Object Oriented Programming

Method (Function inside Object).

Factory function

Constructor function

predicate function

arrow function syntax

methods.includes

methods.push()

methods.unshift()

methods.splice()

methods.pop() removes the last element in an array and return it.

Methods.shift() removes an element from the beginning of an array, and returns it.

Methods.spice() removes an element from the middle of an array.

Method.length

.concat()

Emptying an array - x = [] or x.length = 0. Or x.splice(0, x.length) or x.pop() in a while loop.

method.find

*var array1 = [5, 12, 8, 130, 44}*

*var found = array1****.find****(function(element) {*

*return element > 10;*

*});*

let post = new Post('a', 'b', 'c')

console.log(post)

function Post(title, body, author) {

    this.title = title;

    this.body = body;

    this.author = author;

    this.views = 0;

    this.comments = [];

    this.isLive = false;

}

“To recap. When we use the new operator three things happen. This operator first creates an empty object. Then, it will set this to point to this object. And finally, it will return that object from this function”.

“So, what we get ***here*** is that new object, and we simply set circle to point to that object.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const circle = *new Circle(1);*

const x = {}

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call({}, 1);

function createCircle(radius, location) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const circle1 = createCircle(1);

console.log(circle1);

const circle2 = createCircle(2);

console.log(circle2);

Constructor

string literals

Boolean literals

Object.assign

Spread operator

Constructor - That references the function that was used to construct or create an object. Mosh.

let selectedColor = null;

**Key- Variable Type**

**Word**

Type of Variable.

Statement:

console.log('Hello World');

    </script>

  </body>

The highlighted code is a statement. A statement is a piece of code that expresses an action to be carried out. In this case, we want to log *log* a message *‘Hello World’*, on the *console*.

String:

console.log('Hello World');

What we have here in between single “code?” is called a string. A string is a sequence of characters.

Notation:

// In JavaScript we also have this notation: // We can add two slashes and this represents a comment.  
Here we can add some description to our code and this description is ignored by the JavaScript engine. It is not executed. It is purely for documenting the code when you want to explain to other developers why you have written the code this way. You don’t want to explain what the code does because that should be clear in the code itself. We want to explain Why’s and Hows.

<script>

        // Comment

        console.log('Hello World');

    </script>

Variable:

In programming we use a variable to store data temporarily in a computer’s memory.

In this case, the variable is the name portion.

let name = 'Mosh';

4 Ways to **Declare** a JavaScript Variable:

* Using var
* Using let
* Const vs let:

The best practice is that if you don’t need to reassign, constant is your best choice. If you need to reassign a variable, use let.

Using const

Using nothing

In this example, x, y, and z, are variables, declared with the var keyword:

var x = 5;  
var y = 6;  
var z = x + y;

* camel notation - the first letter of the first word is lowercase, and the first letter of every word after should be uppercase.
* **Object Property\*** - From [W3Schools](https://www.w3schools.com/js/js_object_properties.asp) - Properties are the most important part of any JavaScript object.

**JavaScript Properties**

Properties are the values associated with a JavaScript object.

A JavaScript object is a collection of unordered properties.

Properties can usually be changed, added, and deleted, but some are read only.

**Accessing JavaScript Properties**

The syntax for accessing the property of an object is:

*objectName.property*// person.age

or

*objectName*["*property*"]   // person["age"]

or

*objectName*[*expression*]   // x = "age"; person[x]

The expression must evaluate to a property name.

**Example 1:**

person.firstname + " is " + person.age + " years old.";  
  
  
  
***<h2>JavaScript Object Properties</h2>***

***<p>Access a property with .property:</p>***

***<p id="demo"></p>***

***<script>***

***const person = {***

***firstname: "John",***

***lastname: "Doe",***

***age: 50,***

***eyecolor: "blue"***

***};***

***document.getElementById("demo").innerHTML = person.firstname + " is " + person.age + " years old.";***

***</script>***

**JavaScript Object Properties**

Access a property with .property:

John is 50 years old.

**Example 2:**

person["firstname"] + " is " + person["age"] + " years old.";

***<h2>JavaScript Object Properties</h2>***

***<p>Access a property with ["property"]:</p>***

***<p id="demo"></p>***

***<script>***

***const person = {***

***firstname: "John",***

***lastname: "Doe",***

***age: 50,***

***eyecolor: "blue"***

***};***

***document.getElementById("demo").innerHTML = person["firstname"] + " is " + person["age"] + " years old.";***

***</script>***

**JavaScript Object Properties**

Access a property with ["property"]:

John is 50 years old.

JavaScript for…in Loop

The JavaScript **for...in** statement loops through the properties of an object.

**Syntax:**

for (let *variable* in *object*) {  
*// code to be executed*  
}

The block of code inside of the **for...in** loop will be executed once for each property.

Looping through the properties of an object:  
**Example:**

const person = {  
  fname:" John",  
  lname:" Doe",  
  age: 25  
};  
  
for (let x in person) {  
  txt += person[x];  
}

<h2>JavaScript Object Properties</h2>

<p>Looping object property values:</p>

<p id="demo"></p>

<script>

const person = {

fname:"John",

lname:"Doe",

age:25

};

let txt = "";

for (let x in person) {

txt += person[x] + " ";

}

document.getElementById("demo").innerHTML = txt;

</script>

**JavaScript Object Properties**

Looping object property values:

John Doe 25

**Adding New Properties**

You can add new properties to an existing object by simply giving it a value.

Assume that the person object already exists - you can then give it new properties:

**Example:**

person.nationality = "English";

<h2>JavaScript Object Properties</h2>

<p>Add a new property to an existing object:</p>

<p id="demo"></p>

<script>

const person = {

firstname: "John",

lastname: "Doe",

age: 50,

eyecolor: "blue"

};

person.nationality = "English";

document.getElementById("demo").innerHTML =

person.firstname + " is " + person.nationality + ".";

</script>

**JavaScript Object Properties**

Add a new property to an existing object:

John is English.

**Deleting Properties**

The **delete** keyword deletes a property from an object:

**Example:**

const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};  
  
delete person.age;

<h2>JavaScript Object Properties</h2>

<p>Deleting object properties.</p>

<p id="demo"></p>

<script>

const person = {

firstname: "John",

lastname: "Doe",

age: 50,

eyecolor: "blue"

};

delete person.age;

document.getElementById("demo").innerHTML =

person.firstname + " is " + person.age + " years old.";

</script>

**JavaScript Object Properties**

Deleting object properties.

John is undefined years old.

or delete person["age"];

**Example:**

const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};  
  
delete person["age"];

<h2>JavaScript Object Properties</h2>

<p>Deleting object properties.</p>

<p id="demo"></p>

<script>

const person = {

firstname: "John",

lastname: "Doe",

age: 50,

eyecolor: "blue"

};

delete person["age"];

document.getElementById("demo").innerHTML =

person.firstname + " is " + person.age + " years old.";

</script>

**JavaScript Object Properties**

Deleting object properties.

John is undefined years old.

The **delete** keyword deletes both the value of the property and the property itself.

After **deletion**, the property cannot be used before it is added back again.

The **delete** operator is designed to be used on object properties. It has no effect on variables or functions.

The **delete** operator should not be used on predefined JavaScript object properties. It can crash your application.

**Nested Objects**

Values in an object can be another object:

**Example:**

myObj = {  
  name:"John",  
  age:30,  
  cars: {  
    car1:"Ford",  
    car2:"BMW",  
    car3:"Fiat"  
  }  
}

You can access nested objects using the dot notation or the bracket notation:

**Example:**

myObj.cars.car2;

<h2>JavaScript Objects</h2>

<p>Access nested objects:</p>

<p id="demo"></p>

<script>

const myObj = {

name: "John",

age: 30,

cars: {

car1: "Ford",

car2: "BMW",

car3: "Fiat"

}

}

document.getElementById("demo").innerHTML = myObj.cars.car2;

</script>

**JavaScript Objects**

Access nested objects:

BMW

or:

**Example:**

myObj.cars["car2"];

<h2>JavaScript Objects</h2>

<p>Access nested objects:</p>

<p id="demo"></p>

<script>

const myObj = {

name: "John",

age: 30,

cars: {

car1: "Ford",

car2: "BMW",

car3: "Fiat"

}

}

document.getElementById("demo").innerHTML = myObj.cars["car2"];

</script>

**JavaScript Objects**

Access nested objects:

BMW

**Or:**

**Example:**

myObj["cars"]["car2"];

<h2>JavaScript Objects</h2>

<p>Access nested objects:</p>

<p id="demo"></p>

<script>

const myObj = {

name: "John",

age: 30,

cars: {

car1: "Ford",

car2: "BMW",

car3: "Fiat"

}

}

document.getElementById("demo").innerHTML = myObj["cars"]["car2"];

</script>

**JavaScript Objects**

Access nested objects:

BMW

**Or:**

**Example:**

let p1 = "cars";  
let p2 = "car2";  
myObj[p1][p2];

<h2>JavaScript Objects</h2>

<p>Access nested objects:</p>

<p id="demo"></p>

<script>

const myObj = {

name: "John",

age: 30,

cars: {

car1: "Ford",

car2: "BMW",

car3: "Fiat"

}

}

let p1 = "cars";

let p2 = "car2";

document.getElementById("demo").innerHTML = myObj[p1][p2];

</script>

**JavaScript Objects**

Access nested objects:

BMW

**Nested Arrays and Objects**

Values in objects can be arrays, and values in arrays can be objects:

**Example:**

const myObj = {  
  name: "John",  
  age: 30,  
  cars: [  
    {name:"Ford", models:["Fiesta", "Focus", "Mustang"]},  
    {name:"BMW", models:["320", "X3", "X5"]},  
    {name:"Fiat", models:["500", "Panda"]}  
  ]  
}

To access arrays inside arrays, use a for-in loop for each array:

**Example:**

for (let i in myObj.cars) {  
  x += "<h1>" + myObj.cars[i].name + "</h1>";  
  for (let j in myObj.cars[i].models) {  
    x += myObj.cars[i].models[j];  
  }  
}

<h2>Nested JavaScript Objects and Arrays.</h2>

<p id="demo"></p>

<script>

let x = "";

const myObj = {

name: "John",

age: 30,

cars: [

{name:"Ford", models:["Fiesta", "Focus", "Mustang"]},

{name:"BMW", models:["320", "X3", "X5"]},

{name:"Fiat", models:["500", "Panda"]}

]

}

for (let i in myObj.cars) {

x += "<h2>" + myObj.cars[i].name + "</h2>";

for (let j in myObj.cars[i].models) {

x += myObj.cars[i].models[j] + "<br>";

}

}

document.getElementById("demo").innerHTML = x;

</script>

**Nested JavaScript Objects and Arrays.**

**Ford**

Fiesta  
Focus  
Mustang

**BMW**

320  
X3  
X5

**Fiat**

500  
Panda

**Property Attributes**

All properties have a name. In addition they also have a value.

The value is one of the property's attributes.

Other attributes are: enumerable, configurable, and writable.

These attributes define how the property can be accessed (is it readable?, is it writable?)

In JavaScript, all attributes can be read, but only the value attribute can be changed (and only if the property is writable).

( ECMAScript 5 has methods for both getting and setting all property attributes)

**Property Prototypes**

JavaScript objects inherit the properties of their prototype.

The **delete**keyword does not delete inherited properties, but if you delete a prototype property, it will affect all objects inherited from the prototype.

* Keyword. In JavaScript you cannot use these reserved words as variables, labels, or function names:

Keywords are **reserved words that are part of the syntax in the programming language**. For example, const a = 'hello'; Here, const is a keyword that denotes that a is a constant.

* The Value of a variable:

*let firstName =* ***theValue***

I do believe the value is the second part.

let firstName = undefined;

“In this example, *because* we have set *firstName* to *undefined* as a value, it’s type is also undefined.”

* Variable - In the following statement, the **variable** is age, and the type of age is a string ‘30’?

*Let* ***age*** *= 30;*

* Primitive types type:

1. boolen,
2. null,
3. undefined type   
   number type
4. string type,
5. etc

These are the examples of primitives/value types.

let name = 'Mosh';  //This is a String Literal

let age = 30;  //Number Literal

let isApproved = true;  // Boolean Literal

let firstNames = undefined;  //undefined

let lastNames = null; //null

* Boolean - A boolean can either be true or false. We use Boolean in situations where we want to have some logic. E.g., If the order is approved, it needs to be shipped. So, the value of Boolean variable can be true or false.

E.g., *let isApproved = true;*

* Undefined - (From Mosh’s Primitive Types). When we fail to initialize a variable, it is undefined by default.

E.g.: *let firstName;* or *let firstName = undefined;*

* Null - Used when we Explicitly want to clear the value of a variable. More common that ‘undefined’.

*E.g., let lastNames = null;*

* Dynamic Language - 04 - Dynamic Typing Mosh - In static languages, when we declare a variable, the Type of that variable is set and it cannot be changed in the future. E.g., *string name = ‘John’;* ((The value of name variable (a string type, John) cannot be altered.))

Whereas in a dynamic language, “The type of these variable will be determined at runtime, based on the values we assign to them”   
  
E.g., *let name = ‘John’;* ((Let allows the variable to be changed later. Both the content (value?) and the type, I believe. In the example, Mosh changed the name to a number in the console, changing the variable type to a number.))

* Typeof - An operator that tells us the type of a given variable.   
    
  E.g., for the following code:  
    
    
    
   *type of* name  
  ‘string’
* Floating point number - A decimalized number? Like 30.1. ((In JS, there aren’t floating point numbers and integers. Just *number.*
* Reference types (In contrast to Primitive/ Value Types) (Mosh 05 - Objects)
  + Objects
  + Arrays
  + Functions
* Objects - (Mosh 05 - Objects) - When we’re dealing with multiple related variables, we can put these variables inside an object.

E.g., here we have two variables: name and age.

let name = 'Mosh';

let age = 30;

They are highly related; they are part of the representation of a person. So instead of declaring two variables, we can declare a person Object. Then instead of referencing these two variables, we can simply reference the person object. It makes for cleaner code.

let person = {

    name: 'Mosh',

    age: 30

};

The console output, which Mosh refers to as a “our person object”: {name: “Mosh”, age: 30}

* Object literal - (Mosh - 05 - Objects) - Z def. When defining an object, all the variables (key value pairs?) inside the object (marked by {} ) make an object literal.

In the example below, note how the name and age variables (probably more accurately called key value pairs?) fit nicely inside the object.

let name = 'Mosh';

let age = 30;

let person = {

    name: 'Mosh',

    age: 30

};

* Key value pairs - Z Key:value pairs - (Mosh - 05 - Objects) - “The keys (Key value pairs?) are what we call the properties of this object.” In this case, we want the person object to have two properties, or two keys: name and age.

Name [this is the key]: [after that, we set the value[[type **&** value?]] ] ‘Mosh’ [add a comma], [another key value pair] [the key ‘age’] age: [the value] 30

let name = 'Mosh';

let age = 30;

let person = {

    name: 'Mosh',

    age: 30

};

* Properties - (Mosh - 05 - Objects) - Aka Key:value pairs.  
    
  “Now we have a person object with two properties, or two key:value pairs ((Highlights name: ‘Mosh’)): name and age.”





‘We see our person object {name: “Mosh”, age: 30}. “Note the object literal syntax” - ((He means the curly braces)).’

‘We have a couple key value pairs. ((Mosh wave his mouse over both age and its value, and name and its value)) They are the Properties of the person object.’

*age:30 Name: ‘mosh’*

Key:value Key: Value

Key Value Pair Key Value Pair

Property Property

* Dot notation - Z def - mechanism for altering the properties (listed as name: \_\_\_ value) of objects without directly modifying an object.

E.g.:



To this:

person.name = 'John';

(One can utilize the console to display a specific object property. E.g., instead of *console.log(person)* you can *console.log(person.name)*. ((If you input objectname.property into console.log, Console will pull from “person.name =” before pulling from the object)).  
  
Dot notation is more concise than bracket notation and is therefore preferable.

* Bracket Notation - Mosh - 05 - Objects - See Dot Notation - To access an object’s properties, name the object, and put the desired property in brackets and single quotes.

E.g., *person[‘name’] = ‘Mary’* ((Instead of, say, ‘Mosh’))

Bracket notation is less concise than dot notation, and is less preferable.  
  
However, if you don’t know the name of the target property before runtime, bracket notation can be useful. -

E.g., we can create another variable elsewhere that will fill that data:  
  
*let selection = ‘name’* The user may input the data into the name. And in the bracket, we write:  
  
person[selection] = ‘Mary’;

* Array - *Mosh 06 - Arrays* - “A data structure that we use to represent a list of items”. Object for storing lists. (We can access it’s properties/key:value pairs utilizing dot notation. It also seems to inherit numerous properties, e.g., length). Each element has an index, noting its position in the Array.   
    
  E.g., *let selectedColors = [‘red’, ‘blue’, ‘yellow’];*To access a point in the Array utilizing an index, instead of *console.log(selectedColors)*, use *console.log(selectedColors[0])*

Because Array’s are dynamic, they can be modified with additional elements, and of different types: E.g.,  
  
*selectedColors[3] = ‘6’;*

* Array literal - Mosh 06 - Arrays - Indicate an empty array.
* **Function\*** - Mosh 07 - A function is basically a set of statements that performs the task or calculates a value. (Statements are not variables and do not require semicolons after them).

function greet () {

    console.log('Hello World');

“The parentheses are part of the syntax for declaring functions.”

Inside the curly braces is the body of the function. This is where we add statements to define some kind of logic in our application. E.g. the logic in the function above is to display a message on the console.

The function can be called in the following manner: *greet();* This is a statement.

A variable can be added within the parenthesis. This variable is referred to as a **parameter**. It is only meaningful inside the function. It is not accessible outside this function.  
  
E.g.,

function greet (name) {

    console.log('Hello' + name);

}

greet(‘John’);

* Parameter - The accurate term for the ‘variables’ inside the () portion of a function.
* **Argument\*** - Mosh 07 - Functions - In the following statement:

[[[I explain function thusly: John is an argument that defines the name variable/parameter. Newly defined name variable/parameter equals **undefined name variable**, which is inside the body of the function]]]

So, we can pass John between the brackets. We refer to this as an argument.

function greet (name) {

    console.log('Hello' + **name**);

}

greet(‘John’);

John is an Argument to the greet function. Name is a parameter [[[variable]]] of the greet function. The argument is the actual value we supply for the parameter.

Function can have Multiple parameters, utilizing commas. Likewise, functions can contain multiple arguments. If you don’t define them with an argument, the parameters will display as ‘undefined’.

This function can be reused by simply adding another greet function.

function greet (name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', ‘Smith’);

greet('Mary');

* Concatenations - Mosh - 08 - Types of Function - Whatever all this is:   
    
  console.log(***'Hello ' + name + ' ' + lastName***);
* Function call - Mosh - 08 Types of Functions - Apparently console.log() and square(2) are function calls:

function square(number) {

    return number \* number;

}

console.log(square(2));

Apparently, the return keyword, will return this value to whoever is calling this function.

“Let’s call the square function” We pass 2. This returns a value;

function square(number) {

    return number \* number;

}

square();

With *console.log()* we are calling the log function “which is defined somewhere”, and passing an argument. That argument could be a simple string, or an expression. The expression could be a call to another function, like square(2).

* Operators - Mosh - 01 -JS Operators - We use operators along with Variables to create expressions, which we can use to implement logic and algorithms.
  + Arithmetic
  + Assignment
  + Comparison
  + Logical
  + Bitwise
* Arithmetic Operators - Mosh - 03 - Arithmetic Operators - We use them for performing calculations, just like in mathematics. They usually take at least two operands, like X + Y, and then produce a new value.  
    
  
* Addition Operator - Mosh - 02 - Arithmetic Operators - See below

let x = 10;

let y = 3;

console.log(x + Y);

* More Operators:
  + // console.log(x + Y);
  + // console.log(x - Y);
  + // console.log(x \* Y);
  + // console.log(x / Y);
  + // console.log(x % Y); Remainer of division
  + // console.log(x \*\* Y); Exponential? X to the power of Y
  + // console.log(++x); Increment operator
  + // console.log(x++); Increment operator
  + // console.log(--x);
  + // console.log(x--);
* Increment Operator ++ - Mosh - 02 - Arithmetic Operators - Increment operators are indicated by two plus signs (++). Depending on where we put the plus signs this operator will behave differently.

let x = 10;

let y = 3;

console.log(++x);

Displays 11.

This is where things get Weird. If we put this operator After X like so:

let x = 10;

let y = 3;

console.log(x++);

The value of X displays first. And Then the value of X will be incremented by one. So, if we do a Second log:

let x = 10;

let y = 3;

console.log(x++);

console.log(x)

10, Then 11 displays on the console.

* Addition Assignment Operator += - Not included by Mosh, but in a quiz. From [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Addition_assignment) - The addition assignment operator (+=) adds the value of the right operand to a variable and assigns the result to the variable. The types of the two operands determine the behavior of the addition assignment operator. Addition or concatenation is possible. E.g.,

*let a = 2;*

*let b = 'hello';*

*console.log(a += 3); // addition*

*// expected output: 5*

*console.log(b += ' world'); // concatenation*

*// expected output: "hello world"*

* Expression - Mosh - 02 - Arithmetic Operators - An expression is something that produces a value. E.g., X + Y.
* Comment out - Mosh - 03 - Arithmetic Operators - Highlight a group of text in VSCode and control and slash. It turns text into this:

//Relational

* Comparison Operators - Mosh - 04 - Comparison Operators - The result of an expression that includes a comparison operator is a Boolean… It’s true or false.

let x = 1;

//Relational

console.log(x > 0);

console.log(x >= 1);

console.log(x < 1);

console.log(x <= 1);

//Equality

console.log(x === 1);

console.log(x !== 1);

* Relational Operators - Mosh - 04 - Comparison Operators - Comparison Operators of the following type:

console.log(x > 0);

console.log(x >= 1);

console.log(x < 1);

console.log(x <= 1);

* Equality Operators - Mosh - 04 - Comparison Operators - Comparison Operators of the following type:

//Equality

console.log(x === 1); Is x equal to 1.

console.log(x !== 1); Is x Not equal to 1.

* Strict Equality Operator - Mosh - 05 - Equality Operators - Type and Value must be the same. Represented by === . E.g.,

// Strict Equality Operator (Type + Value)

console.log(1 === 1);

The following example is Not equal

console.log('1' === 1);

Generally, you will use the strict equality operator because it is more precise and accurate.

* Lose Equality Operator - Mosh - 05 - Equality Operators - Value but not type must be the same. Represented by == . E.g.,

// Lose Eqaulity Operator

console.log(1 == 1);

console.log('1' == 1);

JS will convert the type on the right, to match the type on the left. It will also convert Booleans:

console.log(true == 1);

The operator will automatically convert the value on the right side to a Boolean.

The console will also read this as true.

“The lose equality operator does not care about the types matching; if the types don’t match it will convert that type on the right side to match the type on the left side. Then, it will only check if the values are equal”

* Strict Inequality Operator - From [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_inequality) - This is a Strict Inequality Operator. “The strict inequality operator ( !== ) **checks whether its two operands are not equal, returning a Boolean result**. Unlike the inequality operator, the strict inequality operator always considers operands of different types to be different.”

console.log(1 !== 1);

// expected output: false

console.log('hello' !== 'hello');

// expected output: false

console.log('1' !== 1);

// expected output: true

console.log(0 !== false);

// expected output: true

* Turnery or Conditional Operator - Mosh - 06 - ternary Operator - One of Mosh’s favorites.   
    
  Z def. Examine this code:

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

Start with a condition ((using a conditional operator)). If that condition evaluates to true, then the type will receive the value gold. If false, silver.

* Logical Operators - Mosh - 07 - Logical Operators - We use logical operators to make decisions based on multiple conditions. In JavaScript we have three kinds of logical operators:
* Logical and
* Logical or
* Not
* Logical And\* Logical &&\* - Mosh - 07 - Logical Operators - Returns True if Both operands are True:

console.log(true && true)

Or:

let eligibleForLoan = highIncome && goodCreditScore;

let highIncome = true;

let goodCreditScore = true;

let eligibleForLoan = highIncome && goodCreditScore;

console.log(eligibleForLoan);

* Logical Or - Mosh - 07 - Logical Operators - Logical OR is indicated by two vertical lines. || This returns true if one or both of the operands are TRUE.

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

* Logical Not - Mosh - 07 - Logical Operators - The not operator is indicated by an exclamation mark. ! Z def. Inverts the value of true false booleans?

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

let applicationRefused = !eligibleForLoan;

console.log('Eligible', eligibleForLoan);

console.log('Application Refused', applicationRefused);

Executes the opposite of a given value.

* Falsy - Mosh - 08 - Logical Operators with Non-Booleans - If an operand ((pair)) is not a Boolean true or false, it will try to interpret it as what we call truthy or falsy. E.g.,   
    
  I think:  
  0 || ‘’  
    
    
    
   The values of falsy are:
* Undefined
* Null
* The number zero 0
* Boolean false
* Empty string ‘’
* Not a number NaN groups
* Truthy - Mosh - 08 - Logical Operators with Non-Booleans - Anything that is not falsy ((and is not a Boolean)) is truthy.

false || 1 || 2

The console outputs 1, as logical Or returns first true answer it finds.   
  
false || ‘Mosh’  
‘Mosh’  
  
False || 1  
1

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

* Short-Circuiting - Mosh - 08 - Logical Operators with Non-Booleans - Z def - Multiple operands separated by Logical Ors will ignore all but the first true/truthy? operand it encounters, ignoring the rest.
* Bit - Mosh - 09 - Bitwise Operators - One of the 8 digits or bits, that make up a single number:  
  1 = 00000001
* Byte - Mosh - 09 - Bitwise Operators - Eight bits. Eight bits = 1 byte. 0000001 = 1 byte.
* Bitwise Operator - Mosh - 09 - Bitwise Operators - “Bitwise operators in JavaScript or any other programming languages are similar to logical operators but they work on the individual bits of a number.” Primarily we use the Logical Or and Logical & operators.
* Bitwise Or - Mosh - 09 - Bitwise Operators - See Logical Or - Bitwise Or is represented by a single vertical line | , in contrast to a Logical Or with a double vertical line ||

console.log(1 | 2);

With Bitwise Or, the process looks like this:

1 = 0000001

2 = 0000010

R = 0000011 ((AKA 3))

Now, this operator here is going to look at each of these bits in a vertical way. If either of these bits is one, the result will be one. Otherwise it will be zero”

* 0+0=0
* 0+1=1
* 1+1=1

00000001+ 00000010 = 00000011 Or, 1 + 2 = 3

* Bitwise And, Bitwise & - Mosh - 09 - Bitwise Operators - See Logical & - If both numbers are one, one will be returned, otherwise the result will be zero.

console.log(1 & 2); // Bitwise And

1 = 0000001

2 = 0000010

3 = 0000011

R = 0000000

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

In the above, myPermission equals zero Or (bitwise Or) readPermission is 4 aka 0000010, or writePermission 2 aka 00000010.

And message (which is logged on the console) is myPermission aka 0 And (Logical &) readPermission 4. If both are zero, the conditional/turnery operator evaluates the statement as false, and returns no, if readPermission 4 is True, the turnery operator evaluates the statement as true and returns yes.  
  
Hence, the expressions above are utilizing bitwise values to determine statements values.

“So, here’s what I want you to take away. With the bitwise or operator we can add permissions. And with the bitwise & operator we can check to see if we have a given permission.”

* ***Condition\* (In context of if…else)*** - From [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else#syntax) - An [expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#expressions) that is considered to be either [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) or [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy).
* Conditional Statement\* - From [W3Schools](https://www.w3schools.com/js/js_if_else.asp) - Conditional statements are used to perform different actions based on different conditions.

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

* + Use **if** to specify a block of code to be executed, if a specified condition is true
  + Use **else** to specify a block of code to be executed, if the same condition is false
  + Use **else if** to specify a new condition to test, if the first condition is false
  + Use **switch** to specify many alternative blocks of code to be executed

((Zacks example)):

let hour =  4;

if (hour >= 6 && hour < 12 )

    console.log('Good morning');

else if (hour >= 12 && hour < 18)

    console.log('Good Afternoon!');

else

console.log('Good evening');

* if\* - [W3Schools](https://www.w3schools.com/js/js_if_else.asp) - Use ***if*** to specify a block of code to be executed, if a specified condition is true.

**Syntax**:  
*if (condition) {  
  //*  block of code to be executed if the condition is true *}*

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.  
  
**Example**:

Make a "Good day" greeting if the hour is less than 18:00:

*if (hour < 18) {  
  greeting =*"Good day"*;  
}*

The result of greeting will be:

Good day… I said GOOD DAY!!!!  
  
From - [MDM](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) - The **if** statement executes a statement if a specified condition is [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy). If the condition is [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy), another statement can be executed.

* Else\* - See if… else (Conditional Statement) - From [W3Schools](https://www.w3schools.com/js/js_if_else.asp) - Use the ***else*** statement to specify a block of code to be executed if the condition is false.  
    
  if (*condition*) {  
    //  block of code to be executed if the condition is true} else {  
    //  block of code to be executed if the condition is false}

Example:  
If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

if (hour < 18) {  
  greeting = "Good day";  
} else {  
  greeting = "Good evening";  
}

The result of greeting will be:  
  
Good day. I said GOOD DAY!!!

* If…else\* - From [W3Schools](https://www.w3schools.com/js/js_if_else.asp) - Use the else if statement to specify a new condition if the first condition is false.  
    
  **Syntax:**  
  if (*condition1*) {  
    //  block of code to be executed if condition1 is true} else if (*condition2*) {  
    //  block of code to be executed if the condition1 is false and condition2 is true  
  } else {  
    //  block of code to be executed if the condition1 is false and condition2 is false}

**Example:**

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":  
  
*if (time < 10) {  
  greeting =*"Good morning"*;  
} else if (time < 20) {  
  greeting =*"Good day"*;  
} else {  
  greeting =*"Good evening"*;  
}*

The result of greeting will be:  
*Good day*

From - [MDM](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) - The **if** statement executes a statement if a specified condition is [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy). If the condition is [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy), another statement can be executed.  
  
*function testNum(a) {*

*let result;*

*if (a > 0) {*

*result = 'positive';*

*} else {*

*result = 'NOT positive';*

*}*

*return result;*

*}*

*console.log(testNum(-5));*

*// expected output: "NOT positive"*"NOT positive"

**Syntax:**

*if (condition) {*

*statement1*

*} else {*

*statement2*

*}*

**condition**

An [expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#expressions) that is considered to be either [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) or [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy).

**statement1**

Statement that is executed if condition is [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy). Can be any statement, including further nested if statements. To execute multiple statements, use a [block](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) statement ({ /\* ... \*/ }) to group those statements. To execute no statements, use an [empty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/Empty) statement.

**statement2**

Statement that is executed if condition is [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy) and the else clause exists. Can be any statement, including block statements and further nested if statements.

**Description**

Multiple **if...else** statements can be nested to create an **else if** clause. Note that there is no **elseif** (in one word) keyword in JavaScript.

*if (condition1)*

*statement1*

*else if (condition2)*

*statement2*

*else if (condition3)*

*statement3*

*...*

*else*

*statementN*

To see how this works, this is how it would look if the nesting were properly indented:

*if (condition1)*

*statement1*

*else*

*if (condition2)*

*statement2*

*else*

*if (condition3)*

*...*

To execute multiple statements within a clause, use a block statement **({ /\* ... \*/ }**) to group those statements. In general, it is a good practice to always use block statements, especially in code involving nested **if** statements:

*if (condition) {*

*statements1*

*} else {*

*statements2*

*}*

Do not confuse the primitive Boolean values **true** and **false** with truthiness or falsiness of the [**Boolean**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) object. Any value that is not **false**, **undefined**, **null**, **0**, **-0**, **NaN**, or the empty string (**""**), and any object, including a Boolean object whose value is **false**, is considered [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) when used as the condition. For example:

*var b = new Boolean(false);*

*if (b) // this condition is truthy*

**Examples:**

*Using if…else  
  
if (cipher\_char === from\_char) {*

*result = result + to\_char;*

*x++;*

*} else {*

*result = result + clear\_char;*

*}*

**Using else if**

*Note that there is no****elseif****syntax in JavaScript. However, you can write it with a space between****else****and****if****:*

*if (x > 50) {*

*/\* do something \*/*

*} else if (x > 5) {*

*/\* do something \*/*

*} else {*

*/\* do something \*/*

*}*

**Using an assignment as a condtion**

You should almost never have an **if...else** with an assignment like "**x = y**" as a condition:

*if (x = y) {*

*/\* do something \*/*

*}*

However, in the rare case you find yourself wanting to do something like that, the [while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while) documentation has a [Using an assignment as a condition](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while#using_an_assignment_as_a_condition) section with an example showing a general best-practice syntax you should know about and follow

* **Switch… case\* Switch\* Switch and case\*** - Z definition. The case, in switch and case means, “Switch means X. In that Case, do Y”

Abbreviated W3Schools - This is how it works:

* The switch expression is evaluated once.
* The value of the expression [[aka variable?]] is compared with the values of each case.
* *If there is a match, the associated block of code is executed.*
* If there is no match, the default code block is executed.

*let role = 'guest';*

switch (role) {

case 'guest':

    console.log('Guest User');

break;

Break\* Subentry - See Below - From Z [[Utilize break to stop All case statement code from being executed. Otherwise, subsequent code will also be executed. The break “allows us to jump out of the switch block”. Though, if subsequent statements evaluate to true, the may also be executed… not sure ]]

Default\* Subentry - See Below - From Z [[Executes if none of the case statements match]]

From Mosh - 04 - Control Flow - 02 - Switch… Case -

With switch in case, we can compare the value of a variable against multiple other values. They don’t have to be strings, they can be numbers or even Boolean’s… though using Boolean’s here is less common… Because if you want to compare the value of a variable with true and false it makes more sense to use an if statement.

let role = 'guest';

switch (role) {

case 'guest':

    console.log('Guest User');

break;

case 'moderator':

    console.log('Moderator User');

    break;

    default:

        console.log('Unknown User');

}

From [W3Schools](https://www.w3schools.com/js/js_switch.asp) - JavaScript Switch Statement (AKA Switch Case?) - Use the **switch** statement to select one of many code blocks to be executed.

**Syntax:**

switch(expression) {  
  case x:  
    *// code block*    break;  
  case y:  
    *// code block*    break;  
  default:  
    // code block  
}

This is how it works:

* The switch expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* If there is no match, the default code block is executed.

**Example:**  
The **getDay()** method returns the weekday as a number between 0 and 6.

(Sunday=0, Monday=1, Tuesday=2 ..)

This example uses the weekday number to calculate the weekday name:

switch (new Date().getDay()) {  
  case 0:  
    day = "Sunday";  
    break;  
  case 1:  
    day = "Monday";  
    break;  
  case 2:  
     day = "Tuesday";  
    break;  
  case 3:  
    day = "Wednesday";  
    break;  
  case 4:  
    day = "Thursday";  
    break;  
  case 5:  
    day = "Friday";  
    break;  
  case 6:  
    day = "Saturday";  
}

The result of day will be:

*Thursday* ((It outputs Thursday, because it is Thursday today)).

**The break\* Keyword.** (Subentry)

When JavaScript reaches a **break** keyword, it breaks out of the switch block.

This will stop the execution inside the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

**Note:**If you omit the break statement, the next case will be executed even if the evaluation does not match the case.

**The** **default\* Keyword**  (Subentry)

The **default** keyword specifies the code to run if there is no case match:

**Example:**

The **getDay()** method returns the weekday as a number between 0 and 6.

If today is neither Saturday (6) nor Sunday (0), write a default message:

*switch (new Date().getDay()) {  
  case 6:  
    text =*"Today is Saturday"*;  
    break;  
  case 0:  
    text =*"Today is Sunday"*;  
    break;  
  default:  
    text =*"Looking forward to the Weekend"*;  
}*

The result of text will be:

*Looking forward to the Weekend*

The **default** case does not have to be the last case in a switch block:

**Example:**

*switch (new Date().getDay()) {  
  default:  
    text =*"Looking forward to the Weekend"*;  
    break;  
  case 6:  
    text =*"Today is Saturday"*;  
    break;  
  case 0:  
    text =*"Today is Sunday"*;  
}*

If **default** is not the last case in the switch block, remember to end the default case with a break.

**Common Code Blocks**

Sometimes you will want different switch cases to use the same code.

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

**Example:**

*switch (new Date().getDay()) {  
  case 4:  
  case 5:  
    text =*"Soon it is Weekend"*;  
    break;  
  case 0:  
  case 6:  
    text =*"It is Weekend"*;  
    break;  
  default:  
    text =*"Looking forward to the Weekend"*;  
}*

**Switching Details**

Switch cases use strict comparison (===).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

In this example there will be no match for x:

**Example:**

*let x =*"0"*;  
switch (x) {  
  case 0:  
    text =*"Off"*;  
    break;  
  case 1:  
    text =*"On"*;  
    break;  
  default:  
    text =*"No value found"*;  
}*

***For\* loop*** - **From Mosh -** This loop will execute…

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}

…so long as this condition is true:

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}

**From W3Schools -** [**W3Schools**](https://www.w3schools.com/js/js_loop_for.asp) *- Edited intro def* - For loops through a block of code a number of times.  
  
The for loop has the following syntax:

for (*statement 1*;*statement 2*;*statement 3*) {  
  // *code block to be executed*  
}

**Statement 1** is executed (one time) before the execution of the code block.

Statement 1 sets a variable before the loop starts (let i = 0). See example below.

**Statement 2** defines the condition for executing the code block.

Statement 2 defines the condition for the loop to run (i must be less than 5). See example Below

**Statement 3** is executed (every time) after the code block has been executed.

Statement 3 increases a value (i++) each time the code block in the loop has been executed. See Example Below.

**Example:** *for (let i = 0; i < 5; i++) {  
  text +=*"The number is "*+ i +*"<br>"*;  
}*

Loops can execute a block of code a number of times.

**JavaScript Loops**

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

Instead of writing:

*text += cars[0] + "<br>";  
text += cars[1] + "<br>";  
text += cars[2] + "<br>";  
text += cars[3] + "<br>";  
text += cars[4] + "<br>";  
text += cars[5] + "<br>";*

You can write:

*for (let i = 0; i < cars.length; i++) {  
  text += cars[i] + "<br>";*}

Try it:  
  
const cars = ["BMW", "Volvo", "Saab", "Ford"];

let i = 0;

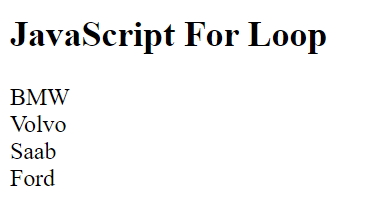
let len = cars.length;

let text = "";

for (; i < len; ) {

text += cars[i] + "<br>";

i++;

}  
  
  
  
  
for - loops through a block of code a number of times

for/in - loops through the properties of an object

for/of - loops through the values of an iterable object

while - loops through a block of code while a specified condition is true

do/while - also loops through a block of code while a specified condition is true

* **While\* loop** - Mosh - 04 - Control Flow - 04 - While - In for loops, the loop variable is part of the loop itself.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

In while loops, we have to declare this variable externally.

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

From [W3Schools](https://www.w3schools.com/js/js_loop_while.asp) - The **while** loop loops through a block of code as long as a specified condition is true.

**Syntax:**

while (condition) {  
*// code block to be executed*  
}

**Example:**

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

while (i < 10) {  
  text += "The number is " + i;  
  i++;  
}

If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

**Comparing For and While**

If you have read the previous chapter, about the for loop, you will discover that a while loop is much the same as a for loop, with statement 1 and statement 3 omitted.

The loop in this example uses a **for** loop to collect the car names from the cars array:

**Example:**

const cars = ["BMW", "Volvo", "Saab", "Ford"];  
let i = 0;  
let text = "";  
  
for (;cars[i];) {  
  text += cars[i];  
  i++;  
}

**Do While\* loop** - Mosh - 04 - Flow Control - 04 - While -   
  
Do while loops are always executed at least once even if this condition is evaluated to false. E.g.,  
  
In do while loops

let i = 0;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

} while (i <= 5);

the *condition* is evaluated at the end. Thus, the middle statements are always executed at least once, even if the following *while* condition is false.



In Contrast, let’s examine our *while loop*. If we were to let I equal nine:

let i = 9;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

The console will display nothing. Because the first time we try to execute the while loop, this condition evaluates to false. The following statements are never executed. So, in while loops this condition is evaluated before the following code block, at the beginning of every iteration.  
  
  
In do while loops

let i = 0;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

} while (i <= 5);

this condition is evaluated at the end. Thus, the middle statements are always executed at least once, even if the while condition is false.

From [W3Schools](https://www.w3schools.com/js/js_loop_while.asp) - The **do while** loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

**Syntax:**

do {  
*// code block to be executed*}  
while (condition);

**Example:**

The example below uses a **do while** loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:  
  
do {  
  text += "The number is " + i;  
  i++;  
}  
while (i < 10);  
  
Do not forget to increase the variable used in the condition, otherwise the loop will never end!

* Infinite Loop - 04 - Control Flow - 06 - Infinite Loops - An infinite loop executes infinitely. If you create one of these loops, you will crash your browser or your computer.

let i = 0;

while(i < 5) {

    console.log(i);

}

while (true) {

}

do {

} while (true);

let x = 0;

do {

} while (x < 5);

for (let i = 0; i > 0: i++)

for (let i = 0; i < 10; )

* **in\* operator** - From [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) - The **in operator** returns true if the specified property is in the specified object or its prototype chain.

**Try it:**

**JavaScript Demo: Expressions - in operator**

const car = { make: 'Honda', model: 'Accord', year: 1998 };

console.log('make' in car);

// expected output: true

delete car.make;

if ('make' in car === false) {

car.make = 'Suzuki';

}

console.log(car.make);

// expected output: "Suzuki"

**Output:**

> true

> "Suzuki"

**Syntax:**

prop **in** object

**Parameters**

**prop**

A string or symbol representing a property name or array index (non-symbols will be coerced to strings).

**object**

Object to check if it (or its prototype chain) contains the property with specified name (prop).

**Examples:**

**Basic usage**

The following examples show some uses of the **in** operator.

// Arrays

let trees = ['redwood', 'bay', 'cedar', 'oak', 'maple']

0 in trees // returns true

3 in trees // returns true

6 in trees // returns false

'bay' in trees // returns false (you must specify the index number, not the value at that index)

'length' in trees // returns true (length is an Array property)

Symbol.iterator in trees // returns true (arrays are iterable, works only in ES2015+)

// Predefined objects

'PI' in Math // returns true

// Custom objects

let mycar = {make: 'Honda', model: 'Accord', year: 1998}

'make' in mycar // returns true

'model' in mycar // returns true

You must specify an object on the right side of the **in** operator. For example, you can specify a string created with the **String** constructor, but you cannot specify a string literal.

let color1 = new String('green')

'length' in color1 // returns true

let color2 = 'coral'

// generates an error (color2 is not a String object)

'length' in color2

**Using *in* with deleted or undefined properties**

If you delete a property with the [delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) operator, the **in** operator returns **false** for that property.

let mycar = {make: 'Honda', model: 'Accord', year: 1998}

delete mycar.make

'make' in mycar // returns false

let trees = new Array('redwood', 'bay', 'cedar', 'oak', 'maple')

delete trees[3]

3 in trees // returns false

If you set a property to [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) but do not delete it, the **in** operator returns true for that property.

let mycar = {make: 'Honda', model: 'Accord', year: 1998}

mycar.make = undefined

'make' in mycar // returns true

let trees = new Array('redwood', 'bay', 'cedar', 'oak', 'maple')

trees[3] = undefined

3 in trees // returns true

The **in** operator will return **false** for empty array slots. Even if accessing it directly returns **undefined**.

let empties = new Array(3)

empties[2] // returns undefined

2 in empties // returns false

To avoid this, make sure a new array is always filled with non-empty values or not write to indexes past the end of array.

let empties = new Array(3).fill(undefined)

2 in empties // returns true

**Inherited Properties**

The **in** operator returns **true** for properties in the prototype chain. (If you want to check for only non-inherited properties, use [Object.prototype.hasOwnProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) instead.)

'toString' in {} // returns true

**Private fields and methods**

You can also use the **in** operator to check whether a particular [private class field or method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields) has been defined in a class. The operator returns **true** if the method is defined, and **false** otherwise.

**Note:** Code will throw if you attempt to access a private class field/method that has not been defined. Using the in operator to check for potentially missing private fields is more compact than using try/catch.

The code fragment below demonstrates a static function that checks whether a specified class has particular private methods and fields.

class ClassWithPrivateFeatures {

#a;

#b = null;

#c() {}

get #d() {}

static f(o) {

return #a in o && #b in o && #c in o && #d in o;

}

}

ClassWithPrivateFeatures.f(new ClassWithPrivateFeatures()) // returns true

ClassWithPrivateFeatures.f({}) // returns false

* **break\*** - From Mosh - With the break keyword we jump out of a loop. [[[We do not finish the code]]]

let i = 0;

while (i <= 10) {

**if (i === 5) break**

    console.log(i);

    i++;

}



**Continue\*** - From Mosh - 04 - Control Flow - 09 - Break and Continue - with the continue keyword we jump to the next iteration. Mosh considers continue legacy code. “It is an ugly way of writing code”.  
  
[[[When i is an even number, it iterates by 1, and then continues, i.e., jumps to the top of the code.]]]

let i = 0;

while (i <= 10) {

   if (i % 2 === 0) {

    i++;

    continue;

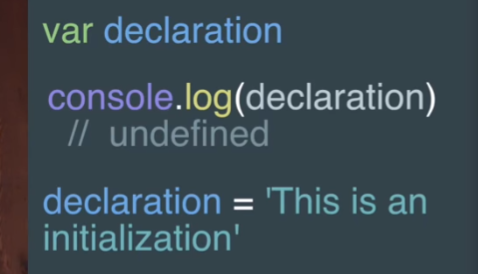
   }

    console.log(i);

    i++;

}



* **Initialize\*** - [uidotdev - youtube](https://www.youtube.com/watch?v=6vBYfLCE9-Q) - “Initialization is when you first assign a value to a variable.”  
    
  “Here, we create a new identifier called declaration. In JS, variables are initialized with the value of undefined when they are created. So, if we log the declaration variable, we get undefined. Initialization is when you first assign a value to a variable. Here, we’re initializing the declaration variable, by assigning it to a string.”  
  

From [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) -

The **var statement** declares a function-scoped or globally-scoped variable, optionally initializing it to a value.  
  
*var x = 1;*

*if (x === 1) {*

*var x = 2;*

*console.log(x);*

*// expected output: 2*

*}*

*console.log(x);*

*// expected output: 2*

From [Teamtreehouse.com](https://teamtreehouse.com/community/variable-initialization-in-javascript#:~:text=Var%20variables%20are%20created%20when,value%20of%20a%20declared%20variable) - I'm extremely confused with the initialization stage when it comes to variable statements:

<http://www.ecma-international.org/ecma-262/6.0/#sec-variable-statement>

*Var variables are created when their containing Lexical Environment is instantiated and are****initialized to undefined****when created.*

Initialization is the means of assigning an initial value to a variable. So undefined is the initial value of a declared variable.

Seeing this though various times;

*var thing = "code";*

The variable *thing* is initialized with *"code".*

How is this possible? Initializing a variable that is **already** initialized does not make any sense to me at all. In other words, **how can a variable be initialized when it already has an initial value**?

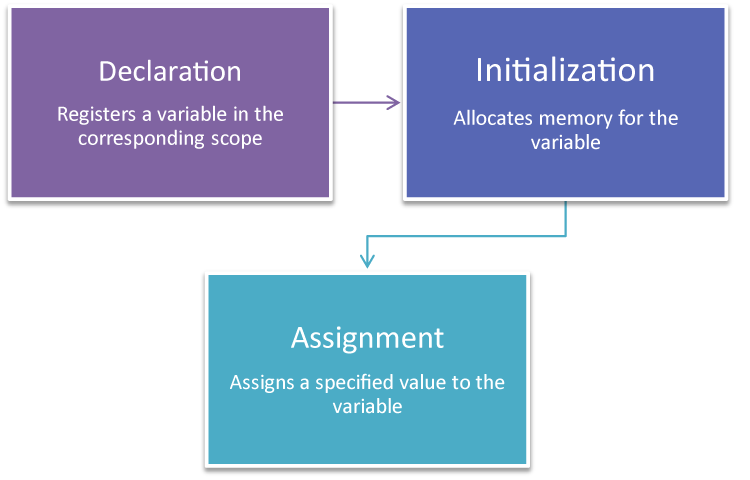
I think what he meant to say was, "Variables are *implicitly* initialized to a value of *undefined* *by default* when they are created." So that only happens if they are not initialized explicitly. For example:

var one; *// this variable is implicitly initialized with "undefined"*

var two = 2; *// this variable is explicitly initialized with the numeric value 2*

From [Sitepoint.com](https://www.sitepoint.com/how-to-declare-variables-javascript/) - **Difference between Declaration, Initialization and Assignment**

Before we start learning the various declarations, lets look at the lifecycle of a variable.



1. **Declaration**: The variable is registered using a given name within the corresponding scope (explained below – e.g. inside a function).
2. **Initialization**: When you declare a variable it is automatically initialized, which means memory is allocated for the variable by the JavaScript engine.
3. **Assignment**: This is when a specific value is assigned to the variable.

let

Syntax:

let x; // Declaration and initialization

x = "Hello World"; // Assignment

// Or all in one

let y = "Hello World";

**let** is the descendant of var in modern JavaScript. Its scope is not only limited to the enclosing function, but also to its enclosing block statement. A block statement is everything inside { and }, (e.g. an if condition or loop). The benefit of let is it reduces the possibility of errors, as variables are only available within a smaller scope.  
  
From [Medium.com](https://medium.com/geekculture/javascript-declaring-and-initializing-variables-how-data-is-stored-and-accessed-in-javascript-2936f4d69ce0) - **Initialization**

Initializing is the term used to describe the process of assignment of a value to a variable (i.e. storing the value (piece of data) in the location in memory which the variable “points” to).

**JavaScript: Declaring and Initializing Variables, How Data is Stored and Accessed in JavaScript**

Declaring and initializing variables in JavaScript are two different concepts. In order to better understand these concepts as well as what a variable actually is, let’s start with discussing how memory is used to store and access data.

**What is a Variable and How does it Work?**

A variable, also known as a “binding” connects (or binds) a piece of data in memory to a name which can later be used to retrieve the value from memory. In computer science, every location in memory has a memory address.

A few things happen when a variable is declared:

1. A location in memory is set aside to store the future value (piece of data) assigned to the variable name.
2. A memory address which “points” to that location in memory is created. This allows us to access the value stored in memory at a later point.
3. The variable name is associated with the memory address.

For example, let’s say that you create a variable **x** using the **let** keyword and set it’s value equal to the string “Hello”. You then use the **console.log()**function to log the value of x to the console. What is actually happening under the hood?

let x = "Hello";

console.log(x);

To make this a little easier to explain and understand, we can break down the 1st line into 2 lines of code, a variable declaration and a value assignment.  
  
*```js*

*// Lines 1 and 2 are equivalent to stating:*

*// let x = "Hello";*

*let x; // Line 1*

*x = "Hello"; // Line 2*

*console.log(x); // Line 3*

*```*

Line 1: **let x;**

* A variable is declared with a name of “x”.
* A location in memory is reserved for the value of the variable “x”.
* A memory address which points to the location in memory is associated with the variable name “x”.

Line 2: x = **“Hello”;**

* JavaScript looks up the variable named **x** and uses the associated memory address to access the location in memory which was reserved for the variable “x”.
* Basically, the variable “points” to the memory location where the value is (or is going to be) stored.
* The value of “Hello” is stored at the specified memory location.

Line 3: **console.log(x)**

* The **console.log()** function calls the variable **x** and uses the memory address associated with **x** to search for the value stored in memory which happens to be “Hello”.
* The value at the specific memory address is returned by **console.log()**.

As a quick summary, when you try to print out the value of the variable x using **console.log(x)**, what is actually happening is that the variable x contains the memory address where the the value “Hello” is stored. JavaScript uses that memory address to go to the specific location in memory which the memory address points to and retrieves the value, which is “Hello”.

So **variables “point” to values stored in memory.**

**Declaring Variables in JavaScript**

To declare (create) a variable, we need to use the **var**, **let**, or **const** keyword, followed by the name we want to provide to the variable. The **var**, **let**, and **const** keywords tell JavaScript to set aside a portion of memory so that we may store a specific piece of data to it later.

The name provided to the variable can be later used to access the location in memory assigned to the variable and retrieve the data which is stored in it. To assign a value to the variable (initialize the variable with a value), use the assignment operator **=** to set the variable name equal to a piece of data (number, boolean, string, array, object, function, etc.)

```js

// Declare a variable named "x" using the var keyword

var x;

// Declare a variable named "y" using the let keyword

let y;

// Declare a variable named "z" using the const keyword

// Assign a value of 2 to the variable "z" using the assignment operator (=)

// Also called initializing "z" with a value of 2 (see section below on initialization)

const z = 2;

```

**Initialization**

Initializing is the term used to describe the process of assignment of a value to a variable (i.e. storing the value (piece of data) in the location in memory which the variable “points” to).

```js

// Initialize the variable x to a value of 1

var x = 1;

// Initialize the variable y to a value of 2

let y = 2;

// Initialize the variable z to a value of 3

// Note that variables with the const keywords

// have to be initialized as soon as they are

// declared otherwise, you will get an error.

const z = 3;

```

From [W3Schools](https://www.w3schools.com/js/js_best_practices.asp) - Best practices - Initialize Variables -   
It is a good coding practice to initialize variables when you declare them.  
  
 This will:

* + Give cleaner code
  + Provide a single place to initialize variables
  + Avoid undefined values

// Declare and initiate at the beginning  
let firstName = "";  
let lastName = "";  
let price = 0;  
let discount = 0;  
let fullPrice = 0,  
const myArray = [];  
const myObject = {};

Initializing variables provides an idea of the intended use (and intended data type).

* See Object Initialization - [JavaPoint](https://www.javatpoint.com/object-and-class-in-java) - Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

**From -** [**MDN**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer) **-** Objects can be initialized using [new Object()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/Object), [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create), or using the *literal* notation (*initializer* notation). An object initializer is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}).  
  
E.g.,  
  
*const object1 = { a: 'foo', b: 42, c: {} };*

*console.log(object1.a);*

*// expected output: "foo"*

*const a = 'foo';*

*const b = 42;*

*const c = {};*

*const object2 = { a: a, b: b, c: c };*

*console.log(object2.b);*

*// expected output: 42*

*const object3 = { a, b, c };*

*console.log(object3.a);*

*// expected output: "foo"*  
  
[Description](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#description)

An object initializer is an expression that describes the initialization of an [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object). Objects consist of *properties*, which are used to describe an object. The values of object properties can either contain [primitive](https://developer.mozilla.org/en-US/docs/Glossary/Primitive) data types or other objects.

[**Creating objects**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#creating_objects)

An empty object with no properties can be created like this:

*const object = {};*

However, the advantage of the *literal* or *initializer* notation is, that you are able to quickly create objects with properties inside the curly braces. You notate a list of key: value pairs delimited by commas.

The following code creates an object with three properties and the keys are "foo", "age" and "baz". The values of these keys are a string "bar", the number 42, and another object.

*const object = {*

*foo: 'bar',*

*age: 42,*

*baz: { myProp: 12 },*

*};*

[Accessing properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#accessing_properties)

Once you have created an object, you might want to read or change them. Object properties can be accessed by using the dot notation or the bracket notation. (See [property accessors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors) for detailed information.)

*object.foo // "bar"*

*object['age'] // 42*

*object.baz // {myProp: 12}*

*object.baz.myProp //12*

**Callback\* function - Call Back\* Function** - Excerpts from [W3Schools](https://www.w3schools.com/js/js_callback.asp) JS Callbacks - See also JS Callbacks -   
  
A callback is a function passed as an argument to another function. This technique allows a function to call another function. A callback function can run after another function has finished.

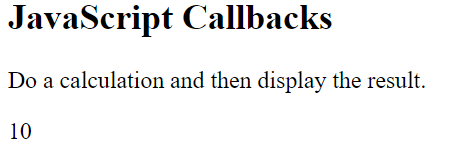
**JavaScript Callbacks**

**A callback is a function passed as an argument to another function.**

Using a callback, you could call the calculator function (myCalculator) with a callback, and let the calculator function run the callback after the calculation is finished: (((myCallback is really myDisplayer.)))

**Example:**

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);



[Test Page Link](https://www.w3schools.com/js/tryit.asp?filename=tryjs_callback4)

In the example above, myDisplayer is the name of a function.

It is passed to myCalculator() as an argument.

When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong: ;

**When to Use a Callback?**

The examples above are not very exciting. They are simplified to teach you the callback syntax.

Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

Asynchronous functions are covered in the next chapter.

From [MDN](https://developer.mozilla.org/en-US/docs/Glossary/Callback_function) - A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Here is a quick example:



The above example is a [synchronous](https://developer.mozilla.org/en-US/docs/Glossary/Synchronous) callback, as it is executed immediately.

Note, however, that callbacks are often used to continue code execution after an [asynchronous](https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous) operation has completed — these are called asynchronous callbacks. A good example is the callback functions executed inside a [.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) block chained onto the end of a promise after that promise fulfills or rejects. This structure is used in many modern web APIs, such as [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch).

From - [Freecodecamp](https://www.freecodecamp.org/news/javascript-callback-functions-what-are-callbacks-in-js-and-how-to-use-them/) - W**hat is a Callback Function?**

In JavaScript, functions are objects. Can we pass objects to functions as parameters? Yes.

So, we can also pass functions as parameters to other functions and call them inside the outer functions. Sounds complicated? Let me show that in an example below:

function print(callback) {

callback();

}

The print( ) function takes another function as a parameter and calls it inside. This is valid in JavaScript and we call it a “callback”. So a function that is passed to another function as a parameter is a callback function. But that’s not all.

**Why do we need Callback Functions?**

JavaScript runs code sequentially in top-down order. However, there are some cases that code runs (or must run) after something else happens and also not sequentially. This is called asynchronous programming.

Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.

In JavaScript, the way to create a callback function is to pass it as a parameter to another function, and then to call it back right after something has happened or some task is completed. Let’s see how…

**How to create a Callback**

To understand what I’ve explained above, let me start with a simple example. We want to log a message to the console but it should be there after 3 seconds.

const message = function() {

console.log("This message is shown after 3 seconds");

}

setTimeout(message, 3000);

There is a built-in method in JavaScript called “setTimeout”, which calls a function or evaluates an expression after a given period of time (in milliseconds). So here, the “message” function is being called after 3 seconds have passed. (1 second = 1000 milliseconds)

In other words, the message function is being called after something happened (after 3 seconds passed for this example), but not before. So the message function is an example of a callback function.

(((Edited remainder… was mention of callback arrow functions)))

Mosh - 06 - Arrays - 04 - Finding Elements (Reference Types) - 5.46  
  
“Back to our example, let’s say we want to see if we have a course with the name a in this array. So, we pass a function here. We call this a predicate or a call back function. Because this function is called back as part of finding an element in this array.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

courses.find(function())

(((Completed Code)))

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a';

});

console.log(course);

**Callbacks\*** - From [W3Schools](https://www.w3schools.com/js/js_callback.asp) - JS Callbacks -   
   
A callback is a function passed as an argument to another function. This technique allows a function to call another function. A callback function can run after another function has finished.

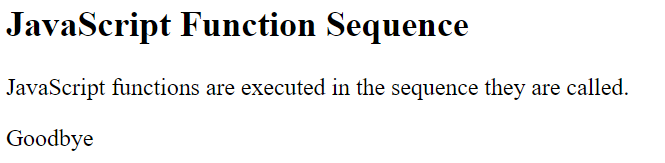
**Function Sequence**

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

This example will end up displaying "Goodbye": (((I presume goodbye is displayed because it is the last, and only the last function is displayed?)))

**Example:**

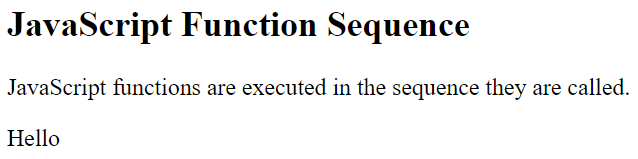
function myFirst() {  
  myDisplayer("Hello");  
}  
  
function mySecond() {  
  myDisplayer("Goodbye");  
}  
  
myFirst();  
mySecond();



This example will end up displaying "Hello":

**Example:**

function myFirst() {  
  myDisplayer("Hello");  
}  
  
function mySecond() {  
  myDisplayer("Goodbye");  
}  
  
mySecond();  
myFirst();



**Sequence Control**

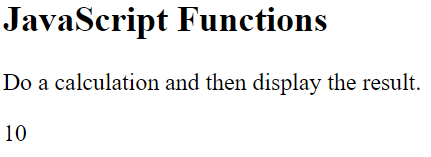
Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation, and then display the result.

You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result: (((So, order is italics, Bold, underline, grey. Green shows the connection. Note that there are Two function calls here… one for calculator, and one for displayer. Last two lines)))

**Example:**

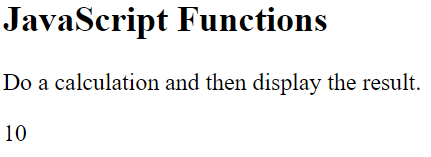
function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
*function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  return sum;  
}*  
  
**let result = myCalculator(5, 5);**  
myDisplayer(result);



Or, you could call a calculator function (myCalculator), and let the calculator function call the display function (myDisplayer): (((Note that the second function call is in the calculator function)))

**Example:**

*function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  myDisplayer(sum);  
}  
  
myCalculator(5, 5);*



The problem with the first example above, is that you have to call two functions to display the result.

**The problem** with the second example, is that you cannot prevent the calculator function from displaying the result.

**Now it is time to bring in a callback. (((Implying the above are Not callbacks?)))**

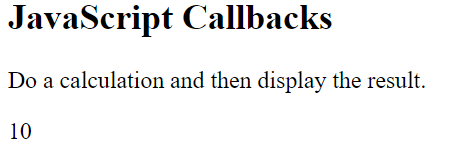
**JavaScript Callbacks**

**A callback is a function passed as an argument to another function.**

Using a callback, you could call the calculator function (myCalculator) with a callback, and let the calculator function run the callback after the calculation is finished: (((myCallback is really myDisplayer.)))

**Example:**

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);



[Test Page Link](https://www.w3schools.com/js/tryit.asp?filename=tryjs_callback4)

In the example above, myDisplayer is the name of a function.

It is passed to myCalculator() as an argument.

When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong: ;

**When to Use a Callback?**

The examples above are not very exciting. They are simplified to teach you the callback syntax.

Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

Asynchronous functions are covered in the next chapter.

**method\*** - From [Tutorialspoint.com](https://www.tutorialspoint.com/What-is-the-difference-between-functions-and-methods-in-JavaScript) - Excerpt only:

Functions and methods are the same in JavaScript, but a method is a function, **which is a property of an object**.

From [programiz.com](https://www.programiz.com/javascript/methods) -   
  
**JavaScript Methods and this Keyword**

In this tutorial, you will learn about JavaScript object methods and this keyword with the help of examples.

In JavaScript, objects can also contain functions. For example,

// object containing method

const person = {

name: 'John',

greet: function() { console.log('hello'); }

};

In the above example, a **person** object has two keys (**name** and **greet**), which have a string value and a function value, respectively.

Hence basically, the JavaScript **method** is an object property that has a function value.

**Accessing Object Methods**

You can access an object method using a dot notation. The syntax is:

objectName.methodKey()

You can access property by calling an **objectName** and a **key**. You can access a method by calling an **objectName** and a **key** for that method along with (). For example,

/ accessing method and property

const person = {

name: 'John',

greet: function() { console.log('hello'); }

};

// accessing property

person.name; // John

// accessing method

person.greet(); // hello

Here, the greet method is accessed as person.greet() instead of person.greet.

If you try to access the method with only person.greet, it will give you a function definition.

person.greet; // ƒ () { console.log('hello'); }

**JavaScript Built-In Methods**

In JavaScript, there are many built-in methods. For example,

let number = '23.32';

let result = parseInt(number);

console.log(result); // 23

Here, the **parseInt()** method of Number object is used to convert numeric string value to an integer value.

To learn more about built-in methods, visit [JavaScript Built-In Methods](https://www.programiz.com/javascript/library).

**Adding a Method to a JavaScript Object**

You can also add a method in an object. For example,

// creating an object

let student = { };

// adding a property

student.name = 'John';

***// adding a method***

***student.greet = function() {***

***console.log('hello');***

}

// accessing a method

student.greet(); // hello

In the above example, an empty **student** object is created. Then, the **name** property is added. Similarly, the **greet** method is also added. In this way, you can add a method as well as property to an object.

(((Truncating definition here)))

From [W3Schools](https://www.w3schools.com/js/js_object_methods.asp) - (((Truncating beginning))) -  
 Methods are functions stored as object properties.

**Accessing Object Methods**

You access an object method with the following syntax:

*objectName.methodName()*

You will typically describe fullName() as a method of the person object, and fullName as a property.

The fullName property will execute (as a function) when it is invoked with ().

This example accesses the fullName() **method** of a person object:

XXXXXXXXXXXXXXXXX Should finish this

From [Tutorialspoint.com](https://www.tutorialspoint.com/What-is-the-difference-between-functions-and-methods-in-JavaScript) -  
  
What is the difference between functions and methods in JavaScript?

Functions and methods are the same in JavaScript, but a method is a function, **which is a property of an object**.

The following is an example of a function in JavaScript:

function functionname(param1, param2){

   // code

}

**Example**

The **method is a function associated with an object**. The following is an example of a method in JavaScript −

<html>

   <head>

      <script>

         var employee = {

            empname: "David",

            department : "Finance",

            id : 002,

            details : function() {

               return this.empname + " with Department " + this.department;

            }

         };

         document.write(employee.details());

      </script>

   </head>

</html>

