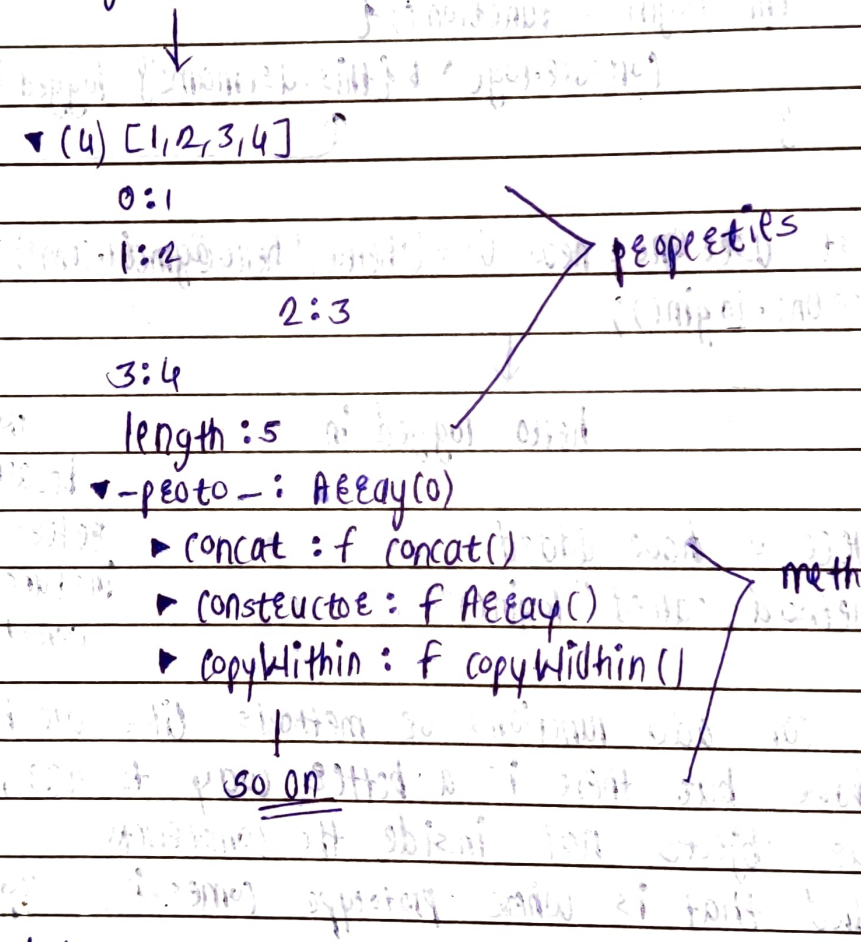
Prototype Model

- all methods are inside the "proto" property

example :-

```
- const num = [1,2,3,4];
- console.log(num);
```

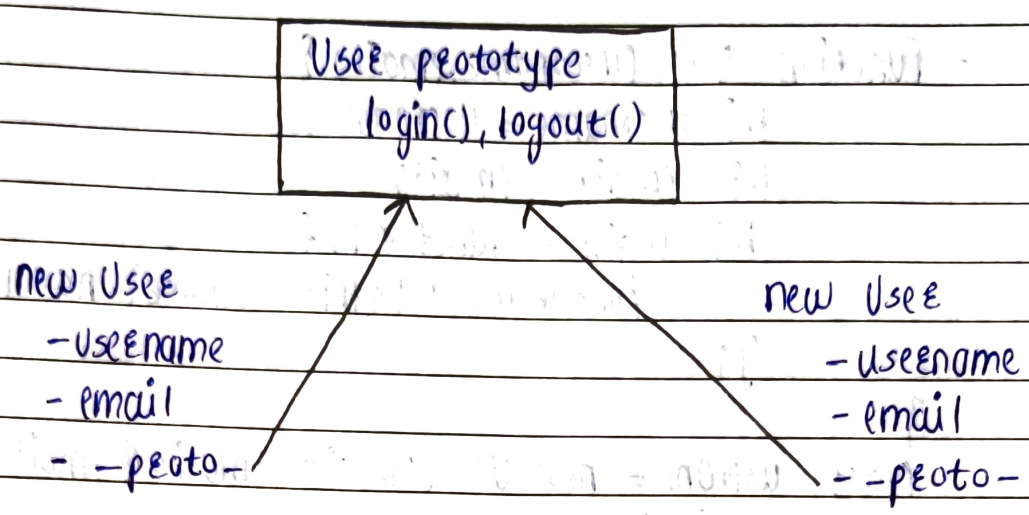


- Prototypes

Every object in Javascript has a prototype. Prototypes contain all the methods for that object type.

example :-

Date prototype	Array Prototype
getDay()	sort()
getMonth()	filter()
etc...	etc....



usee peopeeties will stoee diseectly on object when we ceate them things like useename and email. but the usee object will also have a usee peototype and we want to be able to all of ous methods available to every usee on usee peototype. so that the peoto peopeety of each usee that we ceate points to usee peototype which has accessed to all of the methods e.g. heee we have login() and logout().

so this approach is stoeing the methods on a peototype instead of every individual object that we ceate. on

And it has two main benefits. since we aee stoeing

fiest of all it is moee efficient ... them in one single place once and not many different locations on every single instance that we ceeate and second, it will help us with peototypal inheeitance.

```
function Usee(username, email) {
    this.username = username;
    this.email = email;
    this.login = function() {
        console.log(`${this.username} has logged in`);
    };
}
const usrOne = new User('mario', 'mario@gmail.com');

console.log(usrOne);
```

↓

▼ Usee {username: "mario", email: "mario@gmail.com", login: f}
  email: "mario@gmail.com"
  username: "mario"
  ▶ login: f()
  ▼ __proto__:
    ▶ constructor: f Usee(username, email)
    ▶ __proto__: object

Here we can see that login() function/method is not actually stored inside "__proto__" instead it stored directly on the object and that's because we have defined that directly in the constructor.

So it would be better, if we could define it so that we store inside proto property

and to do that we have to stored the method on the prototype.
when we define methods in a class, javascript automatically took those methods and added them inside "__proto__" property
so that's what we need to do when we are not using class, we need to figure out a way, take our methods and store them inside "__proto__" property.

[in short, if we took our example then login() method we want to have in "__proto__" property]

- just directly moving to our main example, consider below example first

- console.log(Date.prototype);

▼ {constructor: f, toString: f, toDateString: f ...}
  ▶ constructor: f Date()
  ▶ getDate: f getDate()
  ▶ getFullYear: f getFullYear()

and so on

```
function Users(username, email){
    this.username = username;
    this.email = email;
    this.login = function(){
        console.log(`${this.username} logged in`);
    };
}
const useeOne = new Users('masio', 'masio@gmail.com);
console.log(Users.prototype);
```

```
▼ {constructor: f}
    ▶ constructor: f Users(username, email)
    ▶ __proto__: Object
```

```
function Users(username, email){
    this.username = username;
    this.email = email;
}
Users.prototype.login = function (){
    console.log(`${this.username} logged in`);
};

Users.prototype.logout = function(){
    console.log(`${this.username} logged out`);
};
const useeOne = new Users('hello', 'hello1@gmail');
console.log(useeOne);
```

▼ User {username: "hello", mail: "hello1@gmail"); }
email: "hello1@gmail"
username: "hello"
▼ __proto__:
▶ login: f()
▶ constructor: f User(username, email)
▶ __proto__: object

Here we can see now, we have added login() method to "proto" property.
Our login() method for all users is now been stored in one single place on user prototype and if we want to use still we can use it

below console.log(userOne); write below
userOne.login();
↓
hello logged in

[Note that we have deleted login() from constructor because now we no longer want to store them directly instead we want to store that in "proto" property]

We can also do method chaining currently if we do that (we will get eeeoe) i.e.

```
UseeOne.login().logout();
```

↓

⊗ ▶ Uncaught TypeEEEOE : cannot read peopeety 'logout' of undefined

and this why because we have not return "this"

```
- function User(username, email) {
      this.username = username;
      this.email = email;
  }
  User.prototype.login = function () {
      console.log(`${this.username} has logged in`);
  };     return this;
  User.prototype.logout = function() {
      console.log(`${this.username} has logged out`);
      return this;
  };
  const userOne = new User('hello', 'hello1@gmail.com');
   UseeOne.login().logout();
```

↓

hello has logged in
hello has logged out

So remember this is exactly the same happening underthehood when we use class syntax.

before the class keyword came about in javascript, this is how we do using constructor functions and prototype model.

Prototypal inheritance

```
// constructor functions
function User(username, email){
    this.username = username;
    this.email = email;
};
User.prototype.login = function(){
    console.log(`${this.username} has logged in`);
    return this;
};
User.prototype.logout = function(){
    console.log(`${this.username} has logged out`);
    return this;
};

function Admin(username, email, title){
    User.call(this, username, email);   ——*
    this.title = title;
}
Admin.prototype = Object.create(User.prototype);   ——* *

Admin.prototype.deleteUser = function(){   [added new method to "Admin"]
    // delete a user
};
const userOne = new User('hello', 'hello@gmail.com');
const userTwo = new User('no', 'no@gmail.com');
const userThree = new Admin('man', 'man@gmail.com', 'ola');
console.log(userThree);
```

↓

▼ Admin {useename: "man", email: "man@gmail.com", title: "ola"}
email: "man@gmail.com"
title: "ola"
useename: "man"
▼ -peoto-: Usee
▶ deleteUsee: f ()
▼ -peoto-:
▶ login: f ()
▶ logout: f()
▶ consteuctoe: f Usee(useename, email)
▶ -peoto-: object

(fiest peoto)
(fiest level)
(inside 2nd peoto)
(two level deep)

↓

Now heee in this example we have not used "class" instead we have used consteuctoe function.
Heee "Usee" has two propeties "useename" and "email". "Usee" has two methods "login()" and "logout". while "Admin" has three propeties "useename", "email" and "title". "Admin" also has theee methods called "login", "logout" and "deleteUsee". we cannot use "deleteUsee" in "Usee".
in "class" when we weee inheeiting peopeeties, at that time, we have use theee "supee()" function and that function call the consteuctoe of the paeent class.
Now heee in this example we can't use that.
so we have used → ⊛ i.e. we have used "call()" method
and in call() method the fiest aegument is

the context of what "this" keyword will be equal to inside the user constructor function.

So, for inheriting the properties of "User" into "Admin" we have used — ⊛ this line.

similarily now for inheriting methods/functions we have used — ⊛⊛ this line
when we were inheriting methods/functions in "class" i.e. in class inheritance example, we were not doing anything, it was by default inheriting.

⇒ A lot of the time you won't be using this type of working instead you will use javascript classes, but for understanding purpose it is really very important.