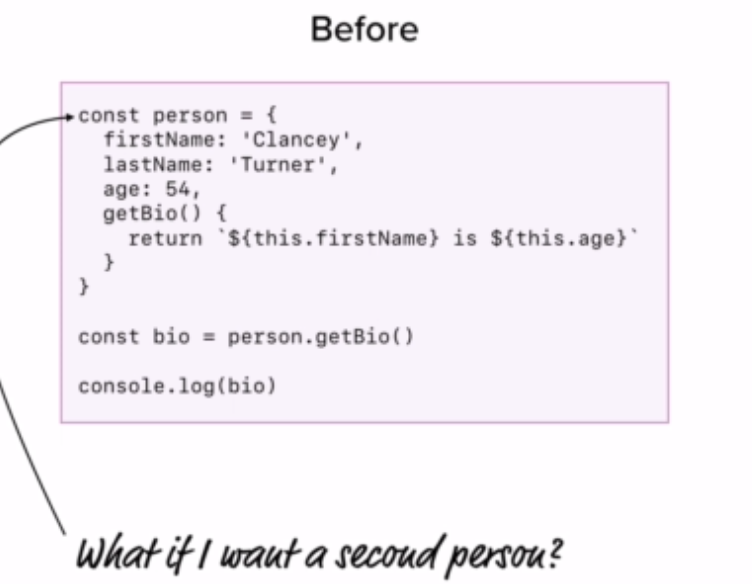
88)Section Intro:Advanced Objects and Functions

In this section we will see advances concepts of objects and functions. We will see how we can combine them to do some pretty interesting stuff. Here we will see object oriented programing and how it works in javascript. Javascript does things differently than most traditional languages.

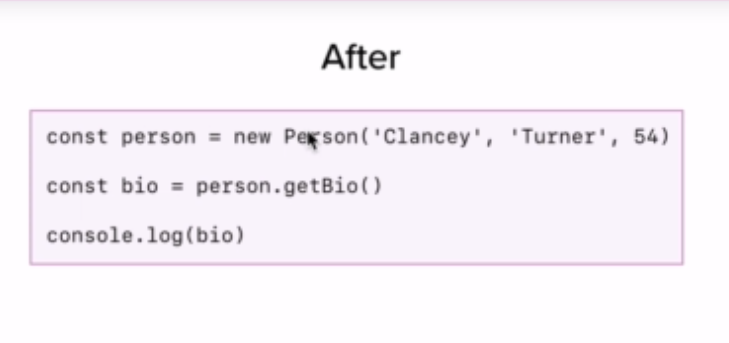
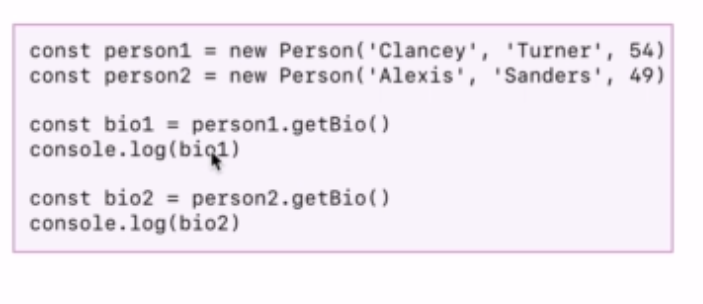
89)Object oriented programming

Here we will see how object oriented programming might be useful. Oriented programming means programming that’s based around some sort of concept. Objects are something, we already know about. W ehave used objects dozens of times in this calss to model real word things. We have used objects to model todo things, notes, cars and person etc. all these can be modeled with various set of properties. So object oriented programming is going to look pretty similar to some of stuff we have already been doing but with a slight twist. The twist is focused on code reusability.



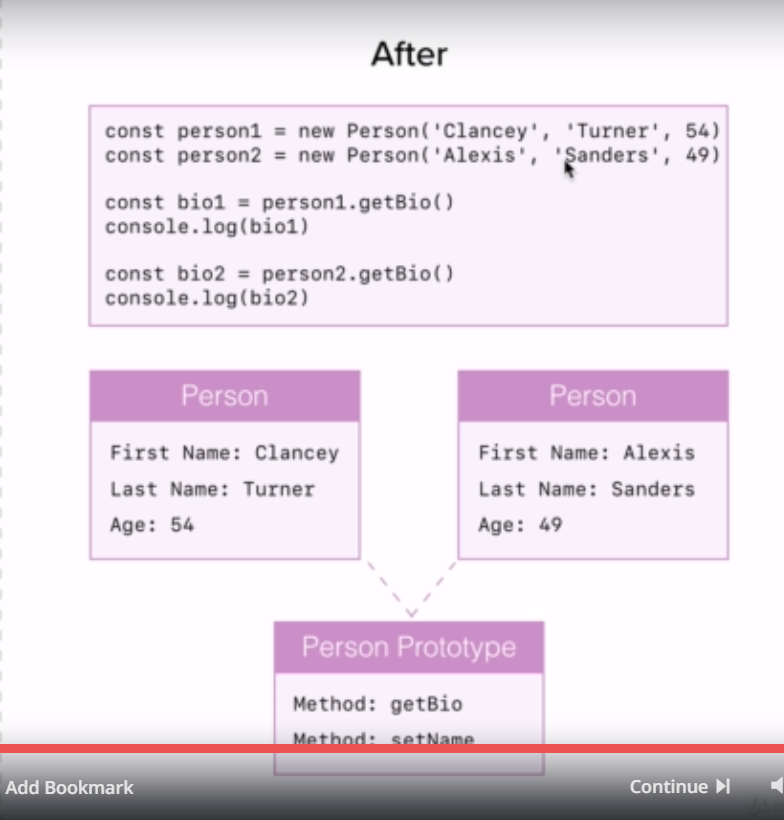
This is something we already know about. Lets say we want to second and 3rd person or someone gave me a list and I had to generate a new person for each one to take advantage of those methods like getBio.

This can be done like this-

we gave function, that serves as template of person. It allows us to easily generate new people and reuse various pieces of functionality like method getBio. We do not have refine everything for each person. We sued new keyword, when we worked with dates.

Lets visualize this. Down below what we really have is 2 people- peroson1 and person2. When we make new version of something we call that an instance. So in this case I have 2 person instances. Now what you will notice about each person is that they come with a standard set of values, we have attributes like firsName ,lastname and age. We have that for all instances. This is all we have in our current apps. In todo we have liest of objects , they all have same pieces of data.

Now what is missing from this visualization is various methods. On code we see that we can call method getBio on all instance. To add this to our visualization, we are going to create a separate box, called person prototype. Person prototype is going to define what all of people should look like. In this case it’s setting up the methods avalaible for each person. We can have various methods like setName to setname. So this is going to be linked like this, every person that we are going to create is going to have access to same set of methods. These methods are going to work with instance data. So getBio is going to be unique for each person depending upon what their first and age values actually are. So what we are seeing here is couple of objects with same properties and different values and somehow magicaly they get a shared set of methods.



90)Constructor Functions

const person = function() {

};

const me = person();

console.log(me);

output – undefined

if we add new i.e-

const person = function() {

};

const me = new person();

console.log(me);

output - person {}

we are getting function name followed by curly braces, with nothing inside. This is our very first custom object type. Object type gets its name from function and currently we have’nt set any values. That’s why there’s nothing showing up right here. What we have done is we have created whats known as constructor function, functions that get used with **new**  operator are constructor functions.

Lets see why we got different resuts, after using new keyword. Behind scenes new operator does very important things, first thing it does is, it generates a new empty oject for this new instance then it gives us access to this empty object in constructor function vi this value. Lets print this-

const person = function() {

console.log(this);

};

const me = new person();

console.log(me);

**person {}**

**person {}**

so this refers to new empty project. Now the only reason we have access to it inside constructor function is to start customizing it. So we can define that our object will have these properties. Like this-

const person = function(firstName) {

this.firstName = firstName;

};

const me = new person('Sumeet');

console.log(me);

one more thing- name of constructor function starts with capital letter. We are not returning anything form constructor function. new object is returned automatically. Tis is also done by new keyword. We can overwrite this behavior be returning a custom value. Code-

const person = function(firstName) {

this.firstName = firstName;

return {name: 'sood'};

};

const me = new person('Sumeet');

console.log(me);

in this case our custom object is returned, which is stored inside me. But this is not what we want. So we do not use return inside constructor functions.

One important thing to note is that we cannot use arrow functions as constructor functions. This is because arrow functions do not bind this.

91)Setting up Prototype Object

Here we will see how we can define a function, which is shared by objects created from our constructor function. so this is inheritance. All objects created from our constructor function will inherit this function.this is prototypal inheritance. Code-

const Person = function(firstName, lastName, age) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

};

Person.prototype.getBio = function() {

return `${this.firstName} is ${this.age}`;

};

const me = new Person('Sumeet', 'Sood', 27);

console.log(me.getBio());

const person2 = new Person('Nitesh','Ganotra', 28);

console.log(person2.getBio());

so this is how we setup things that are same for all objects.

One more thing about arrow functions.

Person.prototype.getBio = function() {

let bio = `${this.firstName} is ${this.age}.`;

this.likes.forEach((like) => {

// bio = bio + ` ${this.firstName} likes ${like}`;

bio += ` ${this.firstName} likes ${like}.`;

});

return bio;

};

Here note that to forEach function we passed a arrow function. but inside arrow function we used this keyword. Note that arrow functions don’t bind this value, it uses this value whatever its parent has. If we use regular function instead of parent function, then our code will fail. because regular functions have different scope of this. So it is usecase where arrow functions are useful.

92)Hangman challenge- Part 1

94)Primitive and Objects: Part 1

We can access methods on strings. But how is this possible? Because methods are only avalaible on obejcts.

const product = {

name: 'Influence'

};

console.log(product.hasOwnProperty('sumit'));

so we can check whether object has some property by using this function. now question is why name of this property is **hasOwnProperty**? It can be **hasProperty** . so why **hasOwnProperty.**

But if we do this-

const product = {

name: 'Influence'

};

console.log(product.hasOwnProperty('hasOwnProperty'));

it means that hasOwnProperty does not exist on product property, means our code should fail. but its not the case. So open mdn documentation for this property. In docs title is-

**Object.prototype.hasOwnProperty()**

So internally when we are using these data types, we are accessing those shared methods like filter, split and others such as hasOwnProperty via prototype inheritance. It is like this-

Each object or primitive type that we create is created by calling Object function in constructor mode. This object has property called prototype which points to prototype of this function. methods like hasOwnproperty exists on this prototype. That is we can access properties on strings. That is why person has property called **hasOwnProperty**, that we have not defined.

Now we will see why its **hasOwnproperty** instead of **hasProperty**. hasOwnProperty returns true if that property exists of objects itself not some where in chain of prototypes.

Using browser we can actually visually explore the prototype chnain for anything by dumping it to console. So that’s why I switched us over to browser right here. Code-

const product = {

name: 'Influence'

};

console.log(product);

output –

Object {name: “Influence”}

When we are making objects by calling functions in constructor mode, we were seeing name of constructor functions instead of Object. That proves that this object is created by calling Object function in constructor mode. If click on our object, we will see this property also-

**\_\_proto\_\_**, I told you that thing that links your instance to prototype property is this guy- [[ptototype]] , but this is for internal use. Browser exposes this property so that ou can view the prototype chain. When we click on \_\_proto\_\_ property, we can see it is object with many properties, one of these is hasOwnProperty, so this explains why we were able to use this property on our object event though object was not having it. Other thing is that Object function’s prototype has \_\_proto\_\_ property tat points to null. So chain ends here.

We can overwrite the properties on Object.Prototype function. we can also add new properties like this-

const product = {

name: 'Influence'

};

Object.prototype.sumit = () => 'This is new property'

console.log(product.sumit());

so these 2 ways of creating the object are similar-

const product = {

name: 'Influence'

};

const product = new Object();

product.name = 'Influence';

shortcut for second one is –

const product = new Object({

name: 'Sumeet'

});

Note that this is object literal syntax-

const product = {

name: 'Influence'

};

Internally js understands this code and it creates the object by calling Object function in constructor mode.

So in this video we explored the prototype chain for objects. In next video we will explore prototype chain for everything else, for ex on arrays we can call filter method. How can we do it?

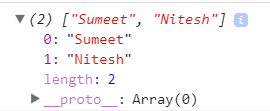
95)Primitives and Objects: Part 2

Primitives value is a value that does not have properties, its not an object. There are 5 primitive values that we already know about. these are number, string, Boolean, null and undefined. Everything not in this link is object. This means arrays and functions are also objects. So we will see how arrays and functions use prototypal inheritance to allow us to access that shared list of functionality. Lets print an array to console-

const team = ['Sumeet', 'Nitesh'];

console.log(team);

Output-



In additional to length property we have \_\_proto\_\_ property to allow us to see that hidden chain. On expanding it we get Array.prototype object. Here we can see all methods that we have seen before like filter,map , find, forEach etc.



That is why all our arrays have access to that shared set of methods. These methods are on Array.prototype. just like methods of Object.ptototype we can chage or manupilate them. Just like objects there is also a constructor function we could use but once again we are never going to do that. We will use literal syntax to create arrays. Now there is one difference. Object.prototype had a prototype that points to null. But that is not the case with Array.prototype. at the bottom of Array.prototype object we have \_\_proto\_\_ property when we click on that we get all the properties that were in Object.prototype. now we know that Object.prototype does not have any prototype. So it means that Array.prototype has \_\_proto\_\_ property that points to Object.protype.

So chain is like this-

myArray -> Array.prototype -> Object.prototype -> null

here myArray is variable which holds a array value. When we try to acces property on myArray then if it has that property then it returns it, otherwise it looks into Array.prototype Object. And chain goes on like this. So when we say that array is technically a object then it what I am talking about. So array is nothing more than customized version of object. We can prove that by accessing hasOwnProperty on aray variable, we know that this property only exist on objects Code-

const team = ['Sumeet', 'Nitesh'];

console.log(team.hasOwnProperty('length'));

we get true. If we do this-

console.log(team.hasOwnProperty('filter'));

we get false, because filter property exist on Array.prototype. it does not exist on array variable. There is also a constructor function for arrays . we can use that to generate array instead of using literal syntax.

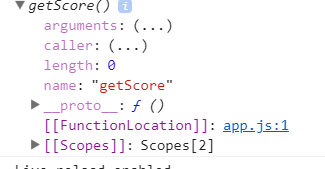
const team = new Array(['sumeet', 'archana']);

now lets move to next type that is function. code-

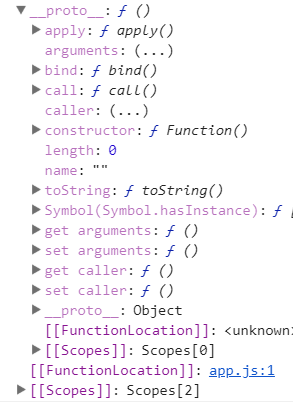
const getScore = () => 1;

console.dir(getScore);

on cosole we can see that we \_\_proto\_\_ property.



This is where we get access to all of shared behavior from functions. There are some methods listed here which we will explore in course.



As we can see this also has a \_\_proto\_\_ property that links to Object. So here is protype chain for function-

myFunc -> Function.prototype -> Object.prototype -> null

so when I say that arrays and functions are objects, this is what I am talking about.

Now lets move to primitive types. There are 2 groups here- first are null and undefined. These are truly primitive values, we will never access a property or method on them. But with string, number and boolen its little different because we know that we have definitely accessed methods on strings. So how can I say that string has a method and same time say that it is not a object. To explain this behavior we will dive into a example-

This is what happens behind the scenes- when we access property on string it converts it to an object. So strings, numbers and Booleans are indeed primitives but they also have what’s known as object wrapper. This object wrapper is what gives it all of the functionality that we are seeing here. For example the split method for string. So lets have alook exactly how that works. Lets run this code-

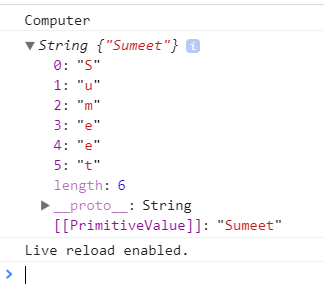
const product = 'Computer';

console.log(product);

const otherProduct = new String('Sumeet');

console.log(otherProduct);

output-



We can see that when we created string using literal syntax, we just get value on console. But when we created string by calling function in constructor mode, we get object on console. Now this object has properties as you can see. It also has \_\_proto\_\_ property. Lets expand it



This has all the properties that we were accessing on strings. So when we create string using literal syntax and we try to access property on it, then it is converted into object thatw e have seen above. So in our code we will use literal syntax to create string and we will let javascript to let us do conversion for us. Now \_\_proto\_\_ property also has \_\_proto\_\_ property. It points to Object.prototype. so this is why strings can be classified as objects as well. We have same conversion happening for number and boolean behind the scenes. So in case of number we have-

myNumber(variable) -> Number.protoype -> Object.prototype -> null

for Boolean-

myBoolean -> Boolean.prototype -> Object.prototype -> null

On web if you will read too much about prototype, you will hear someone saying that everything in javascript is an object. They are kind of right but not quite. there are primitive values even though some of these primitive values do have Object counterparts. But null and undefined will never end up being objects.

96)Hangman Challenge: Part 2

Here challenge was we want to press key from keyboard then we want to check that whether that key exists in puzzle. Here we want to identify alphabets only. We have to ignore numbers, enter, shift ,alt control etc.

97)Hangman Challenge : Part 3

Here we used every function on arrays. Ex-

const finished = this.word.every((letter) => this.guessedLetters.includes(letter));

this method will return true only if each element of array pass the test.

98)Hangman Challenge: Part 4

Here we saw that we can use join method to covert an array into string.

99)The Class Syntax

Here we will see class syntax which is alternate way to create to create constructor function with its methods like we done before. So we are going to explore new syntax and its not going to change functionality. So hangman game will work same. So new syntax is just syntactical sugar. Behind the scenes it is doing same thing as before. So behind the scenes we will still have constructor function and we still add methods onto its prototype property.

So we will convert into new syntax-

const Person = function(firstName, lastName, age, likes = []) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.likes = likes;

};

Person.prototype.getBio = function() {

let bio = `${this.firstName} is ${this.age}.`;

this.likes.forEach((like) => {

// bio = bio + ` ${this.firstName} likes ${like}`;

bio += ` ${this.firstName} likes ${like}.`;

});

return bio;

};

Person.prototype.setName = function(fullName) {

const names = fullName.split(' ');

this.firstName = names[0];

this.lastName = names[1];

};

const me = new Person('Sumeet', 'Sood', 27, ['Teaching', 'Biking']);

me.setName('Nitesh Ganotra');

console.log(me.getBio());

const person2 = new Person('Nitesh','Ganotra', 28);

console.log(person2.getBio());

here is new version-

class Person {

constructor(firstName, lastName, age, likes = []) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.likes = likes;

}

getBio() {

let bio = `${this.firstName} is ${this.age}.`;

this.likes.forEach((like) => {

// bio = bio + ` ${this.firstName} likes ${like}`;

bio += ` ${this.firstName} likes ${like}.`;

});

return bio;

}

setName(fullName) {

const names = fullName.split(' ');

this.firstName = names[0];

this.lastName = names[1];

}

}

const me = new Person('Sumeet', 'Sood', 27, ['Teaching', 'Biking']);

me.setName('Nitesh Ganotra');

console.log(me.getBio());

const person2 = new Person('Nitesh','Ganotra', 28);

console.log(person2.getBio());

100)Creating subclasses

Here we will see how to create subclass using prototypal inheritance. So this is prototypal chain of myPerson object.

myPerson -> Person.prototype -> Object.prototype -> null

now lets say our application have different types of persons like employee, teacher or student. So they all have name and ages but they also have subtle changes to them. Lets ay for employee we want to track how long that employee has been with company. But we would need to do is create an entirely new class with all of same code from person. The problem is duplicate code. Using subclassing, we can actually create a class that inherits behavior from person. So we will create subclass for person that is very similar to person but it is going to be specific to a person who is an employee.

We use extend keyword to inherit properties of class. If we do not define anything in new class then new class is same as class that it extended. They can be used interchanginbly. But now we can use them. But we can start overriding various methods or providing brand new methods first we will override constructor. If we dnt provide constructor in employee class then constructor of person class will be called. But if we define contsuctor in employee class then contructor of Employee will be called when we are making objects of employee class.

Code-

class Person {

constructor(firstName, lastName, age, position,likes) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.likes = likes;

}

getBio() {

let bio = `${this.firstName} is ${this.age}.`;

this.likes.forEach((like) => {

// bio = bio + ` ${this.firstName} likes ${like}`;

bio += ` ${this.firstName} likes ${like}.`;

});

return bio;

}

setName(fullName) {

const names = fullName.split(' ');

this.firstName = names[0];

this.lastName = names[1];

}

}

class Employee extends Person {

constructor(firstName, lastName, age, likes = []) {}

}

const me = new Employee('Sumeet', 'Sood', 27, ['Teaching', 'Biking']);

console.log(me);

error-

**D:\Andrew Mead\js-bbotcamp\basics\oop\person.js:25**

**constructor(firstName, lastName, age, position,likes) {}**

**^**

**ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor**

**at new Employee (D:\Andrew Mead\js-bbotcamp\basics\oop\person.js:25:14)**

**at Object.<anonymous> (D:\Andrew Mead\js-bbotcamp\basics\oop\person.js:28:12)**

**at Module.\_compile (module.js:649:30)**

**at Object.Module.\_extensions..js (module.js:660:10)**

**at Module.load (module.js:561:32)**

**at tryModuleLoad (module.js:501:12)**

**at Function.Module.\_load (module.js:493:3)**

**at Function.Module.runMain (module.js:690:10)**

**at startup (bootstrap\_node.js:194:16)**

**at bootstrap\_node.js:666:3**

it means we need to call parent class constructor in child class. We do this by using super.

So we do this-

class Employee extends Person {

constructor(firstName, lastName, age, position,likes) {

super(firstName, lastName, age,likes);

this.position = position;

}

}

Now whenever we call access a method on Employee class, first it check employee class, if that class has that method, then it is executed. Otherwise it looks to parent class for tat method. Here we will override getBio function. then we define other class Student which inherits person class. Code-

class Person {

constructor(firstName, lastName, age,likes =[]) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.likes = likes;

}

getBio() {

let bio = `${this.firstName} is ${this.age}.`;

this.likes.forEach((like) => {

// bio = bio + ` ${this.firstName} likes ${like}`;

bio += ` ${this.firstName} likes ${like}.`;

});

return bio;

}

setName(fullName) {

const names = fullName.split(' ');

this.firstName = names[0];

this.lastName = names[1];

}

}

class Employee extends Person {

constructor(firstName, lastName, age, position,likes) {

super(firstName, lastName, age,likes);

this.position = position;

}

getBio() {

return `${this.firstName} ${this.lastName} is a ${this.position}`;

}

getYearsLeft() {

return 65 - this.age;

}

}

// const me = new Employee('Sumeet', 'Sood', 27,'architect',['Teaching', 'Biking']);

// console.log(me.getBio());

// me.setName('Nitesh Ganotra');

// console.log(me.getBio());

// const person2 = new Person('Nitesh','Ganotra', 28);

// console.log(person2.getBio());

class Student extends Person {

constructor(firstName,lastName,age, grade, like) {

super(firstName, lastName,age, like);

this.grade = grade;

}

updateGrade(change) {

this.grade += change

}

getBio() {

const status = this.grade >70 ? 'passing' : 'failing';

return `${this.firstName} is ${status} the class`;

}

}

const me = new Student('Sumeet','Sood', 27, 88, ['coding']);

console.log(me.getBio());

me.updateGrade(-20);

console.log(me.getBio());

all code is in persom.js file

101)Getters and Setters

Here we will see custom getters and setters. These allows us to customize what happens when someone seta an object property or reads its value. They are cool new feature of language.

Here is our getter-

const data = {

get location() {

return 'phagwara';

}

};

data.location = 'Pune';

console.log(data.location);

output – **phagwara**

inside getter we can return anything that we want. If someone accessing the location, it will get what we are returning inside getter. If we print data object on console, we get this-

**{ location: [Getter] }**

Lets set a setter also, now setter gets called with a acrgument, this argument is value that user tried to set. Lets name this argument as value. Now its upot us to do something with the value in setter. lets just trim this value.

Code-

const data = {

get location() {

return 'phagwara';

},

set location(value) {

value = value.trim()

}

};

data.location = 'Pune';

console.log(data.location);

now we have to store this value somewhere. We actually cannot store it on location because that’s taken by getters and setters. Lets prove it by code-

const data = {

get location() {

return 'phagwara';

},

set location(value) {

value = value.trim();

}

};

data.location = 'Pune';

console.log(data);

output- { location: [Getter/Setter] }

so location property is already set. It is set to our custom getter and setter. So if we want to store this data, we have to pick different name. so we are going to use property name \_location. Then in getter we return this new property. Code-

const data = {

get location() {

return this.\_location;

},

set location(value) {

this.\_location = value.trim();

}

};

data.location = ' Pune ';

console.log(data.\_location);

console.log(data);

lets add one more thing. Lets we change the value of location. But we want to know what was the value of location in past. With custom setter we don’t just have to set a single property. We can do anthing else , we might want to for example on data object, I can create location array. So we add locations to this array as they end up being set. Code-

const data = {

locations: [],

get location() {

return this.\_location;

},

set location(value) {

this.\_location = value.trim();

this.locations.push(this.\_location);

}

};

data.location = ' Pune ';

console.log(data.\_location);

data.location = 'Phagwara';

console.log(data);

output-

**Pune**

**{ locations: [ 'Pune', 'Phagwara' ],**

**location: [Getter/Setter],**

**\_location: 'Phagwara' }**

Later in video, we used getters and setters in person.js and hangman.js.

So these were getters and setters, these come in handy when we want to run some code when a value is set or when a value is read.