This is related to CoderDost Course on Advance JavaScript

1. Scope and Closure

- We have 3 types of variable in JavaScript ***var***, ***let*** and ***const***
- ☠️ ☠️ *var* is the old one, and should not be used now in any case. As it has many issues with creating scopes
    - why it is still there ?
- Also there are 4 kinds of scope in Javascript - *Block Scope*, *Global Scope*, *Function Scope*, *Module Scope*

Block scope & Global Scope

The **scope** is the current context of execution in which *values* and expressions are "visible" MDN

Global Scope : Any variable/expression which is written outside - i.e. not inside any functions, blocks etc. This is shared across files.

let

- this creates a *block scope*
- *re-declaration* in NOT allowed (in same scope)
- *re-assignment* is allowed

```
{ // block scope
  let x = 0;
  let y = 0;
  console.log(x); // 0
  let x = 1; // Error
}

{
  let x = 1;
  let y = 1;
  x = 2;
  console.log(x);// 2
}

console.log(x); // Error in Global Scope
```

Temporal Dead Zone(TDZ) : the area in which a variable is not accessible. Temporal because it depends on time of excution not position

```
{
  // TDZ starts
  const say = () => console.log(msg); // hi

  let msg = 'hi';
  say();
}
```

const

- this creates a *block scope*
- *re-declaration* in NOT allowed
- *re-assignment* is NOT allowed
- must be assigned at declaration time.

```
{
  const x; //Error
  const y=0;
}

{
  const x=1;
  x=2   // Error
}

console.log(x); // Error
```

Variable Shadowing

```
let x  = 0 // shadowed variable
{
  let x = 1;
  console.log(x)
}

}
```

var

- it doesn't have any block scope, and can be *re-declared*
- it only had *function scope*

- *var* are *hoisted*, so they can be used before the declaration

```
var x = 1;
var x = 2; // valid

console.log(y) // valid
var y = 3

z=4
console.log(z) // valid
var z;
```

NOTE : You should NOT use **var** now ❌

let vs var

```
for(let i=0;i<5;i++){
  setTimeout(
    ()=>console.log(i),
    1000)
} // prints 0,1,2,3,4

for(var i=0;i<5;i++){
  setTimeout(
    ()=>console.log(i),
    1000)
} // prints 5,5,5,5,5
```

Module scope

In modern javascript, a file can be considered as module, where we use *export* and *import* syntax to use variable across files. We

```
<script src="index.js" type="module"></script>
```

```
export { someVar, someFunc}
```

```
import { someVar} from './app.js'
```

global Object

- The global Object is the variable **window** in case of browser. This helps you to use variables across the scopes. Also, it is the **this** value for global functions
  - window.alert
  - window.Promise
- In non-browser environment, **window** doesn't exist. but other global objects exist.
- ***var*** affects this global obejct, also ***function*** declarations.

```
function sayHi(){
   console.log(this) // this will refer to window
}


// Strict mode can change this behaviour;
`use strict`

function sayHi(){
   console.log(window) // this is a better way of code
}
```

function scope

- it is created upon execution a function

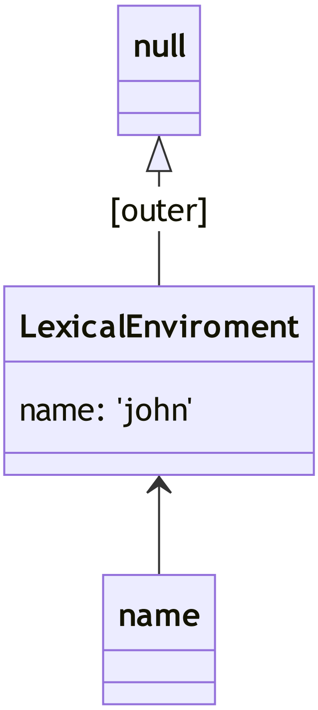
```
function sayHi(name){
    return name;
}

sayHi() // this call will create a function scope

sayHi() // this call will create another function scope
```

Lexical Environment

- Every variable in JavaScript (within global / block / or function) has a reference to an object-like data called *Lexical enviroment*. This object (kind of object) serves as the basis of search for value of variable.

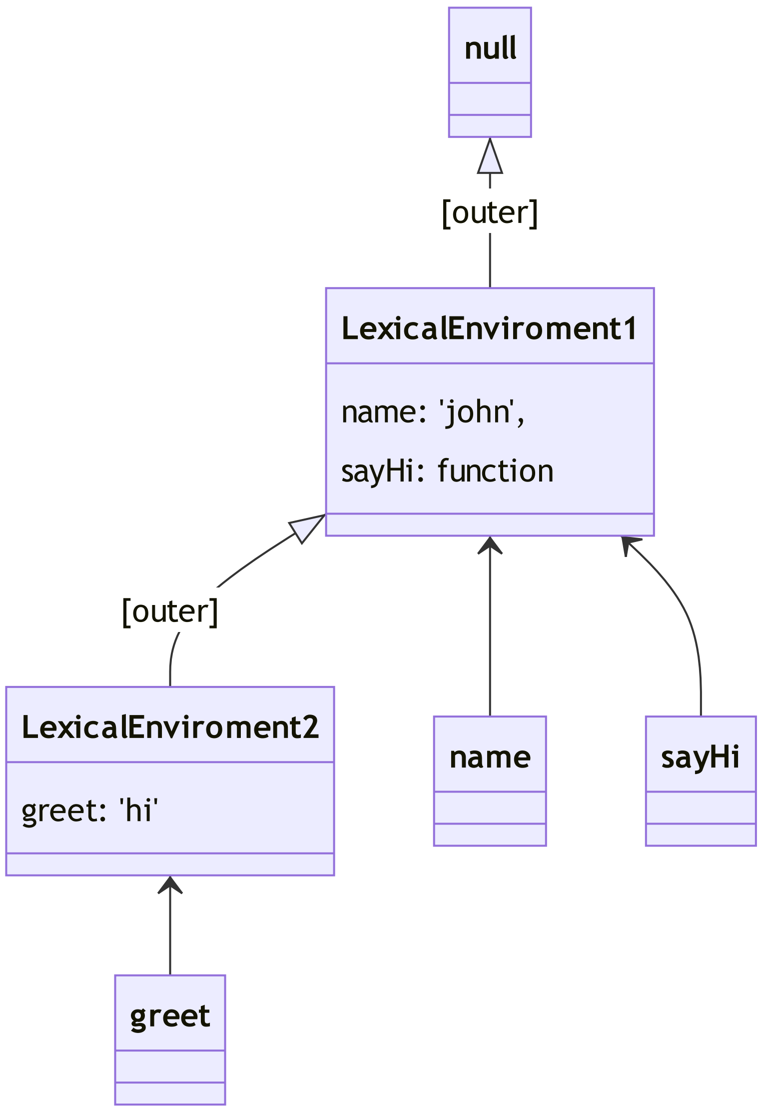
```
let name = 'john'
console.log(name)
```

Lexical Enviroment (Global variable)



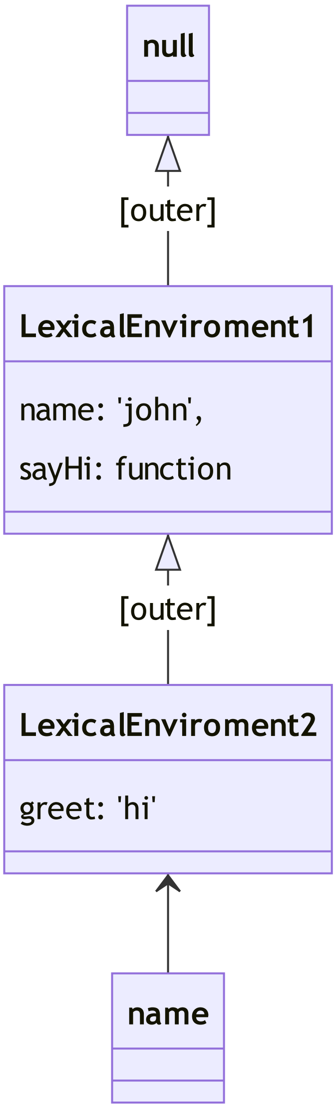

```javascript
let name = 'john';

function sayHi(){
  let greet = "hi"
  console.log(greet)
}

sayHi()
console.log(name, sayHi)
```

Lexical Enviroment (functions)




```
let name = 'john';

function sayHi(){
  let greet = "hi"
  console.log(name)
}

sayHi()
```

Lexical Enviroment (functions)



Hoisting

The movement of *variable declaration* to top of scope - before execution

- *function declarations* are properly hoisted (value accessible)
- *var* is hoisted.

```
let name = 'john';

sayHi() // valid

function sayHi(){
  let greet = "hi"
  console.log(name)
}

sayHello() // error
```

```
let sayHello = function(){
   console.log(name)
}
```

Temporal Dead Zone(TDZ) :

```
let x = 1;

{
  console.log(x) // Reference error
  let x = 2;
}
```

Closures

- we can create nested functions in JavaScript

```
function createUser(name){
  let greeting = 'Hi '
  function greet(){
     return greeting + name + ' is Created';
  }
  return greet()
}

createUser('john') // Hi john is created;
```

- Now more useful work is if we can return the `greet` function itself.

```
function createUser(name){
  let greeting = 'Hi '
  function greet(){
     return greeting + name + ' is Created';
  }
  return greet // returned just definition of function
}

let welcomeJohn = createUser('john')
welcomeJohn() // // Hi john is created;
```

- This is **Closure**

    - *welcomeJohn* function definition has access
        - to outer **params** (*name*) which came for *createUser* function
        - also any other "variables" declared inside *createUser* will also be accessible to this *welcomeJohn*

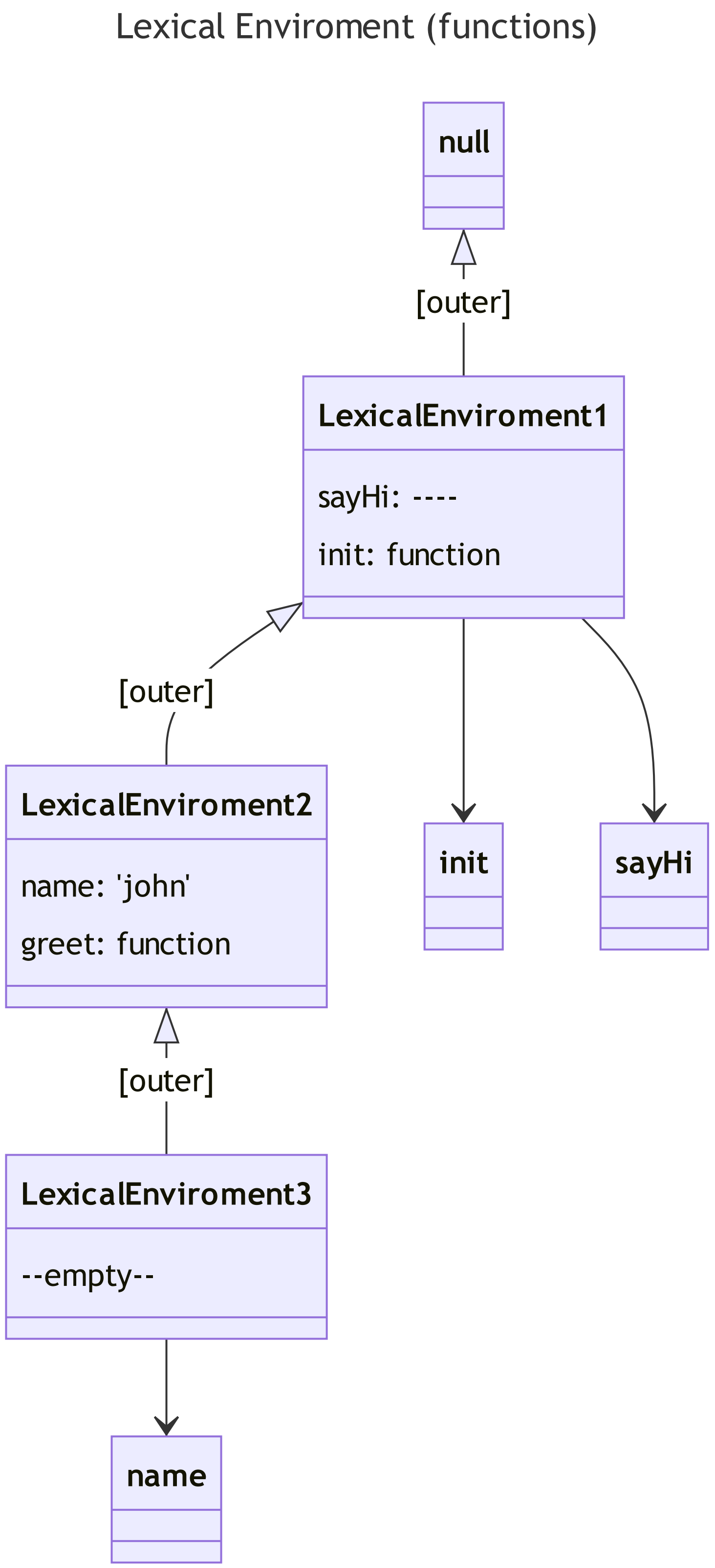
Example

```
function initCounter() {
  let count = 0;
  return function () {
    count++;
  };
}

let counter = initCounter();
counter() // 0
counter() // 1

let counter1 = initCounter();
counter1() // 0
counter1() // 1
```

NOTE : so whenever you have a function which wants to preserve a value over many calls - it's a time for closure.

Lexical Environment

```
function init() {
  let name = 'john';

  function greet() {
    console.log(name)
  }
  return greet;
}

let sayHi = init();

sayHi();
```



Real life example 1

```
function initCounter(id) {
  let count = 0;
  return function () {
    count++;
    document.getElementById(id).innerText = count;
  };
```

```javascript
}
let count = 10;
let counter1 = initCounter('btnCount1');
let counter2 = initCounter('btnCount2');

// here `btn1` and `btn2` are id of HTML buttons.
```

```html
<button onclick="counter1()">1</button>
<p id="btnCount1"></p>
<button onclick="counter2()">2</button>
<p id="btnCount2"></p>
```

Real life example 2

```javascript
function initAddString(inputId, outputId) {
  let str = '';
  return function () {
    str += ' ' + document.getElementById(inputId).value;
    document.getElementById(inputId).value = '';
    document.getElementById(outputId).innerText = str;
  };
}

let strAdder1 = initAddString('text1', 'text-output1');
let strAdder2 = initAddString('text2', 'text-output2');
```

```html
<input type="text" id="text1">
<button onclick="strAdder1()">Add String</button>
<p id="text-output1"></p>

<input type="text" id="text2">
<button onclick="strAdder2()">Add String</button>
<p id="text-output2"></p>
```

IIFE - Immediately Invoked Function Expression

- this practice was popular due to *var*.
- Immediately invoking a function avoids - re-declaration of variables inside it

```
// Immediately invoked function expressions
(function(){
      var x = 1;   // this var is now protected
})()


(function(a){
      var x = a;   // this var is now protected
})(2)
```

Currying

```
function sum(a){
  return function(b){
    return function(c){
       console.log(a,b,c)
       return a+b+c
    }
  }
}

let add = a => b => c => a+b+c

let log = time => type => msg => `At ${time.toLocaleString()}: severity $

log(new Date())('error')('power not sufficient')

let logNow = log(new Date())

logNow('warning')('temp high')

let logErrorNow = log(new Date())('error')

logErrorNow('unknown error')
```

```
function op(operation) {
  return function (a) {
    return function (b) {
      return operation === 'add' ? a + b : a - b;
    };
```

```
  };
}

const add3 = op('add')(3);
const sub3 = op('sub')(3);
const add = op('add');

add3(6);
sub3(6);
add(1)(2);
```

2. Objects

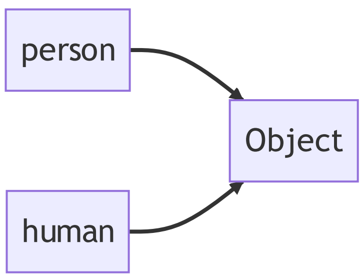
Basic behaviours

Reference Copying

- Variable value is not copied in case of object/arrays

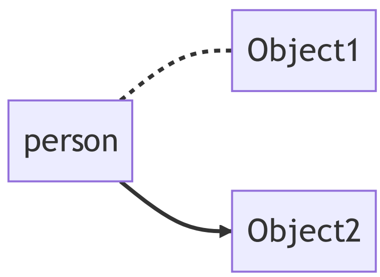
```
let person =  {name:'john'}
let human = person;
```

Reference are point to same value




```
let person =  {name:'john'} // Object1
person = {name:'wick'}; // Object2
```

Reference can be changed for a variable (Garbage collection of Object1)



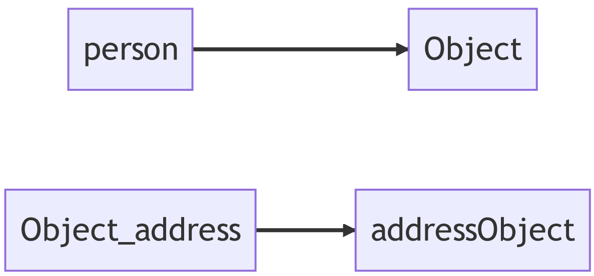
- it a better to use *const* always, and whenever you must need to re-assign change it ot *let*

```
const person  = {name:'john'} // Object1
person = {name:'wick'}; // ERROR
```

Nested Objects

```
let person = {
  name: 'John',
  address: { city: 'delhi', state: 'delhi' },
};
```

Object properties can point to other objects




```
let addressObject = { city: 'delhi', state: 'delhi' }

let person = {
  name: 'John',
  address: addressObject
};
```

Copying objects

Shallow Copy

Many methods can be used to *copy* object without old reference

1. **Object.assign()**

```
let person =  {name:'john'}
let newPerson = Object.assign({}, person)
```

2. **Spread Operator[...]**

```
let person =  {name:'john'}
let newPerson = {...person}
```

But problem which these is they just create a copy of *properties* of that object , but not creating a copy of their references also.

```
let addressObject = { city: 'delhi', state: 'delhi' }

let person = {
  name: 'John',
  address: addressObject
};


let newPerson = Object.assign({}, person)
person === newPerson;  // false
person.address === newPerson.address // true
```

Deep Copy

This is a hard problem to solve in past as there can be multiple level of nested objects and there can be references to functions etc also. few methods which are there:

1. **JSON.stringify and JSON.parse** : this method utilizes the fact that every JSON can be converted to a string value (exception of methods/functions)

```
let addressObject = { city: 'delhi', state: 'delhi' }

let person = {
  name: 'John',
  address: addressObject
};

let str = JSON.stringify(person)
let jsonObject = JSON.parse(str);
```

2. **structuredClone** : Browser API which work even for circular references (but functions not supported)

```
let addressObject = { city: 'delhi', state: 'delhi' }

let person = {
  name: 'John',
  address: addressObject,
```

```
};

person.me = person

let newPerson = structuredClone(person);
```

"this" and Methods

- we can also defined *function* as value to properties of objecy. these will be called *methods*. Methods are just functions but, it means they have been called in "reference" on an Object.

```
let person =  {
  name:'john',
  sayHi: function(){
    return "hi";
  }
}

person.sayHi() // hi
```

- methods can also access the *properties* and other *methods* of same object. To do this we use *this*

```
let person =  {
  name:'john',
  sayHi: function(){
    return "hi "+ this.name;
  }
}

person.sayHi() // hi john
```

- we can also have used *person* instead of *this* but has you know references can be changed. so that could have created a problem

```
let person =  {
  name:'john',
  sayHi: function(){
    return "hi "+ this.name;
  }
}

person.sayHi() // hi john
```

- you can even have *this* without an object

```
function sayHi(){
    return "hi "+ this.name;
}
sayHi() // Error
// here this will "undefined" in Strict mode
let obj1 = {name: 'john'}
let obj2 = {name: 'wick'}

// you can add functional property

obj1.say = sayHi;
obj2.say = sayHi;


obj1.say() //  hi john
obj2.say() //  hi wick
```

- Arrow functions don't have a *this*. they use outer context

```
let person =  {
  name:'john',
  sayHi: ()=> {
    return "hi "+ this.name;
  }
}

person.sayHi() // Error
```

Symbol

- JavaScript also has a *Symbol* data type. This data type is used as *property* name in Objects.
- Object can only have 2 types of *properties* - String and Symbol. If you put any other data type they will convert to String

```
let person =  {
  0:'john',
  sayHi: ()=> {
    return "hi "+ this.name;
  }
}

person["0"] // this number will convert to string
```

- *Symbol* is used for making hidden (library used properties)

```
const id = Symbol("id"); // "id" is descriptor

let person =  {
  name:'john',
  [id]:1
}

person[id] // 1

// note that we have put square [] on property so that it is not confused
```

- *Symbol* are always unique - so there is no chance of collision. Even with same "descriptor" they will be uniquely initialized.
- You can get *Symbol* for some descriptor or key using some methods

```
// get symbol by name
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");
```

for..in loop ignore Symbols. Also methods like *Object.keys()* ignore these properties.

3. Functions

functions are objects

- they already have some predefined properties *name*, *length* etc
- you can also make more properties on functions (but generally it's not required, except for Constructor function)

```
function sayHi(greet){
   return greet
}
sayHi.name // name of function
sayHi.length // length of arguments

sayHi.count =0; // function can have properties
sayHi.count++;
```

```
sayHi.count;
```

function declaration are hoisted

```
sayHi() // works
function sayHi(greet){
   return greet
}

sayHello() // Error
let sayHello = function(){ // functional expression

}

sayHello.name // sayHello
```

function can be called as constructor

```
function Person(name){
    this.name = name
}

const p = new Person('john') // constructor
```

Named function expression (NFE)

```
let sayHello = function fx(user){ // named functional expression

   if(user){
      return "hello " + user
   } else {
      return fx('anonymous')
   }
}

// this can help in case where sayHello is re-assiged to something

let sayHi = sayHello
sayHello = null
sayHi()
```

Decorator (Wrappers)

- It's a *design pattern* in which you modify the functionality of a function by covering it inside a wrapper.

```
let modifiedFx = Decorator(preDefinedFx)
```

Memoization (Caching)

```
function heavy(x) {
  console.log(x + ':heavy');
  return x + ':heavy';
}

function memoized(fx) {
  let map = new Map();

  return function (x) {  // wrapper
    if (map.has(x)) {
      return map.get(x);
    } else {
      let memoValue = fx(x);
      map.set(x, memoValue);
      return memoValue;
    }
  };
}

let memoizedHeavy = memoized(heavy)
memoizedHeavy(2);
memoizedHeavy(2); // take from cache
```

Another Problem

- if you try to use this on a *method* of object, this approach can fail

```
let task = {
  name: 'demo',
  heavy(x) {
    console.log(x + ':heavy:' + this.name);
    return x + ':heavy' + this.name;
```

```
  },
};

function memoized(fx) {
  let map = new Map();
  return function (x) {
    if (map.has(x)) {
      return map.get(x);
    } else {
      let memoValue = fx(x);
      map.set(x, memoValue);
      return memoValue;
    }
  };
}
task.memoizedHeavy = memoized(task.heavy)
task.memoizedHeavy(1) // 1:heavyundefined
```

Solution : use *function.call()*

changing 'this'

Call

```
person = {
  name: 'demo',
  age: 12,
  location: 'delhi',
};

function checkName(a) {
  return !!this.name;
}

checkName() // Error
checkName.call(person)
checkName.call(person, 1) // a = 1
```

apply

```
person = {
  name: 'demo',
  age: 12,
  location: 'delhi',
```

```
};

function checkName(a) {
  return !!this.name;
}

checkName() // Error
checkName.apply(person)
checkName.apply(person, [1]) // a = 1
```

bind

```
person = {
  name: 'demo',
  age: 12,
  location: 'delhi',
};

function checkName(a) {
  return !!this.name;
}

checkName() // Error
let boundCheckName = checkName.bind(person)
boundCheckName();
```

Solution

```
let task = {
  name: 'demo',
  heavy(x) {
    console.log(x + ':heavy:' + this.name);
    return x + ':heavy' + this.name;
  },
};

function memoized(fx) {
  let map = new Map();
  return function (x) {
    if (map.has(x)) {
      return map.get(x);
    } else {
      let memoValue = fx.call(this,x);
      map.set(x, memoValue);
      return memoValue;
```

```
    }
  };
}
task.memoizedHeavy = memoized(task.heavy)
task.memoizedHeavy(1) // 1:heavydemo
```

Debounce

- Run a function only when - if it has not been called again for a fixed *period*
- Suppose you are typing and take a pause of 1 second. Only then that function should be called.

```
let count = 1;
function showCount() {
  count++;
  console.log({ count });
}

function debounce(fx, time) {
  let id = null;
  return function (x) {
    if (id) {
      clearTimeout(id);
    }
    console.log({ id });
    id = setTimeout(() => {
      fx(x);
      id = null;
    }, time);
  };
}

let showCountD = debounce(showCount, 2000);
setTimeout(showCountD, 1000);
setTimeout(showCountD, 1500);
setTimeout(showCountD, 2000);
setTimeout(showCountD, 2500);
setTimeout(showCountD, 5000);
```

Real Example

```
const el = document.getElementById('text1');
const logo = document.getElementById('text-output1');
```

```
el.addEventListener(
  'keyup',
  debounce(function (e) {
    logo.innerText = e.target.value;
  }, 1000)
);
```

```
<input type="text" id="text1">
<p id="text-output1"></p>
```

Throttle

- when you have to only allow 1 execution of a function within a *period* of time
- for example you are *scrolling* fast but only 1 scroll per 100 millisecond is considered.

```
let count = 1;
function showCount() {
  count++;
  console.log({ count });
}

function throttle(fx, time) {
  let id = null;
  let arg = [];
  return function (x) {
    arg[0] = x;
    if (!id) {
      id = setTimeout(() => {
        fx(arg[0]);
        id = null;
      }, time);
    }
    console.log({ id });
  };
}

let showCountT = throttle(showCount, 2000);
setTimeout(showCountT, 1000);
setTimeout(showCountT, 1500);
setTimeout(showCountT, 2000);
setTimeout(showCountT, 2500);
setTimeout(showCountT, 5000);
```

Real Example

```
function throttle(fx, time) {
  let id = null;
  let arg = [];
  return function (x) {
    arg[0] = x;
    if (!id) {
      id = setTimeout(() => {
        fx(arg[0]);
        id = null;
      }, time);
    }
  };
}

function sayHi(){console.log('hi')}
document.addEventListener('scroll',throttle(sayHi,1000))
```

Arrow functions

Differences

- they don't have *this*
- they don't have *arguments*,
- they can't be called with *new* (as constructor)

Similarities

- they have properties like *name*, *length*

4. Iterables, Generators

Iterables and Iterators

Iterable (protocol)

- *Iterables* are objects in which we can make array like iteration (Example using *for..of* loop of *spread operators*)
  - Array are *iterables*
  - String are *iterables*
- To make any object iterable we have these conditions
  - implement a *Symbol.iterator* property, which should be a function which return an *Iterator* Object

```
iterable[Symbol.iterator]() => Iterator
```

Iterator (protocol)

Iterators are objects which have :

- a *next()* method which return a object which is of format `{value:-some-value-, done:-boolean-}` e.g. *{value: 1, done: false}
- **value** is the value we are interested in, while **done** tells us when to stop. Generally when *done:true* the *value:undefined*

Now, making an Iterable is like this:

```
let iterator = {
  i: 0,
  next: function () {
    return { value: this.i, done: this.i++ > 5 };
  },
};

let iterable = {
  name: 'john',
  age: 34,
  [Symbol.iterator]() {
    return iterator;
  },
};
```

Example - Range :

```
let range = {
  start: 0,
  end: 5,
  [Symbol.iterator]() {
    let that = this; // this line is very important
    let i = this.start;
    return { // iterator object
      next: function () {
        return { value: i, done: i++ > that.end };
      }
    };
  },
};
```

Array

```
let num = [1, 2, 3];
let iterator = num[Symbol.iterator]();
iterator.next();
iterator.next();
iterator.next();
iterator.next();
```

Infinite iterators

- As we can see that we can control, how to control the *next()* function. In few cases, it will be useful to have *iterators* which can need to generate the next value infinitely
- If you use such *iterators* in a loop etc. it can be dangerous as can create infinite loop. But can be controlled by `break` etc.
- we will cover all this in generators.

Iterables vs Array-like

- *Iterable* objects are based on *Symbol.iterator* method as defined above
- *Array-like* objects are based on array protocols (index and length)

An object can be

- *Iterable* + *Array-like*
- *Iterable* only
- *Array-like* only
- None of them (not *Iterable* nor *Array-like*)

Example :

```
// iterable + array-like
let arr = [1,2,3]


// only iterable
let range = {
  start: 0,
  end: 5,
  [Symbol.iterator]() {
    let that = this; // this line is very important
    let i = this.start;
    return {
      next: function () {
        return { value: i, done: i++ > that.end };
      },
```

```
    };
  },
};

// only array-like
let array = {
  0: 1,
  1: 5,
  length:2
};


// none
let obj = {
 name:'john'
}
```

Conversions

Array-like to Array

- **Array.from()** : method is used for this

```
let arrayLike = {
  0: 0,
  1: 5,
  length: 2
};

let arr = Array.from(arrayLike);

// also used for general things

let set = new Set()
set.add(1);
set.add(2);
let arr2 = Array.from(set) // [1,2]
```

Map

- this data type is also *iterable*
- special this is can have keys also as *numbers*, *booleans*, *objects*
- also map maintains the *order* of keys added.

```
let map = new Map();


let person = {name:'john'}
let personAccount = {balance: 5000}

map.set('1', 'str1');   //  string key
map.set(1, 'num1');     //  numeric key
map.set(true, 'bool1');
map.set (person, personAccount)

map.get(1)  // 'num1'
map.get('1') // 'str1'
map.get(person) // { balance : 5000 }

map.size // 4

map.keys() // iterable of keys
map.values() // iterable of values
map.entries() // iterable of key-value pair

map.has(1) // key exists
```

Converting Object to Map

- We can use **Object.entries()** method for this.

```
let obj = {a:1,b:2,c:3};
let map = new Map(Object.entries(obj));
```

Converting Map to Object

- We can use **Object.fromEntries()** method for this.

```
let map = new Map();
map.set('a', 1);
map.set('b', 2);
map.set('c', 3);

let obj = (Object.fromEntries(map.entries())); // {a:1,b:2,c:3}
```

Set

- Set is another *iterable*

- Set only contains uniques elements

```
let set = new Set();

let obj1 = { name: "John" };
let obj2 = { name: "Jack" };
let obj3 = { name: "Peter" };

set.add(obj1);
set.add(obj2);
set.add(obj3);
set.add(obj2);
set.add(obj3);

// set keeps only unique values
set.size; // 3

set.keys() // iterable of keys
set.values() // iterable of values (Same as keys)
set.entries()
```

- duplicated values in *values()*, *entries()* etc are maintained to match *Map* compatibility

WeakMap and Weakset

- These are 2 alternative way of creating *Map* or *Set* like data types - when only object keys are considered.
- They have very limited operations and doesn't support all functionality
- Main purpose is that when *keys* are marked as *null* they are garbage collected. So this helps in better memory management

```
let weakMap = new WeakMap()
let person = {name:'john'}

weakMap.set(person, {....});

person = null // in future we decide to remove this key

// so weakMap will remove it from memory space automatically
```

Generators

- Easy way to create an *iterators* and *iterables*

```
function* generatorFunction(){
 yield 1;
 yield 2;
 yield 3
}

let generator = generatorFunction();
generator.next() // {value:1, done:false}
generator.next() // {value:2, done:false}
generator.next() // {value:3, done:false}
generator.next() // {done:true}
```

Infinite iterator

```
function* generator() {
  let i = 0;
  while (true) {
    yield i;
    i++;
  }
}

const gen = generator();

function createID(it) {
  return it.next().value;
}

createID(gen);
createID(gen);
createID(gen);
createID(gen);
createID(gen);
```

Generator objects are "iterables"

```
function* generatorFunction(){
 yield 1;
 yield 2;
 yield 3
}

let generator = generatorFunction();
```

```
let nums = [...generator] // [1,2,3]
```

NOTE: Don't put a *Spread operator* or *for..of* loop on inifinite iterable

Range example - using generator

```
let range = {
  start: 0,
  end: 5,

  *[Symbol.iterator]() {  // * makes it generator function
    for(let value = this.start; value <= this.end; value++) {
      yield value;
    }
  }
};

for(let r of range){
    console.log(r)
}
```

Better version - with function

```
function range(start,end){
  return {
  *[Symbol.iterator]() {
    for(let value = start; value <= end; value++) {
      yield value;
    }
  }
}
};

for(let r of range(1,5)){
    console.log(r)
}

let values = [...range(1,5)]
```

Better - Better version - with function

```
function* range(start,end){
  for(let value = start; value <= end; value++) {
```

```
    yield value;
  }
};


let generator = range(1,5)

console.log([...generator]) // [1,2,3,4,5]
```

return

- only difference it that instantly ends the iterator at that value;

```
function* generatorFunction(){
 yield 1;
 yield 2;
 return 3
}

let generator = generatorFunction();
generator.next()  // {value:1, done:false}
generator.next()  // {value:2, done:false}
generator.next()  // {value:3, done:true} **
```

Generator - composition

- using *generator* inside another *generator* is easy

**Composed Generator using - yield*

```
function* range(start,end){
  for(let value = start; value <= end; value++) {
    yield value;
  }
};


function* multiRange(){
     yield* range(0,5),
     yield* range(100,105)
     yield* range(200,205)
}


let generator = multiRange();
```

```
console.log([...generator]) //[ 0, 1, 2, 3, 4, 5, 100, 101, 102, 103, 104
```

Generator can also take inputs

- **next()** method can also take arguments which act as return value of previous *yield* statement

```
function* generatorFunction(){
  let result = yield 1;
  console.log(result)
  let result2 = yield 2;
  console.log(result2)
  let result3 = yield 3
  console.log(result3)

 }

let generator = generatorFunction();
let r1 = generator.next()
let r2 = generator.next(r1.value)
let r3 = generator.next(r2.value)
generator.next(r3.value)
```

Async Iterators/ Async generators

without generators

```
let range = {
  start: 0,
  end: 5,
  [Symbol.asyncIterator]() {
    let that = this; // this line is very important
    let i = this.start;
    return {
      next: async function () {
        await new Promise((resolve) => setTimeout(resolve, 1000));
        return { value: i, done: i++ > that.end };
      },
    };
  },
};
```

```
(async function () {
  for await (let f of range) {
    console.log(f);
  }
})();
```

with generators

```
let range = {
  start: 0,
  end: 5,
  async *[Symbol.asyncIterator]() {
       for(let i = this.start; i <= this.end; i++) {
       await new Promise((resolve) => setTimeout(resolve, 1000));
        yield i
       };
  },
};

(async function () {
  for await (let f of range) {
    console.log(f);
  }
})();
```

Real-life Example - Paginated API calls

- this example has also used *Composition* of generators

```
async function* getDataAsync(page) {
  let response = await fetch(
    'https://projects.propublica.org/nonprofits/api/v2/search.json?q=x&pa
  );
  let result = await response.json();
  for(let org of result.organizations){
    yield org.name;
  }
}


async function* getData() {
  let response = await fetch(
    'https://projects.propublica.org/nonprofits/api/v2/search.json?q=x'
  );
```

```
    let result = await response.json();


    for (let i = 0; i <= result.num_pages; i++) {
      yield* await getDataAsync(i);
    }
  }

  (async function () {
    let orgs = []
    for await (let f of getData()) {
      orgs.push(f);
    }
    console.log(orgs);  // List of all organization in API
  })();
```

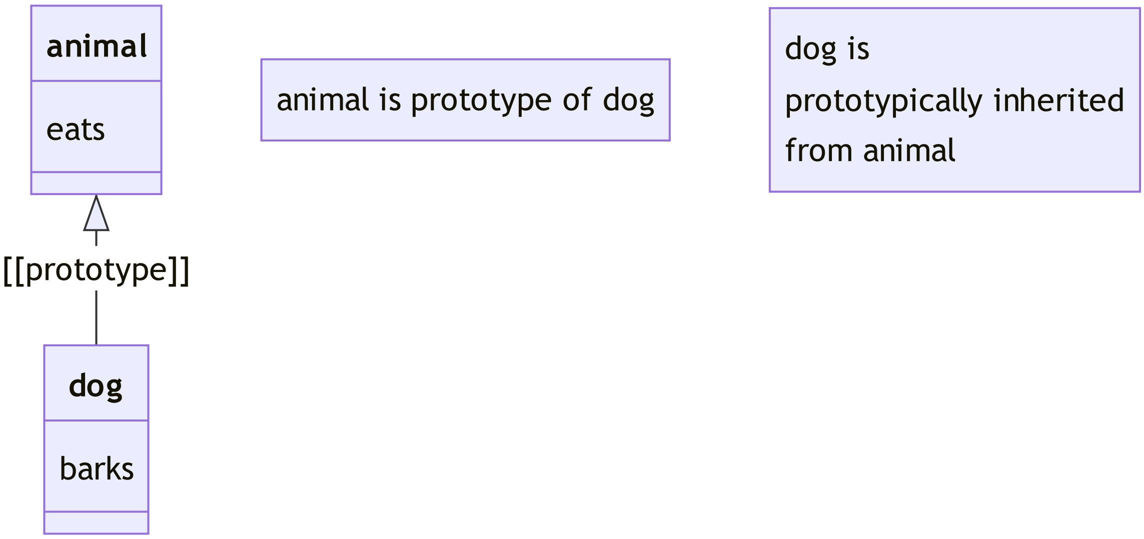
5. ProtoTypes

Prototypical Inheritance

- Objects are *extended* from other Objects. And we can re-use their *properties* and *methods*.
- Object are chained in *prototypical inheritance*
- Objects have a hidden property called `[[Prototype]]`

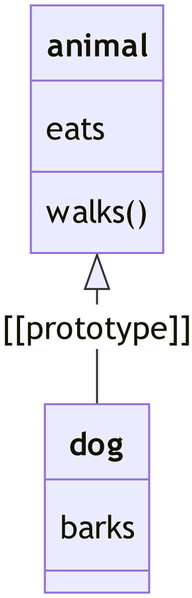
Prototype Inheritance

Prototype example




```javascript
let animal = { eats: true };
let dog = { barks: true };
dog.__proto__ = animal;

dog.barks // true
dog.eats // true
```

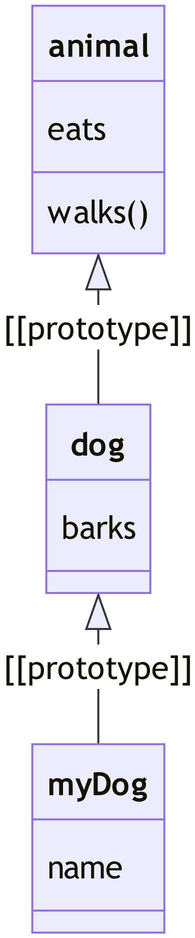
Prototype chaining




```javascript
let animal = {
  eats: true,
  walks: function () {
    return 'walks';
  },
};
```

```
let dog = { barks: true };


dog.__proto__ = animal;

dog.walks() // walks
```

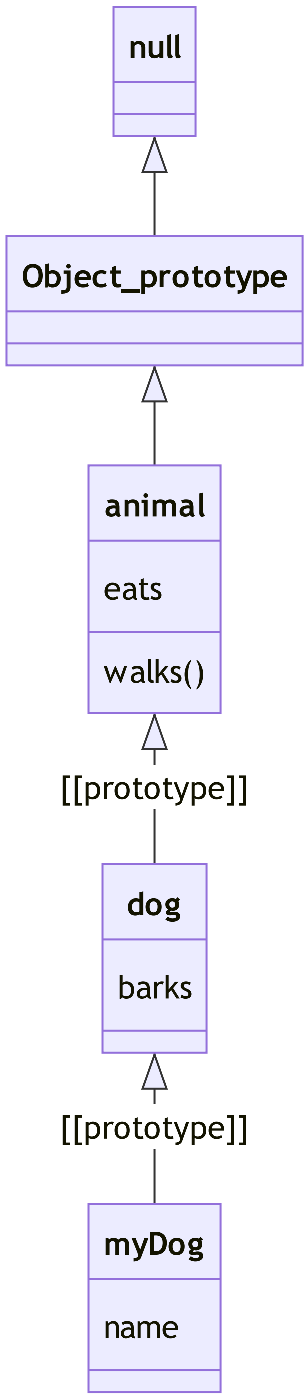
Prototype chain can be longer and longer




```
let animal = {
  eats: true,
  walks: function () {
    return 'walks';
  },
};
let dog = { barks: true };
let myDog = { name: 'sifu' };


dog.__proto__ = animal;
myDog.__proto__ = dog;

myDog.name // sifu
```

```
myDog.barks  // true
myDog.walks() // walks
```

Prototype end at "null"



__proto__

- `__proto__` is a getter/setter for `[[Prototype]]`
- Writing property, doesn't call inherited properties. Except for `getter/setter` properties.
- `__proto__` is not used now , and recommended way is to use *Object.getPrototypeOf()* and *Object.setPrototypeOf*

```
let animal = {
  eats: true,
  walks: function () {
    return 'walks';
  },
};
let dog = { barks: true };
let myDog = { name: 'sifu' };


dog.__proto__ = animal;
myDog.__proto__ = dog;

myDog.walks = function(){
   return 'walks slowly'; // this will not affect prototype
}

myDog.walks() // walks slowly
dog.walks() // walks
```

- *for..in* loop works on all properties which are *enumerable* - *inherited* or *own*
- if you want to avoid looping on inherited ones use `Object.hasOwn` or `Object.prototype.hasOwnProperty`
- *Object.keys()* and *Object.value()* these will avoid `inherited` properties.

.prototype property, constructor

properties

```
// simple object initialization

let usr = {
   name : 'john'
}


// now using a constructor function

function User(name){
     this.name = name
}

let user = new User('john');
```

```
console.log(user)
// User{ name :  'john'}
console.log(usr)
// {name : 'john'}
```

- Step 1 : **.prototype** proptery is automatically created (on *User*) and is assigned an object (empty Object)

```
function User(name){
     this.name = name
}

let user = new User('john');

console.log(User.prototype) // prototype object
```

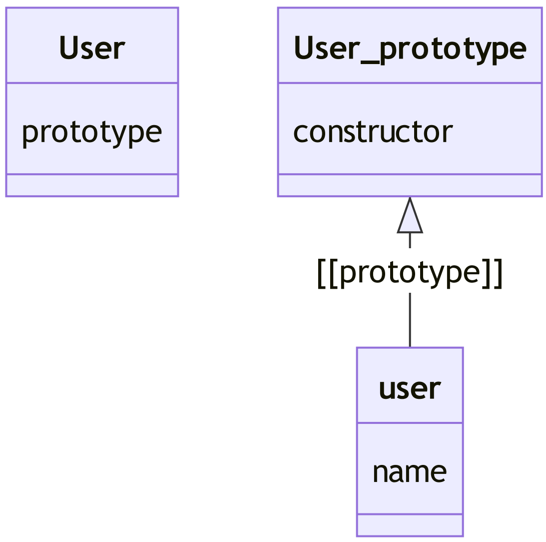
- Step 2 :**constructor** method is assigned to this prototype, and that is *User* function itself.

```
// User.prototype.constructor = User

// this above assignment is done by the constructor call itself
User.prototype.constructor === User // true
```

- Step 3: **.prototype** property's object is assigned to created instances.

```
// user.__proto__ = User.prototype
// this above assignment is done by the constructor call itself

user.__proto__ === User.prototype // true
```

.prototype property

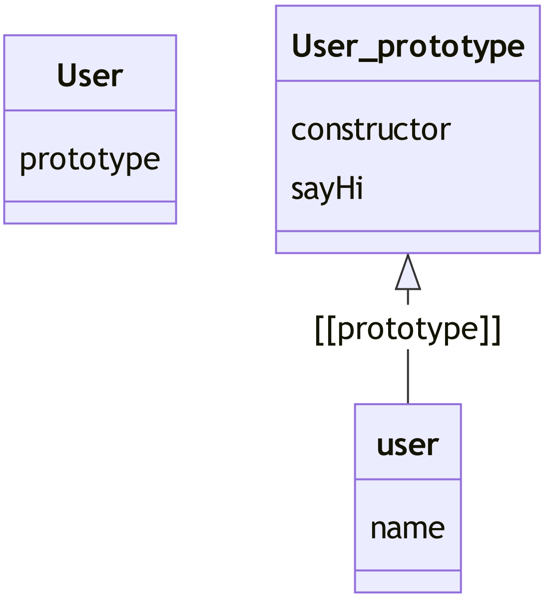


methods

```
function User(name){
     this.name = name
}

User.prototype.sayHi = function () {
  return this.name;
};

let user = new User('john');
let user1 = new User('wick');

user.sayHi()
// 'john'
user1.sayHi();
// 'wick'
```

- this the main benefit of `prototypes`. you can have inherited methods.

.prototype property



A useful method : reverseString

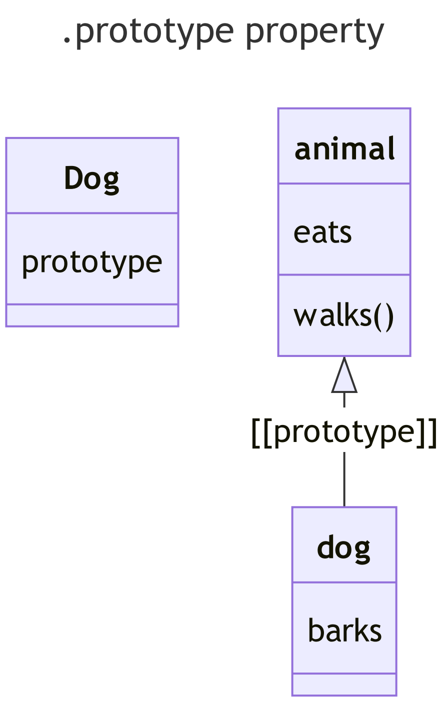
methods

```
function User(name){
     this.name = name
}

User.prototype.reverseName = function () {
  return this.name.split('').reverse().join('');
};

let user = new User('john');
let user1 = new User('wick');

user.reverseName()
// 'nhoj'
user1.reverseName();
// 'kicw'
```

- remember *prototype* based *methods* are directly available on their created *object instances*.
- you can also change the *prototype* completely, not recommended though

```
let animal = {
  eats: true,
  walks: function () {
```

```
    return 'walks';
  },
};

function Dog(){
  this.barks = true
}

Dog.prototype = animal;

let dog =  new Dog();

dog.walks()
// walks

dog.__proto__ === animal;  // true
Dog.prototype === dog.__proto__ // true
```



Native Prototypes

- Object.prototype
- Array.prototype
- Function.prototype

Object.prototype

```javascript
let obj = {}

let obj1 =  new Object();

Object.prototype === obj1.__proto__ // true
Object.prototype === obj.__proto__ // true
```

- toString()
- isPrototypeOf()
- toLocaleString()

Array.prototype

```javascript
let arr = []

let arr1 =  new Array();

Array.prototype === arr1.__proto__ // true
Array.prototype === arr.__proto__ // true
```

- push()
- pop()
- slice()
- splice()
- reverse()
- ....and many more

Function.prototype

```javascript
function Fx(){

}

Function.prototype === Fx.__proto__ // true
```

- call()
- apply()
- bind()
- arguments
- caller
- length

Date.prototype

```
let d = new Date();

d.getTime(); // getTime is given by Date.prototype
```

- getTime()
- getDay()
- getDate()
- .... more

Primitives

Primitive types also get wrapped into a Object when used as an Object

String.prototype

```
"hello".toString()
```

Number.prototype

```
10.1111.toFixed(2)
```

Boolean.prototype

Polyfills

- polyfill is a way of providing futuristic API not available in browser.
- polyfills are made often Native prototype modifications, so that we can get a feature/API (which is not available in current browser)
- This can help us write code / libraries which can run on many systems (old or modern)

```
if(!Array.prototype.contains){
      Array.prototype.contains = function(searchElement) {
         return this.indexOf(searchElement)>=0 ? true : false
      }

}
// similar to includes()
```

NOTE : `Shims` are piece of code to correct some existing behaviour, while `Polyfills` are new API/ behaviours.

Static properties and methods

Some properties and methods are directly created on these Native constructors.

- **Object.create()**
- **Object.keys()**
- **Object.values()**
- **Object.hasOwn()**
- **Array.from()**
- **Date.now()**

These are not available on instances, and only available on Native contructors

6. Class

Classes are *easier* way to implement inheritance in JavaScript.

Syntactic Sugar

It's a syntactic sugar to *Protypical Inheritance* BUT more functionalities than it.

ProtoType Version

```
function User(name){
     this.name = name
}

User.prototype.sayHi = function () {
  return this.name;
};

let user = new User('john');
user.sayHi() // john
```

Class Version

```
class User {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
```

```
    return this.name;
  }
}

let user = new User('john');
user.sayHi() // john
```

Similarities:

1. Same kind of *prototype* property with `constructor` method is added when called with new;
2. you can use *prototype* also on class based things

```
class User {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return this.name;
  }
}
User.prototype.sayHello = function(){
    return "hello "+this.name;
}


let user = new User('john');
user.sayHello() // hello john
```

Differences:

1. Class methods are *non-enumerable*
2. Class *toString()* is different
3. Class can only be called with *new* . Not as a normal function
4. Class is always is *use strict* mode.

getter/setters

- Accessor properties can also be used in class

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
```

```
    this.lastName = lastName;
  }

  get fullName(){
      return this.firstName + ' ' + this.lastName;
  }
  set fullName(_fullName){
      this.firstName = _fullName.split(' ')[0];
      this.lastName = _fullName.split(' ')[1];
  }
}

let user = new User('john', 'wick');
user.fullName // john wick
user.fullName =  "john cena"

user.firstName // john
user.lastName // cena
```

Computed property names

- properties which don't have a fixed name and assigned by [ ]

```
let variableName = "hello"
class User {
  constructor(name) {
    this.name = name;
  }

  [variableName]() {
    return this.name;
  }
}



let user = new User('john');
user.hello() // john```
```

"this" binding issue

```js
class Button {
  constructor(value) {
    this.value = value;
  }

  click() {
    return this.value;
  }
}

let button = new Button("play");

button.click()  // play

setTimeout(button.click, 1000);
// this has issue - this has changed here
```

- here we lose the context of `this`.

2 Solution exists :

1. Arrow functions : use arrow function wrappers.

```js
setTimeout(()=>button.click(), 1000);
```

2. use *.bind()* to `constructor` object.

```js
setTimeout(button.click.bind(button), 1000);
```

Also you can add this **arrow** style function in class definition - which will act as class field

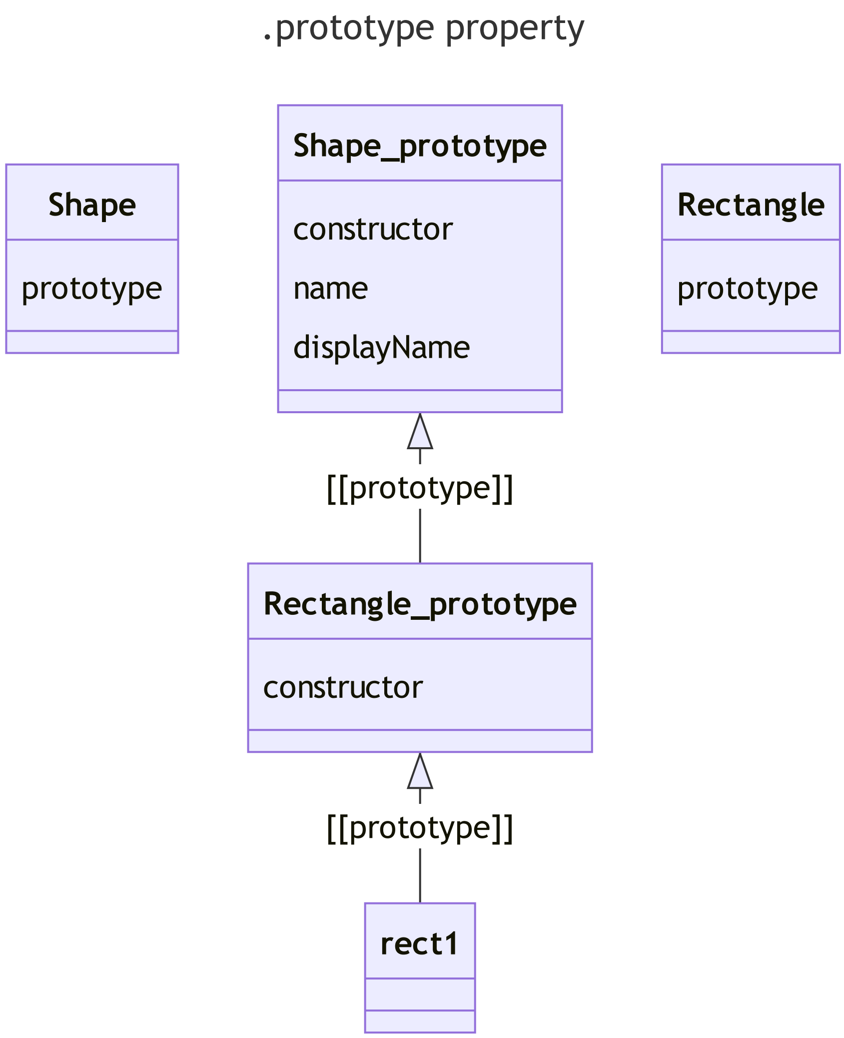
```js
class Button {

  constructor(value) {
    this.value = value;
  }

  click = () => { // this is a class field
    return this.value;
  }
}

let button = new Button("play");

button.click()  // play
```

```
setTimeout(button.click, 1000);
```

Inheritance

- We can inherit Parent Class properties and metods in a Child Class. using *extends* keyword
- Here we have ***Shape*** as *Parent* and ***Rectangle*** as *Child* :

```
class Shape {
  constructor(name) {
    this.name = name;
  }
  displayShape() {
    return 'Shape ' + this.name;
  }
}

class Rectangle extends Shape {
}

let rect1 = new Rectangle('rect1');

rect1.displayShape(); // Shape rect1

// constructor of Child is implicitly created and it calls constructor of


// constructor(...args){
//     super(..args)
// }
```



- Now adding more properties to *constructor* of Rectangle. You have to call *super* constructor - which will call *Shape* constructor.

```
class Shape {
  constructor(name) {
    this.name = name;
  }
  displayShape() {
    return 'Shape ' + this.name;
  }
}

class Rectangle extends Shape {
  constructor(name, width, height) {
    super(name);
    this.width = width;
    this.height = height;
    this.area = width * height;
  }
}

let rect1 = new Rectangle('rect1', 10, 11);
```

```
rect1.displayShape();
rect1.area;
```

Static Methods

- We can have *methods* on *constructor* function also.
- These methods are called *static* methods and they don't apply on *prototype*. So they are not accessible to created objects also.
- Use of such methods is limited to Class wide applications
- *this* remains same as the *class*

```
class Shape {
  constructor(name,area) {
    this.name = name;
    this.area = area;
  }
  static areEqual(shape1, shape2){
    return shape1.name === shape2.name && shape1.area === shape2.area
  }

}


let s1 = new Shape('rectangle',100)
let s2 = new Shape('rectangle',100)

Shape.areEqual(s1,s2) // true
```

- *static* property are also available as a new feature, but rarely used.

Private and Protected properties

- in Object Oriented Programming there is a concept of *Encapsulation* or *Data Hiding* - so that you just interact with object via given methods/properties. This avoids changing some internal properties which are not meant for public use.

```
class User {
  type = "admin"
  constructor(name) {
    this.name = name;
  }
}
```

```
let user = new User('john')
user.type = "normal"
```

- properties `type` and `name` both are accessible - so they are called public

Protected

- this is something not provided by javascript but by convention and *get/set* method we can create it
- you have to use convention of _ in front of property name - making is known to developer that this property is not directly accessible and used only via *get/set* accessors.

```
class User {
  _type = "admin"
  constructor(name) {
    this.name = name;
  }
  get type(){
      return this._type
  }
  set type(type){
      this._type = type;
  }
}

let user = new User('john')
user.type = "normal"
```

But what is benefit ?

```
class User {
  _type = "admin"
  constructor(name) {
    this.name = name;
  }
  get type(){
      return this._type
  }
  set type(type){
      if(type==('normal' || 'admin')){
          this._type = type;
      } else {
         throw Error('admin / normal ?')
      }
```

```
  }
}

let user = new User('john')
user.type = "normal"
```

Private

- this is a new feature and is not very frequently used.
- you can name any proptery with #

```
class User {
  #type = "admin"
  constructor(name) {
    this.name = name;
  }
  get type(){
      return this.#type
  }
  set type(type){
      if(type==('normal' || 'admin')){
           this.#type = type;
      } else {
         throw Error('admin / normal ?')
      }
  }
}

let user = new User('john')
user.type = "normal"
user.#type // Error
```

instanceOf

- to check if object is instance of a Class or inherited from a Class

```
class Shape {
  constructor(name) {
    this.name = name;
  }
  displayShape() {
    return 'Shape ' + this.name;
  }
}
```

```
class Rectangle extends Shape {
  constructor(name, width, height) {
    super(name);
    this.width = width;
    this.height = height;
    this.area = width * height;
  }
}

let rect1 = new Rectangle('rect1', 10, 11);


rect1 instanceof Rectangle // true
rect1 instanceof Shape // true
```

7. Async JavaScript

Asynchronous APIs

- JavaScript itself is not asynchronous langauge it uses some API from browser or enviroment to achieve this behaviour

```
console.log(1)
setTimeout(console.log,1000,3); // Timer API
console.log(2)
```

- Now suppose, we have a function which does something meaningful and return a value - but *asynchronously* .

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

let asyncFx =(a,b)=>setTimeout(()=>sum(a,b),1000)
```

How to get that value back in program ??

Callbacks

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

```
let asyncFx = (a,b,cb)=>setTimeout(()=>cb(sum(a,b)),1000)
// callback is passed from outside, and called from inside of async funct

asyncFx(3, 1, function (result) {
    console.log({result})
})
```

Errors

```
function sum(a, b) {
    if(a>0 && b>0){
        return [null,a + b]
    } else{
        return ['input', null]
    }
}

let asyncFx = (a,b,cb)=>setTimeout(()=>cb(...sum(a,b)),1000)

asyncFx(3, 1, function (error,result) {
    if(error){
        console.log({result})
    } else{
        console.log({error})
    }
})
```

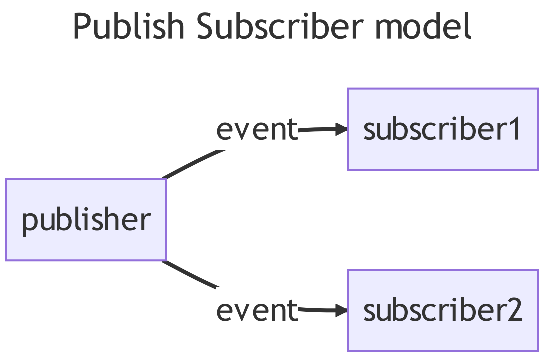
Multiple callbacks

```
function sum(a, b) {
    if(a>0 && b>0){
        return [null,a + b]
    } else{
        return ['input', null]
    }
}

let asyncFx = (a,b,cb)=>setTimeout(()=>cb(...sum(a,b)),1000)

let x = 4;
let y = 5;
```

```
asyncFx(3, 1, function (error, result) {
  console.log({ result });
  asyncFx(x, result, function (error, result) {
    console.log({ result });
    asyncFx(y, result, function (error, result) {
      console.log({ result });          // Callback hell
    });
  });
});
```

Promise

- Promise are based on *Publish-Subscribe* pattern.

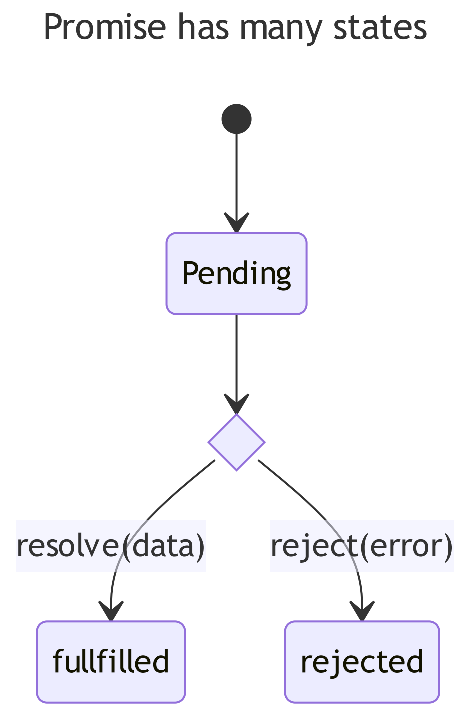


Example : Youtube video release

- *subscribers* are people who are subscribed to channel (with bell icon)
- *publisher* is video uploader channel
- When the *release event* happens, automatically people are notified about the released video.

Promise constructor

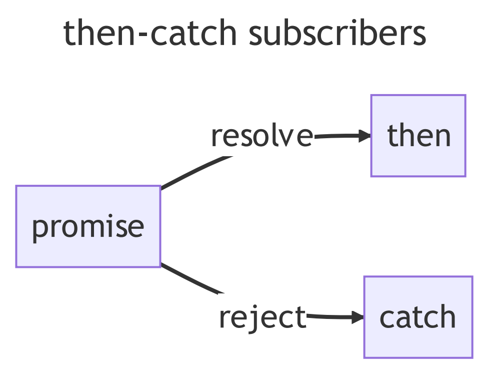
```
let promise = new Promise((resolve, reject)=>{
    // async task is inside this
   // if async task is successful
      resolve(data);
   // else task is having error
      reject(error)
})
```



Promise Consumers

```
let promise = new Promise((resolve, reject)=>{
    // async task is inside this
   // if async task is successful
      resolve(data);
   // else task is having error
      reject(error)
})
```

```
promise.then(successCallback).catch(errorCallback)
```



Callback version

```
function sum(a, b) {
    if(a>0 && b>0){
        return [null,a + b]
```

```
    } else{
        return ['input', null]
    }
}

let asyncFx = (a,b,cb)=>setTimeout(()=>cb(...sum(a,b)),1000)

asyncFx(3, 1, function (error,result) {
    if(error){
        console.log({result})
    } else{
        console.log({error})
    }
})
```

Promise version

```
function sum(a, b) {
  if (a > 0 && b > 0) {
    return [null, a + b];
  } else {
    return ['input not correct', null];
  }
}

let asyncFx = (a, b) =>
  new Promise((resolve, reject) => {
    setTimeout(() => {
      let output = sum(a, b);
      if (output[0]) {
        reject(output[0]);
      } else {
        resolve(output[1]);
      }
    }, 1000);
  });

asyncFx(-2,4)
.then(data=>console.log(data))
.catch(err=>console.log(err))
```

Promise chain

```
asyncFx(1, 4)
  .then((data) => {
    console.log(data);
    return asyncFx(1, 4);
  })
  .then((data) => {
    console.log(data);
    return asyncFx(3, 6);
  })
  .then((data) => {
    console.log(data);
  })
  .catch((err) => console.log(err));
```

Note : *catch* is only one it catches for all above then. Also note that `catch` works for `reject` and also any `error` throw by code.

finally

```
asyncFx(1, 4)
  .then((data) => {
    console.log(data);
    return asyncFx(1, 4);
  })
  .then((data) => {
    console.log(data);
    return asyncFx(3, 6);
  })
  .then((data) => {
    console.log(data);
  })
  .catch((err) => console.log(err));
  finally(()=>{
      doSomething() // after everything is completed
  })
```

Promise API

Promise.all

Parallel execution of async functions - only work when *all promises are fullfiled*

```
Promise.all([
    asyncFx(1,2),
    asyncFx(2,3),
    asyncFx(5,6)
]).then(results=>{
    console.log(results) // array of resolved value, same order
})
```

Promise.allSettled

Parallel execution of async functions - only work when *all promises are fullfiled or rejected*

```
Promise.allSettled([
    asyncFx(1,2),
    asyncFx(2,3),
    asyncFx(5,6)
]).then(results=>{
    console.log(results) // array of resolved/reject objects, same order
})
```

Promise.race

Parallel execution of async functions - works when *any one of promises are fullfiled or rejected*

```
Promise.race([
    asyncFx(1,2),
    asyncFx(2,3),
    asyncFx(5,6)
]).then(results=>{
    console.log(results) // value of first settled (resolved/rejected) pr
})
```

Promise.any

Parallel execution of async functions - works when *any one of promises are fullfiled*

```
Promise.race([
    asyncFx(1,2),
    asyncFx(2,3),
    asyncFx(5,6)
]).then(results=>{
```

```
    console.log(results) // value of first fullfilled promise
})
```

Promise.reject

created already promise which gets rejected just after creation

```
let promise = Promise.reject('error')
```

Promise.resolve

created already promise which gets resolved just after creation

```
let promise = Promise.resolve(123)
```

Async/Await

- *async* keywords makes every function to return promise.

```
async function sayHi(){
    return "hi"
}

sayHi().then(result=>console.log(result)) // hi
```

- `"hi"` is wrapped inside using `Promise.resolve`
- we can use *await* only inside a *async* function
- *await* is a *syntatic sugar* for Promise *.then()**

```
function sum(a, b) {
  if (a > 0 && b > 0) {
    return [null, a + b];
  } else {
    return ['input not correct', null];
  }
}

let asyncFx = (a, b) =>
  new Promise((resolve, reject) => {
    setTimeout(() => {
      let output = sum(a, b);
      if (output[0]) {
```

```
        reject(output[0]);
      } else {
        resolve(output[1]);
      }
    }, 1000);
  });

async function init() {
  let result = await asyncFx(4, 5);
  console.log({ result });
}

init();
```

Handling Error in Async/Await function

```
async function init() {
  try {
    let result = await asyncFx(4, 5);
    console.log({ result });
  } catch (err) {
    console.log(error);
  }
}

init();
```

- **async** works for all promise-compatible things

```
async function init() {
  let results = await Promise.all([
    asyncFx(1, 2),
    asyncFx(2, 3),
    asyncFx(5, 6),
  ]);
  console.log(results)
}
```

Move to Async Generators ==

Property of Object

3 criteria to check on every property

1. own or `inherited`
2. `enumerable` or `non-enumerable`
3. `String` or `Symbol`

Property configurations

1. writeable - `true/false`
2. configurable - `true/false`
3. enumberable - `true/false`
4. value : value of property

```
object1 = {property1:42}

Object.defineProperties(object1, {
  property1: {
    value: 42,
    writable: true,
    enumerable: true,
    configurable: true
  },
  property2: {}
});
```

Strict Mode

It shows up many *slient* errors in JavaScript.

```
'use strict' // file level strict mode


function myStrictFunction() {
  // Function-level strict mode syntax
  "use strict";

}
```

- *window* global object is not available
- *assigning* a variable *without declaration* cause issues

```
variable = 10
```

- *duplicate property name* throw error

```
let obj = {a:1,a:2}
```

Object Constructor API

1. **Object()** : new Object() and Object() are same
2. **Object.prototype.constructor** : instance of object created will have `constructor` set to the reference of creator function. Not enumerable

```
const o1 = {};
o1.constructor === Object; // true

const o2 = new Object();
o2.constructor === Object; // true

const a1 = [];
a1.constructor === Array; // true

const a2 = new Array();
a2.constructor === Array; // true

const n = 3;
n.constructor === Number; // true
```

3. **Object.prototype.__proto__** : .

- it' simple an accessor property of `Object.prototype`
- should not be used as `deprecated`
- use instead `Object.getPrototypeOf` and `Object.setPrototypeOf`
- It will gave same results are `Array.prototype` if applied on array object.

4. **Object.assign()** : used to copy all the property from source object (objects) to a target object.

- copies `enumerable` and own properties ONLY
- not suitable to copy getter/accessors - as it only copies the value.
- `String` and `Symbol` both type of properties are copied.
- Only for `Shallow` Copy

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
```

```
// Expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget === target);
// Expected output: true
```

5. **Object.create()** : creates a new empty object, with an existing object as `prototype`

- should not be used these days, better to use `class` syntax
- don't set contstructor automatically its an issue
- {} (Object initializer syntax) is syntactic suger to this syntax only

```
o = {};
// Is equivalent to:
o = Object.create(Object.prototype);

o = Object.create(Object.prototype, {
  // foo is a regular data property
  foo: {
    writable: true,
    configurable: true,
    value: "hello",
    enumerable: true,
  },
  // bar is an accessor property
  bar: {
    configurable: false,
    get() {
      return 10;
    },
    set(value) {
      console.log("Setting `o.bar` to", value);
    },
  },
});


o = Object.create(null);
// Is equivalent to:
o = { __proto__: null };


function Constructor() {}
o = new Constructor();
// Is equivalent to:
o = Object.create(Constructor.prototype);
```

6. **Object.defineProperties()**: defines new properties or modifies old ones,directly on object, return the object

```
const object1 = {};

Object.defineProperties(object1, {
  property1: {
    value: 42,
    writable: true,
    enumerable: true,
    configurable: true
  },
  property2: {}
});

console.log(object1.property1);
// Expected output: 42
```

7. **Object.defineProperty()**: defines a new property or modifies old one ,directly on object, return the object

```
Object.defineProperty(object1, 'property1', {
  value: 42,
  writable: false
});

object1.property1 = 77;
// Throws an error in strict mode

console.log(object1.property1);
// Expected output: 42
```

8. **Object.entries()** : array of array of key-value pairs on an object proptery (own, enumerable)
9. **Object.freeze()** : Freezing objects makes properties non-writeable and non-configurable.

- Highest integrity level of JS object. *Object.isFrozen()* checks if object is frozen.

10. **Object.fromEntries()** : key-value pairs (inside an iterable, array or Map) are converted in object.

- convert Map to an Object
- convert Array to an Object
- tranform object

```
// Map to Object
const map = new Map([
```

```
  ["foo", "bar"],
  ["baz", 42],
]);
const obj = Object.fromEntries(map);
console.log(obj); // { foo: "bar", baz: 42 }

// Transform object

const object1 = { a: 1, b: 2, c: 3 };

const object2 = Object.fromEntries(
  Object.entries(object1).map(([key, val]) => [key, val * 2]),
);

console.log(object2);
// { a: 2, b: 4, c: 6
```

11. **Object.getOwnPropertyDescriptor()** : return configuration object of a specific property. that object is mutable but won't affect the original configurations
    Object.getOwnPropertyDescriptors() - is similar to this but return configuration of all properties at once.

```
const object1 = {
  property1: 42
};

const descriptor1 = Object.getOwnPropertyDescriptor(object1, 'property1')

console.log(descriptor1.configurable);
// Expected output: true

console.log(descriptor1.value);
// Expected output: 42
```

12. **Object.getOwnPropertyNames()** : array of all properties *including non-enumberable* but not "Symbols" only "Strings". Similary to this is *Object.getOwnPropertySymbols()* which takes only "Symbols"

```
// Only getting enumerable properties - trick
const target = myObject;
const enumAndNonenum = Object.getOwnPropertyNames(target);
const enumOnly = new Set(Object.keys(target));
const nonenumOnly = enumAndNonenum.filter((key) => !enumOnly.has(key));
```

```
console.log(nonenumOnly);
```

13. **Object.getPrototypeOf()** : get the prototype of an Object

```
const proto = {};
const obj = Object.create(proto);
Object.getPrototypeOf(obj) === proto; // true
```

14. **Object.hasOwn()** : return true if own property. *Object.hasOwnProperty()* is older version of same.

```
const object1 = {
  prop: 'exists'
};

console.log(Object.hasOwn(object1, 'prop'));
// Expected output: true

console.log(Object.hasOwn(object1, 'toString'));
// Expected output: false

console.log(Object.hasOwn(object1, 'undeclaredPropertyValue'));
// Expected output: false
```

15. **Object.is()** : two values are same or not, including primitives

- Its almost same as === but it also differentiate +0 and -0 and NaN

16. **Object.isExtensible()** : true if you can add more properties to an object.

17. **Object.prototype.isPrototypeOf()** : check if object exists in another object's proto chain

```
function Foo() {}
function Bar() {}

Bar.prototype = Object.create(Foo.prototype);

const bar = new Bar();

console.log(Foo.prototype.isPrototypeOf(bar));
// Expected output: true
console.log(Bar.prototype.isPrototypeOf(bar));
// Expected output: true
```

18. **Object.keys**() : array of keys (own, enumberable, string type)

19. **Object.preventExtensions()** : prevents adding of new properties, also prevents re-assignment of prototype value.
20. **Object.prototype.propertyIsEnumerable()** : check if `enumerable` own property.
21. **Object.seal()** : seals objects for further addition of new properties, and also make `configurable: false` for all properties. but allow old property value modifications.
22. **Object.setPrototypeOf()** :

```
const obj = {};
const parent = { foo: 'bar' };

console.log(obj.foo);
// Expected output: undefined

Object.setPrototypeOf(obj, parent);

console.log(obj.foo);
// Expected output: "bar"
```

23. **Object.prototype.LocaleString() :

```
const date1 = new Date(Date.UTC(2012, 11, 20, 3, 0, 0));

console.log(date1.toLocaleString('ar-EG'));
// Expected output: "٢٠‏/١٢‏/٢٠١٢ ٤:٠٠:٠٠ ص"

const number1 = 123456.789;

console.log(number1.toLocaleString('de-DE'));
// Expected output: "123.456,789"
```

24. **Object.prototype.toString()** : for converting object in String format
25. **Object.prototype.valueOf()** : for converting Object in primitive values by when primitive value is expected.
26. **Object.values**() : array of vaules (own, enumberable,string keyed)