

Introduction To JavaScript

JavaScript (or "JS") is a programming language used most often for dynamic client-side scripts on webpages, but it is also often used on the server-side, using a runtime such as Node.js.

JavaScript should not be confused with the Java programming language. Although "Java" and "JavaScript" are trademarks (or registered trademarks) of Oracle in the U.S. and other countries, the two programming languages are significantly different in their syntax, semantics, and use cases.

JavaScript is primarily used in the browser, enabling developers to manipulate webpage content through the DOM, manipulate data with AJAX and IndexedDB, draw graphics with canvas, interact with the device running the browser through various APIs, and more. JavaScript is one of the world's most commonly-used languages, owing to the recent growth and performance improvement of APIs available in browsers.

JavaScript is a powerful programming language that can add interactivity to a website. It was invented by Brendan Eich (co-founder of the Mozilla project, the Mozilla Foundation, and the Mozilla Corporation).

Millions of webpages are built on JavaScript and it's not going anywhere at least for now. On one side HTML and CSS give styling to the web pages but on the other side, it's the magic of JavaScript that makes your web page alive. Today this language is not just limited to your web browser. You can also use it for server-side applications. Isn't it cool to use a single language for both client-side and server-side applications? A single language fulfills both of the purposes and this is the main reason TON of job posting is there for javascript developer in the tech industry.

<https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
https://www.ecma-international.org/wp-content/uploads/ECMA-262_12th_edition_june_2021.pdf
<https://www.ample.co/blog/javascript-history>
<https://stackoverflow.com/questions/912479/what-is-the-difference-between-javascript-and-ecmascript>
<https://ultimatecourses.com/blog/an-introduction-to-javascript>
<https://www.educative.io/blog/javascript-versions-history>

ECMA Script versions

ES1	Jun 1997		ES6	ES2015
ES2	Jun 1998		ES7	ES2016
ES3	Dec 1999		ES8	ES2017
ES5	Dec 2009		ES9	ES2018
ES5.1	Jun 2011		ES10	ES2019
			ES11	ES2020
			ES.Next	

List of JavaScript Engines By Browser

Browser	Name of Javascript Engine
Google Chrome	V8
Edge (Internet Explorer)	Chakra
Mozilla Firefox	Spider Monkey
Safari	Javascript Core Webkit

Installation of Node Js:

- <https://nodejs.org/en/download/>
- *Preferably download zip 64 bit version instead of msi*
- *Extract it and put inside **C:\node-vXx.Xx.Xx-win-x64** (Avoid deep path inside nested folder structure)*
- *Set Path for you node.exe inside environment variable*
- *Check if node is installed using below commands:*
 - *C:\Users\GD>node -v*
v16.14.2
 - *C:\Users\GD>npm -v*
8.5.0

Browser based JavaScript code vs Node Js Based JavaScript Code

Browser Based JavaScript:

- Using browser based JS we can manipulate dom
- We have two object available which are commonly used.
 - **window** object which is root (top most) object in browser
 - **document** object which is root (top most) object representing DOM inside browser to be manipulated using JS. (document object falls below windows object)

Node Based JavaScript:

- We can write pure JS code not dependent on browser.
- We have set of JS libraries provided by node js internally e.g File System, Operating System, Network & HTTP library.
- The libraries inside node js are commonly called as modules.
- **global** object tis the root(top most) object available inside node js.

JavaScript

```
// For HTML Document
document.write("Hello, World!");

// For Browser Console
console.log("Hello, World!");
```

To run the java code easily & dynamically see code changes do following:

- *In Visual Studio Code create index.html file.*
- *Add <script> tag to refer to your file.*
- *Launch live Server plugin*
- *Now as you change code and the moment save file the changes will reflect in browser console.*

Semicolons in JavaScript: To Use or Not to Use? -- **Preferably Use Them**
<https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli>



There are 7 primitive data types:

- ***string***,
- ***number***,
- ***bigint***,
- ***boolean***,
- ***undefined***,
- ***symbol***, and
- ***null***.

All primitives are **immutable**, i.e., they cannot be altered.

It is important not to confuse a primitive itself with a variable assigned a primitive value. The variable may be reassigned a new value, but the existing value can not be changed in the ways that objects, arrays, and functions can be altered.

<https://developer.mozilla.org/en-US/docs/Glossary/Primitive>

Primitive wrapper objects in JavaScript

Except for null and undefined, all primitive values have object equivalents that wrap around the primitive values:

- ***String*** for the string primitive.
- ***Number*** for the number primitive.
- ***BigInt*** for the bigint primitive.
- ***Boolean*** for the boolean primitive.
- ***Symbol*** for the symbol primitive.

<https://developer.mozilla.org/en-US/docs/Glossary/Primitive>

Scope means variable access. What variable do I have access to when a code is running? In javascript by default, you're always in root scope i.e. the window scope. The scope is simply a box with a boundary for variables, functions, and objects. These boundaries put restrictions on variables and determine whether you have access to the variable or not. It limits the visibility or availability of a variable to the other parts of the code. You must have a clear understanding of this concept because it helps you to separates logic in your code and also improves the readability.

A scope can be defined in two ways...

Local Scope allow access to everything within the boundaries (inside the box)

Global Scope is everything outside the boundaries (outside the box). A global scope can not access a variable defined in local scope because it is enclosed from the outer world, except if you return it.

```
> function greeting(){  
  var newYear="Happy New Year!"  
}  
greeting();  
console.log(newYear);
```

✖ ▶ Uncaught ReferenceError: newYear is not defined
at <anonymous>:5:13

```
> var newYear=2022;  
function greeting(){  
  var message="Happy New Year";  
  return message+" "+newYear;  
}  
console.log(greeting());  
console.log(newYear);
```

Happy New Year 2022

2022

The Browser Object Model (BOM)

The **window** object is the Global Object in the Browser. Any Global Variables or Functions can be accessed as properties of the window object.

Access Global Variables

```
var foo = "foobar";  
foo === window.foo; // Returns: true
```



After defining a Global Variable `foo`, we can access its value directly from the `window` object, by using the variable name `foo` as a property name of the Global Object `window.foo`.

Explanation:

The global variable `foo` was stored in the `window` object, like this:

```
foo: "foobar"
```



Access Global Functions

```
function greeting() {  
  console.log("Hi!");  
}  
  
window.greeting(); // It is the same as the normal invoking: greeting();
```



The example above explains how Global Functions are stored as *properties* in the `window` object. We created a Global Function called `greeting`, then invoked it using the `window` object.

Explanation:

The global function `greeting` was stored in the `window` object, like this:

```
greeting: function greeting() {  
  console.log("Hi!");  
}
```



var const & let

var is old way of creatin variables in JS, (avoid using it due to its scoping issues it should be used till ES5)

New way is to use **let** or **const** (from ES6 use let as replacement to var).

let is reassign able variable with block scope

const values are not reassign able.

What is **this**?

In JavaScript, the **this** keyword refers to an **object**.

Which object depends on how **this** is being invoked (used or called).

The **this** keyword refers to different objects depending on how it is used:

In an object method, **this** refers to the **object**.

Alone, **this** refers to the **global object**.

In a function, **this** refers to the **global object**.

In a function, in strict mode, **this** is **undefined**.

In an event, **this** refers to the **element** that received the event.

Methods like **call()**, **apply()**, and **bind()** can refer **this** to **any object**.

Note

this is not a variable. It is a keyword. You cannot change the value of **this**.

```
> function gd(){  
    console.log(this)  
}  
gd();
```

VM156:2

```
▶ Window {0: Window, window: Window, self: Window, document: document, n  
ame: '', location: Location, ...}
```

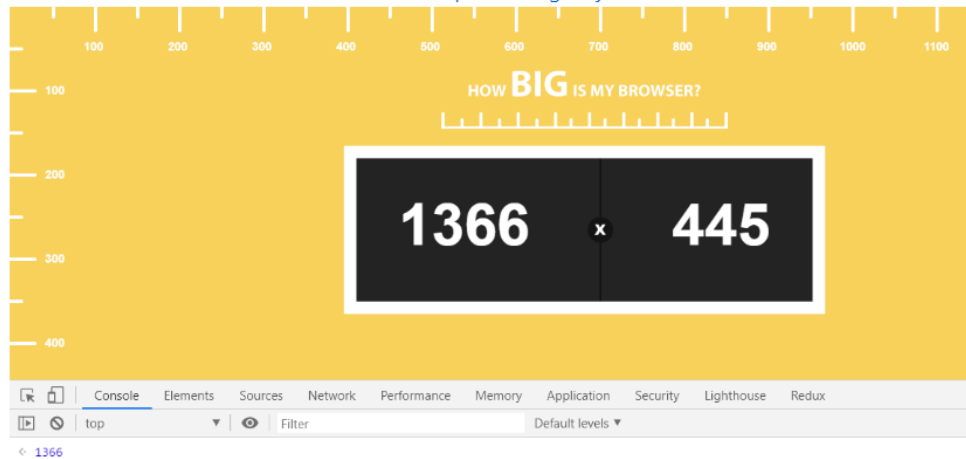
https://www.w3schools.com/js/js_this.asp

Screen resolution

if you mean browser resolution then

window.innerWidth gives you the browser resolution

you can test with <http://howbigismybrowser.com/> try changing your screen resolution by zoom in / out browser and check resolution size with <http://howbigismybrowser.com/>



Window.innerWidth should be same as screen resolution width

<https://www.mydevice.io/#compare-devices>

Inline function in JavaScript

In JavaScript, inline function is a special type of anonymous function which is assigned to a variable, or in other words, an anonymous function with a name.

Syntax:

- Function:

```
function func() {  
    //Your Code Here  
}
```

- Anonymous function:

```
function() {  
    //Your Code Here  
}
```

- Inline function

```
var func = function() {  
    //Your Code Here  
};
```


JavaScript Anonymous Functions with Inline Function Expressions

Anonymous Function is a function that does not have any name associated with it. Normally we use the function keyword before the function name to define a function in JavaScript, however, in anonymous functions in JavaScript, we use only the function keyword without the function name.

An anonymous function is not accessible after its initial creation, it can only be accessed by a variable it is stored in as a function as a value. An anonymous function can also have multiple arguments.

Example 1: In this example, we define an anonymous function that prints a message to the console. The function is then stored in the *greet* variable. We can call the function by invoking *greet()*.

```
> var greet = function (title) {  
    var user='Admin';  
    console.log("Welcome " +title+user+"!");  
};
```

```
greet('Mrs.');
```

```
Welcome Mrs.Admin!
```

JavaScript Demo: Expressions - function expression

```
1 const getRectArea = function(width, height) {  
2   return width * height;  
3 };  
4  
5 console.log(getRectArea(3, 4));  
6 // expected output: 12  
7
```

```
/* Function declaration */
```

```
foo(); // "bar"
```

```
function foo() {  
    console.log('bar');  
}
```

```
/* Function expression */
```

```
baz(); // TypeError: baz is not a function
```

```
var baz = function() {  
    console.log('bar2');  
};
```

JavaScript Arrow Function

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Hoisting (Variable Hoisting vs Function Hoisting)

A lot of developers get unexpected results when they are not clear with the concept of Hoisting in javascript. In javascript, you can call a function before it is defined and you won't get an error 'Uncaught ReferenceError'. The reason behind this is hoisting where the javascript interpreter always moves the variables and function declaration to the top of the current scope (function scope or global scope) before the code execution. Let's understand this with example.

Example: Take a look at the code given below.

```
> sendMessage("+919898989898");  
function sendMessage(mobileNo){  
    console.log(mobileNo);  
}
```

+919898989898

```
> function sendMessage(mobileNo){  
    console.log(mobileNo);  
}  
sendMessage("+919898989898");
```

+919898989898

```
> var a = 5;  
    console.log(5);
```

5

```
> a = 5;  
    console.log(5);  
    var a;
```

5

For var variable JS automatically adds Below line on top
var a= undefined
Then
a=5
happens

IIFE (Immediately Invoked Function Expression)

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts:

- The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
- The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.

IIFE	Arrow function IIFE	async IIFE
<pre>(function () { /* ... */ })();</pre>	<pre>((() => { /* ... */ }))();</pre>	<pre>(async () => { /* ... */ })();</pre>

IIFE (Immediately Invoked Function Expression)

Use Cases

Avoid polluting the global namespace

Because our application could include many functions and global variables from different source files, it's important to limit the number of global variables. If we have some initiation code that we don't need to use again, we could use the IIFE pattern. As we will not reuse the code again, using IIFE in this case is better than using a function declaration or a function expression

```
((() => {  
  // some initiation code  
  let firstVariable;  
  let secondVariable;  
})();  
  
// firstVariable and secondVariable will be discarded after the function is executed.
```

IIFE (Immediately Invoked Function Expression)

Use Cases

Execute an async function

An async IIFE allows you to use await and for-await even in older browsers and JavaScript runtimes that have no top-level await:

```
const getFileStream = async (url) => { /* implementation */ };

(async () => {
  const stream = await getFileStream('https://domain.name/path/file.ext');
  for await (const chunk of stream) {
    console.log({ chunk });
  }
})();
```

IIFE (Immediately Invoked Function Expression)

Use Cases

The module pattern

We would also use IIFE to create private and public variables and methods. For a more sophisticated use of the module pattern and other use of IIFE, you could see the book *Learning JavaScript Design Patterns* by Addy Osmani.

```
const makeWithdraw = (balance) => ((copyBalance) => {  
  let balance = copyBalance; // This variable is private  
  const doBadThings = () => {  
    console.log('I will do bad things with your money');  
  };  
  doBadThings();  
  return {  
    withdraw(amount) {  
      if (balance >= amount) {  
        balance -= amount;  
        return balance;  
      }  
      return 'Insufficient money';  
    },  
  };  
})(balance);  
  
const firstAccount = makeWithdraw(100); // "I will do bad things with your money"  
console.log(firstAccount.balance); // undefined  
console.log(firstAccount.withdraw(20)); // 80  
console.log(firstAccount.withdraw(30)); // 50  
console.log(firstAccount.doBadThings); // undefined; this method is private  
const secondAccount = makeWithdraw(20); // "I will do bad things with your money"  
console.log(secondAccount.withdraw(30)); // "Insufficient money"  
console.log(secondAccount.withdraw(20)); // 0
```

IIFE (Immediately Invoked Function Expression)

Use Cases

For loop with var before ES6

We could see the following use of IIFE in some old code, before the introduction of the statements `let` and `const` in ES6 and the block scope. With the statement `var`, we have only function scopes and the global scope. Suppose we want to create 2 buttons with the texts Button 0 and Button 1 and we click them, we would like them to alert 0 and 1. The following code doesn't work:

```
for (var i = 0; i < 2; i++) {  
  const button = document.createElement('button');  
  button.innerText = 'Button ' + i;  
  button.onclick = function() {  
    console.log(i);  
  };  
  document.body.appendChild(button);  
}  
console.log(i); // 2
```

When clicked, both Button 0 and Button 1 alert 2 because `i` is global, with the last value 2. To fix this problem before ES6, we could use the IIFE pattern: When clicked, Buttons 0 and 1 alert 0 and 1. The variable `i` is globally defined.

```
for (var i = 0; i < 2; i++) {  
  const button = document.createElement('button');  
  button.innerText = 'Button ' + i;  
  button.onclick = (function(copyOfI) {  
    return () => {  
      console.log(copyOfI);  
    };  
  })(i);  
  document.body.appendChild(button);  
}  
console.log(i); // 2
```

Demo: [dynamicControl.html](#)

IIFE (Immediately Invoked Function Expression)

Use Cases

For loop with var before ES6

Using the statement let, we could simply do:

```
for (let i = 0; i < 2; i++) {  
  const button = document.createElement("button");  
  button.innerText = 'Button ' + i;  
  button.onclick = function() {  
    console.log(i);  
  };  
  document.body.appendChild(button);  
}  
console.log(i); // Uncaught ReferenceError: i is not defined.
```



Demo: dynamicControl.html

A closure is simply a function inside another function that has access to the outer function variable even after the outer function execution is over. Now, this definition sounds pretty much straightforward but the real magic is created with the scope. The inner function (closure) can access the variable defined in its scope (variables defined between its curly brackets), in the scope of its parent function, and the global variables. Now here you need to remember that the outer function can not have access to the inner function variable (we have already discussed this in scope concept). Let's take an example and understand it in a better way

In the example, the inner function 'second()' is a Closure. This inner function will have access to the variable 'greet' which is the part of the outer function 'first()' scope. Here the parent scope won't have the access of child scope variable 'name'.

Now the question is why do we need to learn closures? What's the use of it? Closures are used when you want to extend behavior such as pass variables, methods, or arrays from an outer function to an inner function. In the above example, second() extends the behavior of the function first() and also has access to the variable 'greet'. Javascript is not pure object-oriented language but you can achieve object-oriented behavior through closures. In the above example, you can think const 'newFunc' as an Object having property 'greet' and 'second()' a method as in an OOP language.

Here you need to notice that after first() statement is executed, variables inside the first() function will not be destroyed (even if it has 'return' statement) because of closures as the scope is kept alive here and child function can still access the properties of the parent function. So closures can be defined in simple terms as "a function run, the function executed. It's never going to execute again but it's going to remember that there are references to those variables so the child scope always has access to the parent scope."

[Closures - JavaScript | MDN \(mozilla.org\)](https://developer.mozilla.org/en-US/docs/Closures)

```
> const first = () => {  
    const greet = "Hi";  
    const second = () => {  
        const name = "GD!";  
        console.log(greet + " " + name);  
    }  
    return second;  
} // first  
  
const newFunc = first();  
  
newFunc();  
  
Hi GD!
```

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.

In the following example, we just call the print function from the add function.

```
function print( ans ){  
    console.log(ans) ; // 7  
}  
function add(a, b){  
    print(a+b) ;  
}  
  
add(2,5);
```

What if we use the print function as a parameter?

Without using print function from global scope we just pass the print function as an argument.

```
function print( ans ){  
    console.log(ans) ; // 7  
}  
function add(a, b, callback){ // here callback = print  
    callback(a+b) ;  
}  
add(2,5,print); // print function as a parameter
```

Generally, JavaScript allows function as a parameter, so any function that is passed as an argument is called a callback function.

Another example:

```
function greeting(name) {  
  alert('Hello ' + name);  
}  
  
function processUserInput(callback) {  
  var name = prompt('Please enter your name.');
```

```
  callback(name);  
}  
  
processUserInput(greeting);
```

The above example is a synchronous callback, as it is executed immediately.

Note, however, that callbacks are often used to continue code execution after an asynchronous operation has completed — these are called asynchronous callbacks. A good example is the callback functions executed inside a `.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()`.

Asynchronous example:

Here is one example where the usage of callback function is easy to understand.

- A login function which performs user login with the server asynchronously.
- Due to asynchronous call , we need to get the result of login once the date receives from the server.

```
const axios = require('axios');

function login(loginData,callbackSuccess,callbackFailure) {
  axios
    .post("/api/login",loginData)
    .then((response) => {
      callbackSuccess(response);
    })
    .catch((error) => {
      callbackFailure(error);
    });
}

function callbackSuccess(data) {
  console.log("Login response :",data);
}

function callbackFailure(error) {
  console.log("Login failed :",error);
}

let userData = {
  username : "test",
  password : "abcd123"
}

login(userData,callbackSuccess,callbackFailure);
```

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

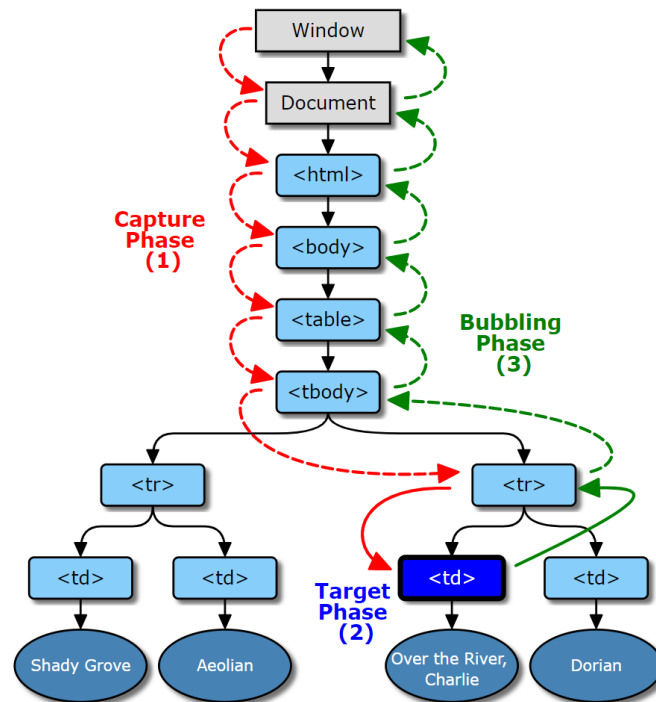
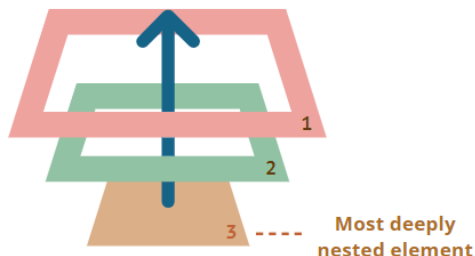
Capturing Phase --> Target Phase --> Bubbling Phase

<https://javascript.info/bubbling-and-capturing>

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>
<form onclick="alert('form')">
  FORM
  <div onclick="alert('div')">
    DIV
    <p onclick="alert('p')">
      P
    </p>
  </div>
</form>
```

A click on the inner `<p>` first runs `onclick`:

1. On that `<p>`.
2. Then on the outer `<div>`.
3. Then on the outer `<form>`.
4. And so on upwards till the `document` object.



Capturing Phase --> Target Phase --> Bubbling Phase

The code sets click handlers on *every* element in the document to see which ones are working.

If you click on `<p>`, then the sequence is:

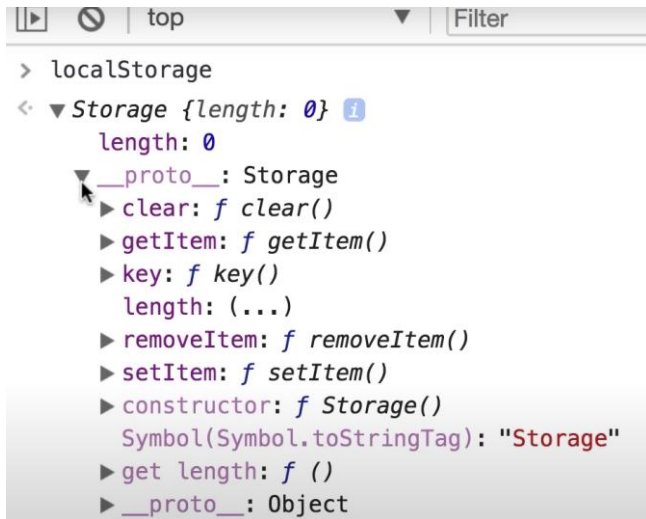
1. `HTML` → `BODY` → `FORM` → `DIV` (capturing phase, the first listener):
2. `P` (target phase, triggers two times, as we've set two listeners: capturing and bubbling)
3. `DIV` → `FORM` → `BODY` → `HTML` (bubbling phase, the second listener).

The standard [DOM Events](#) describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.

Inbuild html object called localStorage

Every browser has local storage object which allows you to store data locally.
the localStorage object has properties and methods like below:



Encapsulation

Reduce complexity + increase reusability

Abstraction

Reduce complexity + isolate impact of changes

Inheritance

Eliminate redundant code

Polymorphism

Refactor ugly switch/case statements

Encapsulation

```
let baseSalary = 30_000;
let overtime = 10;
let rate = 20;

function getWage(baseSalary, overtime, rate) {
  return baseSalary + (overtime * rate);
}

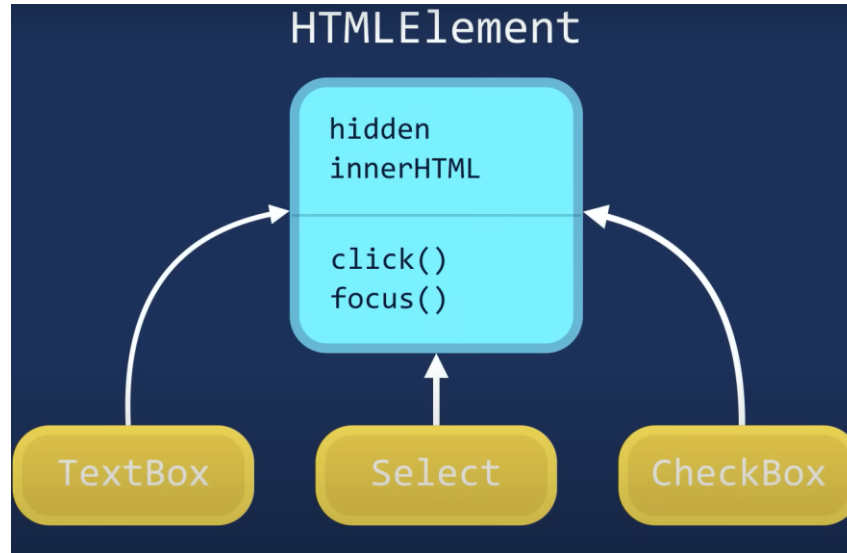
let employee = {
  baseSalary: 30_000,
  overtime: 10,
  rate: 20,
  getWage: function() {
    return this.baseSalary + (this.overtime * this.rate);
  }
};

employee.getWage();
```

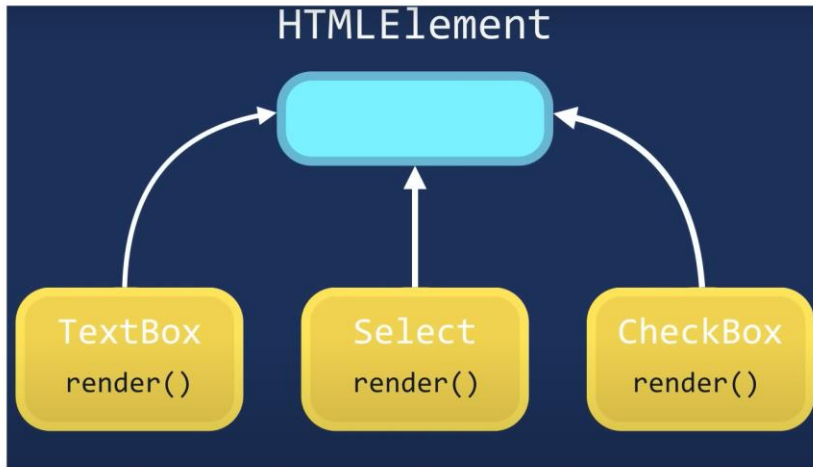
Procedural

OOP
Encapsulation

Inheritance



Polymorphism



```
switch (element.type) {  
  case 'select': renderSelect();  
  case 'text': renderTextBox();  
  case 'checkbox': renderCheckBox()  
  case  
    element.render();  
}
```

Creating Objects using “Object Literal Syntax”

Object in java is essentially collection of key value pair.

We can define object using **object literal syntax**

let/const <object-name>= {<key>:<value>;}

Note: value can be another object literal syntax & any JS function.

Drawback: Duplicate of functions in each object and modifying becomes difficult if multiple functions

Properties

Method

```
const circle = {  
  radius: 1,  
  location: {  
    x: 1,  
    y: 1  
  },  
  draw: function() {  
    console.log('draw')  
  }  
};
```

```
circle.draw();
```

Creating Objects using “Function Factory”

In ES6 if the name of the argument and the property in object are same, we don't need to use **radius=radius** we can simply return **radius**.

*If the input parameter is rad and property is radius we need to mention them like **radius:rad***

```
// Factory Function
function createCircle(radius) {
  return {
    radius,
    draw: function() {
      console.log('draw');
    }
  };
}

const circle = createCircle(1);
circle.draw();
```

Creating Objects using “Constructor Function”

Rules:

1. Naming convention is we need to use method name as pascal notation.
2. Here **this** is the reference to an object in memory that is executing the function if we mention **this** inside the function.
3. *Imagine if we have empty object in memory in order to refer to its property, we use **this** followed with **dot**.*
4. *You need to create object using new keyword, the new keyword basically create the empty object {} inside memory.*
5. *Later we try to add the properties to that object inside function call.*
6. *If you don't write new instead const another = Circle(1) , this will refer to global object. (In case of browser-based JS its window, in Node its global).*
7. Check it using console.log both with and without new keyword.
8. *Additionally, we do not mention return this in code since we have used = new Circle(1) this will return the newly created object automatically unlike normal functions.*

Which one to choose?

Chose any option i.e. Factory Function or Constructor Function there is no hard and fast rule to use specific one, developed from java love the way to write new and this keyword to better understand logic.

```
// Constructor Function
function Circle(radius) {
  this.radius = radius;
  this.draw = function() {
    console.log('draw');
  }
}

const another = new Circle(1); {}
```

```
function Circle(radius) {
  console.log('this', this);
}
```

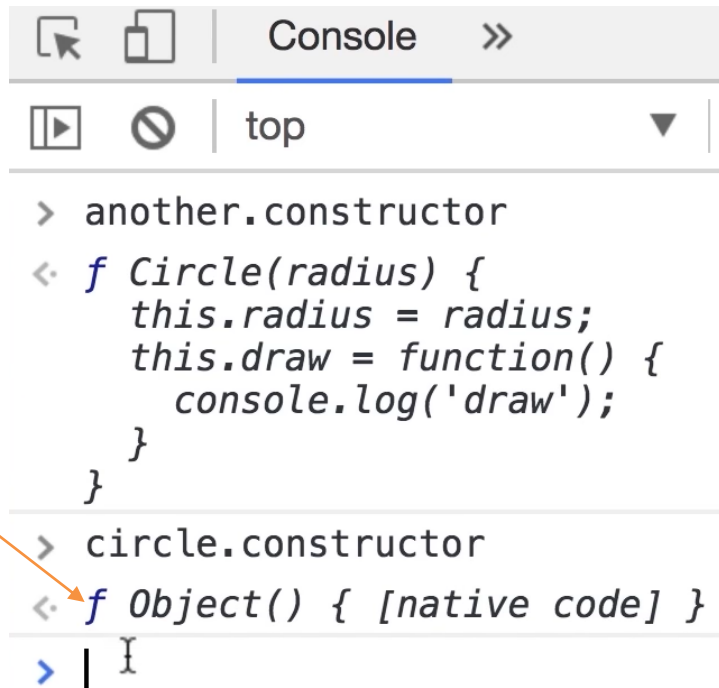


Using “Constructor Property”

- Every object in JS has property called **constructor**
- This property reference the function that is used to create/construct that object.
- Try printing previous two objects property on console.
- There is built in constructor function in JS.
- We can create object explicitly using the new Object() syntax as below:

```
let x = {};  
  
// let x = new Object();  
  
new String(); // '', '', ''  
new Boolean(); // true, false  
new Number(); // 1, 2, 3, ...
```

- In JS we have few other built in constructor like String, Boolean, Number etc.
- But usually we don't create string, Boolean, number using them instead directly assigning literal values to them.



```
> another.constructor  
< f Circle(radius) {  
  this.radius = radius;  
  this.draw = function() {  
    console.log('draw');  
  }  
}  
  
> circle.constructor  
< f Object() { [native code] }  
> |
```

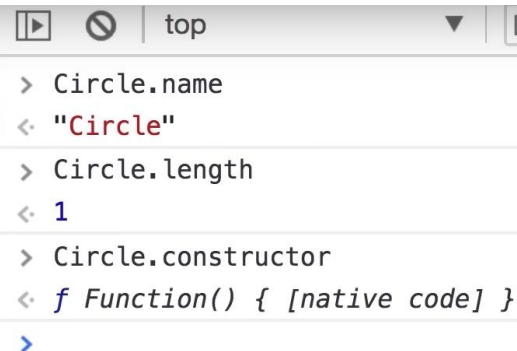
Function() & Objects

- In JS functions are objects., that is they have properties as well as method to them.
- When we declare function the object of that function is created internally by JS by make use of Function() constructor.
- How the newly created function object will be internally represented is shown using Circle1 object.
- The first parameter is argument and second is body with backtick operator for multiline method body.
- You can check on console it logs the object details.
- Invoking the function can be done using call method available on function object like below:
- The first argument to call is empty object i.e this object which we need to pass as empty object.
- The second argument onwards any no of arguments will come as per no of arguments declared in function definition.
- Hence when we say **new Circle(1)** internally it is represented as **Circle.call({},1)**

```
Circle.call({}, 1)
```

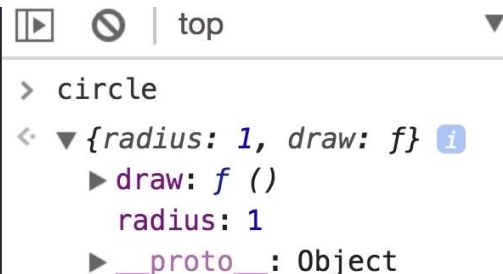
```
const another = new Circle(1);
```

```
function Circle(radius) {  
  this.radius = radius;  
  this.draw = function() {  
    console.log('draw');  
  }  
}
```



```
top  
> Circle.name  
< "Circle"  
> Circle.length  
< 1  
> Circle.constructor  
< f Function() { [native code] }
```

```
const Circle1 = new Function('radius', `  
this.radius = radius;  
this.draw = function() {  
  console.log('draw');  
}  
`);  
  
const circle = new Circle1(1);
```



```
top  
> circle  
< {radius: 1, draw: f} ⓘ  
  ▶ draw: f ()  
    radius: 1  
  ▶ __proto__: Object
```

Note: The backtick character is different that single quote character

Function() & Objects Cont.. Call() & apply() methods.

- Invoking the function can be done using call method available on function object like this:
- The call() method calls the function with a given **this** value and arguments provided individually.
- The first argument to call in our example is empty object i.e this object which we need to pass as empty object.
- The second argument onwards any no of arguments will come as per no of arguments declared in function definition.
- Hence when we say **new Circle(1)**, internally it is represented as **Circle.call({},1)**
- We also have another method called apply similar to call which takes second parameter as array instead of multiple comma separate parameters.

```
Circle.call({}, 1)

const another = new Circle(1);
```

```
Circle.call({}, 1);
Circle.apply({}, [1, 2, 3]);
```

```
function Product(name, price) {
  this.name = name;
  this.price = price;
}
```

```
let pizza = {};
console.log(pizza);
Product.call(pizza, "Pepperoni Pizza", 250);
console.log(pizza.name);
console.log(pizza.price);
```

```
function Food(name, price) {
  Product.call(this, name, price); // adds two more properties into Food object
  this.category = "food";
}

let junk = new Food("cheese", 5);
console.log(junk);
console.log(junk.name);
```

JavaScript Data Types (Primitive or Value Types & Reference Types)

```
let x = 10;  
let y = x;
```

```
x = 20;
```

```
top  
> x  
< 20  
> y  
< 10
```

Value Types

Number
String
Boolean
Symbol
undefined
null

Reference Types

Object
Function
Array

Primitives are copied by their **value**

Objects are copied by their **reference**

*X & Y store the address location
of object in memory*

```
let x = { value: 10 };  
let y = x;
```

```
x.value = 20;
```

```
top  
> x  
< {value: 20}  
> y  
< {value: 20}
```



```
let number = 10;  
  
function increase(number) {  
  number++;  
}
```

```
increase(number);  
console.log(number);
```

▶ ⓧ | top

10

>

```
let obj = { value: 10 };  
  
function increase(obj) {  
  obj.value++;  
}
```

```
increase(obj);  
console.log(obj);
```

▶ ⓧ | top

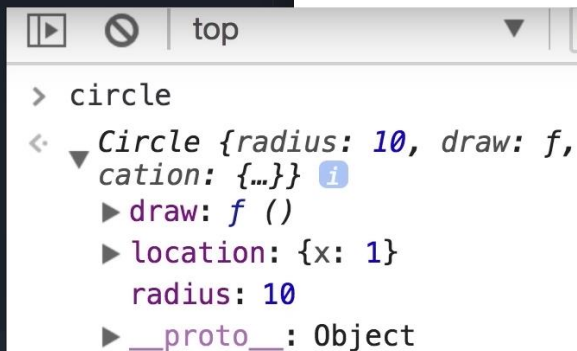
{value: 11}

>

Adding Properties to Object is Dynamically Possible

- In Java or OOP language if you want to add the property you need to update the class.
- In JS we can add properties adhocly simply by assigning them as below, the location is added dynamically.

```
function Circle(radius) {  
  this.radius = radius;  
  this.draw = function() {  
    console.log('draw');  
  }  
}  
  
const circle = new Circle(10);  
  
circle.location = { x: 1 };
```



```
> circle  
◀ Circle {radius: 10, draw: f,  
  cation: {...}} ⓘ  
  ▶ draw: f ()  
  ▶ location: {x: 1}  
    radius: 10  
  ▶ __proto__: Object
```

- We can add properties using bracket notation like `['propertyname']` or `[variablename]` when the properties has special character like dash or space, or when we want to add them dynamically through variables.

```
const propertyName = 'center |location';  
circle.center-location  
circle[propertyName] = { x: 1 };
```

Adding Properties to Object is Dynamically Possible

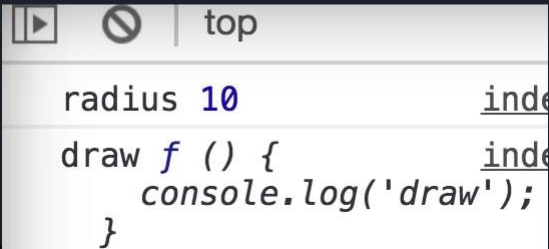
- We can delete properties e.g DB retrieve confidential details before sending object to client/UI using delete operation.

```
delete circle['location'];
```

- Iterate over properties using for .. In loop

```
const circle = new Circle(10);

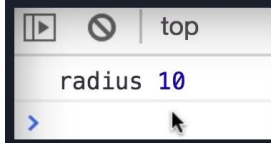
for (let key in circle) {
  console.log(key, circle[key]);
}
```

A screenshot of a code editor's console window. The console shows the output of the code above. The first line of output is "radius 10" followed by a link "index.html". The second line of output is "draw f () {" followed by a link "index.html", and then "console.log('draw');" and a closing brace "}" on the next line. The console window has a "top" button and a "stop" icon.

Adding Properties to Object is Dynamically Possible

- Iterate over only properties not function we use typeof operator-

```
for (let key in circle) {  
  if (typeof circle[key] !== 'function')  
    console.log(key, circle[key]);  
}
```



- Iterate over all properties including function another approach-

```
const keys = Object.keys(circle);  
console.log(keys);  
▶ (2) ["radius", "draw"]
```

- To check if specific property is in the object we use **if with in**-

```
if ('radius' in circle)  
  console.log('Circle has a radius.');
```

Circle has a radius.

Hiding properties from accessing using object name

- You remove them as properties that is no more **this.properties** rather use them **let properies** as local variables.
- Closure feature in JS will allow inner function to use outer function variables

Getter/Setter Old Way

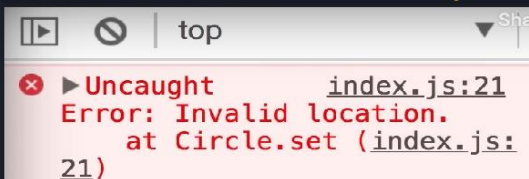
- Read property from outside using custom method no modify allowed

```
function Circle(radius) {  
  this.radius = radius;  
  let defaultLocation = { x: 0, y: 0 };  
  this.getDefaultLocation = function() {  
    return defaultLocation;  
  };  
}  
  
circle.getDefaultLocation();
```

Getter/Setter New Way

```
function Circle(radius) {  
  this.radius = radius;  
  let defaultLocation = { x: 0, y: 0 };  
  this.getDefaultLocation = function() {  
    return defaultLocation;  
  };  
  Object.defineProperty(this, 'defaultLocation', {  
    get: function() {  
      return defaultLocation;  
    },  
    set: function(value) {  
      if (!value.x || !value.y)  
        throw new Error('Invalid location.');      defaultLocation = value;  
    }  
  });  
}
```

```
circle.defaultLocation = 1;
```



Uncaught Error: Invalid location.
at Circle.set (index.js:21)

Objects Basics in JavaScript

An object is a collection of related data and/or functionality. These usually consist of several variables and functions (which are called properties and methods when they are inside objects).

Basic Code:

```
const person = {};
```

Output:

```
[object Object]  
Object { }  
{ }
```

Syntax:

```
const objectName = {  
  member1Name: member1Value,  
  member2Name: member2Value,  
  member3Name: member3Value  
};
```

Example:

```
> const person = {  
  name: ['Bob', 'Smith'],  
  age: 32,  
  bio: function() {  
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years old.`);  
  },  
  introduceSelf: function() {  
    console.log(`Hi! I'm ${this.name[0]}.`);  
  }  
};  
console.log(person);
```

```
▼ {name: Array(2), age: 32, bio: f, introduceSelf: f} ⓘ  
  age: 32  
  ▶ bio: f ()  
  ▶ introduceSelf: f ()  
  ▶ name: (2) ['Bob', 'Smith']  
  ▶ [[Prototype]]: Object
```

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>

Objects Basics in JavaScript

Create object using generic method

Example:

```
function createPerson(name) {  
  const obj = {};  
  obj.name = name;  
  obj.introduceSelf = function() {  
    console.log(`Hi! I'm ${this.name}.`);  
  }  
  return obj;  
}
```

Objects:

```
const salva = createPerson('Salva');  
salva.name;  
salva.introduceSelf();  
  
const frankie = createPerson('Frankie');  
frankie.name;  
frankie.introduceSelf();
```

This works fine but is a bit long-winded: we have to create an empty object, initialize it, and return it. A better way is to use a **constructor**

Objects Basics in JavaScript

Create object using Constructor method.

A constructor is just a function called using the new keyword. When you call a constructor, it will:

- *create a new object*
- *bind this to the new object, so you can refer to this in your constructor code*
- *run the code in the constructor*
- *return the new object.*

Constructors, by convention, start with a capital letter and are named for the type of object they create. So we could rewrite our example like this:

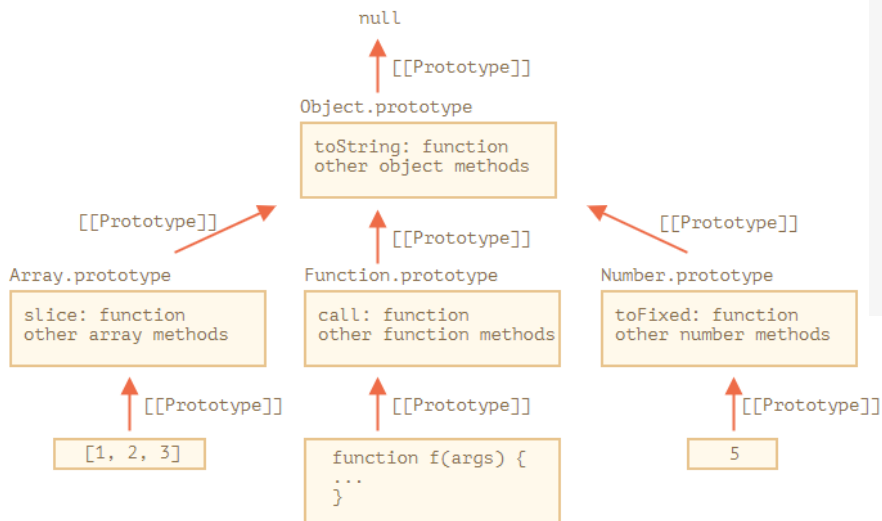
Example:

```
function Person(name) {  
  this.name = name;  
  this.introduceSelf = function() {  
    console.log(`Hi! I'm ${this.name}.`);  
  }  
}
```

Objects:

```
const salva = new Person('Salva');  
salva.name;  
salva.introduceSelf();  
  
const frankie = new Person('Frankie');  
frankie.name;  
frankie.introduceSelf();
```

JavaScript inherits from Object



```
function Rabbit() {  
  this.color = 'White'  
}
```

```
let obj = new Rabbit();
```

```
alert(Rabbit.__proto__ === Function.prototype) //true  
alert(obj.__proto__ === Rabbit.prototype) //true  
alert(obj.__proto__.__proto__ === Object.prototype) //true
```

```
function Rabbit() {  
  this.color = 'White'  
}
```

```
let obj = new Rabbit();
```

```
console.log(Rabbit.__proto__ === Function.prototype) //true  
console.log(obj.__proto__ === Rabbit.prototype) //true  
console.log(obj.__proto__.__proto__ === Object.prototype) //true
```

```
console.log(Function.prototype.__proto__ === Object.prototype) //true  
console.log(Object.getPrototypeOf(Function.prototype) === Object.getPrototypeOf(Object.prototype)) //false
```

JavaScript Prototype and Prototype Chain

Inheritance in JavaScript is implemented through the prototype chain.

Every normally created object, array, and function has a prototype chain of `__proto__` properties ending with `Object.prototype` at the top.

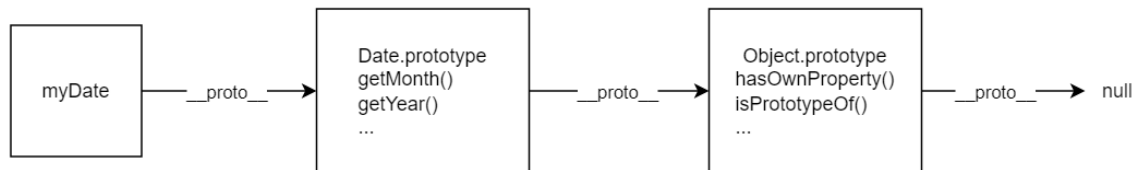
<https://chamikakasun.medium.com/javascript-prototype-and-prototype-chain-explained-fdc2ec17dd04>

<https://chamikakasun.medium.com/javascript-factory-functions-vs-constructor-functions-585919818afe>

```
const myDate = new Date();
let object = myDate;

do {
  object = Object.getPrototypeOf(object);
  console.log(object);
} while (object);

// Date.prototype
// Object { }
// null
```



Viewer does not support full SVG 1.1

Inheritance and Prototype Javascript

From the Top Down

There is an object in Javascript where all other objects inherits default properties and methods from.

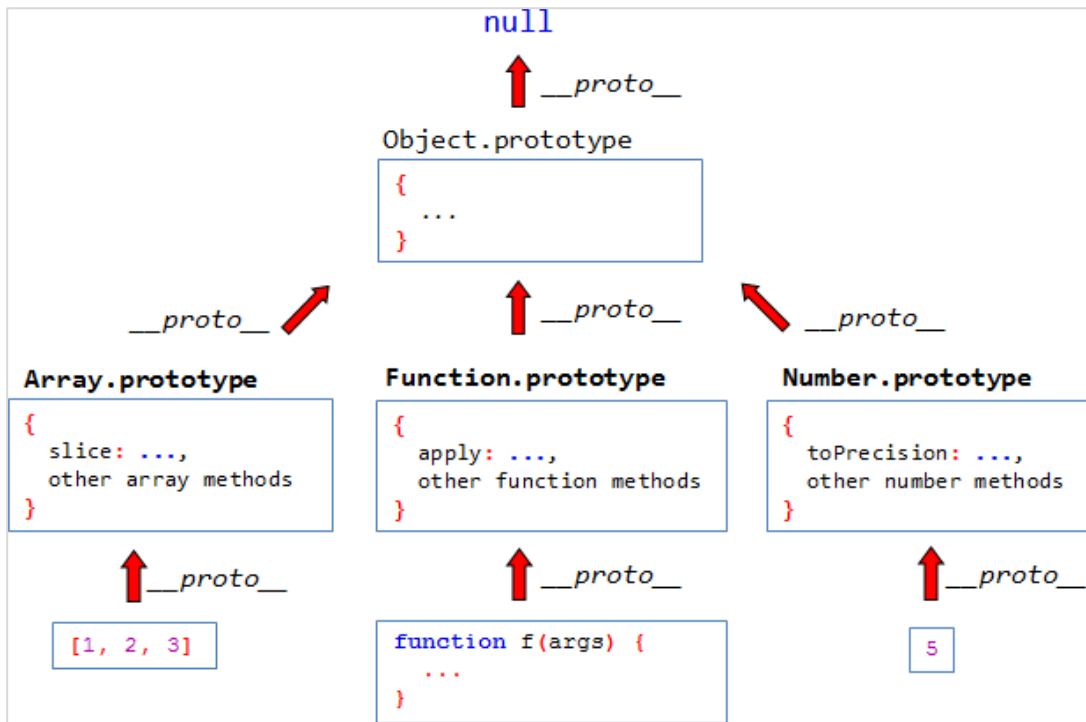
will call this object the “Object prototype” or Object with a capital “O”, all objects are descendent from Object.

All objects have defaults given by the Object; it is a hierarchy. All objects have the `__proto__` property that gives us access to it's prototype.

What sits at the top of all object is the Object prototype; but above that is null.

A prototype can be thought of as a grouping of methods that specific object has access to. Arrays have their own prototype, so do functions and numbers.

Most everything is an object in Javascript. Arrays have methods and properties as do functions and primitives like numbers, strings, and boolean values. It's important to note that although these data types have their own separate prototype; they all inherit the default prototype as well from the Object.

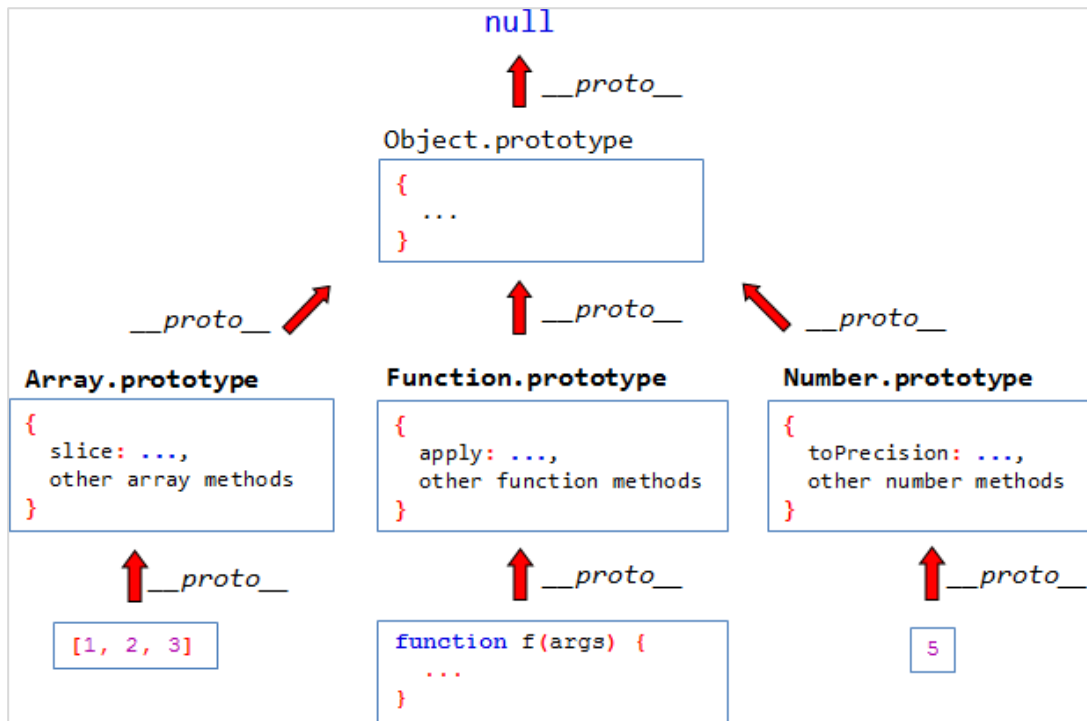


Inheritance and Prototype Javascript

Inside all objects we have the property `__proto__` that accesses the internal prototype designated as `[[Prototype]]`, it is the default inherited property of all objects.

This default gets overwritten when the object was constructed using "new", the `[[Prototype]]` gets replaced with the prototype property of the function used as constructor.

<https://javascript.info/prototype-inheritance>



OOP in JavaScript

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript

Demo -- > folder oop-in-javascript

Private in JavaScript

Underscore (`_`) is just a plain valid character for variable/function name, it does not bring any additional feature. However, it is a good convention to use underscore to mark variable/function as private

In 2019, It appears a [proposal](#) to extend class syntax to allow for `#` prefixed variable to be private was accepted. **Chrome 74** ships with this support.

`_` prefixed variable names are considered private by convention but are still public.

JavaScript actually does support encapsulation, through a method that involves hiding members in closures.

- *JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.*

JSON is built on two structures:

- 1. A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.*
 - 2. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.*
- *These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.*

Java

- | | |
|--------------------------------------|---|
| • <u>jackson</u> | • <u>fastjson</u> |
| • <u>google-gson</u> | • <u>mjson</u> |
| • <u>JSON-java</u> | • <u>jjson</u> |
| • <u>JSONUtil</u> | • <u>json-simple</u> |
| • <u>jsonp</u> | • <u>json-io</u> |
| • <u>Json-lib</u> | • <u>FOSS Nova JSON</u> |
| • <u>Stringtree</u> | • <u>Corn CONVERTER</u> |
| • <u>SOJO</u> | • <u>Apache johnzon</u> |
| • <u>json-taglib</u> | • <u>Genson</u> |
| • <u>Flexjson</u> | • <u>cookjson</u> |
| • <u>Argo</u> | • <u>progbase</u> |
| • <u>jsonij</u> | • <u>MOXy</u> |

JavaScript

- [JSON](#)
- [json2.js](#)
- [clarinet](#)
- [Oboe.js](#)
- [progbase](#)

Uncaught TypeError: Cannot set property 'innerHTML' of null

Why do I get an error or Uncaught TypeError: Cannot set property 'innerHTML' of null?

The browser always loads the entire HTML DOM from top to bottom. Any JavaScript code written inside the script tags (present in head section of your HTML file) gets executed by the browser rendering engine even before your whole DOM (various HTML element tags present within body tag) is loaded. The scripts present in head tag are trying to access an element having id hello even before it has actually been rendered in the DOM. So obviously, JavaScript failed to see the element and hence you end up seeing the null reference error. Resolving in 2 ways:

1) Allow HTML load before the js code.

```
<script type = "text/javascript">
  window.onload = function what(){
    document.getElementById('hello').innerHTML = 'hi';
  }
</script>

//or set time out like this:
<script type = "text/javascript">
  setTimeout(function(){
    what();
    function what(){
      document.getElementById('hello').innerHTML = 'hi';
    }
  }, 50);
  //NOTE: 50 is millisecond.
</script>
```

2) Move js code under HTML code

```
<div id="hello"></div>
<script type = "text/javascript">
  what();
  function what(){
    document.getElementById('hello').innerHTML = 'hi';
  }
</script>
```

What structure should your website have?

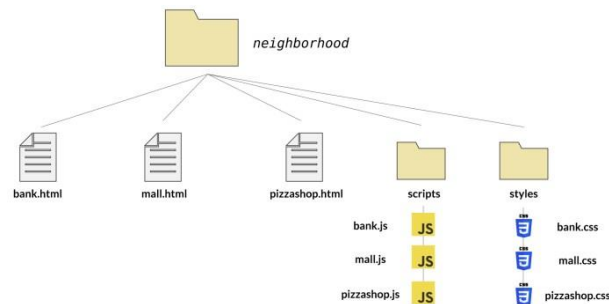
The most common things we'll have on any website project we create are an index HTML file and folders to contain images, style files, and script files. Let's create these now:

index.html: This file will generally contain your homepage content, that is, the text and images that people see when they first go to your site. Using your text editor, create a new file called index.html and save it just inside your test-site folder.

images folder: This folder will contain all the images that you use on your site. Create a folder called images, inside your test-site folder.

styles folder: This folder will contain the CSS code used to style your content (for example, setting text and background colors). Create a folder called styles, inside your test-site folder.

scripts folder: This folder will contain all the JavaScript code used to add interactive functionality to your site (e.g. buttons that load data when clicked). Create a folder called scripts, inside your test-site folder.



https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Dealing_with_files

JavaScript Common Guidelines (Good Practices)

<https://www.educative.io/blog/javascript-tips-simplify-code>

https://www.w3schools.com/js/js_best_practices.asp

https://www.w3schools.com/js/js_conventions.asp

Synchronous and Asynchronous code in JavaScript

Synchronous code runs in sequence. This means that each operation must wait for the previous one to complete before executing.

Asynchronous code runs in parallel. This means that an operation can occur while another one is still being processed.

Asynchronous code execution is often preferable in situations where execution can be blocked indefinitely. Some examples of this are network requests, long-running calculations, file system operations etc. Using asynchronous code in the browser ensures the page remains responsive and the user experience is mostly unaffected.

This below code will print “Three” before “Two”. Second Line will run after 100 milliseconds.

setTimeout is asynchronous, so the last line will not wait for setTimeout.

JavaScript

```
console.log('One');  
console.log('Two');  
console.log('Three');  
// LOGS: 'One', 'Two', 'Three'
```

JavaScript

```
console.log('One');  
setTimeout(() => console.log('Two'), 100);  
console.log('Three');  
// LOGS: 'One', 'Three', 'Two'
```

What is setTimeout():

setTimeout() executes a function once the timer expires.

Suppose, you want a to run code after 2 seconds, you can use setTimeout()

Syntax:

setTimeout(function[, delay]);

function: can be an anonymous function, or, you can run another function.

delay: is the number of milliseconds.

Promises, async Function & await keyword

"async and await make promises easier to write"

async makes a function return a Promise

await makes a function wait for a Promise

The keyword async before a function makes the function return a promise:

```
// async function myFunction() {  
  // return "Hello";  
}
```

Is the same as:

```
// function myFunction() {  
  // return Promise.resolve("Hello");  
}
```

Await Syntax

The keyword await before a function makes the function wait for a promise:

```
// let value = await promise;
```

The await keyword can only be used inside an async function.

An async function is a function declared with the async keyword, and the await keyword is permitted within it.

The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Async functions and the await keyword act like syntactic sugar on top of promises, making asynchronous code both easier to read and write.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

<https://medium.com/swlh/set-a-time-limit-on-async-actions-in-javascript-567d7ca018c2>



Let us assume, your web page has a DIV element and you want to auto-refresh its contents every few seconds.

You can easily do this using jQuery or any other library.

Let see how to auto-refresh a DIV every few seconds using plain JavaScript and Ajax.

The advantage in using JavaScript is that it will make coding simple and you do not have to inject any library to your web page.

The DIV will get data (content) extracted from an external JSON file.

And to do this, we will use JavaScript Ajax with GET method.

To auto-refresh the DIV element every few seconds, we'll call a method (which will refresh the DIV's content) from within the `window.setInterval()` function, at a specified interval (few seconds).

How to Auto Refresh Page Every 10 Seconds using JavaScript setInterval() Method.

The JavaScript setInterval() method calls a function or executes a code repeatedly at specified time intervals.

Here in this post, we'll show you a simple example on how to refresh or reload a web page every 10 Seconds using the JavaScript setInterval() method.

Syntax of setInterval() Method

window.setInterval(code, delay)

The method setInterval() takes two parameters. The code to execute and a delay in milliseconds (to execute the code) . The code can also be a function. In the demo example, we are calling a function (a user defined function).

```
<script>
    window.setInterval('refresh()', 10000);    // Call a function every 10000 milliseconds (OR 10
seconds).

    // Refresh or reload page.
    function refresh() {
        window.location.reload();
    }
</script>
```

Create a Countdown using setInterval() Method

You can create a countdown using the setInterval() method, which will show the seconds left before the page will automatically load.

The method will update a element every second. Use this countdown example to notify users for upcoming events or a deal that is going end very soon, etc.

Demo:

```
<div>This page will reload in <span id="cnt" style="color:red;">10</span> Seconds</div>
```

```
<script>
```

```
    var counter = 10;
```

```
    // The countdown method.
```

```
    window.setInterval(function () {
```

```
        counter--;
```

```
        if (counter >= 0) {
```

```
            var span;
```

```
            span = document.getElementById("cnt");
```

```
            span.innerHTML = counter;
```

```
        }
```

```
        if (counter === 0) {
```

```
            clearInterval(counter);
```

```
        }
```

```
    }, 1000);
```

```
    window.setInterval('refresh()', 10000);
```

```
    // Refresh or reload page.
```

```
    function refresh() {
```

```
        window .location.reload();
```

```
    }
```

```
</script>
```

You can Auto Refresh a page using the <meta> tag. This in case you do not want to use JavaScript or jQuery for this purpose. Add the below tag inside the <head> tag.

```
<meta https-equiv="refresh" content="10">
```

The content attribute has a value in seconds.

The period it will wait before refreshing the page automatically.

However, the JavaScript method that we discussed earlier has other benefit. For example, you can explicitly invoke the auto refresh procedure, when every necessary.

<https://danlevy.net/you-may-not-need-axios/>

JavaScript Good Learning Links:

<https://www.javascripttutorial.net/>

<https://www.javascripttutorial.net/javascript-bom/javascript-window/>

<https://www.javascripttutorial.net/es-next/javascript-globalthis/>

<https://www.javascripttutorial.net/es6/difference-between-var-and-let/>

<https://www.javascripttutorial.net/es6/>

<https://www.javascripttutorial.net/javascript-array-length/#>

<https://dorey.github.io/JavaScript-Equality-Table/>

How To Get Screen Resolution

Common need for website developer is to know the screen resolution of the person who is accessing their website- This help to properly align the layout for website.

```
function gd(){
    console.log(this)
    console.log("Your browser screen resolution is (width x height):"+
        window.screen.width+"px"+
        window.screen.height+"px");
}
gd();
function getResolution() {
    console.log("Your OS screen resolution is: " + window.screen.width * window.devicePixelRatio + "x" +
        window.screen.height * window.devicePixelRatio);
}
getResolution();
```

Screen Resolution \neq Window Width : Most OS changing screen dpi so screen.width mostly return screen size with OS dpi, for example one screen resolution is 1920x1080 and windows default dpi is 125 so JS screen.width return 1600px.

How To Publishing Your Website

https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Publishing_your_website

What's the Difference?

 HTML Hypertext Markup Language	Create the structure <ul style="list-style-type: none">Controls the layout of the contentProvides structure for the web page designThe fundamental building block of any web page
 CSS Cascading Style Sheet	Stylize the website <ul style="list-style-type: none">Applies style to the web page elementsTargets various screen sizes to make web pages responsivePrimarily handles the "look and feel" of a web page
 Javascript	Increase interactivity <ul style="list-style-type: none">Adds interactivity to a web pageHandles complex functions and featuresProgrammatic code which enhances functionality

