Chapter 5: Advanced JavaScript Objects and Functions:

V.IMP

3. Everything is an Object: Inheritance and the Prototype Chain:

Objects in Javascript:

In JavaScript, everything is an object. Well, almost everything.

Primitives:	Everything else:
- Numbers	- Arrays
- Strings	- Functions
- Booleans	- objects
- Undefined	- Dates
- Null	- Wrappers for numbers, strings, booleans

.... is an object.

Object oriented programming:

In very simple terms, object oriented programming makes heavy use of objects,

properties and methods and these objects interact with one another to form complex applications. We use objects to store data, structure our code, and keep our code clean.

So far, we have only created simple objects holding some data. Remember the john object from the intro lectures:

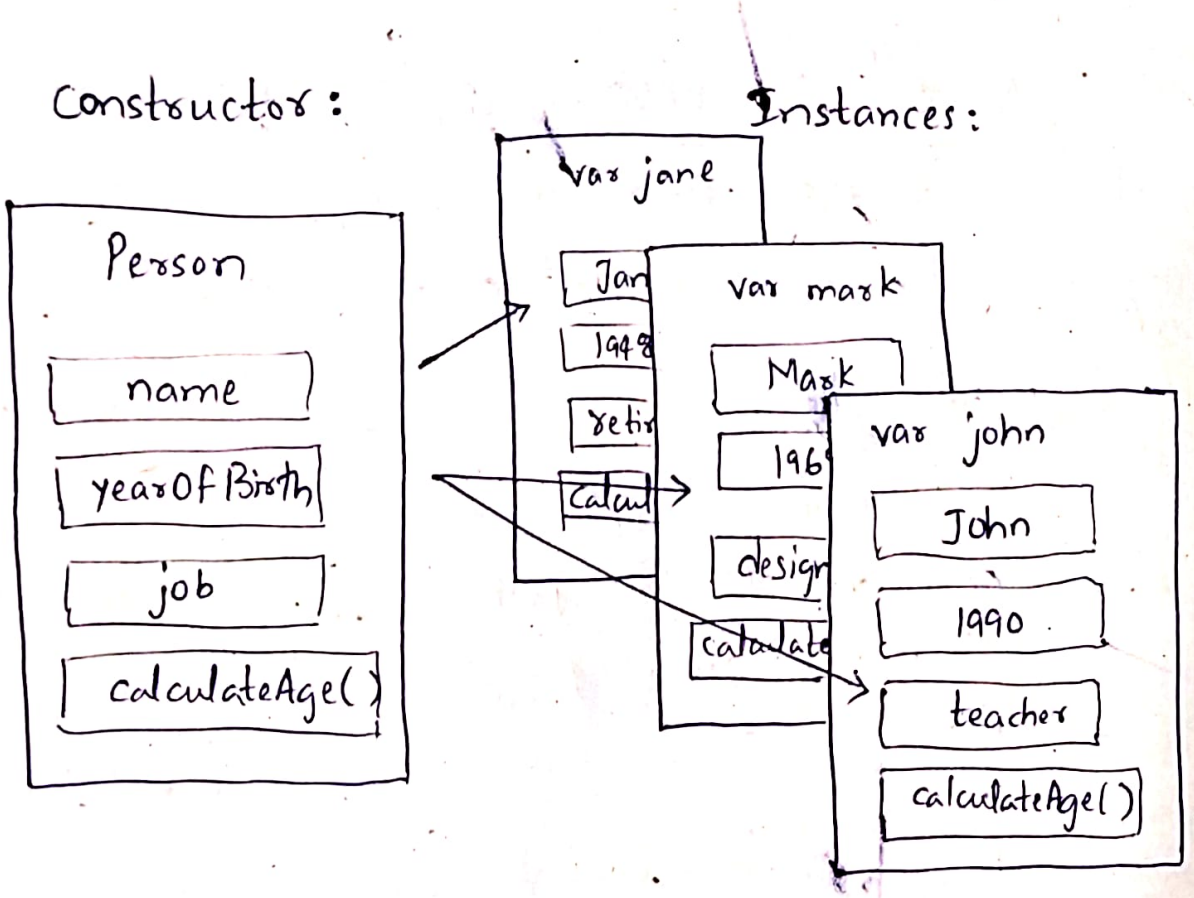
```
var john = {
    name: 'John',
    yearOfBirth: 1990,
    isMarried: false
};
```

and if we had created other persons back then we have simply written them one by one like mark, jane etc.

```
var mark = {
    name: 'Mark',
    yearOfBirth: 1948,
    isMarried: true
};
```

```
var jane = {
    name: 'Jane',
    yearOfBirth: 1969,
    isMarried: true
};
```

But there is a better way. Imagine something like a blueprint from which we can generate as many objects as we want, and we can do that in JavaScript:



This is a special person object that we can basically use as a blueprint to create a lot of person objects. In other programming languages, this is called a class, but in JS, we call it a constructor or prototype.

So based on this constructor, we can create as many instances as we want.

Here, 'john, mark and jane were created from the person constructor. So they are effectively person instances. And now their name, yearOfBirth and job variables are defined. And they all have access to the calculateAge() method.

So, the constructor acts like a blueprint and is used to create instances which of course are also objects.

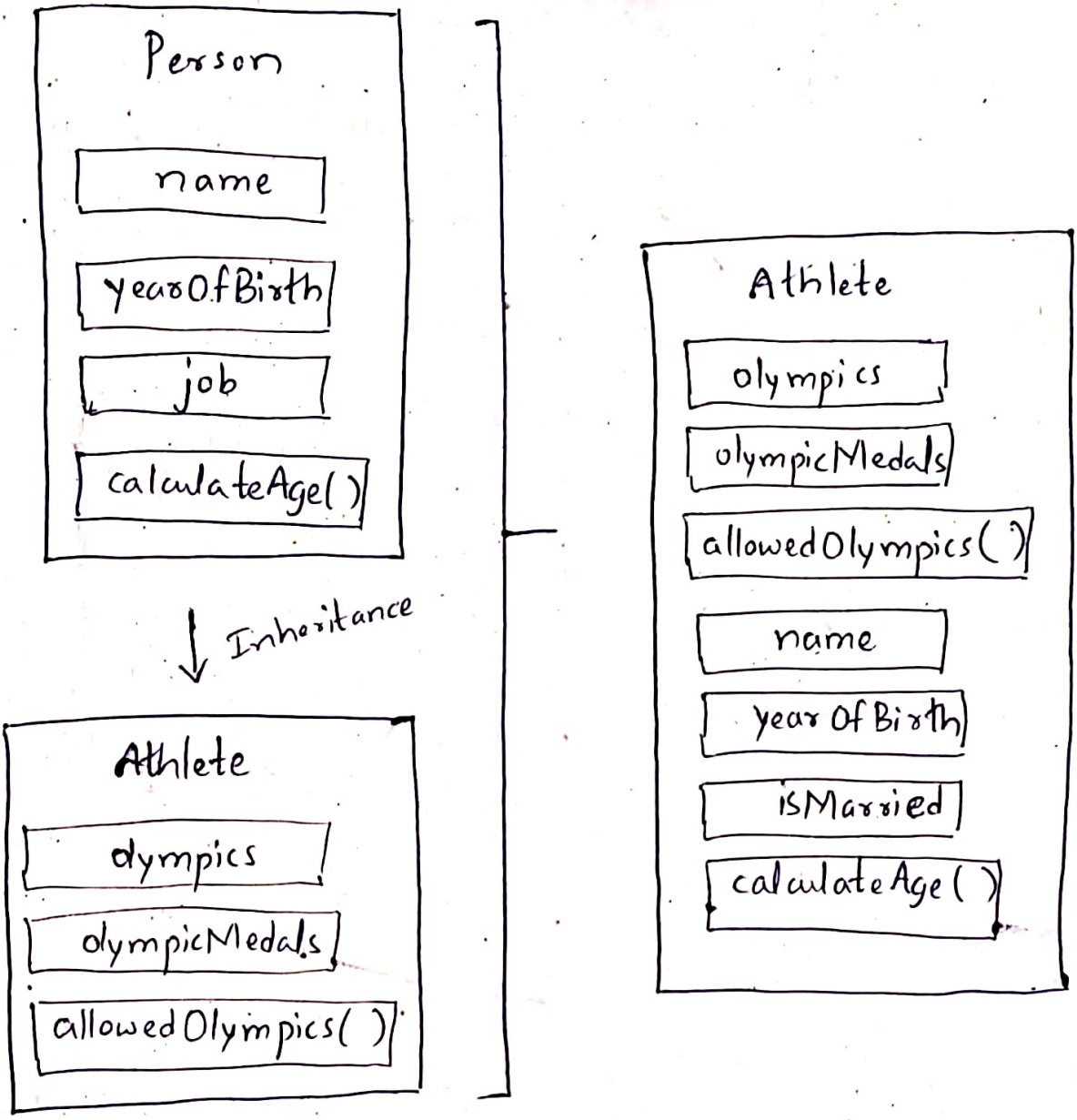
Inheritance:

In simple terms, inheritance is when one object is based on another object. It's when one object gets access to another object's properties and methods.

Back to our person example, imagine that you also wanted to have an athlete constructor, besides the person constructor, with a couple of different properties and methods. Now, an athlete is also a person, right?

There are just some particular properties and methods for an athlete, like for example how many participations in Olympics they have or how many medals they have won.

So when we define the blueprint, so the constructor, for an athlete, then why repeat the same stuff that we have in our person constructor? An athlete also has a name, year of birth and job, right? So what we can do is to use inheritance We make the athlete object inherit the properties and methods from the person object, so then the athlete not only has access to its own properties and methods but also the ones from the person object. This allows us to write less code and more logical programs.



Inheritance in JavaScript: Prototypes and Prototype chains:

JavaScript is a prototype-based language, which means that inheritance works by using something called prototypes. In practice, it means that each and every JavaScript object has a prototype

property which makes inheritance possible in JavaScript. So again, inheritance is made possible through the prototype property that every object has.

How does inheritance actually work? To better understand this, let's go back to our person example, where the person object is the constructor and john, is one of the instances.

Now, if we want John to inherit a method or a property from the person object we have to add that method or property to the person's prototype property.

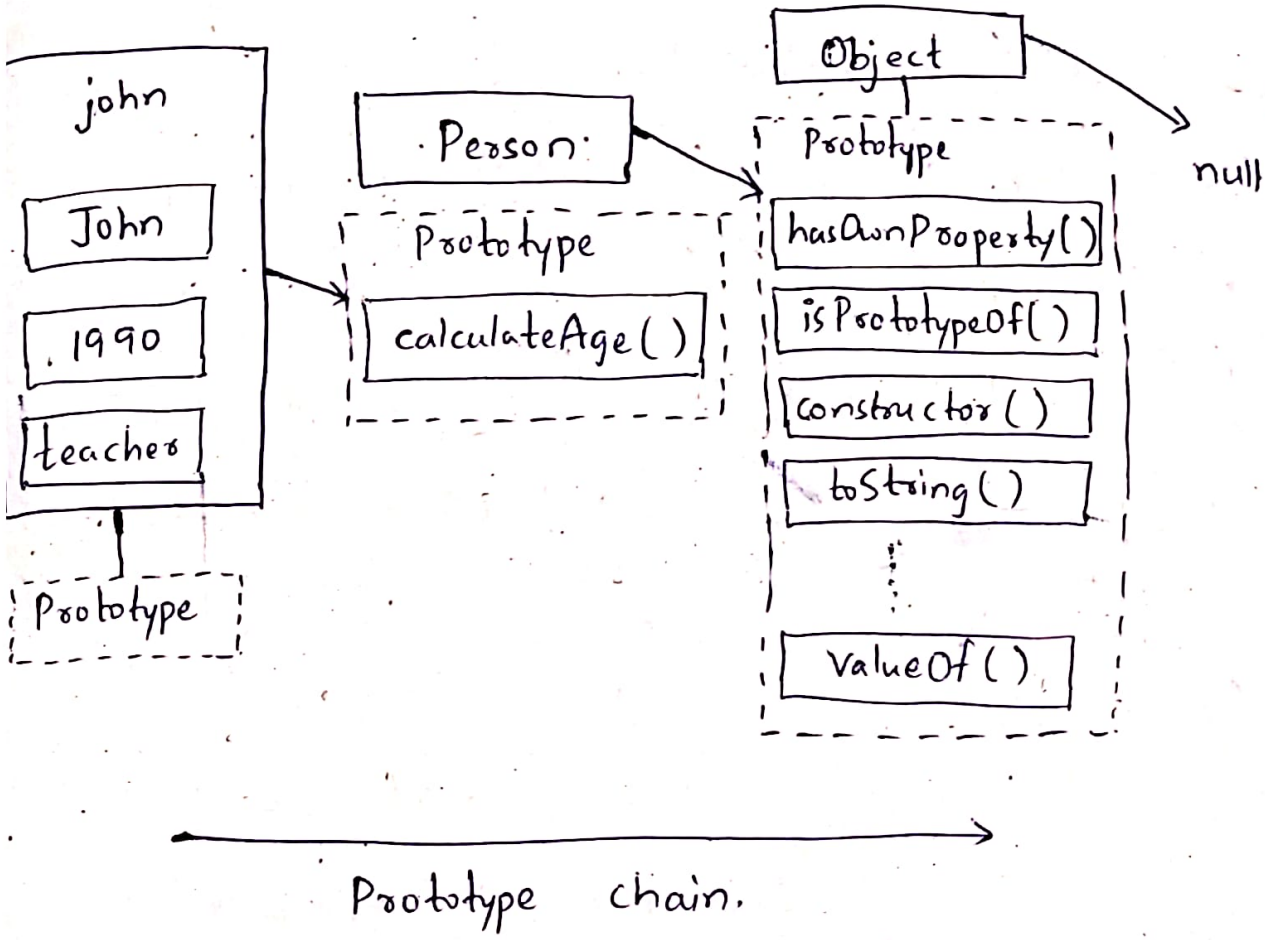
So in this example, we have the calculateAge() method in the person's prototype property and therefore john inherits the method and can then call it. And any other object created by the person constructor would inherit this method as well.

So again, the prototype property of an object is where we put methods and properties that we want other objects to inherit. What is really important to note here is that the person prototype is not the prototype of the person itself, but of all instances that we created thru the person blueprint; like for example, john.

So in other words, the person's prototype property is the prototype of john. And that's not even all, because the person object itself is an instance of an even bigger constructor, which is the Object object. Each and every object that we ever create is an instance of the Object constructor which has a bunch of methods in its prototype property. And as we can guess, the person object inherits these methods and can call them. Plus, the john object also inherits these methods and can also use them.

With this, we have actually explained the prototype chain. The prototype chain is what makes all this inheritance possible, and here is how it works: When we try to access a certain method or property or an object, JavaScript will first try to find that method on that exact object. But if it can't find it, it will look in the object's prototype., which is the prototype property of its parent. So it moves up in the prototype chain. If the method is still not there, this continues until there is no more prototype to look at, which is null. Null is the only one that has no prototype and is therefore the final link in the prototype chain. And in this case, undefined is returned. This is the reason why, for example, the john object could call the hasOwnProperty method that is stored in

the object prototype property



Summary:

- Every JavaScript object has a prototype property, which makes inheritance possible in JavaScript.
- The prototype property of an object is where we put methods and properties that we want other objects to inherit.
- The Constructor's prototype property is NOT the prototype of the Constructor itself, it's the prototype of ALL instances

that are created through it.

- When a certain method or property is called, the search starts in the object itself, and if it cannot be found, the search moves on to the object's prototype. This continued until the method is found: prototype chain.

V.V.IMP

4. Creating Objects: Function Constructors:

Import 5-Advanced-JS starter code from Github.

script.js:

```
// Function constructor

var john = {
    name: 'John',
    yearOfBirth: 1990,
    job: 'teacher'
};
```

object literal way of writing objects.

```
var Person = function(name, yearOfBirth, job) {
```

constructor name (convention to write function const. names in Capital)

Function constructor: popular way of creating objects.

The parameters for this function constructors will be the variables that we want to set

```
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}

var john = new Person('John', 1990,
                      'teacher');
```

This is how it works:

- In order to understand how this works, we need to first understand what the new operator does. So when we use the new operator first a brand new empty object is created.
- Then after that, the constructor function, which in this case is Person, is called with the arguments that we specified.
- So first an empty object is created and then the function is called.
- As we already know, calling a function creates a new execution context that also has a this variable.

We know that in a regular function call the this variable points to the global object.

- But if we now look at our function constructor here, then having the this variable pointing at the global object wouldn't be so useful. Because we would simply set all these properties on the global object. And that's not what we want. And the new operator takes care of this. And it makes it so that the this variable of the function points to the empty object that was created in the beginning by the new operator

- So again, what the new operator does is to point the this variable not to the global object but to the new empty object that was created in the beginning when we used this operator

- So then after that when we set the name, year and job properties to this, then that's the same as setting them right on our new empty object.
- And finally if the constructor function does not return anything, and as we can see in this:

```
var Person = function (name, ...
        ⋮
}
```

that is clearly the case, then the result is simply the object that was created in the first step. So the new empty object here. And this empty object which was created here now has the properties that we defined. So the name, yearOfBirth and job in this case.
- Finally, all of this is simply assigned to the john variable.

And just like this, we created an object called john simply by called a function constructor.

Summary:

→ Our new operator here first creates an empty object, then it calls our function with the this variable not pointing to the global object but to the new object that was created here:

```
var john = new Person('John', 1990, 'teacher');
```

And then when this code runs, the this variable is no longer the global variable. So these are all set on the new object which is then in the end assigned to this variable.

Let's now add inheritance to the game. And for that, we are gonna bring back our calculateAge() function.

Let's now suppose that we wanted to add a method to our objects. We could just add it to our function constructor like we would do with a normal object. Just like this:

```
...

var Person = function (name, ....

    ...

    this.calculateAge = function () {
        console.log (2016 - this.yearOfBirth);
    }
}

var john = new Person ('John, 1990, 'teacher');
john.calculateAge ();
```

Let's now create a couple of more objects here for more people:

```
var jane = new Person ('Jane', 1969, 'designer');
var mark = new Person ('Mark', 1948, 'retired');
```

Now if we run this, each of the object that is now created has the calculateAge method

attached to them. Because it's right in the constructor function of all of the objects. So each of these objects now has the method. In this case it's just one function with one line of code, so this is no big deal. But imagine that we had 20 functions in each object and that each of them would have 100 lines of code. That would not be very efficient bcoz then we would have three copies of the exact same thing. And that is exactly why we use inheritance.

We have to add all the methods and properties that we want to be inherited into the constructor's prototype property. This is what we have to do:

script.js:

```
var Person = function(name, yearOfBirth, job) {
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}
```

```
Person.prototype.calculateAge = function() {
    console.log(2016 - this.yearOfBirth);
};

var john = new Person('John', 1990, 'teacher');
var jane = new Person('Jane', 1969, 'designer');
var mark = new Person('Mark', 1948, 'retired');

john.calculateAge();
jane.calculateAge();
mark.calculateAge();
```

Output: 26
47
68.

The method is not anymore in the constructor. But we can still use it bcoz it is in the prototype property of our function constructor. So inheritance actually works.

We can also add properties instead of methods but that's rarely used. But anyway, we will have a look at that:

script.js:

```
var Person = function (name, .....
    :

Person.proto....
    :
}

Person.prototype.lastName = 'Smith';   ← we are adding a property.

var john = new Person.....
    :

mark.calculateAge();

console.log(john.lastName);
console.log(jane.lastName);
console.log(mark.lastName);
```

Output:

```
26
47
68

Smith
Smith
Smith.
```

So this property is not directly in the object. But we have access to it bcoz it's in the prototype property of the function constructor. So all john, jane and mark inherit this property.

V.IMP

5. The Prototype Chain in the Console:

The JavaScript console in our browser is a very powerful tool to inspect objects and the prototype chain. Let's now see how we can do that:

Lets use the last lecture's code: Go to console and type:

```
> john ↵
  ▶ Person {name: "John", yearOfBirth: 1990, job: "teacher"}
```

We hit that arrow and we'll get a lot of additional information about the object.

```
> john
  ▼ Person {name....}
      job: "teacher"
      name: "John"
      yearOfBirth: 1990
```

▼ __proto__ : Object

▶ calculateAge: function ()

▶ constructor: function (name, yearOfBirth, job)

lastName: "Smith"

▶ __proto__ : Object

If we type this:

> Person.prototype ↵

▼ Object {lastName: "Smith"}

▶ calculateAge: function ()

▶ constructor: function (name, yearOfBirth, job)

lastName: "Smith"

▶ __proto__: Object

To prove that john's prototype is the prototype property of the person, write this:

> john.__proto__ === Person.prototype ↵

true

If we expand the last __proto__: Object from john object, we can look at a bunch of methods that are associated to the prototype property of the object function constructor.

This is the prototype property of the function constructor and the last expanded __proto__: Object is the prototype property of the object function constructor.

And that's because the person function constructor is infact an instance of object function constructor.

hasOwnProperty:

> john.hasOwnProperty('job')↵

true

This is true bcoz the john object has a ~~prototype~~ property called job.

If we try this:

> john.hasOwnProperty('lastName')↵

false.

This is false bcoz lastName is not john's own property.

hasOwnProperty method looks for own properties.

Instanceof:

> john instanceof Person ↵

true

This is true bcoz john is an instance of Person bcoz we created it through the person function constructor.

Almost everything is an object!

var x = [2,4,6] ↵

console.info(x) ↵

▼ [2,4,6]

0: 2

1: 4

2: 6

length: 3 ↵ This is where length comes from. It's always stored in this array instance and that's why we can use this property.

▼ __proto__: Array [0]

▶ concat: ...

▶ constru...

:

In this prototype property, there are

all of the methods that we have already used before like pop, push, shift, unshift etc. All these methods are right here in the prototype property of the array. So that's why we can use them.

So all the arrays that we ever create will always have access to these methods bcoz they are stored in their prototype.

6. Creating Objects: Object.create:

In this lecture, we will talk about another way that we can create objects that inherit from a prototype. That is: object.create method.

script.js:

```
var personProto = {
    calculateAge: function() {
        console.log(2016 - this.yearOfBirth);
    }
};

var john = Object.create(personProto);
john.name = 'John';
john.yearOfBirth = 1990;
john.job = 'teacher';

var jane = Object.create(personProto,
{
    name: { value: 'Jane' },
    yearOfBirth: { value: 1969 },
    job: { value: 'designer' }
});
```

Output (in the console):

> john ↵

▶ Object { name: "John", year. . . .

> jane ↵

▶ Object { name: "Jane", ye. . . .

→ First we wrote the prototype as a simple object. We called it personProto. Here, we didnot use capital 'P' in the beginning because it's not a function constructor.

→ What we want in a prototype is the calculateAge() method.

→ Then we are creating john. We simply say Object.create and then we simply pass it the object that we define to be the prototype object which will be personProto.

→ If we now type john in the console, the --proto__: object will have calculateAge()

method inside it.

→ Now we have to fill john object with our data.

The object.create accepts a second parameter. So let's now do that for another person.

→ In jane, we are passing a second argument. Which is an object specifying the data that we want to have in the object.

If we now look at the console, both john and jane share the same prototype bcoz both have the calculateAge method and then they have both their own type of data in there.

The difference between Object.create and the function constructor pattern is that object.create builds an object that inherits directly from the one that we passed into the first argument. On the other hand, the function constructor, the newly created

object inherits from the constructor's prototype property.

One of the biggest benefits of Object.create is that it allows us to implemant a really complex inheritant structures in an easier way than function constructors bcoz it allows us to directly specify which object should be a prototype.

7. Primitives vs. Objects:

We basically know that only numbers, strings, booleans, undefined and null are primitives and that everything else are objects.

A big difference between primitives and objects is that variables containing primitives actually hold that data inside of the variable itself. On objects, it's very different variables associated with objects donot actually contain the object but instead they contain

a reference to the place in memory where the object sits, so where the object is stored. So, a variable declared as an object doesnot have a real copy of the object, it just points to that object. So let's see this in practice:

script.js:

```
// Primitives

var a = 23;
var b = a;

a = 46;
console.log(a);
console.log(b);
```

Output: 46
23

'a' is 46 bcoz we changed it from 23 to 46 and 'b' is 23 because we said that 'b' should be equal to 'a'. What we did here was to simply copy the value of 'a' to 'b' and when we then changed 'a' to

46, this did not affect the value of variable 'b' which still is 23.

This means that each of the variables actually hold their own copy of the data. They donot reference anything. So two variables holding primitives are really two different things

script.js:

```
// Objects
var obj1 = {
    name: 'John',
    age: 26
};
var obj2 = obj1;
obj1.age = 30;
console.log(obj1.age);
console.log(obj2.age);
```

Output: 30
30.

They are both 30 because when we said

object one should be equal to object two, we did not actually create a new object. No copy was created here. All that we did was to create a new reference which points to the first object. So the object one and object two variables both hold a reference that point to the exact same object in the memory and that's why when we change the age in object one, this change is also reflected on object two because it's the exact same object.

Now let's see what happens when we pass an object and a primitive into a function.

script.js:

```
// Functions
var age = 27;
var obj = {
    name: 'Jonas',
```

```
    city: 'Lisbon'
};

function change(a, b) {
    a = 30;
    b.city = 'San Francisco';
}

change(age, obj);

console.log(age);
console.log(obj.city);
```

Output: 27
San Francisco

We passed the age variable holding a primitive and the object variable holding a reference to an object into our function. This function then, as it was invoked attempted to change the arguments that we passed into it. So when we then console log. the values, we see the primitive has remained

unchanged but the city and the object has changed from Lisbon to San Francisco.

This shows us that when we pass a primitive into the function, a simple copy is created. So we can change 'a' as much as we want. It will never affect the variable on the outside because it is a primitive. But when we pass the object it's not really the object that we pass, but the reference to the object.

So we do not pass an object into a function, but only the reference that points to the object. When we then change the object inside of the function, it is still reflected outside of the function.

8. First Class Functions: Passing Functions as Arguments:

With functions, we can do the same things that we can do with objects.

Functions are also objects in JavaScript.

- A function is an instance of the object type.
- A function behaves like any other object.
- We can store functions in a variable.
- We can pass a function as an argument to another function.
- We can return a function from a function.

Because of all that, we say that in JavaScript we have first class functions.

Script.js:

```
// Pass-ing functions as arguments.
var years = [1990, 1965, 1937, 2005, 1998];
function arrayCalc(arr, fn) {
    var arrRes = [];
    for (var i=0; i< arr.length; i++) {
```

```
        arrRes.push(fn(arr[i]));
    }

    return arrRes;
}

function calculateAge(el) {
    return 2016 - el;
}

function isFullAge(el) {
    return el >= 18;
}

function maxHeartRate(el) {
    if (el >= 18 && el <= 81) {
        return Math.round(206.9 - 0.67 * el));
    } else {
        return -1;
    }
}

var ages = arrayCalc(years, calculateAge);
```

call back function.

```
var fullAges = arrayCalc(ages, isFullAge);
var rates = arrayCalc(ages, maxHeartRate);

console.log(ages);
console.log(rates);
console.log(fullAges);
```

Output: [26, 51, 79, 11, 18]

[189, 173, 154, -1, 195]

[true, true, true, false, true].

Let's now imagine that we have a couple of arrays filled with values and that we wanted to do some calculations with them.

→ We first wrote an array with some years in it.

→ We want to do some calculations with these values. We could do a huge function which does all of these calculations that we want to perform at the same time and it then resolves all the result arrays

at the same time. But that would not be really good practice. Instead, we can write a function that will receive an array and return a new result array and do the calculations based on a function that we pass into the calculation function.

→ So we wrote a calculation function and called it arrayCalc. The arguments that we passed into the arrayCalc function are. the array ie, 'years' array and a function which does the actual calculations.

→ In this arrayCalc function, we simply loop over the array and return a result. So a new empty array is created which is the one we're gonna fill. This empty array is arrRes.

→ Using 'for' loop, we are looping over the array. Inside of this loop, we will then use the function that we can pass into

this function. We will push something into our result array here. We used the push method to insert an element at the end of the array.

→ We are going to push the result of calling fn function and then we pass the current element of our input array into this function.

→ Then we have returned the result array.

→ We will write a couple of simple functions that do only one single task and these functions will actually be called "callback" functions because they are functions that we pass into functions that will then call them later. In this case, our callback function, fn, will be called here when we want to push a new element into our array.

→ We wrote the calculateAge function with

a parameter 'el' and we're returning

`2016 - el'

→ This function has only one task: it receives the 'el' argument and then returns the age based on that argument.

→ Lets see whats happening in arrayCalc function: We have our empty array here, and then we start looping through the array that we input i.e., 'years' array. When we get to this line:

```
arrRes.push(fn(arr[i]));
```

what happens is that our callback function 'fn' will be called with the 'i' element of the input array. Imagine in the first loop, 'i' is zero. So with the arr[i], we're going to retrieve the first value of the years array, so 1990 and then we'll push the 1990 into the fn function which will be calculateAge. Then the

calculateAge function does it's work and it returns '2016 - 1990' in this case and then the result will get pushed into the array and after this is completed five times, the result will be returned.

→ When we then call the arrayCalc function

```
arrayCalc(years, calculateAge);
```

Here, calculateAge is the callback function. We are not going to call the function right here because, that way we would need parentheses. We don't want to call a function here, we want it to be called later by the arrayCalc function. That's why it's called a callback function. Because it is called later We simply pass this variable which is this function right into this other function.

→ So when we console log the ages, we have our ages.

→ So our arrayCalc function looped through the years array five times and five times the calculateAge function here was called and then pushed the result right into the array that we then returned.

We can imagine this function as a generic machine that loops through an array and we can tell that machine what to do with each element in the array. That's what we do with our 'fn' parameter, which is to callback.

Then we wrote another callback function. We wrote a function which can determine if someone is of full age. Here also, we get 'el' as a parameter.

We are calling it fullAges. Again we use our generic function which we can think of as a machine that then we can use as an input the ages array that we calculated before. Then we pass the isFullAge function

into our generic calculation function. Then we console logged fullAges.

The third one is the maximum heart rate. We wrote the function maxHeartRate for that and we are taking 'el' as an input. To return as a decimal, we are using Math.round. This formula is only valid for people between 18 and 81 in ages. So we used an if-else statement for that.

Summary: We created a generic function which loops over an input array. Then we gave it a function as input which is used to calculate something based on each element of the array. We have created a bunch of different callback functions for this and we could create even more. This is way better than having one big function calculating all of this stuff at the same time because it creates more modular and readable code. Each function here, of these

three that we coded here, each one has a simple task. So this is a good practice.

9. First Class Functions: Functions Returning Functions:

script.js:

```
function interviewQuestion(job) {
    if (job === 'designer') {
        return function (name) {
            console.log(name + ', can you please
            explain what UX design is?');
        }
    } else if (job === 'teacher') {
        return function (name) {
            console.log('What subject do you teach,'
            + name + '?');
        }
    } else {
        return function (name) {
            console.log('Hello ' + name + ', what
            do you do?');
        }
    }
}
```

```
var teacherQuestion = interviewQuestion('teacher');
var designerQuestion = interviewQuestion('designer');

teacherQuestion('John');
designerQuestion('John');
designerQuestion('Jane');
designerQuestion('Mark');

interviewQuestion('teacher')('Mark');
```

This will return a function. We don't need to store this in a variable. We can call it right away. This entire thing is evaluated from left to right. First the interviewQuestion('teacher') gets called, it returns a function and with ('Mark'), we call the function that was returned before with mark

Output:

```
What subject do you teach, John?
John, can you please explain what UX is?
Jane, can you ..... UX is?
Mark, " " ..... UX is?
Mike, can you " " " is?
What subject do you teach, Mark?
```

10. Immediately Invoked Function Expressions (IIFE):

Imagine that we wanted to build a little game where we win the game if a random score from zero to nine is greater or equal to five and lose if it's smaller. But we want to keep the score hidden in this game. The answer we would think is to write something like this:

```
function game() {
    var score = Math.random() * 10;
    console.log(score >= 5);
}
game();
```

Output: false.

But this can be done in a different way. Because there's a problem with the above method.

If the only purpose is to hide the score variable from the outside, which means creating a private variable, then we don't need to

declare a whole function with a name and then call it. We can do this using IIFE.

script.js:

```
(function () {
    var score = Math.random() * 10;
    console.log(score >= 5);
})();
console.log(score);
```

invoking the function.

Output: true

▶ 'score is not defined. → we cannot see the score.

If we wrote something like this:

```
function () {

}
```

Something with out a name, parentheses then the JavaScript parser would think that this is a function declaration. But since we don't

have any name for the function declaration, then it will throw an error. So we basically need to trick the parser and make it believe that what we have here is an expression and not a declaration.

The solution is to wrap the entire thing into parentheses because in JavaScript, what's inside of parenthesis cannot be a statement. Then we invoked the function.

We can also pass arguments into our IIFE so that we can extend our function. by adding a parameter called goodLuck to the game. And the more goodLuck we add to the game, higher the chance that we win the game.

Script.js:
```
(function (goodLuck) {
    var score = Math.random() * 10;
    console.log(score >= 5 - goodLuck);
})(5);
```
↳ This is the argument.

Output: true → This will always be true.

11. Closures:

script.js:

```
function retirement (retirementAge) {
    var a = ' years left until retirement.';
    return function (yearOfBirth) {
        var age = 2016 - yearOfBirth;
        console.log ((retirementAge - age) + a);
    }
}

retirement (66) (1990);
```

Output: 40 years left until retirement.

Our inner function here is able to use the retirement variable and 'a' variable of retirement( ) function that is already returned and gone off the execution stack. But somehow the variables are still there. This is the Closure. Let's see how this works.

Closures summary: An inner function has always access to the variables and parameters of its outer function even after the outer function has returned. This is because the scope chain always stays intact.

A new function gets a new execution context that is put on top of the execution stack. That execution context has an object which stores the variables.

After the function has returned, its execution context is gone. and with it, the variable object & the entire should be gone, right? Actually, no! The secret to closures is that even after a function returns and execution context is gone, the variable object is still there. It still sits in memory and it can be accessed. The scope chain is a pointer to variable objects. The current execution context has closed in on the outer variable object, so that it can use it, so it's called Closure.

Lets write the interview Question funiction using closures.

script.js:

```
function interviewQuestion (job) {
    return function (name) {
        if (job === 'designer') {
            console.log (name + ', can you please
            explain what UX design is?');
        } else if (job === 'teacher') {
            console.log ('What subject do you
            teach, ' + name + '?');
        } else {
            console.log ('Hello ' + name +
            ', what do you do?');
        }
    }
}.
interviewQuestion ('teacher') ('John');
```

Output: What subject do you teach, John?

12. Bind, Call and Apply:

We'll talk about bind, call and apply methods. These methods allow us to call a function and set the 'this' variable manually.

script.js:

```
var john = {
    name: 'John',
    age: 26,
    job: 'teacher',
    presentation: function(style, timeOfDay) {
        if (style === 'formal') {
            console.log('Good ' + timeOfDay + ', Ladies
            and gentlemen! I\'m ' + this.name
            + ', I\'m a ' + this.job + ' and I\'m
            ' + this.age + ' years old.');
        } else if (style === 'friendly') {
            console.log('Hey! What\'s up? I\'m ' +
            this.name + ', I\'m a' + this.job + 'and
            I\'m ' + this.age + ' years old. Have
            a nice ' + timeOfDay + '.');
        }
```

```
    }
};

var emily = {
    name: 'Emily',
    age: 35,
    job: 'designer'
};

john.presentation('formal', 'morning');

john.presentation.call(emily, 'friendly', 'afternoon');

//jonn.presentation.apply(emily, ['friendly', 'afternoon']);
```

↳ We'll use apply method in our project.

```
var johnFriendly = john.presentation.bind(john,
                                          'friendly');

johnFriendly('morning');
johnFriendly('night');

var emilyFormal = john.presentation.bind(emily,
                                         'formal');

emilyFormal('afternoon');
```

Output:

Good mooning, Ladies and g.... I'm John...

Hey! ... Emily ... designer .... afternoon.

Hey! ... John ... teacher .... morning.

Hey! ... John ... teaches .... night.

Good .... Emily. .... designer ....

Bind allows us to preset some arguments, exactly the way we did here. It has a name: Carrying. Carrying is a technique in which we create a function based on another function, but with some preset parameters.

Call allows us to set the 'this' variable in the first argument as we did here:

john.presentation.call(emily, 'friendly', afternoon);

and we set it to emily bcoz we wanted to use John's presentation method. but setting the 'this' variable to "emily".

Apply accepts arguments as an array. So that's only two arguments: first 'this' variable and then an array where all the other

arguments go.

Let's look at an other example:

script.js:

```
var years = [1990, 1965, 1937, 2005, 1998];

function arrayCalc (arr, fn) {
    var arrRes = [];
    for (var i=0; i< arr.length; i++){
        arrRes.push (fn(arr[i]));
    }
    return arrRes;
}

function calculateAge (el){
    return 2016- el;
}

function isFullAge (limit, el) {
    return el >= limit;

var ages = arrayCalc (years, calculateAge);

var fullJapan = arrayCalc (ages, isFullAge.bind
                    This is the preset limit. ← (this, 20));
```

```
console.log(ages);
console.log(fullJapan);
```

Output: [26, 51, 79, 11, 18]

[true, true, true, false, false].