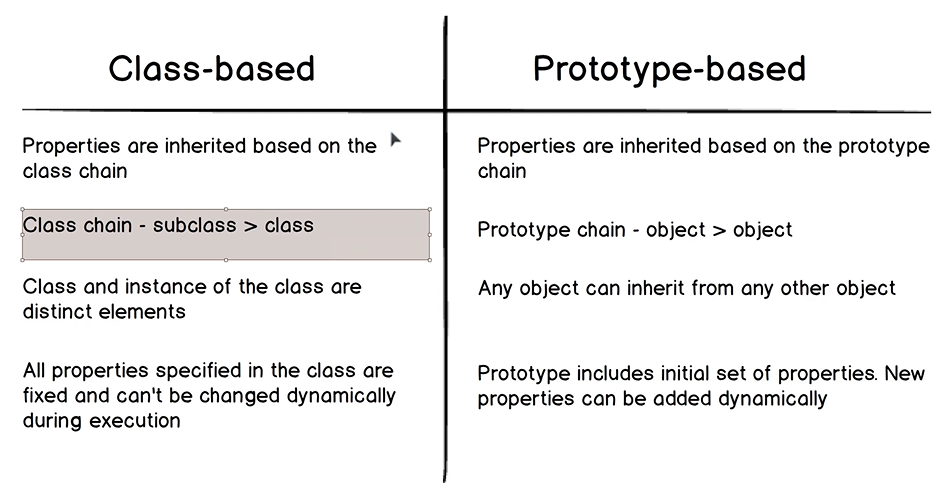
**Section 20**: Classes, Prototypes and Function Constructors

**Lecture 246**:

Class based languages (C++, Java),

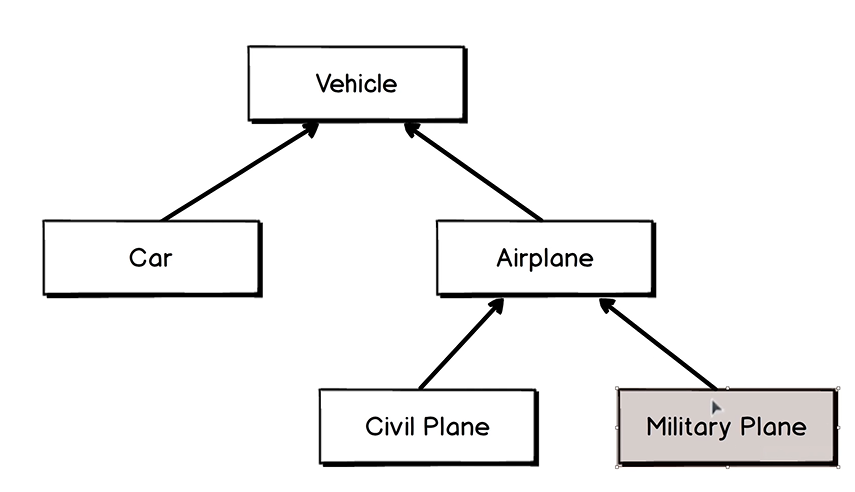
Prototype based languages (JS)



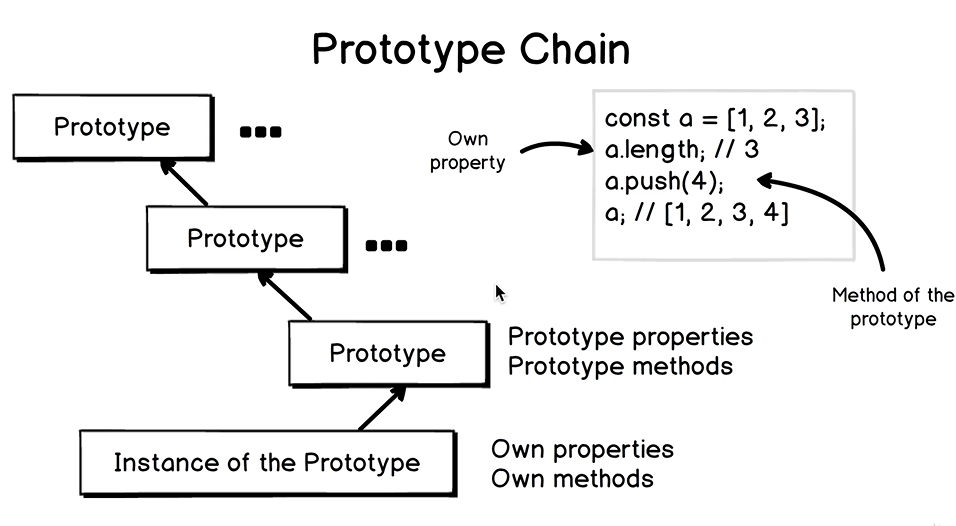
ES6 Classes: Prototype based inheritance

JS still remains prototype based.

**Lecture 247**:



**Lecture 248**: Prototype Chain



Each instance knows about its prototype.

Prototype does not know about all instances of itself.

Relations in a prototype chain are built from bottom to up.

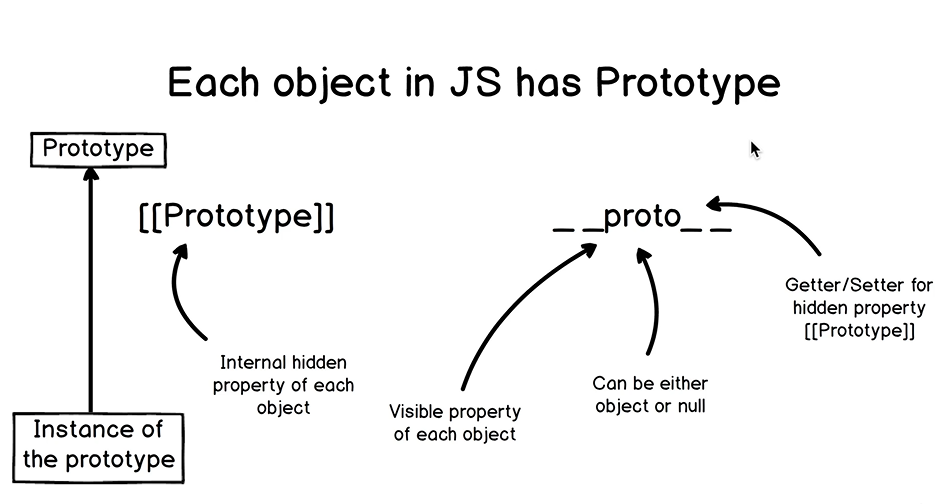
push() method is absent in the instance of the Array prototype.

We move up the prototype chain and check whether the prototype has a particular method.

It could happen that the bottom level prototype does not have a requested method or property. Then we move one level up, if it exists in the prototype chain to check for it.

At the end of the prototype chain in Javascript, there is always null.

**Lecture 249**: \_proto\_ and [[Prototype]]



Each object in Javascript has prototype.

There is a hidden property called as [[Prototype]] and we cannot access it directly.

There is a visible property called as \_\_proto\_\_ for each object.

This name was chosen on purpose to avoid occassional change of it.

This property can be either object or null. It cannot be equal to number or string.

By default, in each prototype chain in javascript, last prototype is equal to null.

\_\_proto\_\_ is just a getter/setter for the hidden property, [[Prototype]]

hasOwnProperty returns a boolean value indicating whether the object on which you are calling it has a property with the name of the argument

Primitive variable types like string also have \_\_proto\_\_ property.

Primitive variable types are also objects in javascript.

const c = 2

c.\_\_proto\_\_.\_\_proto\_\_ gives Object prototype.

c.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ gives null.

string and number also have \_\_proto\_\_ property.

null and undefined primitives do not have \_\_proto\_\_ property.

Father of all prototypes: Object.prototype

**Lecture 250**: What is Prototype?

Object.prototype: Top-level prototype

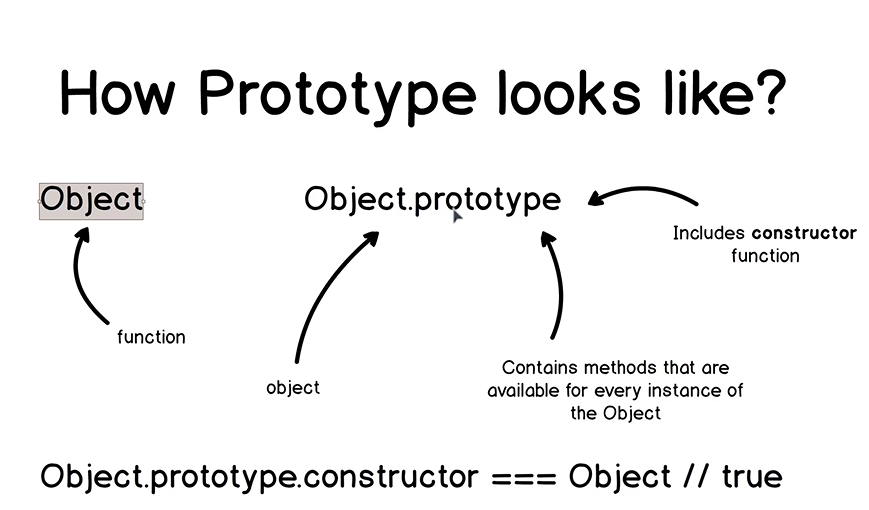
If we create an instance of an Array, then prototype of this instance will be Array.prototype.

Prototype of Array.prototype will be Object.prototype.

Prototype of Object.prototype is null.

Object.prototype.\_\_proto\_\_ === null returns true.

When js engine tries to find a specific property in the prototype chain, it stops when it reaches prototype equal to null.



typeof Object returns “function”

typeof Object.prototype is an “object”

Object.prototype object contains all methods and properties that are available for every instance of the Object.

This object also has “constructor” function which is invoked when we create an instance of Object.prototype.

Object.prototype.constructor === Object // true

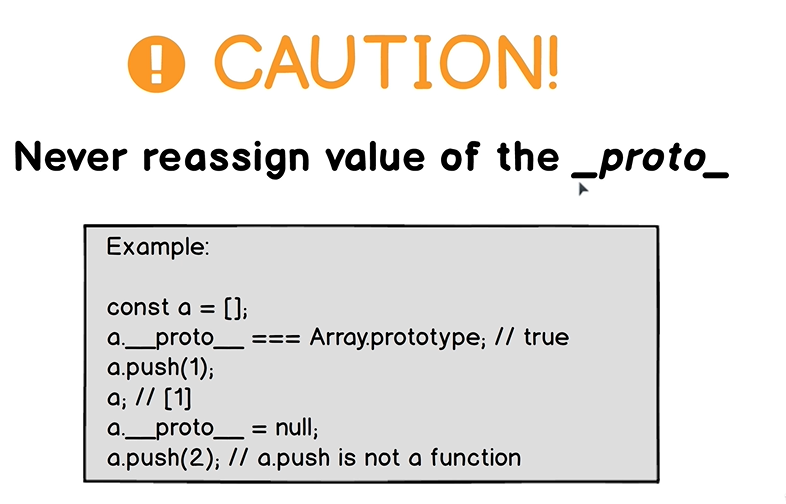
When we create an object using,

const a = { } ; or

const b = new Object()

In both cases, we invoke the constructor function.

**Lecture 251**: Don’t reassign \_\_proto\_\_



When we re-assign value of the prototype of any instance of the prototype or of any prototype, we break the prototype chain and this can lead to unexpected behaviour.

And we will not be able to access methods and properties.

We can assign \_\_proto\_\_ property something other than Object or null.

If we try to assign it some other value, the value of \_\_proto\_\_ will remain unchanged.

We can reassign prototype of any object using Object.create() function.

**Lecture 252**: Native Prototypes

Object

Array

Function

Date

String

Number

Boolean

RegExp

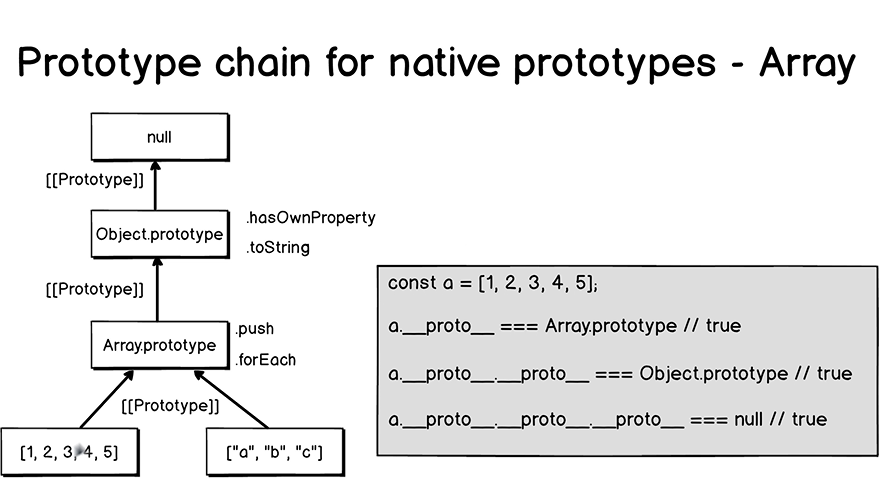
Promise

Map

Set

Error

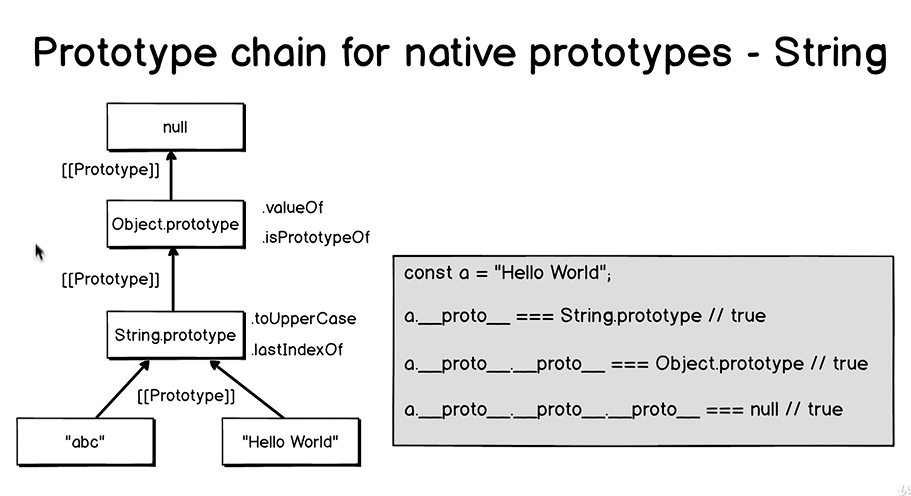
There are native prototypes for primitive types such as String, Number and Boolean.



When we create an array for example using array literal notatin or new Array notation, we create an instance of Array.prototype.

Array.prototype has its own prototype, Object.prototype.

Under the hood, javascript engine uses [[Prototype]] property when it moves from bottom to up in the prototype chain.



const a = “Hello World”

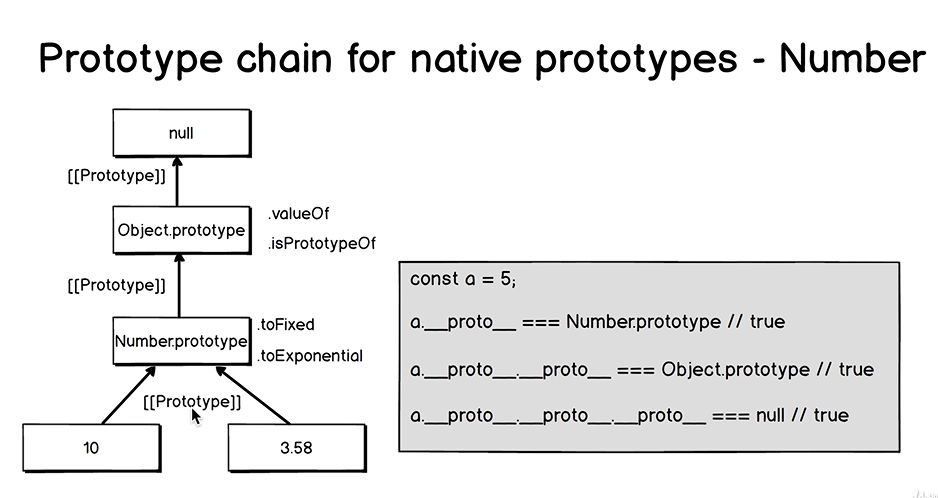
a.\_\_proto\_\_ === String.prototype // true

a.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype

a.\_\_proto\_\_.\_\_proto\_\_ .\_\_proto\_\_ === null

a.toUpperCase()

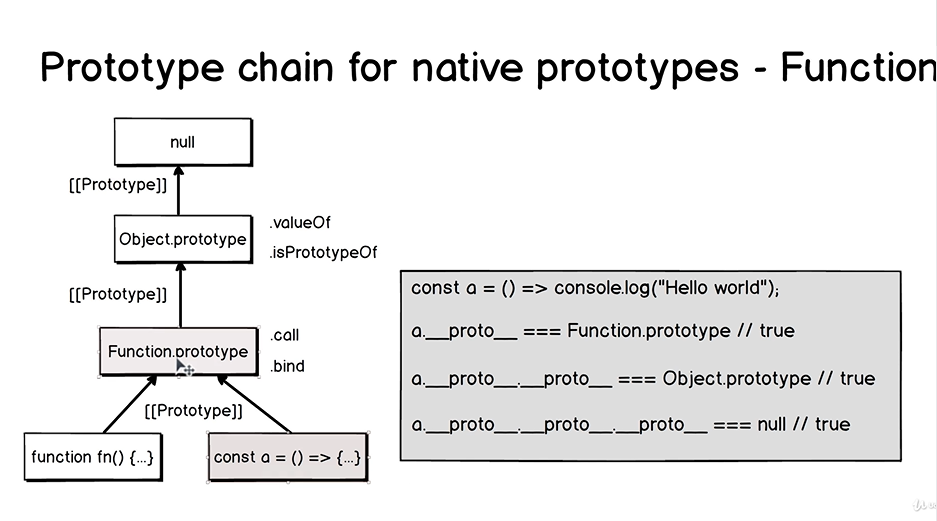
“Hello World” is converted to an instance of String under the hood.



We should create numbers, strings and booleans without using new keyword.

Under the hood, we create instances of the corresponding prototypes.

**Lecture 253**:



When we create a function, we create an instance of the Function prototype.

const a = () => console.log(“Hello World”)

a.\_\_proto\_\_ === Function.prototype // true

a.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype

// true

a.\_\_proto\_\_.\_\_proto\_\_ .\_\_proto\_\_ === null

// true

We can assign a function to any variable or use function in the key value pair of any object.

We can expand function as an object using console.dir() method.

For traditional functions, there is another property called as “prototype”, which has “constructor” function.

const fn2 = (a, b) => console.log(a+b)

fn2.\_\_proto\_\_ === Function.prototype // true

This fn2 function does not have a “prototype” property.

This means that we cannot create instances of fn2 function because it is anonymous arrow function expression.

fn2.prototype: undefined

Arrow functions do not have “prototype” property.

fn2.length returns the number of parameters that are configured for any function in Javascript.

Traditional functions and arrow functions, both have \_\_proto\_\_ property.

Function.prototype is the prototype for both regular functions and arrow function expressions assigned to the variables.

Regular functions are function constructors. They can be used as prototypes for other instances of the functions as they have the “prototype” property.

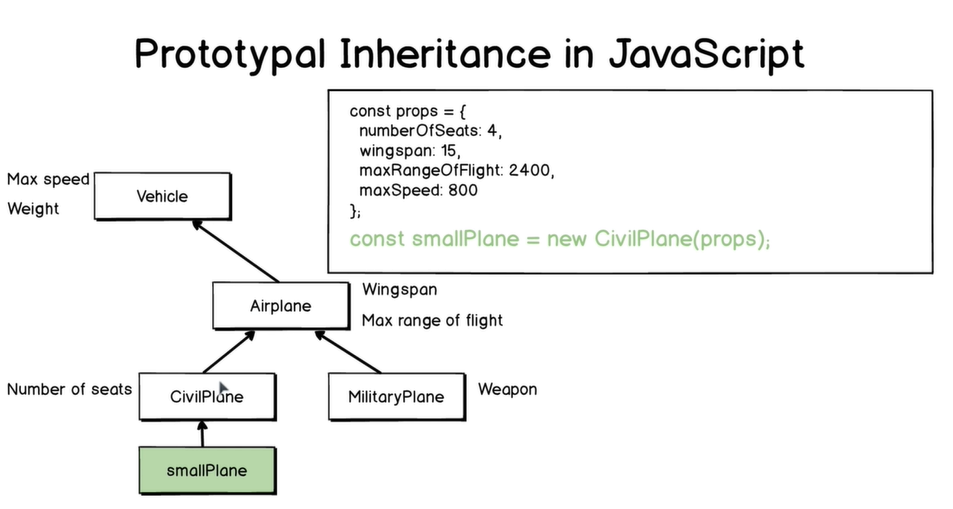
Arrow functions are not function constructors and cannot be used as Prototypes.

**IMP**:

Every function in Javascript (except arrow functions) is a constructor function.

When we create any function in Javascript, we can use this as prototype for new instance of this function.

**Lecture 254**: Prototypal Inheritance in Javascript



const smallPlane = new CivilPlane(props)

smallPlane.\_\_proto\_\_ === CivilPlane.prototype

// true

smallPlane.\_\_proto\_\_.\_\_proto\_\_ === Airplane.prototype

smallPlane.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_

=== Vehicle.prototype

smallPlane.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype

smallPlane.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ === null

The ‘smallPlane’ instance will have its own properties (from CivilPlane.prototype), properties from Airplane, Vehicle and Object.

const smallPlane = new CivilPlane(props);

smallPlane.wingspan; // 15

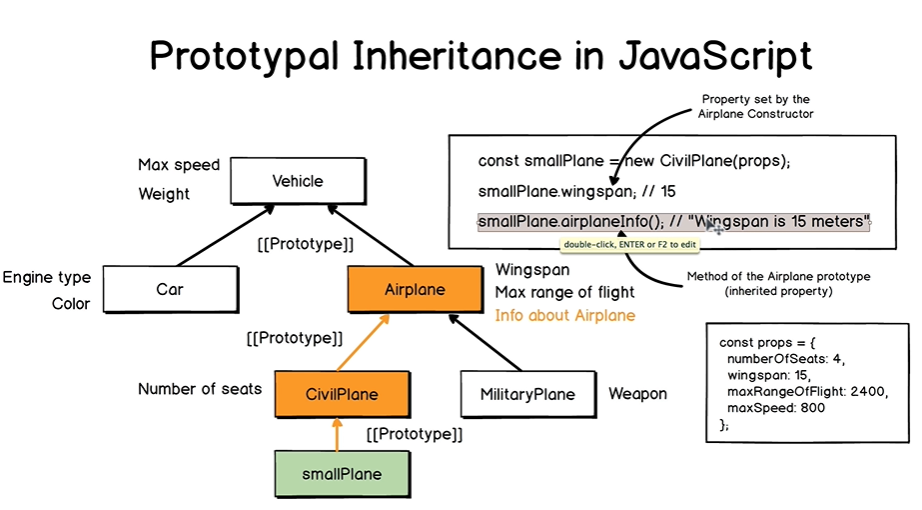
smallPlane.airplaneInfo()

// “Wingspan is 15 metres”

“wingspan” property is set by the Airplane constructor.

airplaneInfo() is the method of the Airplane prototype (inherited property).

Javascript engine moves up through the prototype chain and finds the airplaneInfo() in the Airplane.prototype.



Value 15 comes from the smallPlane object. It is not the property of Airplane.

It is taken dynamically from smallPlane using reference, “this”.

Refer to smallPlane object using “this” in airplaneInfo method.

Suppose we want to invoke another method, vehicleInfo() and it is located in the Vehicle prototype.

const smallPlane = new CivilPlane(props)

smallPlane.vehicleInfo();

// “Max speed is 800 km/hour”

Again we go up through the prototype chain, checking whether the method exists in that prototype.

In javascript, prototype only contains initial list of properties and methods.

And we can adjust them as we want.

We can add or remove properties and methods.

Methods are located in the prototypes.

When we create an instance of any prototype, then those methods are not copied to any instance.

They are located in a particular prototype. There is no need to copy each method of the prototype to 100s or 1000s of instances of this prototype.

Methods are available using prototypal inheritance.

**Lecture 255**:

For function constructors, start their names with capital names.

Following functions have “prototype” property and are function constructors:

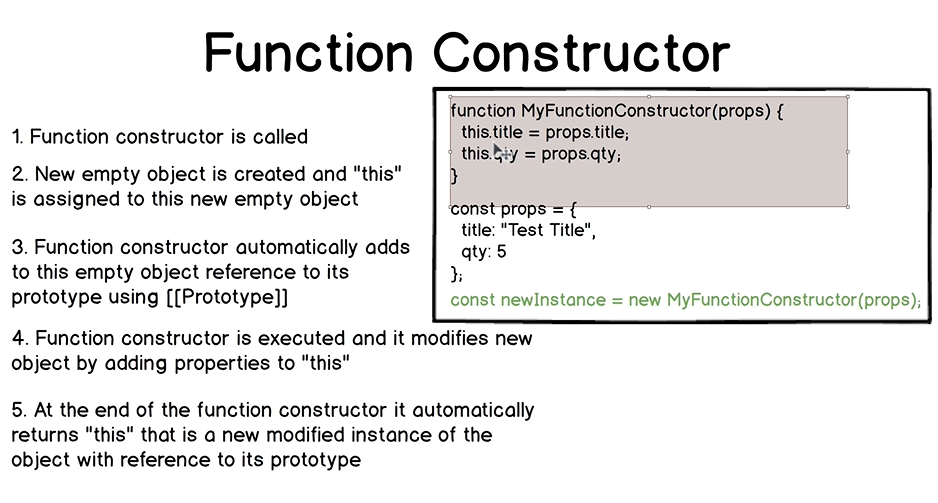
* Functions declared using function declaration
* Anonymous function expressions assigned to the variable.

Each function in Javascript (except arrow functions) has .prototype property and can be used as a constructor function.

It has prototype chain.

When we use “new” keyword with a function name, then instead of normal function call, Function constructor is called.

**Lecture 256**: Function Constructors Overview



Function constructor is executed and it modifies new object by adding properties to “this”

At the end of the function constructor it automatically returns “this” that is a new modified instance of the object with reference to its prototype.

Using Function constructor, we can create unlimited number of instances.

And each time, Function constructor will be called, new instance will be created.

[[Prototype]] 🡪 MyFunctionConstructor.prototype

The new modified instance of the object will contain the properties title, qty, \_\_proto\_\_ and hidden property [[Prototype]]

Each time this process is repeated.

/\*

Create new instance of the prototype using "new" syntax:

- Function constructor is called.

- New object {} is called.

- \_\_proto\_\_ is added

- New props are added

- Object is returned by constructor function

\*/

**Lecture 258**:

function CivilPlane(props) {

this.numberOfSeats = props.numberOfSeats

this.seatsInfo = function(){

console.log(**`**Number of seats in the plane is ${this.numberOfSeats}**`**)

}

}

const propsForSmallPlane = {

numberOfSeats: 4

}

const smallPlane = new CivilPlane(propsForSmallPlane)

seatsInfo() method is added to each instance of the CivilPlane prototype.

Example:

function CivilPlane(props) {

this.numberOfSeats = props.numberOfSeats

}

CivilPlane.prototype.seatsInfo = function(){

console.log(`Number of seats in the plane is ${this.numberOfSeats}`)

}

const propsForSmallPlane = {

numberOfSeats: 4

}

const smallPlane = new CivilPlane(propsForSmallPlane)

console.log(smallPlane)

If we try to print smallPlane now, we will see only 2 properties, numberOfSeats and \_\_proto\_\_

If we try to do, smallPlane.info(), then we get an unexpected console result, “Number of seats in the plane is 4”.

This seatsInfo() method is inherited by all instances of CivilPlane.prototype.

If we change this anonymous function expression to anonymous arrow function expression,

we will get “Number of seats in the plane is undefined”

In the arrow function, “this” points to window.

Arrow functions don’t have own “this”.

When using traditional functions, “this” refers to the object where the function is called.

We can also create a function to change a property of the object.

CivilPlane.prototype.modifySeatsNumber = function(newSeatsQty) {

this.numberOfSeats = newSeatQty

}

smallPlane.modifySeatsNumber(10)

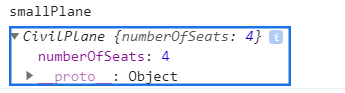
// modifies “numberOfSeats” property of the “smallPlane”

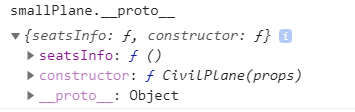
seatsInfo and modifySeatsNumber are the methods of the CivilPlane prototype.

They are inherited automatically by any instance of this prototype. This is the prototypal inheritance in action.

We can also change the value of an object property directly.

With prototype, we have flexibility.





**Lecture 259**: Object.create()

Create prototype of a prototype.

“Object” in Javascript is a function.

Each function in javascript is an object.

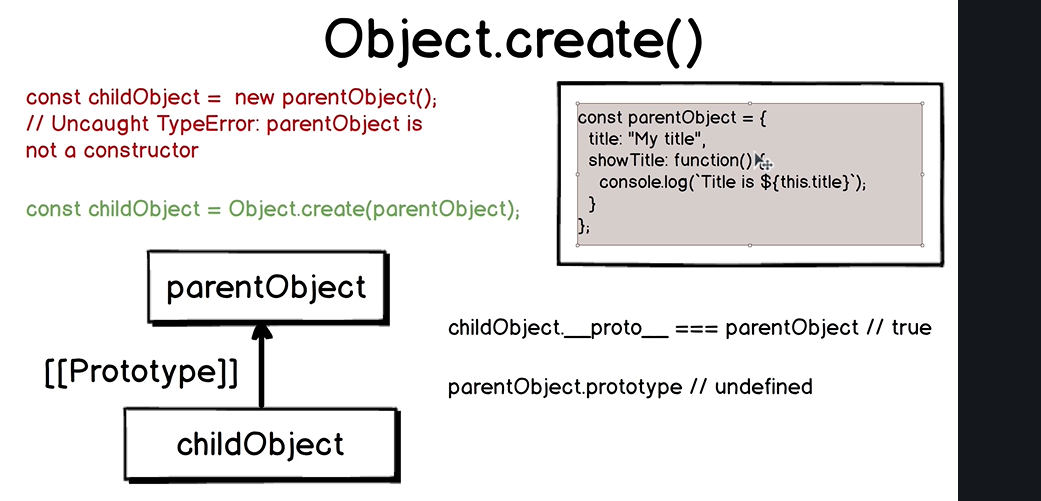
And that is why each function may have properties and methods.

Using create(), we can create an object from another object.

We create new object and set its prototype equal to the object that we pass an argument.

In other words, we create a prototype chain.

We do this manually without using prototypes.



This object is not a function constructor and it does not have the “prototype” property.

parentObject.\_\_proto\_\_ is an Object

childObject.\_\_proto\_\_ === parentObject // true

All methods and properties available in the parentObject will be available on childObject reference because of inheritance.

Parent is the prototype for the child objects.

Name of prototypes should start with capital letters. (Just a convention)

**Lecture 260**:

“prototype” property is only defined for normal functions.

When we see \_\_proto\_\_ property, at first sight we see something like:

\_proto\_\_: Object

(What this means that \_\_proto\_\_ property is itself an object, that is it has various fields and information).

The underlying information is hidden inside it.

(different for constructors and objects).

const Parent = {

type: "Parent",

typeInfo: function(){

console.log(`Hello from ${this.type}`)

}

}

const child = Object.create(Parent)

child.\_\_proto\_\_ === Parent // true

And not Parent.prototype

Parent.prototype is undefined.

“child” object does not have ‘type’ or ‘typeInfo’ properties by itself. They are inherited from the Parent prototype.

Parent object is not a constructor function and thus cannot modify new instance of the object.

It cannot set initial properties of the objects which have Parent as their prototype.

We will have to set the “type” property for the ‘child’ object manually.

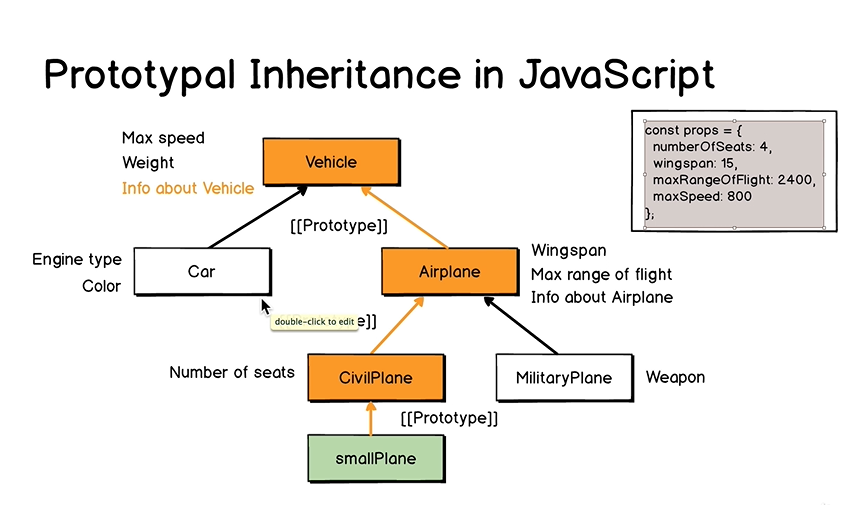
child.type = “Child”

If we try to print “child”, then at its first level, we will see 2 properties called as “type” and “\_\_proto\_\_”

Parent.\_\_proto\_\_ === Object.prototype // true

“prototype” property is only defined for function constructors.

**Lecture 261:**



In this example, CivilPlane is a function constructor.

Suppose, here initially prototype of CivilPlane is Object.

Lets add Airplane to this prototype chain.

And prototype of Airplane will be Object.

function Airplane(props){

this.wingspan = props.wingspan

this.maxRangeOfFlight = props.maxRangeOfFlight

}

Airplane.prototype.airplaneInfo = function(){

console.log(`Wingspan of the airplane is ${this.wingspan} and

maxRangeOfFlight is ${this.maxRangeOfFlight}`)

}

function CivilPlane(props){

this.numberOfSeats = props.numberOfSeats

}

CivilPlane.prototype.seatsInfo = function(){

console.log(this)

console.log(`Number of seats in the plane is ${this.numberOfSeats}`)

}

CivilPlane.prototype.modifySeatsNumber = function(newSeatsQty){

this.numberOfSeats = newSeatsQty

}

const propsForSmallPlane = {

numberOfSeats: 4

}

const smallPlane = new CivilPlane(propsForSmallPlane)

// smallPlane is itself CivilPlane prototype.

Now, we need to insert Airplane.prototype in our prototype chain.

Each function is equal to the function constructor.

CivilPlane.prototype = Object.create(Airplane.prototype)

\_\_proto\_\_ property of CivilPlane.prototype will be equal to Airplane.prototype.

Object.create() sets the \_\_proto\_\_ property and hidden [[Prototype]] property to the object that we pass in as an argument.

CivilPlane.prototype.\_\_proto\_\_ === Airplane.prototype // true

With CivilPlane.prototype = Object.create(Airplane.prototype),

we have broken the rule that the function should be equal to the constructor.

CivilPlane.prototype.constructor === CivilPlane will give false.

The value of CivilPlane.prototype.constructor is Airplane function.

Object.create() creates empty object with one property called as \_\_proto\_\_

“constructor” is the property of the prototype.

We need to assign property to CivilPlane.prototype.constructor and make it equal to CivilPlane.

CivilPlane.prototype.constructor = CivilPlane

Now we have correctly inserted Airplane.prototype in the prototype chain and it is the prototype of CivilPlane.prototype.

Suppose we set some properties in CivilPlane and other in Airplane constructor.

If we just call constructor of CivilPlane, then properties set by Airplane constructor will not be set.

function CivilPlane(props){

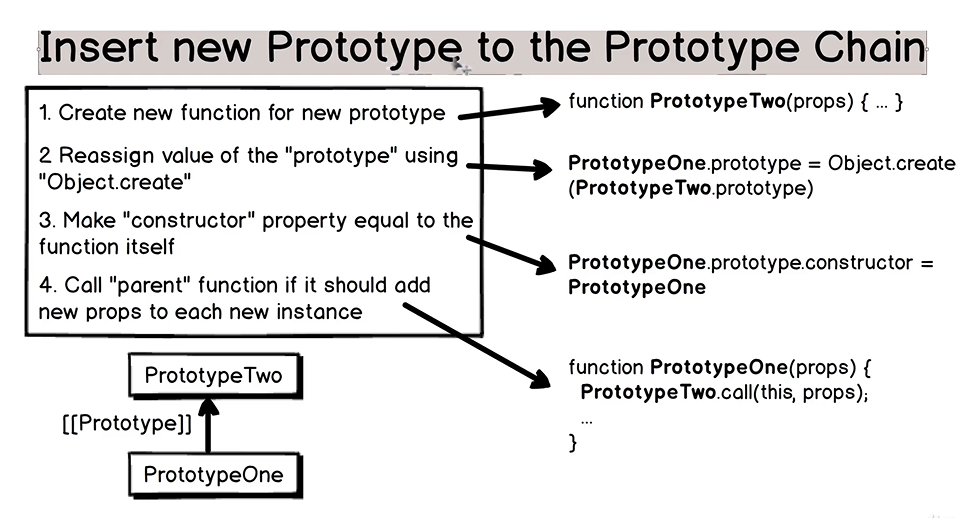
Airplane.call(this, props)

this.numberOfSeats = props.numberOfSeats

}

1. Reassign value of the “prototype” using “Object.create”
2. Make “constructor” property equal to the function itself because Object.create() creates a brand new object with only 1 property, \_\_proto\_\_
3. Call “parent” function if it should add new props to each new instance. We simply call the function and not the function constructor using call()

**Lecture 262**:



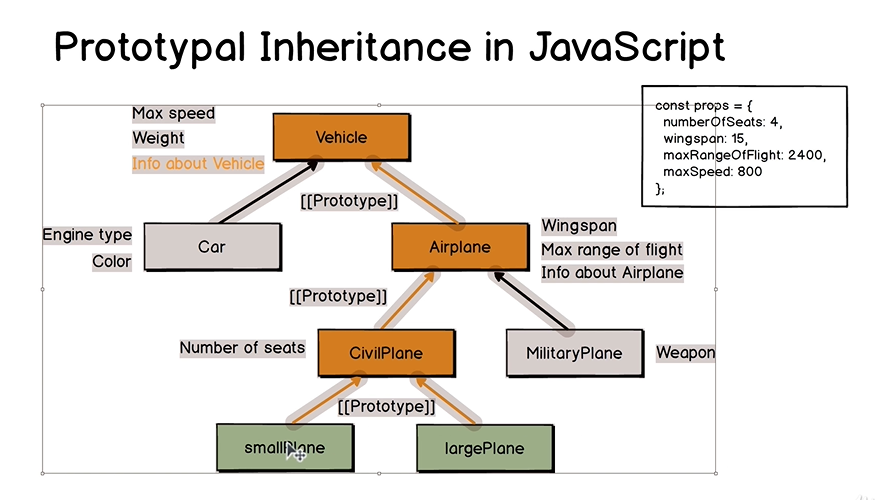
If we want to add another prototype in this prototype chain, then we should repeat these steps for the PrototypeTwo.

Imperative programming paradigm

Declarative programming paradigm

**Lecture 265**:

“instanceof” and “typeof” in the Prototype Chain



Variables of Lower prototypes are instances of the prototypes above them.

CivilPlane instanceof Airplane // false

These are functions. We cannot initialize CivilPlane as a new instance of Airplane.

**Lecture 266**:

Methods should be in the .prototype property of a Function rather than being in the function declaration. This prevents adding the method to every instance of the Function.

Method is a name-value pair where value is a function.

If we want to build correct prototype chain, we should add methods to the prototype and each instance should have only name-value pairs without any methods.

We should call the parent function only if it sets additional properties of the instance.

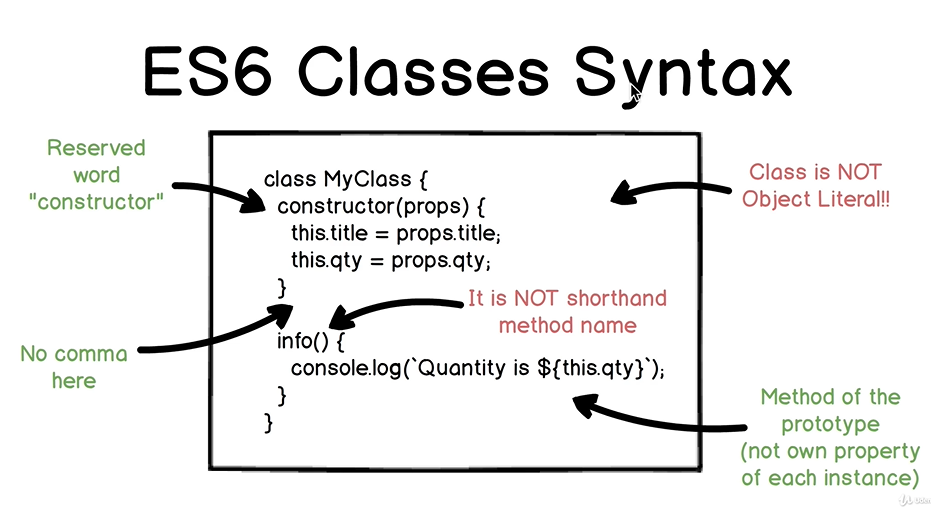
**Lecture 268**: ES6 Classes

ES6 classes do not introduce any new functionality in the function prototype chain behaviour.

It is just more simple and declarative way to create Prototype chain.

When we create prototype chain using function constructor, we follow imperative programming paradigm.

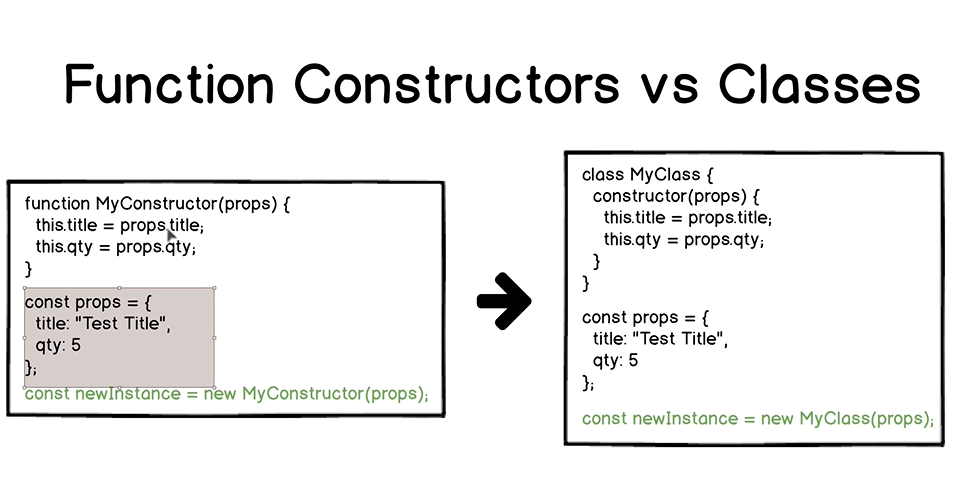
We describe step by step about what we want from Javascript engine and it simply performs those tasks.



Class is not an object literal.

Class consists of methods and methods are listed one by one without any commas.

**Lecture 269**: Function Constructors vs Classes

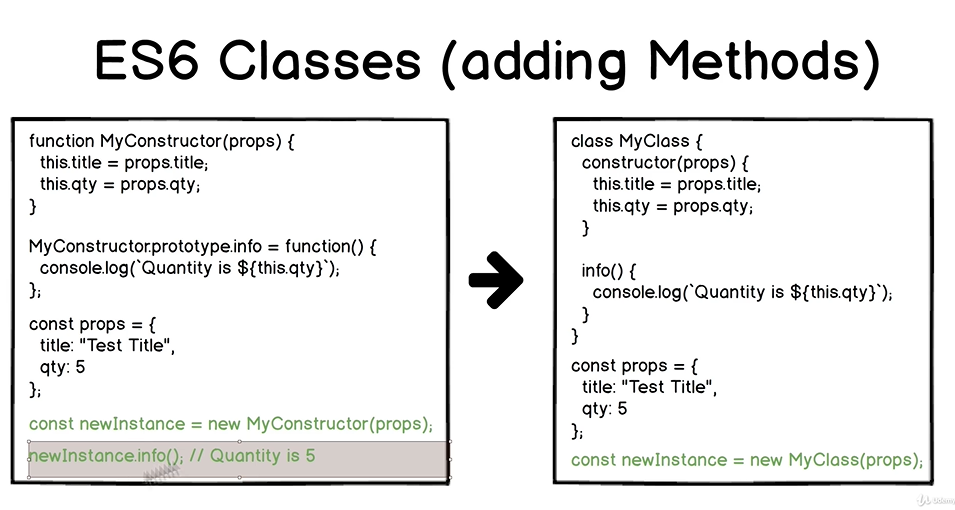


newInstance.\_\_proto\_\_ === MyConstructor.prototype // true

In these 2 blocks of code, we perform the same action of creating new prototype chain.

The 2 separate instances that we create are equal.

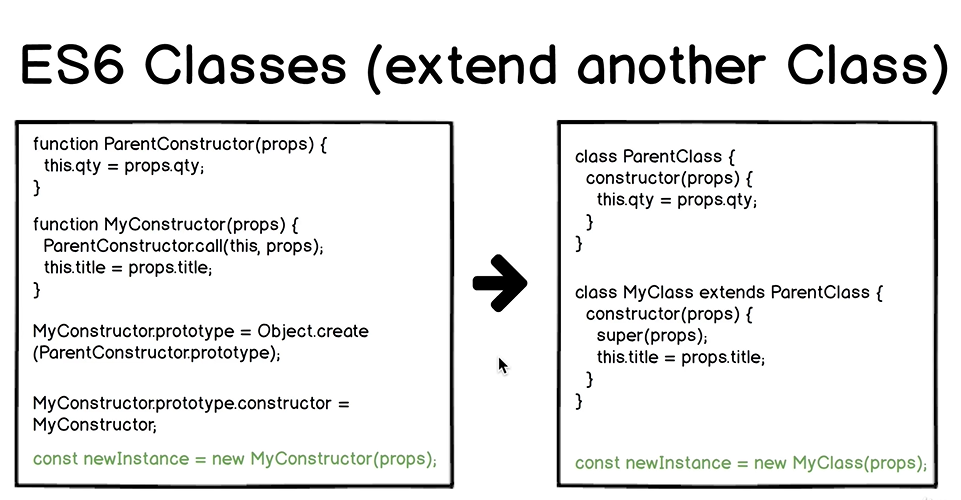
* Another difference is with adding methods.



In class declaration, we do not use keywords such as “prototype”, “\_\_proto\_\_” or Function.

Prototypes with methods are created automatically behind the scenes and we need to specify only list of methods here.

* Extension of other prototype and class



If constructor of the parent class does not set any additional properties of the instance, we must call it using super() in the child class that has constructor.

**Lecture 270**:

class MyClass { }

const firstInstance = new MyClass();

console.log(firstInstance)

The constructor property points to the class.

If I see Object written, then it could mean object which means that the reference has properties and methods and nothing else.

firstInstance.\_\_proto\_\_ === MyClass.prototype

// true

MyClass.prototype.constructor === MyClass // true

Under the hood, behaviour of classes is same as behaviour of Function constructors.

Functions are hoisted.

Class declarations are not hoisted. Behavior is similar to with let and const.

(Temporal Dead Zone)

// Example: Class Expression

const MyClass = class {}

const firstInstance = new MyClass();

console.log(firstInstance)

With classes, we can use both class declarations and expressions and there is no difference between the two as a class is not hoisted.

With regular functions, there is a difference between function declaration and function expression. That is, difference in terms of hoisting.

Class is not object literal and class methods are not shorthand method names.

Classes do not use traditional object literal notation. They use brand new own syntax and this syntax only allows us to add methods.

We cannot add property-value pair in a class.

The syntax which classes use only allows to add methods.

class c{

constructor(){ }

method1(){ }

method2(props){ }

}

**Lecture 271**:

Setting default properties of each instance of a class.

When assigning property using ‘this’ keyword, then we can use || to set default property.

Harder check, we can use ternary operator ?:

this.interface = props.interface !== undefined ? props.interface : “Not Specified”

**Recommendation**:

Never assign “undefined” as a value for any variable. If variable should not have any value, use “null” instead.

Value “undefined” should be reserved for cases when variable does not exist.

Behaviour of methods of a class is same as the behaviour of methods of a prototype in a function constructor.

// Example: Class extension (Add new class to the prototype chain)

class ComputerAccessories{

constructor(props){

this.compatibility = props.compatibility || ["PC", "Mac"]

}

}

class ComputerMouse extends ComputerAccessories{}

const myMouse = new ComputerMouse( { compatibility: ["Mac"] } )

Constructor of the parent class is called automatically if child class does not have a constructor.

class ComputerMouse extends ComputerAccessories{

constructor(...allProps){ // REST operator

super(...allProps) // Spread operator

}

}

super() must be present in the constructor.

If we set properties of the same instances in different classes, we must call super(…) in the child class.

**Lecture 272**:

typeof and instanceof

typeof ComputerMouse // function

myMouse instanceof ComputerMouse // true

myMouse instanceof ComputerAccessories // true

myMouse instanceof Object // true

typeof ComputerMouse.prototype // object

typeof ComputerMouse.prototype.constructor

// function

ComputerMouse.prototype.constructor === ComputerMouse // true

// Example: Static methods

Static methods can be called directly on class names.

Calling them on a variable name gives an error.

If error is thrown, Javascript code is not executed further.

**Lecture 274**:

Convert function constructor to classes having same functionality.

**Lecture 279**: React and ES6 Classes

Classes are syntactic sugar for prototypal inheritance and prototypal chains.

First we will load react, then react-dom, then material-ui and finally index.js

React.Component is a function.

“Component” is a function constructor.

Prototype of React.Component is Function.prototype.

React.Component.prototype.constructor === React.component

Reaction.Component is instance of Function.prototype.

button1.type === MyButton

The class “MyButton” is assigned to ‘type’ property of button1.

To tell that this object is actually a react element,

$$typeof property is added.

button1.$$typeof === Symbol.for(“react.element”)

// true

React.createElement ( … ) takes in 3 arguments, Type, Props and Children.

Type (either tag name such as “button” or “div” or other React Component).

External **React component** Button located in material-ui,

window[“material-ui”].Button

Using 2nd argument, we pass properties to this component.

onClick defines the event that happens when we click the button.

JSX alternative, allows to insert html in js code.

Order of <script> tags is important.

**Lecture 281**: