JS_Beginning_to_Mastery_Part1_3

April 21, 2023

[1]:
```
const myArray = ["value1", "value2", "value3", "value4", 1, 2, 3, 4, 5, 6];

// delete item 2
myArray.splice(1,1); // starting from index 1, delete 1 item
console.log(myArray);
```

```
[ 'value1', 'value3', 'value4', 1, 2, 3, 4, 5, 6 ]
```

[2]:
```
// delete 6 items starting from index 3
myArray.splice(3,6); // starting from index 3, delete 6 items
console.log(myArray);
```

```
[ 'value1', 'value3', 'value4' ]
```

[3]:
```
const deletedItem = myArray.splice(1,1); // starting from index 1, delete 1 item
console.log(deletedItem);
```

```
[ 'value3' ]
```

[4]:
```
console.log(myArray);
```

```
[ 'value1', 'value4' ]
```

[5]:
```
// insert
myArray.splice(1, 0, "inserted item"); // insert at index 1, no delete as 0 is␣
↪passed, and the item to be inserted
console.log(myArray);
```

```
[ 'value1', 'inserted item', 'value4' ]
```

[6]:
```
// insert and delete at the same time
// will delete two items from index 1 and then add two elements
myArray.splice(1, 2, "new inserted item1", "new inserted item4");
console.log(myArray);
```

```
[ 'value1', 'new inserted item1', 'new inserted item4' ]
```

[7]:
```
// I can also store the deleted items when using splice
const deletedItems2 = myArray.splice(1, 2, "new inserted item5", "new inserted␣
↪item6");
console.log(deletedItems2);
```

```
[ 'new inserted item1', 'new inserted item4' ]
```

[8]:
```
console.log(myArray);
```

```
[ 'value1', 'new inserted item5', 'new inserted item6' ]
```

[9]:
```
// object is not iterable (cannot read property Symbol(Symbol.iterator))

// string, arrays are iterable

// array like objects
// which has length property and index based access
// string is array like object

const firstName = "John";
console.log(firstName.length);
console.log(firstName[2]);
```

```
4
h
```

0.0.1 Working with Set

[10]:
```
const numbers = new Set([1,2,3,3]);
console.log(numbers);
```

```
Set(3) { 1, 2, 3 }
```

no index-based access, no duplicates, no order

[11]:
```
// add
numbers.add(55);
console.log(numbers);
```

```
Set(4) { 1, 2, 3, 55 }
```

[12]:
```
// delete
numbers.delete(2);
console.log(numbers);
```

```
Set(3) { 1, 3, 55 }
```

[13]:
```
// has
console.log(numbers.has(3));
```

```
true
```

[14]:
```
// clear
numbers.clear();
console.log(numbers);
```

```
Set(0) {}
```

[15]:
```
// size
console.log(numbers.size);
```

```
0
```

[16]:
```
const numbers2 = new Set([1,2,3,4,5,6,7,8,9,10]);
// forEach
numbers2.forEach((value, key) => {
    console.log(key, value);
})
```

```
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
```

[17]:
```
// convert to array
const numbersArray = Array.from(numbers2);
console.log(numbersArray);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

[18]:
```
// convert to array
const numbersArray2 = [...numbers2];
console.log(numbersArray2);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

[19]:
```
// convert to array
const numbersArray3 = [...new Set([1,2,3,3,4,5,6,7,8,9,10])];
console.log(numbersArray3);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

[20]:
```
// convert to array
const numbersArray4 = Array.from(new Set([1,2,3,3,4,5,6,7,8,9,10]));
console.log(numbersArray4);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

0.0.2 Optional chaining

[21]:
```
// optional chaining

const user = {
    firstName: "John",
    address: {houseNumber: '1234'}
};

console.log(user.firstName);
console.log(user.address.houseNumber);
```

```
John
1234
```

[22]:
```
const user2 = {
    firstName: "John",
    // address: {houseNumber: '1234'}
};

console.log(user2.firstName);
console.log(user2.address); // returns undefined
// console.log(user2.address.houseNumber);
// the above line will returns error as can't read property of undefined
```

```
John
undefined
```

we want to retun undefined and not error if a property is not there

[23]:
```
console.log(user2?.firstName);
// Does user exist? if yes then return firstName
```

```
John
```

[24]:
```
console.log(user2?.addres?.houseNumber);
// Does user exist? if yes then return address.
// Does address exist? if yes then return houseNumber
```

```
undefined
```

Methods – function inside object

[25]:
```
// methods
// function inside object

const person = {
    firstName: "John",
    age: 33,
    about: function(){
        console.log("Person's name is John and age is 33");
    }
};
```

[26]:
```
// Access  this function or method inside person
console.log(person.about); // returns or print the function
```

```
[Function: about]
```

[27]:
```
// calling the function inside person
person.about(); // returns or print the function
```

```
Person's name is John and age is 33
```

this keyword `this` keyword refers to the object that is executing the current function

[28]:
```
const person2 = {
    firstName: "Lipon",
    age: 43,
    about: function(){
        console.log(`Person's name is ${this.firstName} and age is ${this.
↪age}`);
    }
    // We can't write console.log(`Person's name is ${firstName} and age is␣
↪${age}`), rather have to use this keyword to access the object's property
    // here this will refer to the object person2 which will be calling the␣
↪method about
};

person2.about();
```

```
Person's name is Lipon and age is 43
```

[29]:
```
const person3 = {
    firstName: "Lipon",
    age: 43,
    about: function(){
        console.log(this);
    }
};
```

```
person3.about(); // prints the object person3
```

```
{ firstName: 'Lipon', age: 43, about: [Function: about] }
```

[30]:
```
// methods -- continued
const person4 = {
    firstName: "Shammun",
    age: 33,
    about: function(){
        console.log(this.firstName, this.age);
    }
};
person4.about(); // prints the object person4
```

```
Shammun 33
```

[31]:
```
// in below, this refers to the object that is executing the current function
function personInfo(){
    console.log(`person name is ${this.firstName} and age is ${this.age}`);
}
```

[32]:
```
const person5 = {
    firstName: "Shammun",
    age: 37,
    about: personInfo
};

const person6 = {
    firstName: "Mohsin",
    age: 47,
    about: personInfo
};

const person7 = {
    firstName: "Mokhtar",
    age: 57,
    about: personInfo
};
```

[33]:
```
person5.about();
person6.about();
```

```
person name is Shammun and age is 37
person name is Mohsin and age is 47
```

[34]:
```
person7.about();
```

```
person name is Mokhtar and age is 57
```

0.0.3 use strict

`Use strict` is a special string that can be used in JavaScript code to enable a stricter version of the language, which enforces certain rules and restrictions. When the `use strict` mode is enabled, the JavaScript engine will perform stricter syntax checking and will generate more errors in certain situations that might have been ignored before.

Some of the benefits of using `use strict` include:

Preventing the accidental creation of global variables. In non-strict mode, it's possible to accidentally create a global variable by forgetting to declare it with the `var`, `let`, or `const` keyword. `use strict` mode prevents this by throwing an error whenever a variable is used without being declared first.

Making it easier to write secure code. In strict mode, certain actions that could be used maliciously are disallowed, such as assigning a value to the `eval` function, which can be used to execute arbitrary code.

Improving code quality. Strict mode helps catch common coding mistakes and encourages good coding practices.

To use `use strict` in your JavaScript code, simply include the string `use strict` at the beginning of your script or function. For example:

[1]:
```
"use strict";
function myFunction() {
  // Code in here will be executed in strict mode
}
```

[1]:
```
'use strict'
```

```
console.log(this) // returns window object

console.log(window) // also returns window object

console.log(this === window) // returns true

console.log("Hello") // returns "Hello"
```

[4]:
```
function myFunc(){
    "use strict"
    console.log(this);
}
```

```
//window.myFunc(); // returns window object
myFunc(); // returns undefined
window.myFunc(); // returns window object

// with strict mode, to access window object, we need to use window.myFunc()

function myFunc2(){
    //"use strict"
    console.log(this);
}
```

```
//window.myFunc(); // returns window object
myFunc2(); // returns window object
```

0.0.4 More on functions

[5]:
```
function hello(){
    console.log("Hello world");
}
```

```
hello(); // returns Hello world
window.hello(); // returns Hello world
hello.call(); // returns Hello world
```

0.0.5 call(), apply() and bind()

0.0.6 `call()`

`call()` method is used to call a function with a given this value and arguments provided individually

[7]:
```
const user1 = {
    firstName: "John",
    age: 8,
    about:  function(){
        console.log(this.firstName, this.age);
    }
}

const user2 = {
    firstName: "Jane",
    age: 9,
}
```

using ‘about()“ method of user1 object from user1 object

[8]:
```
user1.about.call(user2); // returns Jane 9
```

```
Jane 9
```

[9]:
```
user1.about.call(); // returns undefined undefined
```

```
undefined undefined
```

[10]:
```
user1.about.call(user1) // returns John 8
```

```
John 8
```

[11]:
```
user1.about() // returns John 8
```

```
John 8
```

[12]:
```
const  user3 = {
    firstName: "John",
    age: 8,
    about:  function(hobby, favMusician){
        console.log(this.firstName, this.age, hobby, favMusician);
    }
}

const  user4 = {
    firstName: "Jane",
    age: 9,
}
```

using ‘about()“ method of user3 object from user4 object

[13]:
```
// returns Jane 9 reading Michael Jackson
user3.about.call(user4, ["reading", "Michael Jackson"]);
```

```
Jane 9 [ 'reading', 'Michael Jackson' ] undefined
```

[14]:
```
user3.about.call(user4, "reading", "Michael Jackson", "reading"); // returns␣
 ↪Jane 9 reading Michael Jackson
```

```
Jane 9 reading Michael Jackson
```

[15]:
```
user3.about.call(user4, "reading"); // returns Jane 9 reading undefined
```

```
Jane 9 reading undefined
```

[16]:
```
user3.about.call(user4, "reading", "Michael Jackson")
```

```
Jane 9 reading Michael Jackson
```

[17]:
```
user3.about.call(user4, ["reading", "Michael Jackson"], "Michael Jackson"); //␣
 ↪returns Jane 9 reading Michael Jackson Michael Jackson
```

```
Jane 9 [ 'reading', 'Michael Jackson' ] Michael Jackson
```

[18]:
```
function about2(hobby, favMusician){
    console.log(this.firstName, this.age, hobby, favMusician);
}

const  user5 = {
    firstName: "Monty",
    age: 8,

}

const  user6 = {
    firstName: "Ronty",
    age: 9,
}
```

[19]:
```
about2.call(user5, "guitar", "Aiyub Bacchu"); // returns Monty 8 guitar Aiyub␣
  ↪Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

0.0.7 apply method

In JavaScript, the `apply()` method is a built-in function that allows you to call a function with a specified `this` value and arguments provided as an array. This method is similar to the `call()` method, but **the arguments are passed as an array rather than individually**.

The syntax for using the `apply()` method is as follows:

```
function myFunction(arg1, arg2, arg3) {
  // function body
}

myFunction.apply(thisValue, [arg1, arg2, arg3]);
```

The first argument passed to `apply()` is the value to be used as the `this` value inside the function, and the second argument is an array containing the arguments to be passed to the function.

Here's an example of how to use `apply()`:

[21]:
```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

const args = [1, 2, 3];
const result = myFunction.apply(person, args);
```

In the example above, we have an object `person` with a `fullName` function. We use `apply()` to set the `this` value of the `fullName` function to the `person` object, so that it can access the `firstName` and `lastName` properties of the object. The `args` array is then passed as the second argument to `apply()`.

The `apply()` method can also be used with functions that are not attached to objects, such as:

[23]:
```
function myFunction2(a, b, c) {
  return a + b + c;
}

const args2 = [1, 2, 3];
const result2 = myFunction2.apply(null, args); // 6
```

In the example above, we use `apply()` to call the `myFunction` function and pass `null` as the `this` value (since the function is not attached to an object), and pass `[1, 2, 3]` as the second argument. The result is `6`, which is the sum of the arguments.

[24]:
```
// using call
about2.call(user5, "guitar", "Aiyub Bacchu"); // returns Monty 8 guitar Aiyub␣
  ↪Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

[25]:
```
// using apply
about2.apply(user5, ["guitar", "Aiyub Bacchu"]); // returns Monty 8 guitar␣
  ↪Aiyub Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

0.0.8 bind()

In JavaScript, the `bind()` method is a built-in function that allows you to create a new function with a specified this value and pre-set arguments. This method is useful when you want to create a new function based on an existing function but with some of its arguments pre-set.

The syntax for using the `bind()` method is as follows:

```
function myFunction(arg1, arg2, arg3) {
  // function body
}

const boundFunction = myFunction.bind(thisValue, arg1, arg2);
```

The first argument passed to `bind()` is the value to be used as the `this` value inside the function, and subsequent arguments are the arguments to be pre-set in the new function. The `bind()` method returns a new function that can be called later.

Here's an example of how to use `bind()`:

[27]:
```
const person11 = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

const printName11 = function(prefix, suffix) {
  console.log(prefix + this.fullName() + suffix);
}

const boundPrintName11 = printName11.bind(person11, 'Mr. ', ' Jr.');
boundPrintName11(); // "Mr. John Doe Jr."
```

```
Mr. John Doe Jr.
```

In the example above, we have an object `person` with a `fullName` function and a `printName` function that takes two arguments. We use `bind()` to create a new function `boundPrintName` that is based on `printName` but with the `this` value set to `person`, and with the first argument pre-set to `'Mr. '` and the second argument pre-set to `' Jr.'`. When we call `boundPrintName()`, it logs `"Mr. John Doe Jr."` to the console.

The `bind()` method is particularly useful when you want to create a new function that can be passed as a callback to another function with some arguments pre-set.

The `call()` method sets the this value and immediately invokes the function, while the `bind()` method creates a new function with the `this` value set, but does not invoke the function immediately. Instead, the new function can be called later.

[29]:
```
printName11.call(person11, 'Mr. ', ' Jr.');
```

```
Mr. John Doe Jr.
```

[31]:
```
about2.call(user5, "guitar", "Aiyub Bacchu"); // returns Monty 8 guitar Aiyub␣
  ↪Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

[32]:
```
about2.apply(user5, ["guitar", "Aiyub Bacchu"]); // returns Monty 8 guitar␣
  ↪Aiyub Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

[33]:
```
const func = about2.bind(user5, "guitar", "Aiyub Bacchu"); // returns a function
func()
```

```
Monty 8 guitar Aiyub Bacchu
```

[34]:
```
about2.bind(user5, "guitar", "Aiyub Bacchu")();
```

```
Monty 8 guitar Aiyub Bacchu
```

[36]:
```
const user12 = {
    firstName: "Shammunul",
    age: 37,
    about: function(){
        console.log(this.firstName, this.age);
    }
}

user12.about(); // returns Shammunul 37
```

```
Shammunul 37
```

[38]:
```
// don't do the following mistake

const myFunc12 = user12.about; // this is not bounded yet
myFunc12(); // returns undefined undefined
```

```
// here this value is window object
```

```
undefined undefined
```

[39]:
```
// do the following if you have to save the function inside an object as a␣
↪variable
// if we want to bind this to user1 object, we need to use bind method

const myFunc13 = user12.about.bind(user12);
myFunc13(); // returns Shammunul 37
```

```
Shammunul 37
```

[40]:
```
// Without arrow function

// If the arrow function is not used, then this value will be the object itself␣
↪where the function is called

const user0 = {
    firstName: "Shammunul",
    age: 37,
    about: function(){
        console.log(this); // this is user0 object
        console.log(this.firstName, this.age);
    }
}

user0.about(); // returns Shammunul 37
```

```
{ firstName: 'Shammunul', age: 37, about: [Function: about] }
Shammunul 37
```

// arrow function

// arrow function takes this function from the surrounding // scope, so it is not bound to the function itself

```
const user13  = {
    firstName: "Shammunul",
    age: 37,
    about: () => {
        console.log(this); // this is window object
        console.log(this.firstName, this.age);
    }
}

// here this value is window object
user13.about(); // returns window followed by undefined undefined

// For arrow function, even if we use call method, it will not work
user1.about.call(user1); // returns window followed by undefined undefined
```

```
user1.about(user1); // returns window followed by undefined undefined
```

[44]:
```
/*
const user1  = {
    firstName: "Shammunul",
    age: 37,
    about: function(){
        //console.log(this); // this is window object
        console.log(this.firstName, this.age);
    }
}
*/

// The above code is equivalent to the following code

const user14  = {
    firstName: "Shammunul",
    age: 37,
    about(){
        //console.log(this); // this is window object
        console.log(this.firstName, this.age);
    }
}

user14.about();
```

```
Shammunul 37
```

0.0.9 Object oriented programming

[45]:
```
// Object oriented programming concept starts from here
```

[46]:
```
const user = {
    firstName: "Shammunul",
    lastName: "Islam",
    email: "sha_is113@yahoo.com",
    age: 37,
    address: "House 40/10, ABCD R/A, Sylhet, Bangladesh",
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    }
}
```

But, if we want to do this for different persons, it will become tedious. A more efficient way is described below.

0.0.10 Factory approach

Now we will create a function that will create object, add key value pairs and return that object.

[47]:
```
function createUser(firstName, lastName, email, age, address){
    const user = {};
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    user.about = function(){
        return `${this.firstName} is ${this.age} years old`;
    };
    user.is18 = function(){
        return this.age >= 18;
    };
    return user;
}
```

[49]:
```
const user17 = createUser("Shammunul", "Islam", "sha_is13@yahoo.com", 37, "ABC␣
↪R/A");
```

[50]:
```
console.log(user17);
```

```
{
  firstName: 'Shammunul',
  lastName: 'Islam',
  email: 'sha_is13@yahoo.com',
  age: 37,
  address: 'ABC R/A',
  about: [Function (anonymous)],
  is18: [Function (anonymous)]
}
```

[51]:
```
const is18 = user17.is18();
console.log(is18);
```

```
true
```

[52]:
```
const about = user17.about();
console.log(about);
```

```
Shammunul is 37 years old
```

Factory apporach is not a good approach, because it creates a new function for each object.

A better approach is to create a new object containing the two common methods.

[53]:
```
const userMethods = {
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    }
}

function createUser(firstName, lastName, email, age, address){
    const user = {};
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    user.about = userMethods.about; // we are not calling the function here␣
 ↪rather we are pointing to the address of the function
    user.is18 = userMethods.is18; // referencing to the function
    return user;
}
```

[55]:
```
const user18 = createUser("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
 ↪Dhaka");
const user19 =  createUser("Miron", "Manna", "miron@gmail.com", 44, "Dhanmondi,␣
 ↪Dhaka");
console.log(user18.about());
console.log(user19.about());
console.log(user19.is18())
```

```
Liton is 29 years old
Miron is 44 years old
true
```

0.0.11 A better approach

0.0.12 Object.create()

[57]:
```
const userMethods20 = {
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    },
    sing: function(){
        return "la la la la";
    }
}
```

[58]:
```
function createUser20(firstName, lastName, email, age, address){
    const user = Object.create(userMethods);
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
}
```

[59]:
```
const user20 = createUser("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
 ↪Dhaka");
const user21 =  createUser("Miron", "Manna", "miron@gmail.com", 44, "Dhanmondi,␣
 ↪Dhaka");
console.log(user20.about());
console.log(user21.about());
console.log(user21.is18())
```

```
Liton is 29 years old
Miron is 44 years old
true
```

[60]:
```
const obj1 = {
    key1: 'value1',
    key2: 'value2'
}

const obj2 = {
    key3: 'value3',
}

console.log(obj2.key3); // prints value3
console.log(obj2.key1); // prints undefined
```

```
value3
undefined
```

we want to see that if key1 is in obj3 and if it is not there go to obj1 and get the value of key1

[61]:
```
const obj3 = Object.create(obj1); // creates an empty object
// console.log(obj3);
obj3.key3 = "value3";
console.log(obj3.key1); // prints value1 from obj1
```

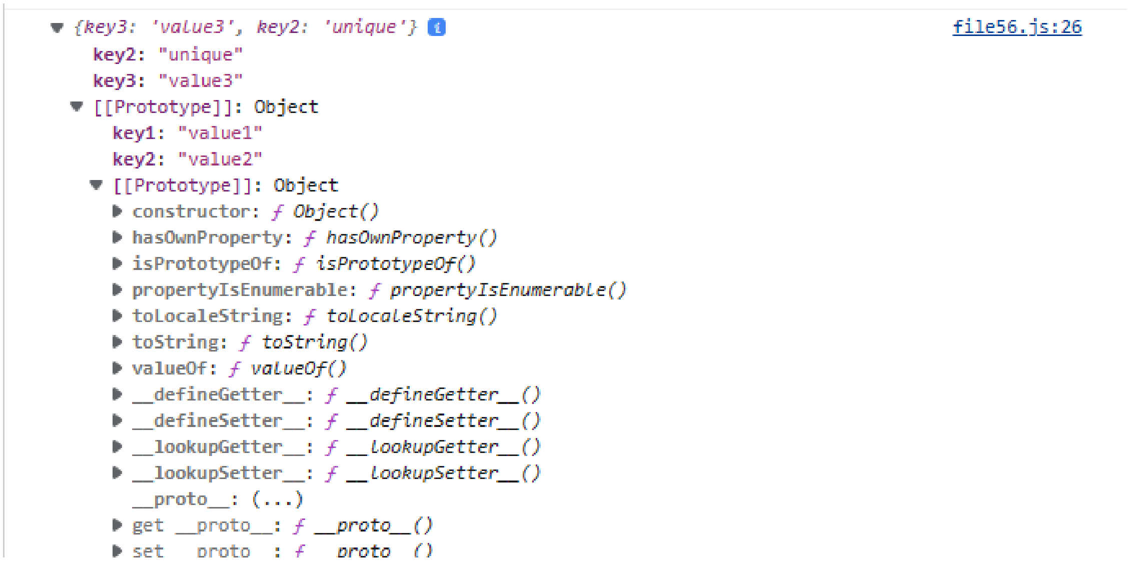
```
value1
```

[62]:
```
obj3.key2 = "unique";
console.log(obj3.key2); // prints unique
console.log(obj3);
```

```
unique
{ key3: 'value3', key2: 'unique' }
```

Note that `key1` is not here.

We will see the following in a browser if we see the output of `console.log(obj3)`.



```
▼ {key3: 'value3', key2: 'unique'}                                    file56.js:26
    key2: "unique"
    key3: "value3"
  ▼ [[Prototype]]: Object
      key1: "value1"
      key2: "value2"
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
        __proto__: (...)
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
```

`__proto__` is a property of an object also written as `[[prototype]]` in some versions of javascript. It is a reference to the object that is used as a prototype.

[63]:
```
console.log(obj3.__proto__); // prints { key1: 'value1', key2: 'value2' }
```

```
{ key1: 'value1', key2: 'value2' }
```

- `prototype` is different than `__proto__` or `[[prototype]]`
- `prototype` is a property of a function
- `__proto__` is a property of an object – also known as dunderscore proto
- `[[prototype]]` is a property of an object
- `__proto__` is a reference while prototype is an object of function
- `proto` is a reference to the chain being created

[66]:
```
const userMethods22 = {
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    },
    sing: function(){
        return "la la la la";
    }
}

function createUser22(firstName, lastName, email, age, address){
    const user = Object.create(userMethods22);
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
}
```

The line `const user = Object.create(userMethods22);` creates a new object user and sets its prototype to the userMethods object.

The `Object.create()` method creates a new object and sets its prototype to the object passed as an argument. In this case, we're passing the `userMethods22` object as the argument, so the `user` object will inherit all the properties and methods defined on the userMethods object.

By setting the user object's prototype to `userMethods22`, we're creating a "prototype chain". This means that when we call a method on the `user` object, JavaScript first looks for the method on the `user` object itself. If the method isn't found, JavaScript looks for the method on the user object's prototype (`userMethods22`). If the method still isn't found, JavaScript continues up the prototype chain until it reaches the Object.prototype object, which is the default prototype for all objects in JavaScript.

In this example, we're using the `Object.create()` method to create a new user object with its prototype set to `userMethods`. This means that the `user` object will inherit the `about()`, `is18()`, and `sing()` methods defined on `userMethods`, and we can call these methods on the `user` object as if they were defined directly on the `user` object itself.

[69]:
```
const user23 = createUser22("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
↪Dhaka");
const user24 =  createUser22("Miron", "Manna", "miron@gmail.com", 44,␣
↪"Dhanmondi, Dhaka");
```

[70]:
```
console.log(user23.about());
```

```
Liton is 29 years old
```

[71]:
```
console.log(user24.about());
```

```
Miron is 44 years old
```

[72]:
```
console.log(user24.is18())
```

```
true
```

[73]:
```
console.log(user23);
```

```
{
  firstName: 'Liton',
  lastName: 'Miah',
  email: 'liton22@yahoo.com',
  age: 29,
  address: 'Mirpur, Dhaka'
}
```

0.0.13 Prototype property of function

In JavaScript, every function is an object, and like any object, it has properties and methods. One of those properties is the prototype property. The prototype property of a function is used to add properties and methods to all instances of the function.

The prototype property is an object that will become the prototype of any objects created using the function as a constructor. This means that any properties or methods added to the prototype will be shared by all instances of the function.

[74]:
```
function hello(){
    console.log("Hello world!");
}
```

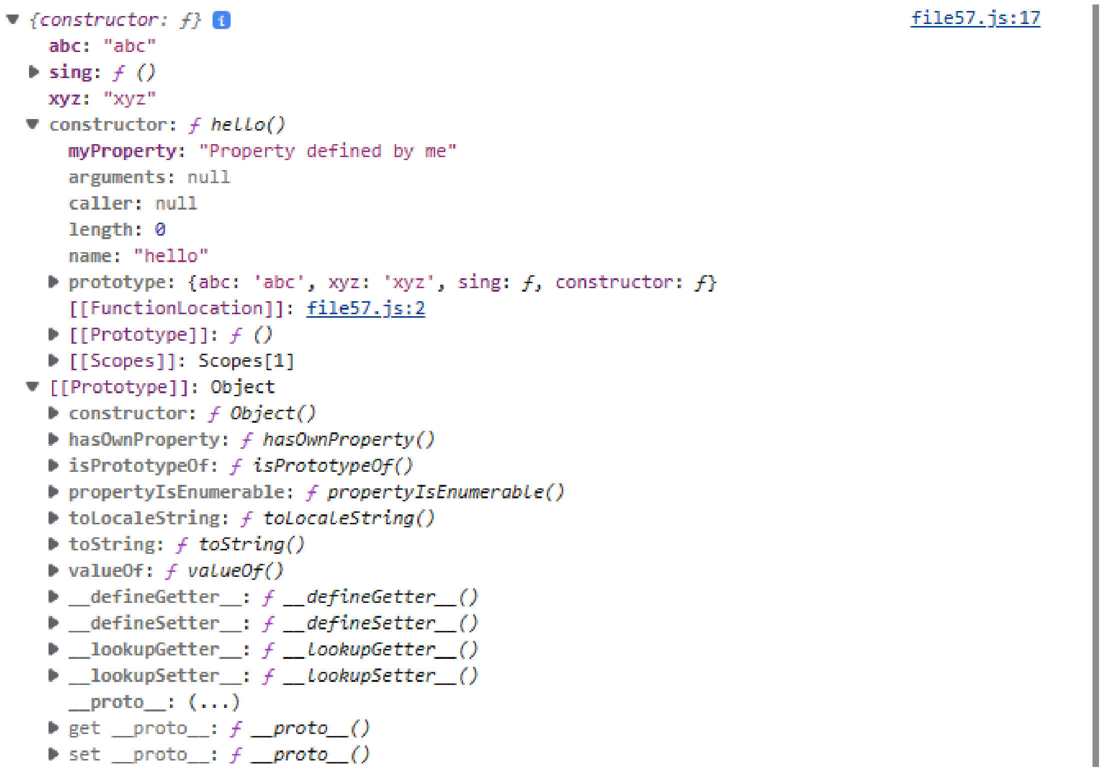
[75]:
```
// js function -- function + object

// console.log(hello.name); -- function name

// add your own properties

hello.myProperty = "Property defined by me";
console.log("hello.myProperty = " + hello.myProperty);
```

```
hello.myProperty = Property defined by me
```

[76]:
```
// only functions provide prototype property

console.log(hello.prototype); // returns constructor function
```

```
{}
```

We will see the following in a browser if we see the output of `console.log(hello.prototype);`.



```
▼ {constructor: ƒ} ℹ                                                   file57.js:17
    abc: "abc"
  ▶ sing: ƒ ()
    xyz: "xyz"
  ▼ constructor: ƒ hello()
      myProperty: "Property defined by me"
      arguments: null
      caller: null
      length: 0
      name: "hello"
    ▶ prototype: {abc: 'abc', xyz: 'xyz', sing: ƒ, constructor: ƒ}
      [[FunctionLocation]]: file57.js:2
    ▶ [[Prototype]]: ƒ ()
    ▶ [[Scopes]]: Scopes[1]
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
      __proto__: (...)
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
```

[77]:

```
hello.prototype.abc = "abc";
console.log(hello.prototype.abc);

hello.prototype.xyz = "xyz";
console.log(hello.prototype.xyz);

hello.prototype.sing = function(){
    return "la la la";
}
console.log(hello.prototype.sing());
```

```
abc
xyz
la la la
```

proto is property of object and it is a reference to an object

We will add the functionalities of the `userMethods` object to the prototype of the `createUser` function. `prototype` is simply an object

[78]:

```
// createUser is a constructor function
function createUser(firstName, lastName, email, age, address){
    // creates a new object user and sets its prototype to the createUser.
↪prototype object
    // __proto__ is being set to the prototype of the createUser function
    const user = Object.create(createUser.prototype); // create sets the␣
↪proto's value. Here it sets to prototype of createUser function
```

```
    // proto is a reference to the chain being created
    // In this case, proto will be an object

    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
}
```

The line `const user = Object.create(createUser.prototype);` creates a new object user and sets its prototype to the `createUser.prototype` object.

In JavaScript, every function has a special prototype property that is used as the prototype for objects created using the function as a constructor. When you create a new object using the new keyword and a constructor function, the object's prototype is automatically set to the constructor's prototype property.

In this example, we're creating a new object `user` that has its prototype set to `createUser.prototype`. This means that the `user` object will inherit any properties or methods defined on `createUser.prototype`.

The `Object.create()` method is used to create a new object and set its prototype to a specific object. In this case, we're passing `createUser.prototype` as the argument to `Object.create()`, so the `user` object will inherit from `createUser.prototype`.

By setting the user object's prototype to `createUser.prototype`, we're creating a prototype chain that allows us to define shared methods and properties for all objects created using the `createUser` constructor function.

In this example, `createUser.prototype` is an object that can be used to define shared methods and properties for all `user` objects created using the `createUser` constructor function. This means that any methods or properties defined on `createUser.prototype` will be inherited by all user objects created using the `createUser` constructor function.

[79]:
```
createUser.prototype.about = function(){
    return `${this.firstName} is ${this.age} years old`;
}

createUser.prototype.is18 = function(){
    return this.age >= 18;
}

createUser.prototype.sing = function(){
    return "la la la la";
}
```

[79]:
```
[Function (anonymous)]
```

[80]:
```
console.log(createUser.prototype);
```

```
{
  about: [Function (anonymous)],
  is18: [Function (anonymous)],
  sing: [Function (anonymous)]
}
```

[81]:
```
const user28 = createUser("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
  ↪Dhaka");
const user29 =  createUser("Miron", "Manna", "miron@gmail.com", 44, "Dhanmondi,␣
  ↪Dhaka");
console.log(user28);
console.log(user28.about());
console.log(user29.about());
console.log(user29.is18())

console.log(user28.sing())
```

```
createUser {
  firstName: 'Liton',
  lastName: 'Miah',
  email: 'liton22@yahoo.com',
  age: 29,
  address: 'Mirpur, Dhaka'
}
Liton is 29 years old
Miron is 44 years old
true
la la la la
```

0.0.14 new

`new` keyword – creates empty object

[82]:
```
function createUser(firstName, age){
    this.firstName = firstName;
    this.age = age;
}

createUser.prototype.about = function(){
    console.log(`${this.firstName} is ${this.age} years old`);
}
```

[82]:
```
[Function (anonymous)]
```

[83]:
```

```

```
// Object.create(createUser.prototype) -- this work is automatically done by␣
↪the new keyword
// But, we had to do it separately in the previous file
const user31 = new createUser('John', 25);
console.log(user31); // returns createUser {firstName: "John", age: 25}
```

```
createUser { firstName: 'John', age: 25 }
```

[84]:
```
// this is why, we don't need to do this "Object.create(createUser.prototype)"
// before calling the about method

console.log(user31.about())
```

```
John is 25 years old
undefined
```

0.0.15 Constructor function

In JavaScript, a constructor function is a special function that is used to create and initialize objects. The purpose of a constructor function is to provide a blueprint for creating new objects of a specific type.

To create a constructor function, you can define a function with a capitalized name, which conventionally indicates that it is a constructor function. Within the constructor function, you can use the "this" keyword to define properties and methods that will be associated with instances of the object created by the constructor.

[85]:
```
// constructor function name should start with capital letter and should be in␣
↪camel case

// constructor function
function CreateUser(firstName, lastName, email, age, address){
    // using new to create new empty object will set something like "this = {}"␣
↪here
    // that's why the following line is not needed
    //const user = Object.create(createUser.prototype)
    // and because of the new keyword, we can just use this to set the␣
↪properties
    // and this is different than what we have done
    // previously (or, in file58.js)
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.age = age;
    this.address = address;
    // we can write ""return this;"" here but it is not needed
}
```

[86]:
```
CreateUser.prototype.about = function(){
    return `${this.firstName} is ${this.age} years old.`;
};
CreateUser.prototype.is18 = function (){
    return this.age >= 18;
}
CreateUser.prototype.sing = function (){
    return "la la la la ";
}
```

[86]:
```
[Function (anonymous)]
```

[88]:
```
const user34 = new CreateUser('harshit', 'vashsith', 'harshit@gmail.com', 18,␣
 ↪"my address");
const user35 = new CreateUser('harsh', 'vashsith', 'harshit@gmail.com', 19, "my␣
 ↪address");
const user36 = new CreateUser('mohit', 'vashsitha', 'harshit@gmail.com', 17,␣
 ↪"my address");
console.log(user34);
console.log(user34.is18());
```

```
CreateUser {
  firstName: 'harshit',
  lastName: 'vashsith',
  email: 'harshit@gmail.com',
  age: 18,
  address: 'my address'
}
true
```

[89]:
```
for(let key in user34){
    console.log(key); // will print all the properties of user1, even in the␣
 ↪prototypes
}
```

```
firstName
lastName
email
age
address
about
is18
sing
```

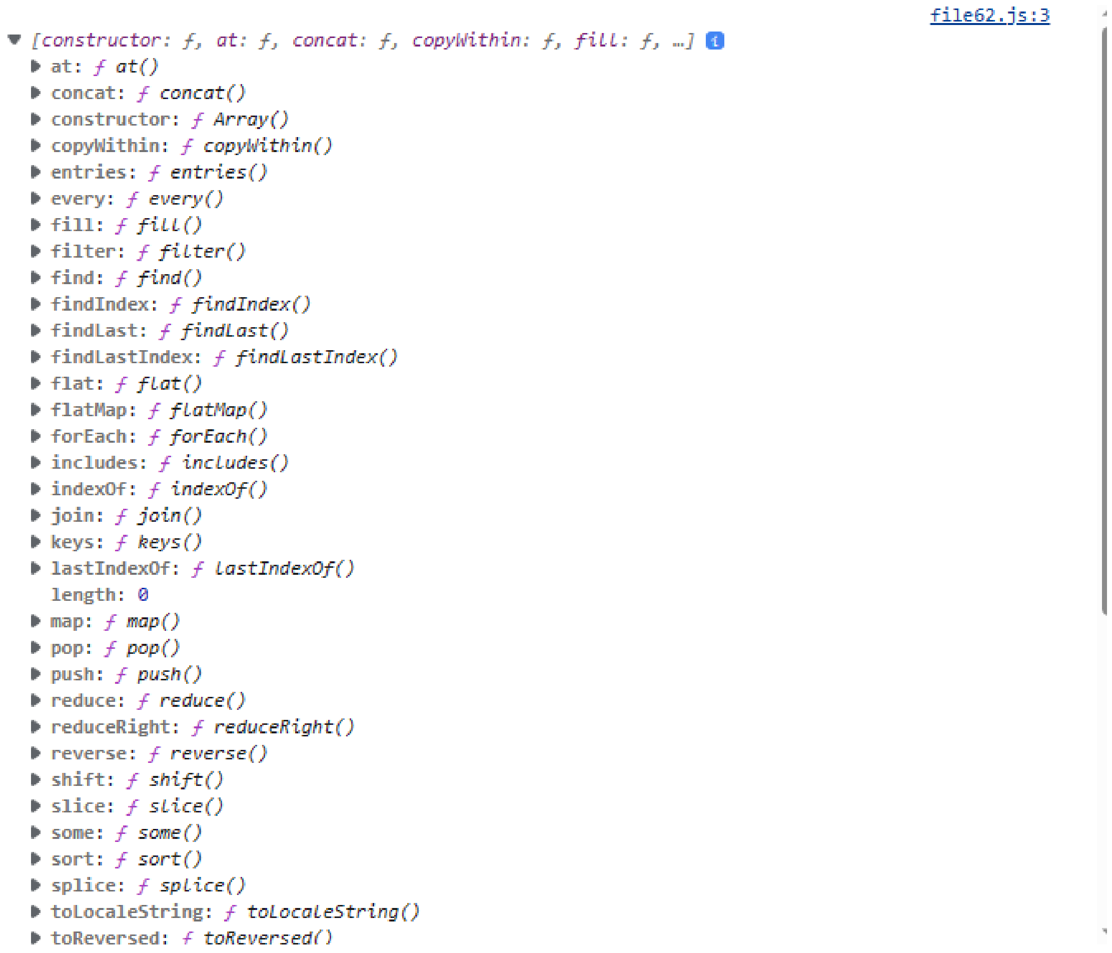
[91]:
```
// we can also use the following method to get all the properties of an object
// but this will not print the properties of the prototypes

for(let key in user34){
    if (user34.hasOwnProperty(key)){
        console.log(key);
    }
}
```

```
firstName
lastName
email
age
address
```

[92]:
```
let numbers = [1, 2, 3];
console.log(Array.prototype);
// In fact, Array is called internally by JS as new Array()
// and thus we get prototype of Array
```

```
Object(0) []
```

We will see the following in a browser if we see the output of `console.log(Array.prototype);`.



```
                                                                       file62.js:3
▼ [constructor: ƒ, at: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, …] i
  ▶ at: ƒ at()
  ▶ concat: ƒ concat()
  ▶ constructor: ƒ Array()
  ▶ copyWithin: ƒ copyWithin()
  ▶ entries: ƒ entries()
  ▶ every: ƒ every()
  ▶ fill: ƒ fill()
  ▶ filter: ƒ filter()
  ▶ find: ƒ find()
  ▶ findIndex: ƒ findIndex()
  ▶ findLast: ƒ findLast()
  ▶ findLastIndex: ƒ findLastIndex()
  ▶ flat: ƒ flat()
  ▶ flatMap: ƒ flatMap()
  ▶ forEach: ƒ forEach()
  ▶ includes: ƒ includes()
  ▶ indexOf: ƒ indexOf()
  ▶ join: ƒ join()
  ▶ keys: ƒ keys()
  ▶ lastIndexOf: ƒ lastIndexOf()
    length: 0
  ▶ map: ƒ map()
  ▶ pop: ƒ pop()
  ▶ push: ƒ push()
  ▶ reduce: ƒ reduce()
  ▶ reduceRight: ƒ reduceRight()
  ▶ reverse: ƒ reverse()
  ▶ shift: ƒ shift()
  ▶ slice: ƒ slice()
  ▶ some: ƒ some()
  ▶ sort: ƒ sort()
  ▶ splice: ƒ splice()
  ▶ toLocaleString: ƒ toLocaleString()
  ▶ toReversed: ƒ toReversed()
```

[93]:
```
console.log(Object.getPrototypeOf(numbers)); //
```

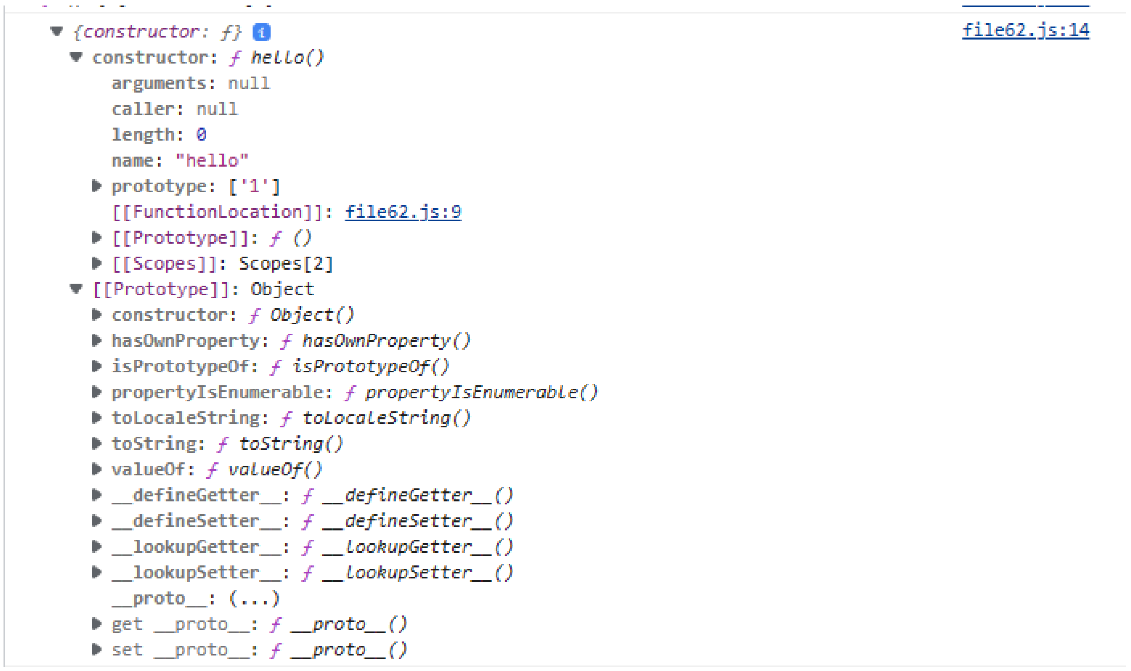
```
Object(0) []
```

The above code will also similar out put in the console as can be seen in the above image.

[94]:
```
function hello(){
    console.log("Hello");
}

console.log(Object.getPrototypeOf(hello)); //
```

```
{}
```

We will see the following in the console of a browser

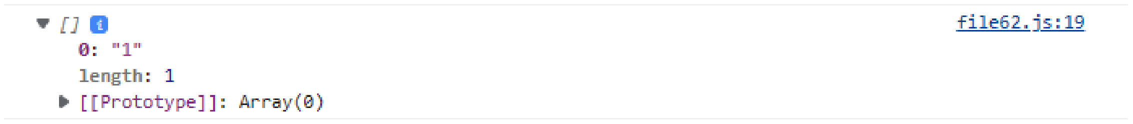


```
▼ {constructor: ƒ}                                        file62.js:14
  ▼ constructor: ƒ hello()
      arguments: null
      caller: null
      length: 0
      name: "hello"
    ▶ prototype: ['1']
      [[FunctionLocation]]: file62.js:9
    ▶ [[Prototype]]: ƒ ()
    ▶ [[Scopes]]: Scopes[2]
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
      __proto__: (...)
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
```

[95]:
```
console.log(hello.prototype);
```

```
{}
```

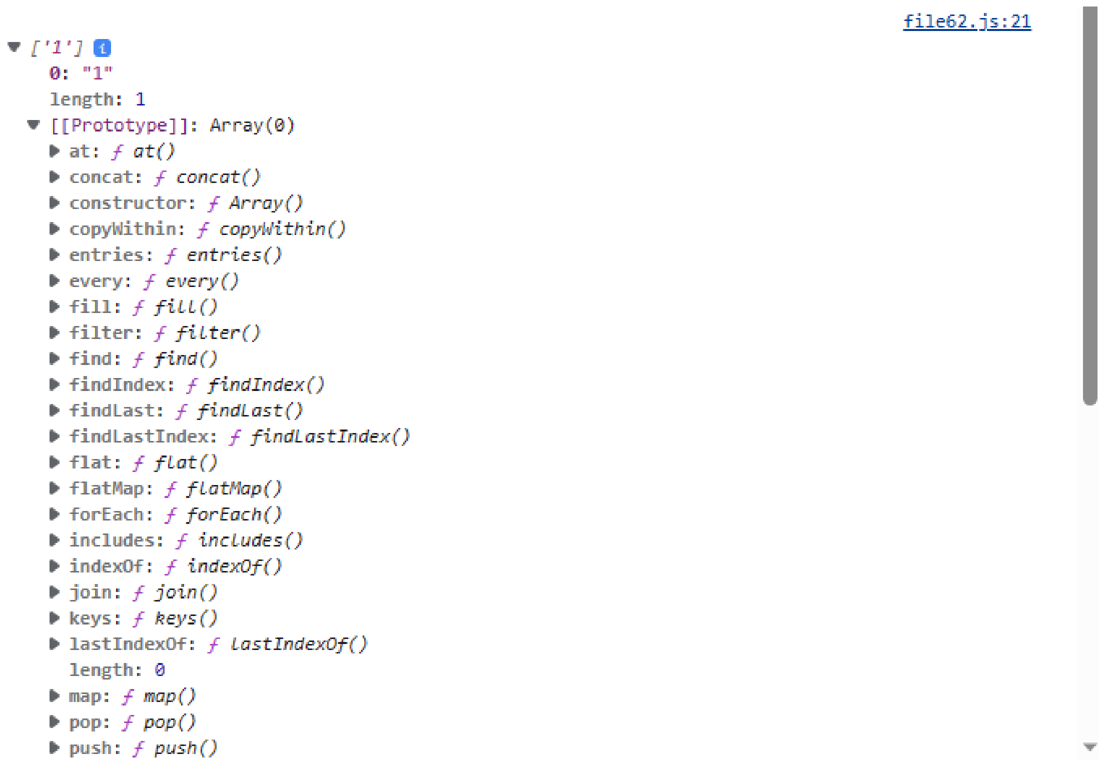
We will see the following in the console of a browser



```
▼ []                                                      file62.js:19
    0: "1"
    length: 1
  ▶ [[Prototype]]: Array(0)
```

```
hello.prototype.push('1');
console.log(hello.prototype);
```

The above will show the output below in the console of a browser:



```
file62.js:21
▼ ['1'] ℹ
    0: "1"
    length: 1
  ▼ [[Prototype]]: Array(0)
    ▶ at: ƒ at()
    ▶ concat: ƒ concat()
    ▶ constructor: ƒ Array()
    ▶ copyWithin: ƒ copyWithin()
    ▶ entries: ƒ entries()
    ▶ every: ƒ every()
    ▶ fill: ƒ fill()
    ▶ filter: ƒ filter()
    ▶ find: ƒ find()
    ▶ findIndex: ƒ findIndex()
    ▶ findLast: ƒ findLast()
    ▶ findLastIndex: ƒ findLastIndex()
    ▶ flat: ƒ flat()
    ▶ flatMap: ƒ flatMap()
    ▶ forEach: ƒ forEach()
    ▶ includes: ƒ includes()
    ▶ indexOf: ƒ indexOf()
    ▶ join: ƒ join()
    ▶ keys: ƒ keys()
    ▶ lastIndexOf: ƒ lastIndexOf()
      length: 0
    ▶ map: ƒ map()
    ▶ pop: ƒ pop()
    ▶ push: ƒ push()
```

We have already done this:

```
function CreateUser(firstName, lastName, email, age, address){
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.age = age;
    this.address = address;
    // we can write ""return this;"" here but it is not needed
}
CreateUser.prototype.about = function(){
    return `${this.firstName} is ${this.age} years old.`;
};
CreateUser.prototype.is18 = function (){
    return this.age >= 18;
}
CreateUser.prototype.sing = function (){
    return "la la la la ";
}


const user1 = new CreateUser('harshit', 'vashsith', 'harshit@gmail.com', 18, "my address");
const user2 = new CreateUser('harsh', 'vashsith', 'harshit@gmail.com', 19, "my address");
const user3 = new CreateUser('mohit', 'vashsitha', 'harshit@gmail.com', 17, "my address");
```

But, we can also do this using `class`.

classes are fake in JS, but they are useful for documentation

[98]:
```
class CreateUser30{
    constructor(firstName, lastName, email, age, address){
        this.firsName = firstName;
        this.lastName = lastName;
        this.email;
        this.age = age;
        this.address = address;
    }

    about(){
        return `${this.firstName} is ${this.age} years old.`;
    }

    is18(){
        return this.age >= 18;
    }

    sing(){
        return "la la la la";
    }
}
```

[99]:
```
const user41 = new CreateUser('harshit', 'vashsith', 'harshit@gmail.com', 18,␣
 ↪"my address");
const user42 = new CreateUser('harsh', 'vashsith', 'harshit@gmail.com', 19, "my␣
 ↪address");
const user43 = new CreateUser('mohit', 'vashsitha', 'harshit@gmail.com', 17,␣
 ↪"my address");
```

[100]:
```
console.log(user41.about());
```

```
harshit is 18 years old.
```

0.0.16 class inheritance

0.0.17 extends

In JavaScript, the `extends` keyword is used to create a new class that is a child or subclass of an existing class. This is known as class inheritance.

The syntax for creating a subclass using the `extends` keyword is as follows:

```
class Subclass extends Superclass {
  // subclass definition goes here
}
```

In this syntax, the `Subclass` is the new class that is being created, and `Superclass` is the class that the new class is inheriting from. The subclass definition goes inside the curly braces.

By using `extends`, the subclass inherits all of the properties and methods of the superclass, and can also add new properties and methods, or override existing ones. This allows for code reuse and helps to keep the code organized.

Here is an example of using `extends` to create a subclass of a Person class:

[104]:
```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log("Hello, my name is " + this.name + " and I am " + this.age + "␣
 ↪years old.");
  }
}

class Student extends Person {
  constructor(name, age, grade) {
    super(name, age);
    this.grade = grade;
  }

  study() {
    console.log(this.name + " is studying for the " + this.grade + " grade.");
  }
}

var john = new Student("John", 15, "9th");

john.greet(); // outputs "Hello, my name is John and I am 15 years old."
john.study(); // outputs "John is studying for the 9th grade."
```

```
Hello, my name is John and I am 15 years old.
John is studying for the 9th grade.
```

In this example, the `Student` class is created as a subclass of the `Person` class using the `extends` keyword. The `Student` class has a new property `grade` and a new method `study`, while still inheriting the `name`, `age` and `greet` method from the Person class.

The `super` keyword is used in the constructor of the `Student` class to call the constructor of the `Person` class and pass in the `name` and `age` arguments. This ensures that the `name` and `age` properties are initialized properly in the `Student` object.

[101]:
```
class Animal{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }

    eat(){
        return `${this.name} is eating`;
    }

    isSuperCute(){
        return this.age <= 1;
    }

    isCute(){
        return true
    }
}
```

[102]:
```
class Dog extends Animal{

}
```

[103]:
```
const tommy = new Dog("tommy", 3);
console.log(tommy);
console.log(tommy.isCute());
```

```
Dog { name: 'tommy', age: 3 }
true
```

[106]:
```
// super
class Animal3{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }

    eat(){
        return `${this.name} is eating`;
    }

    isSuperCute(){
        return this.age <= 1;
    }

    isCute(){
        return true;
    }
}

class Dog3 extends Animal3{
    constructor(name, age, speed){
        super(name, age)
        this.speed = speed;
    }

    run(){
        return `${this.name} is running at ${this.speed} km/hr`;
    }
}

// objects/instances

const tommy3 = new Dog3("tommy", 3, 44);
console.log(tommy3.run());
```

```
tommy is running at 44 km/hr
```

0.0.18 Getter and setter methods

In JavaScript, getter and setter methods are used to get and set the values of an object's properties. They are typically used within classes to provide a way to access and manipulate the internal state of an object, while still maintaining control over how that state is accessed and modified from outside the class.

Getter methods are used to get the value of a property, while setter methods are used to set the value of a property. They are defined using the `get` and `set` keywords respectively, followed by the name of the property. Here is an example:

[110]:
```
class Person4{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    // By using get
    // we can use fullName like property
    // now, we don't need to call it like function or use ()
    // person.fullName will give back the full name
    // we can call this later without ()
    get fullName(){
        return `${this.firstName} ${this.lastName}`;
    }

    // By using set, we can set the value of
    // the property
    // now we can call this like property
    // we can call this later without ()
    set fullName(fullName){
        const [firstName, secondName] = fullName.split(" ");
        this.firstName = firstName;
        this.lastName = secondName;
    }

    setName(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

[111]:
```
const person13 = new Person4("harshit", "vashsith", 18);
// console.log(person1.fullName()); // will give error

// will give the full name without using () -- this is due to the get method
console.log(person13.fullName);
```

```
harshit vashsith
```

[112]:
```
person13.fullName = "Motin Miah"; // this is due to the set method
console.log(person13);
console.log(person13.fullName);
console.log(person13.firstName);
```

```
Person4 { firstName: 'Motin', lastName: 'Miah', age: 18 }
Motin Miah
Motin
```

[118]:
```
person13.setName("Poran Theon", "Dilan")
```

[119]:
```
console.log(person13);
console.log(person13.fullName);
console.log(person13.firstName);
```

```
Person4 { firstName: 'Poran Theon', lastName: 'Dilan', age: 18 }
Poran Theon Dilan
Poran Theon
```

[117]:
```
person13.fullName = "Thierry Henry"
console.log(person13);
console.log(person13.fullName);
console.log(person13.firstName);
```

```
Person4 { firstName: 'Thierry', lastName: 'Henry', age: 18 }
Thierry Henry
Thierry
```

0.0.19 Static methods and properties

In JavaScript, static methods and properties are class-level members that are associated with the class rather than with instances of the class.

Static methods and properties are declared using the `static` keyword within a class declaration. Here's an example:

[120]:
```
class MyClass {
  static myStaticProperty = 42;

  static myStaticMethod() {
    console.log('Hello from my static method!');
  }
}
```

In the example above, `myStaticProperty` and `myStaticMethod` are both defined as static members of the `MyClass` class.

Static properties can be accessed using the class name directly, like this: `MyClass.myStaticProperty`.

Static methods can also be called directly on the class, like this: `MyClass.myStaticMethod()`.

One common use case for static methods and properties is when you want to define utility functions or constants that are not tied to any particular instance of the class. Another use case is when you want to implement factory methods that create instances of a class in a specific way.

Here's an example of using a static method to create instances of a class:

[123]:
```
class Person14 {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  static createFromFullName(fullName) {
    const [firstName, lastName] = fullName.split(' ');
    return new Person14(firstName, lastName);
  }
}

const john14 = Person14.createFromFullName('John Smith');
console.log(john14.firstName); // Output: John
console.log(john14.lastName); // Output: Smith
```

```
John
Smith
```

In this example, the `createFromFullName` method is a static method that creates instances of the `Person14` class from a full name string. The method is called directly on the `Person14` class, and it returns a new instance of the class.

[124]:
```
// static method and properties

// they are related to class

// static methods and properties are not related to the object

class Person15{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    static classInfo(){
        return "This is a class for Person";
    }

    static desc = "static properties";

    get fullName(){
        return `${this.firstName} ${this.lastName}`;
    }

    set fullName(fullName){
        const [firstName, secondName] = fullName.split(" ");
        this.firstName = firstName;
        this.lastName = secondName;
    }
}

const person15 = new Person15("harshit", "vashsith", 18);
const info = Person15.classInfo();
console.log(info);
```

```
This is a class for Person
```

[126]:
```
console.log(Person15.desc);
```

```
static properties
```

In the console of a web browser, `console.log(Person15.desc)` will show **this is person class**.

// We can't do the following with static method

`person15.classInfo() // this will throw error`

Static methods help in application initialization.

[ ]:
```

```