");

// Use sort() method to sort the strings

string.sort();

document.write("</br>After sorting</br>");



// Print sorted string array

document.write(string);
```

```
</script>
```

Output:

```
Original String

Suraj, Sanjeev, Rajnish, Yash, Ravi

After sorting

Rajnish, Ravi, Sanjeev, Suraj, Yash
```

Using loop: We will use a simple approach of sorting to sort the strings. In this method, we will use a loop and then compare each element and put the string at its correct position. Here we can store numbers in an array and apply this method to sort the array.

Examples:

```
   Original String

   Suraj,Sanjeev,Rajnish,Yash,Ravi

   After sorting

   Rajnish,Ravi,Sanjeev,Suraj,Yash

   Original String

   40, 100, 1, 5, 25, 10

   After sorting

   1,5,10,25,40,100
```

Below program illustrates the above approach:

```
<script>

// JavaScript code to sort the strings

// Function to perform sort

function string_sort(str) {

   var i = 0, j;

while (i < str.length) {

      j = i + 1;
```

```
            while (j < str.length) {
                if (str[j] < str[i]) {
                    var temp = str[i];
                    str[i] = str[j];
                    str[j] = temp;
                }
                j++;
            }
            i++;
        }
    }

// Driver code

// Original string
var string = ["Suraj", "Sanjeev", "Rajnish", "Yash", "Ravi"];

// Print original string array
document.write("Original String</br>");
document.write(string);
document.write("</br>");
// Call string_sort method
string_sort(string);
document.write("</br>After sorting</br>");
// Print sorted string array
document.write(string);

</script>
```

Output:

Original String
Suraj, Sanjeev, Rajnish, Yash, Ravi

After sorting
Rajnish, Ravi, Sanjeev, Suraj, Yash

Another way:

```
const arr = [1, 2, 3, 4, 5, 6];
const alpha = ['e', 'a', 'c', 'u', 'y'];

// sort in descending order
descOrder = arr.sort((a, b) => a > b ? -1 : 1);
console.log(descOrder); // output: [6, 5, 4, 3, 2, 1]

// sort in ascending order
ascOrder = alpha.sort((a, b) => a > b ? 1 : -1);
console.log(ascOrder); // output: ['a', 'c', 'e', 'u', 'y']
```

9. Array.from()

This change all thing that are array-like or iterable into true array especially when working with DOM, so that you can use other array methods like reduce, map, filter and so on.

```
const name = 'frugence';
const nameArray = Array.from(name);

console.log(name); // output: frugence
console.log(nameArray); // output: ['f', 'r', 'u', 'g', 'e', 'n', 'c', 'e']
```

10. Array.of()

This create array from every arguments passed into it.

```
const nums = Array.of(1, 2, 3, 4, 5, 6);
console.log(nums); // output: [1, 2, 3, 4, 5, 6]
```

11. toString():

toString() converts an array to a string of (comma separated) array values.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Result:

```
Banana,Orange,Apple,Mango
```

12. Join():

The join() method also joins all array elements into a string.

It behaves just like toString(), but in addition you can specify the separator:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
```

Result:

```
Banana * Orange * Apple * Mango
```

13. pop():

The pop() method removes the last element from an array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();// Removes the last element ("Mango") from fruits
```

The pop() method returns the value that was "popped out":

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.pop();// the value of x is "Mango"
```

14. push():

The push() method adds a new element to an array (at the end):

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");// Adds a new element ("Kiwi") to fruits
```

The push() method returns the new array length:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.push("Kiwi");   //  the value of x is 5
```

15. shift():

Shifting is equivalent to popping, working on the first element instead of the last.

The shift() method removes the first array element and "shifts" all other elements to a lower index.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();            // Removes the first element "Banana" from fruits
```

The shift() method returns the string that was "shifted out":

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.shift();    // the value of x is "Banana"
```

16. unshift():

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");    // Adds a new element "Lemon" to fruits
```

The unshift() method returns the new array length.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");    // Returns 5
```

17. Delete:

Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator delete:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];           // Changes the first element in fruits to undefined
```

Note: Using **delete** may leave undefined holes in the array. Use pop() or shift() instead.

18.splice():

The splice() method can be used to add new items to an array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

The splice() method returns an array with the deleted items:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 2, "Lemon", "Kiwi");
```

Using splice() to Remove Elements

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);        // Removes the first element of fruits
```

The first parameter (0) defines the position where new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

19. slice():

The slice() method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(1);
```

The slice() method creates a new array. It does not remove any elements from the source array.

This example slices out a part of an array starting from array element 3 ("Apple"):

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(3);
```

The slice() method can take two arguments like slice(1, 3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(1, 3);
```

If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(2);
```

ES6 Javascript

1. Default Parameters in ES6

Remember we had to do these statements to define default parameters:

```
var link = function (height, color, url) {
    var height = height || 50
    var color = color || 'red'
    var url = url || 'http://azat.co'
    ...
}
```

In ES6, we can put the default values right in the signature of the functions:

```
var link = function(height = 50, color = 'red', url = 'http://azat.co') {
  ...
}
```

2. Template Literals in ES6

Template literals or interpolation in other languages is a way to output variables in the string. So in ES5 we had to break the string like this:

```
var name = 'Your name is ' + first + ' ' + last + '.'
var url = 'http://localhost:3000/api/messages/' + id
```

Luckily, in ES6 we can use a new syntax ${NAME} inside of the back-ticked string:

```
var name = `Your name is ${first} ${last}.`
var url = `http://localhost:3000/api/messages/${id}`
```

3. Multi-line Strings in ES6

Another yummy syntactic sugar is multi-line string. In ES5, we had to use one of these approaches:

```
var roadPoem = 'Then took the other, as just as fair,\n\t'
    + 'And having perhaps the better claim\n\t'
    + 'Because it was grassy and wanted wear,\n\t'
    + 'Though as for that the passing there\n\t'
    + 'Had worn them really about the same,\n\t'

var fourAgreements = 'You have the right to be you.\n\
    You can only be you when you do your best.'
```

While in ES6, simply utilize the backticks:

```
var roadPoem = `Then took the other, as just as fair,
    And having perhaps the better claim
    Because it was grassy and wanted wear,
    Though as for that the passing there
    Had worn them really about the same,`

var fourAgreements = `You have the right to be you.
    You can only be you when you do your best.`
```

4. **const and let**

const (Constants) are block scoped (only available in local scope). You can’t reassign value of const variable but properties of const can be mutated for example.

```
const a = {namv : "Jhon Doe"}

a = {  // this not allowed

name : "Jen Doe"

}

a.name = "Jen Doe" // this allowed
```

The let statement allows you to declare a variable with block scope.

```
var x = 10;
// Here x is 10
{
let x = 2;
 // Here x is 2
}
// Here x is 10
```

5. Arrow Function:
Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

```
// ES5
var x = function(x, y) {
return x * y;
}
```

```
// ES6
const x = (x, y) => x * y;
```

Arrow functions do not have their own this. They are not well suited for defining object methods.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

const x = (x, y) => { return x * y };

6. Classes:

ES6 introduced classes.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

Use the keyword class to create a class, and always add a constructor method.

The constructor method is called each time the class object is initialized.

A simple class definition for a class named "Car":

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
}
```

Now you can create objects using the Car class:
Create an object called "mycar" based on the Car class:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
}
mycar = new Car("Ford");
```

7. Import and export

Import export are more important where modularity is essential. Usual syntax for import is import module from 'path/to/module'. Similarly, you can also export any module. let’s take an example for more clarification. Suppose there is two files one is sum.js containing method that return sum of two numbers and other is index.js where we want to sum two numbers.

```
---------------------sum.js---------------------------------

export default function sum(a,b){
 return a+b

}

--------------------------index.js--------------------------------

import sum from './sum' // no need to provide format of file

console.log(sum(4,5))

--------------------------console---------------------------------

>>9
```

Higher-Order Functions

A function that accepts and/or returns another function is called a **higher-order function**.

It’s *higher-order* because instead of strings, numbers, or booleans, it goes *higher* to operate on functions.

A Higher-order function is a function that may receive a function as an argument and can even return a function. Higher-order functions are just like regular functions with an added ability of receiving and returning other functions are arguments and output.

Functions Operate on Data

Strings Are Data

```
sayHi = (name) => `Hi, ${name}!`;
result = sayHi('User');

console.log(result); // 'Hi, User!'
```

Numbers Are Data

```
double = (x) => x * 2;
result = double(4);

console.log(result); // 8
```

Booleans Are Data

```
getClearance = (allowed) => (allowed ? 'Access granted' : 'Access denied');

result1 = getClearance(true);
result2 = getClearance(false);
```

```
console.log(result1); // 'Access granted'
console.log(result2); // 'Access denied'
```

Objects Are Data

```
getFirstName = (obj) => obj.firstName;

result = getFirstName({
  firstName: 'Yazeed'
});

console.log(result); // 'Yazeed'
```

Arrays Are Data

```
len = (array) => array.length;
result = len([1, 2, 3]);

console.log(result); // 3
```

These 5 types are first-class-citizens in every mainstream language.

What makes them first-class? You can pass them around, store them in variables and arrays, use them as inputs for calculations. You can use them like *any piece of data*.

Functions Can Be Data Too:

4 ways function can be data in javascript:

1. Pass them to other function.

2. Set them as object property.

3. Store them in array.

4. Set them as variable.

Functions as Arguments

```
isEven = (num) => num % 2 === 0;
result = [1, 2, 3, 4].filter(isEven);

console.log(result); // [2, 4]
```

See how filter uses isEven to decide what numbers to keep? isEven, *a function*, was a parameter *to another function*.

It’s called by filter for each number, and uses the returned value true or false to determine if a number should be kept or discarded.

Returning Functions

```
add = (x) => (y) => x + y;
```

add requires two parameters, but not all at once. It’s a function asking for just x, that returns a function asking for just y.

Again, this is only possible because JavaScript allows functions to be a return value—just like strings, numbers, booleans, etc.

You can still supply x and y immediately, if you wish, with a **double invocation**

```
result = add(10)(20);

console.log(result); // 30
```

Or x now and y later:

```
add10 = add(10);
result = add10(20);

console.log(result); // 30
```

Greater Reusability

Probably the greatest benefit of HOFs is greater reusability. Without them, JavaScript’s premiere Array methods—map, filter, and reduce—wouldn’t exist!

Here’s a list of users. We’re going to do some calculations with their information.

```
users = [
  {
    name: 'Yazeed',
    age: 25
  },
  {
    name: 'Sam',
    age: 30
  },
  {
    name: 'Bill',
    age: 20
  }
];
```

Map

Without higher-order functions, we’d always need loops to mimic map's functionality.

```
getName = (user) => user.name;
usernames = [];

for (let i = 0; i < users.length; i++) {
  const name = getName(users[i]);

  usernames.push(name);
}

console.log(usernames);
// ["Yazeed", "Sam", "Bill"]
```

Or we could do this!

```
usernames = users.map(getName);

console.log(usernames);
// ["Yazeed", "Sam", "Bill"]
```

Filter

In a HOF-less world, we’d still need loops to recreate filter's functionality too.

```
startsWithB = (string) => string.toLowerCase().startsWith('b');

namesStartingWithB = [];

for (let i = 0; i < users.length; i++) {
  if (startsWithB(users[i].name)) {
    namesStartingWithB.push(users[i]);
  }
}

console.log(namesStartingWithB);
// [{ "name": "Bill", "age": 20 }]
```

Or we could do this!

```
namesStartingWithB = users.filter((user) => startsWithB(user.name));

console.log(namesStartingWithB);
// [{ "name": "Bill", "age": 20 }]
```

Reduce

Yup, reduce too… Can’t do much cool stuff without higher-order functions!! ?

```
total = 0;

for (let i = 0; i < users.length; i++) {
  total += users[i].age;
}

console.log(total);
// 75
```

How’s this?

```
totalAge = users.reduce((total, user) => user.age + total, 0);

console.log(totalAge);
// 75
```

- Strings, numbers, bools, arrays, and objects can be stored as variables, arrays, and properties or methods.
- JavaScript treats functions the same way.
- This allows for functions that operate on other functions: **higher-order functions**.
- Map, filter, and reduce are prime examples—and make common patterns like transforming, searching, and summing lists much easier!

Currying:

Currying is a technique of evaluating function with multiple arguments, into sequence of function with single argument.

In other words, when a function, instead of taking all arguments at one time, takes the first one and return a new function that takes the second one and returns a new function which takes the third one, and so forth, until all arguments have been fulfilled.

That is, when we turn a function call add(1,2,3) into add(1)(2)(3) . By using this technique, the little piece can be configured and reused with ease.

Why it’s useful ?

- Currying helps you to avoid passing the same variable again and again.
- It helps to create a higher order function. It extremely helpful in event handling.
- Little pieces can be configured and reused with ease.

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

You can call it with too few (with odd results), or too many (excess arguments get ignored).

```
add(1,2) --> 3
add(1) --> NaN
add(1,2,3) --> 3 //Extra parameters will be ignored.
```

How to convert an existing function to curried version?

The curry function does not exist in native JavaScript. But library like lodash makes it easier to convert a function to curried one.

```
function curry(f) { // curry(f) does the currying transform
        return function(a) {
                return function(b) {
                                return f(a, b);
                };
        };
}

// usage
function sum(a, b) {
        return a + b;
}

let curriedSum = curry(sum);
console.log(curriedSum(1)(2));//3
console.log(curriedSum(4)(7));//11
```

How does currying work?

Currying works by natural closure.The closure created by the nested functions to retain access to each of the arguments.So inner function have access to all arguments.

> Note: We can achieve the same behavior using bind.The problem here is we have to alter this binding.

```
//same thing we can achieve by bind method
var addBy2 = sum.bind(this,1);
console.log(addBy2(2));//3

var addBy2 = sum.bind(this,4);
console.log(addBy2(7));//11
```