

**Basics**:

If we have properties that are highly related, we want to encapsulate them inside of an object.

For Example, We are building an application for drawing different kind of shapes like circles, rectangles and so on.

So we could declare multiple variables for a circle like,

let radius = 1;

let x = 1;

let y = 1;

and all these variables are highly related, they represent a circle.

Better approach is to put these variables inside of an object which we can send anywhere in our program or pass to any function and all these variables will be available in that object.

So let us define a circle object using object literal syntax,

const circle = {

}

Inside it we will define one or more *key-value* pairs.

const circle = {

  radius: 1,

};

Now this value 1(one) can be of **any** type in JavaScript. It can even be another object!

const circle = {

  radius: 1,

  location: {

    x: 1,

    y: 1,

  },

};

Here we defined another key called location and set its value to another object.

One of the purpose of an object is to group related variables but it is not the only purpose. *Quite often we have functions that should operate on these variables*.

For example, we can have a draw function for drawing a circle.

function draw() {}

or a function for moving the circle.

function move() {}

Again, these functions are highly related to the variables we have defined in circle object. So instead of defining these functions in a standalone way, It is better to put these functions inside the circle object.

const circle = {

  radius: 1,

  location: {

    x: 1,

    y: 1,

  },

  draw: function () {

    console.log("draw");

  },

};

To access this draw function, we use dot notation,

circle.draw();

In console we can see “draw” printed out.

This style of implementing logic is called *Object – Oriented – Programming* or OOP.

“*OOP is basically a style of programming where we see a program as a collection of objects that talk to each other to perform some functionality*”.

Note: If a function is a part of an object, we call that function a method

**Factory Functions**:

Just like a factory producing products, these factory functions produce/*return* objects.

Let us name this function *createCircle* which will create our circle object.

function createCircle(){

}

For simplicity purpose we will be using only radius and draw functions as parameters.

function createCircle() {

  return {

    radius: 1,

    draw: function () {

      console.log("Draw");

    },

  };

}

Here we are simply returning an object from *createCircle* factory function. We make some more changes,

🡪 Instead of hardcoding *radius* value pass *radius* as function parameter.

🡪 *draw* method can be defined in a simpler way, instead of defining it as a key value pair.

function createCircle(radius) {

  return {

    radius: radius,

    draw() {

      console.log("Draw");

    },

  };

}

Note: In modern JavaScript if key-value pair are same as in our case radius : radius, we can remove value and only show key.

function createCircle(radius) {

  return {

    radius,

    draw() {

      console.log("Draw");

    },

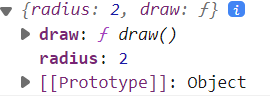
  };

}

Now we have a factory function, we can simply call it to create a circle object.

const circle2 = createCircle(2);

console.log(circle2);



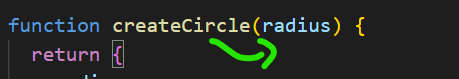
This circle2 object has radius 2 and a draw method.

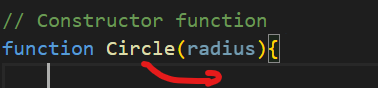


Beauty of this factory function is that we have defined our logic in one place. So we can call this function with different arguments and get different circle objects in return.

**Constructor Functions**:

Another pattern for creating object is by using a constructor function. Its job is same as of factory function however naming convention is different.

🡨Factory function names are in *camel case* notation(*first letter in first word is in lowercase*).

🡨 Constructor function names are in *Pascal* notation(*first letter in first word is in Uppercase*).

Instead of returning an object as we did in factory function, we are going to use a different approach to initialize an object.

In JavaScript we have a keyword called *this*, *which is a reference to the object that is currently executing our code*. For now just imagine that *this* references an empty object.

function Circle(radius){

    this 🡨reference current object

}

Now, we know that using dot notation we can access properties of an object. We can read them , set them etc.

function Circle(radius) {

  this.radius = radius;

}

On this empty object, we have added a new property called *radius* and set it to *radius* argument.

Note: In JavaScript our objects are dynamic, once we create them we can always add additional properties or methods to them.

Same approach is used to add a *draw* method to this empty object.

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

    console.log("draw");

  };

}

Finally in order to create a circle object using Circle constructor function, we are going to define a constant called circle and set it to new Circle()

const circle = new Circle(1);

When we use new operator here, three things happen in background,

🡪 new operator creates an empty JavaScript object, like

const x = {}

🡪 It will use *this* to point to this newly created empty object. So radius property and draw method are added in this new object.

🡪 Finally, new operator will return *new object* (*with new properties and methods*) from constructor function.

In this line

const circle = new Circle(1);

*circle* is just pointed to that new object.

**Dynamic Nature of objects**:

Objects in JavaScript are dynamic, which means once you create them you can always add new properties and methods or remove existing ones.

const circle = {

  radius: 1,

};

console.log(circle);

here we have a *circle* object with a single property, *radius*.

We can add a new property called *color* and a method called draw,

const circle = {

  radius: 1,

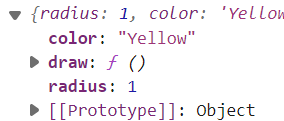
};

circle.color = "Yellow";

circle.draw = function () {};

console.log(circle);

We can see two properties and a method in console.



We can also delete properties after adding them,

const circle = {

  radius: 1,

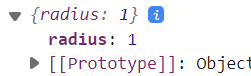
};

circle.color = "Yellow";

circle.draw = function () {};

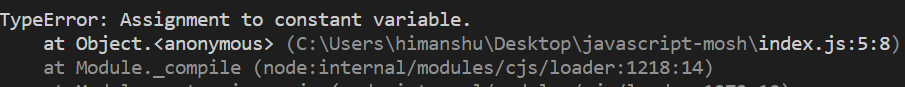
delete circle.color;

delete circle.draw;

🡨Only radius property is here.

Note: Some developers find it confusing that we have defined a constant object (*const circle = {}*) but still able to add new values and delete existing ones.

It is because *circle* variable itself is a constant so we cannot define another object with same name or else we will get



Otherwise changing values inside the object are supported.

**Constructor Property**:

Every object in JavaScript has a property called constructor and that references the function that was used to construct or create that object.

So here we have two objects created by Factory and Constructor functions respectively

//Factory function

function createCircle(radius) {

  return {

    radius,

    draw: function () {

      console.log("Draw");

    },

  };

}

const circleFac = createCircle(1);

//Constructor function

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

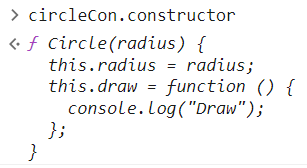
    console.log("Draw");

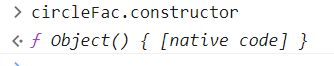
  };

}

const circleCon = new Circle(1);

Let us look at the constructor property of these two objects.

🡨Object created by constructor, returns *Circle* function that we used to create this object.

🡨 Object created by factory, return a built in constructor function Object().

*What exactly is happening here*?

When we create an object using object literal syntax { },

Internally in JavaScript engine this constructor function is used.

Let us create an empty object using this syntax,

let someObject = {};



It is same as saying

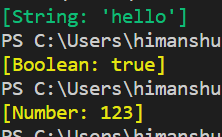
let someObject = new Object();

In JavaScript we have a few built in constructors,

let word = new String("hello");

let choice = new Boolean(true);

let number = new Number(123);

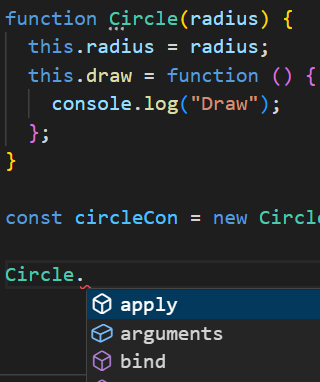


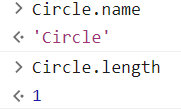
Here String, Boolean and Number are constructors as you can notice they start with capital letters(Pascal Convention)

**Functions are Objects**:

One of the confusing concepts in JavaScript is that functions are objects!

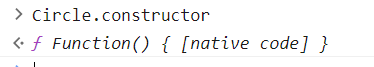
When you use dot operator on a function, you will see properties and methods available for object.

 Note: Purple icons are *methods* and blue icons are *properties*.

🡨 Circle.length returns number of arguments.

As we know that every JavaScript object has a constructor property which references to the function that was used to create that object.

*Who do you think created this* ***Circle*** *object*?



Here we have another built in constructor called Function. When we declare our very own constructor function, internally JavaScript engine will use Function constructor to create this object.

Let us construct a constructor function using Function constructor,

const Circle1Fun = new Function(

  "radius",

  ` this.radius = radius;

this.draw = function () {

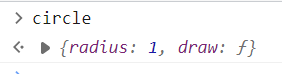
  console.log("Draw");

};`

);

const circle = new Circle1Fun(1);

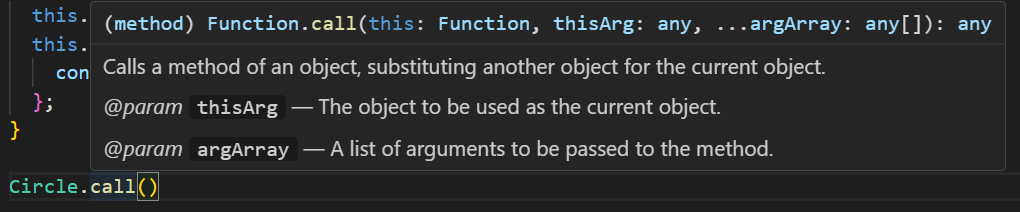
First argument is radius parameter and second argument is body of the function.

We created a real circle object with two parameters.

*Some other methods available in our functions*:

call method:

With the call method, we can call a function. Look at the arguments,



First argument is *this* argument.

Here we need to pass an empty object {} and *this* will reference the object that we passed.

Circle.call({})

After that we add our arguments explicitly,

Circle.call({}, 1);

Note: Both these expressions are exactly the same

Circle.call({}, 1);

new Circle(1);

When we use new operator, it will internally create an empty object {} and pass that as the first argument of call method which will determine the context of this.

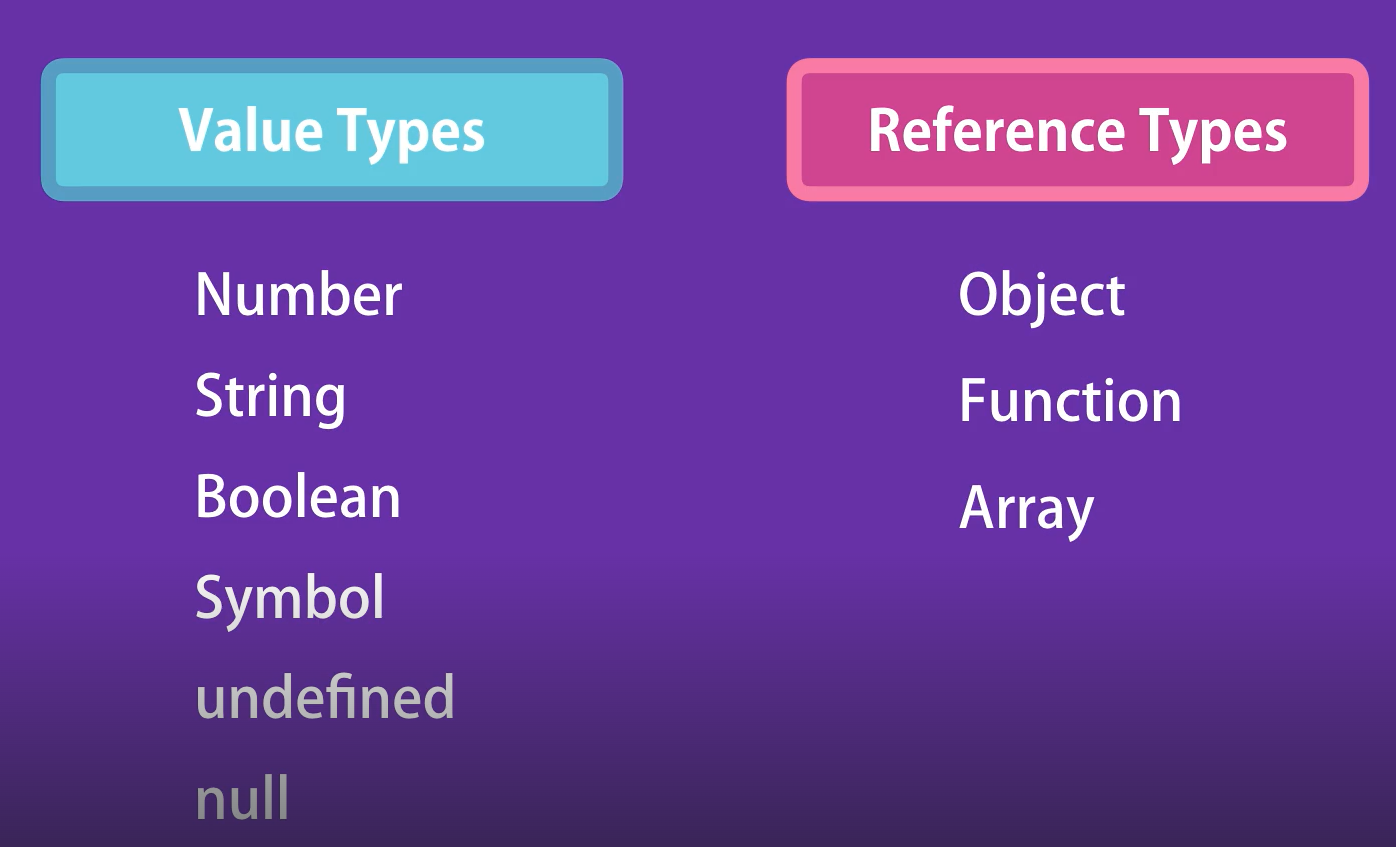
apply method:

Similar to call method, we have a apply method, Only difference is that instead of passing arguments explicitly we can pass them in form of an array.

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

**Value vs Reference types**:

In JavaScript we have two categories of types. On one side we have Value types called primitives and on other side we have Reference types.



In last lecture we learned that functions are objects, same is true for arrays as well. So concisely we have *primitives and objects*. We need to learn how both of these types behave differently.

*primitive types example*:

let x = 10;

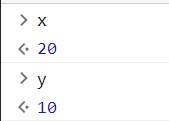
let y = x;

x = 20;

What should be the value of y?

Hint: Both x and y are *independent* variables.

Result:



Explanation:

The value 10 was stored initially inside variable X and when we copy that variable, the value that was stored 🡪10 was copied in to the new variable Y.

So both these variables X and Y are independent of each other.

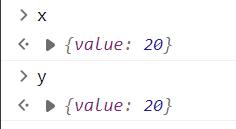
*reference / objects example*:

let x = { value: 10 };

let y = x;

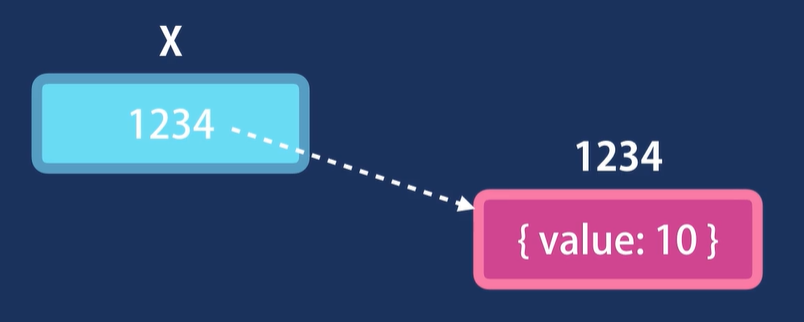
x.value = 20;

Result:

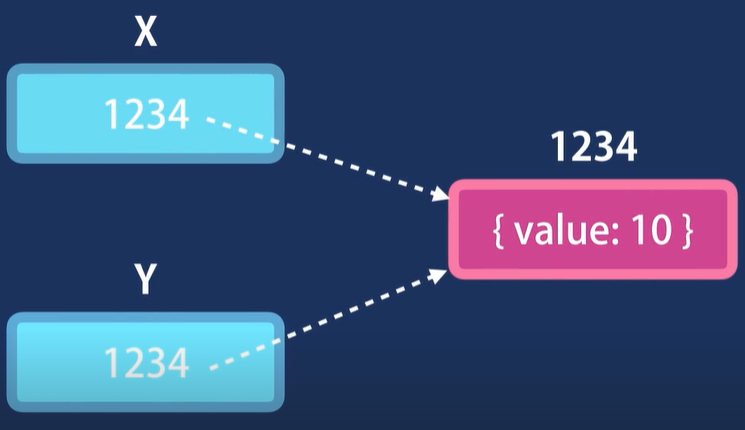


Explanation:

When we use an object {value: 10} that object is not stored in the variable X. That object is stored somewhere else in the memory and the address of that memory location is stored inside that variable.

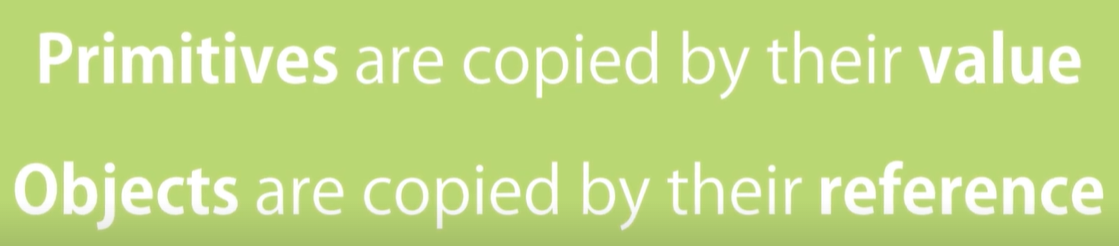


So when we copy X in to Y, it is the address of the reference that is copied. In other words both X and Y are pointing to the same object in memory.



And when we modify that object by either X or Y, our changes are immediately visible to the other variables.

*Conclusion*:



*Another example of primitive types*:

let number = 10;

function increase(number) {

  number++;

}

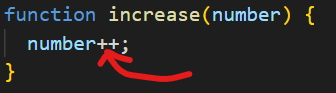
increase(number);

console.log(number);

What should be the value of number?

Solution: 10

Explanation: When we call *increase* function and pass this *number* variable, its value is copied into the parameter which is local in this function.

So this number 🡪 

and this number 🡪 

are completely independent of each other. Inside the function value of number is 11 but outside the scope of this function value of number is still 10.

*Another example of reference/object types*:

let obj = { value: 10 };

function increase(obj) {

  obj.value++;

}

increase(obj);

console.log(obj);

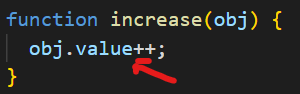
What should be the value of *obj* object now ?

Solution:

{ value: 11 }

Explanation:

It is because when we call increase function and pass this object *obj*, this object is passed by its reference.

So local parameter 🡪 

and 

are same.

Here we are not dealing with two independent copies instead we have two variables that are pointing to the same object.

Any changes we make to this object (**let obj = { value: 10 };**) will be visible to the other object.

**Enumerating properties of an Object**:

Here we have circle object with radius property and draw method.

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

Earlier in the course we learned how to use *for – in* , *for – of* loops to iterate over the properties in object.

for (let key in circle) {

  console.log(key, circle[key]);

}

We get the name of the key and its corresponding value.



We can use a *for – of* loop with iterables, *but a JavaScript Object is not iterable*,

for (let keys of circle) {

  console.log(keys);

}

TypeError: circle is not iterable

So instead we use Object.keys method, which returns an array and *Arrays are iterable*.

for (let keys of Object.keys(circle)) {

  console.log(keys); //radius draw

}

*What happens in background of Object.keys method*?

Since *Object* is a built in constructor function and each function in JavaScript is an object. So this *Object* has properties and methods that we can access using dot notation.

Another useful method of *Object* is ***entries***,

for (let entry of Object.entries(circle)) {

  console.log(entry);

}

This method returns each key – value pair as an array.

O/P:

[ 'radius', 1 ]

[ 'draw', [Function: draw] ]

Sometimes we want to check if an object has a given property or a method, we can use *in* operator,

if ("radius" in circle) {

  console.log("Yes"); //Yes

}

**Cloning an Object**:

Suppose we want to create another circle object which is a copy of our original circle object.

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

So we create an empty object and call it *another*.

const another = {};

Now we use for – in loop to iterate over all the properties of circle object and copy them into this new object.

for (let key in circle) {

  another[key] = circle[key];

}

Here we use bracket notation to access a property with a given key.

In the first iteration, this code will be equivalent to,

*another[“radius”] = circle[“radius”]*

So *on the right side we are reading the value of radius property of the circle (which is one) and pasting it in radius property of another object*.

console.log(another); // { radius: 1, draw: [Function: draw] }

Our cloning of circle object works but this approach is a little bit old. In modern JavaScript we have a better ways to achieve the same thing. One way is to use *Object.assign* method.

const another = Object.assign(circle);

console.log(another); // { radius: 1, draw: [Function: draw] }

We can also add other properties while cloning object,

const another = Object.assign({ color: "Yellow" }, circle);

console.log(another); // { color: 'Yellow', radius: 1, draw: [Function: draw] }

This *another* object now has the *color* property as well as the members of *circle* object.

*Cloning by using spread operator(****simplest way****)*:

const another = { ...circle };

console.log(another); // { radius: 1, draw: [Function: draw] }

We put *…circle* inside { } and that’s it

**Garbage Collection**:

In Low level languages like C or C++ when creating an object, we need to allocate memory to it and when we are done we have to deallocate memory.

But in JavaScript, we do not have this concept. We can easily create an object and at the time when we initialize this object, the memory is automatically allocated to this object and when we are done we do not have to deallocate the memory.

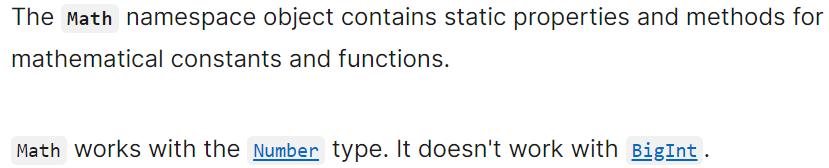
So our JavaScript engine has a ***Garbage Collector***,

***“****The job of this garbage collector is to find the variables or constants that are no longer used and then deallocate the memory that allocated to them earlier****”*.**

Based on some complex algorithms, this garbage collector runs in the background, we as a JavaScript developer has no part in it.

**Math**:

Go to <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math>



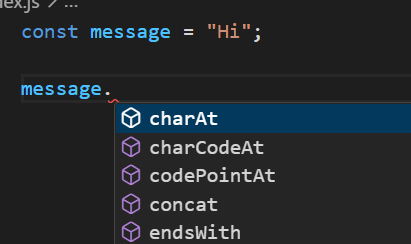
Try methods like *random*, *max*, *min*, and *round* on your own.

**String**:

I will define a constant *message* here and set it to a string.

const message = "Hi";

when I use dot notation on this , I get various properties and methods,



But earlier in the course we studied that string is a primitive type and primitive types do not have properties and methods only objects do.

*Then why we are seeing this*?

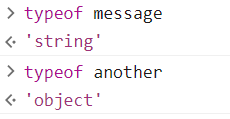
The reason for this is that in JavaScript we have two kinds of strings. *String primitives* and *String objects*.

//string primitive

const message = "Hi";

//string object

const another = new String("Hi");



***“****However when we use dot notation JavaScript engine internally wraps the string primitive with a string object****”***.

To know more about strings,

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String>

*To get the length of string*:

const message = "Hi";

console.log(message.length); //2

particularly useful in cases where we want user to input a certain number of characters in a field or limit the user input.

*To get a character at a specific index*:

const message = "Hi";

console.log(message[0]); //H

*To check if string has a specific word*:

const message = "Hi";

console.log(message.includes("H")); //true

*To get the index of a string*:

const message = "Hi";

console.log(message.indexOf("i")); //1

*To replace part of string*:

const message = "Hi";

console.log(message.replace("Hi", "Hello")); //Hello

*replace* method returns a new string and does not modify the original one.

startsWith, endsWith, toUpperCase, toLowerCase.

🡪 trim method and its variations like trimLeft, trimRight

🡪 split method.

**Template Literals**:

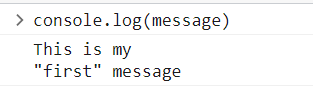
If you want to write a long message with many special characters , starting from ES6 we have template literals.

These are denoted by **` `** (*back ticks*)

Example:

*A string using escape sequences*:

const message = "This is my\n" + '"first" message';



We get this type of string in our console with new line and “ ” in between our string using escape sequences.

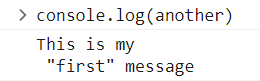
But the code looks ugly.

*A string using template literal syntax*:

const another = `This is my

 "first" message

`;



We get the same output and the code is cleaner.

Another benefit of using template literals is that we can use ***placeholders***.

user = "Himanshu";

const message = `Hi John,

Thank you for the invite

Regards,

${user}

`;

console.log(message);

O/P:

Hi John,

Thank you for the invite

Regards,

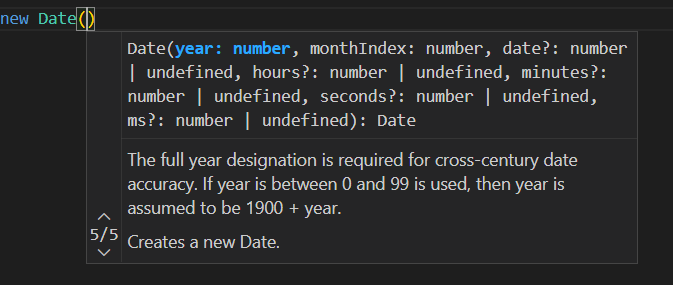
Himanshu 🡪Here the value is dynamically changed

Inside the placeholder we can place any valid expression in between the curly braces.

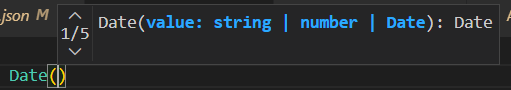
**Date**:

Let us look at different ways to create a *Date* object.

Here we first use Date constructor and we can see different versions of this date constructor.



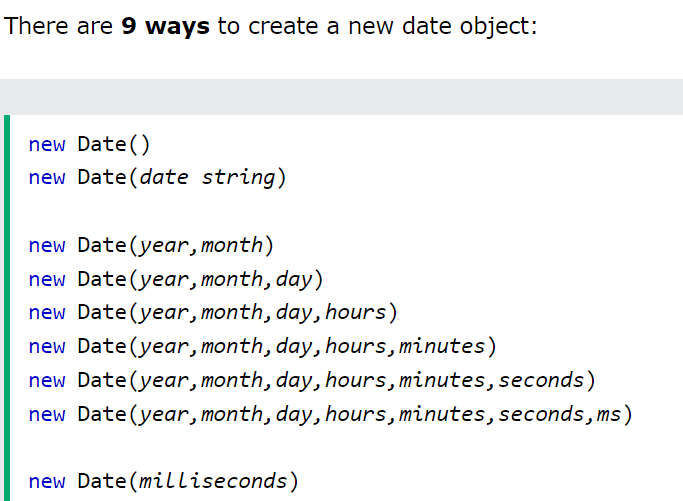
*Or*,



If we do not use, Date constructor *without any parameters it returns the current time*.

const now = new Date();

console.log(now); //2023-07-23T07:54:45.216Z

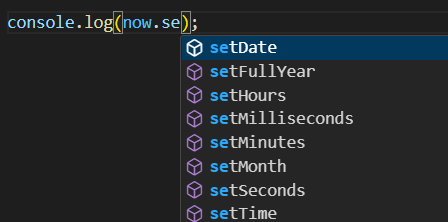
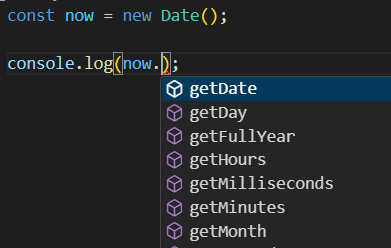


Note: One peculiar thing about Date objects is that months are *0 based*, means 0 is for January and 11 for December.

const date2 = new Date(2018, 11, 11, 9);

console.log(date2); //2018-12-11T03:30:00.000Z

All these date objects have *get and set* methods,



Also we have methods to convert the date objects into string like *toDateString*, *toTimeString*, *toISOString*.

const now = new Date();

console.log(now); //2023-07-23T08:33:14.966Z

console.log(now.toDateString()); //Sun Jul 23 2023

console.log(now.toTimeString()); //14:06:17 GMT+0530 (India Standard Time)

console.log(now.toISOString()); //2023-07-23T08:36:17.909Z

Note: The *toISOString*, it returns a string in a very particular format, 2023-07-23T08:36:17.909Z. *It is a standard ISO format and is commonly used to transfer Date and Time data between client and server*.

**Exercise 1: Address Object**:

Create an address object with three properties; street, city and zip code. Then create a function called showAddress that takes an address object and displays all the properties of this object along with their values.

*My way*:

const address1 = {

  street: 2,

  city: "delhi",

  zipcode: 110094,

};

function showAddress(address) {

  let output = {};

  for (let key of Object.keys(address)) {

    output[key] = address[key];

  }

  return output;

}

console.log(showAddress(address1)); { street: 2, city: 'delhi', zipcode: 110094 }

*Mosh way*:

function showAddress(address) {

  for (let key in address) console.log(key, address[key]);

}

showAddress(address1);

**Exercise 2: Factory & Constructor functions**:

Initialize the movies object using factory function and then using constructor function.

const movies = [

  { title: "a", year: 2018, rating: 4.5 },

  { title: "b", year: 2018, rating: 4.7 },

  { title: "c", year: 2018, rating: 3 },

  { title: "d", year: 2017, rating: 4.5 },

];

function getMovies(title, year, rating) { //Factory function

  return {

    title,

    year,

    rating,

  };

}

console.log(getMovies("a", 2018, 4.5));

function Movies(title, year, rating) { // Constructor function

  this.title = title;

  this.year = year;

  this.rating = rating;

}

const movie1 = new GetMovies("b", 2018, 4.7);

console.log(movie1);

**Object Equality**:

Check if two objects are same.

function Address(street, city, zip) {

  this.street = street;

  this.city = city;

  this.zip = zip;

}

const address1 = new Address(2, "delhi", 110094); //Object 1

const address2 = new Address(2, "delhi", 110094); //Object 2

const address3 = new Address(3, "delhi", 110094); //Object 3

function areEqual(address1, address2) {

  return (

    address1.street === address2.street &&

    address1.city === address2.city &&

    address1.zip === address2.zip

  );

}

console.log(areEqual(address1, address2)); //true

console.log(areEqual(address1, address3)); //false

Check if two objects are referencing the same memory space,

const address1 = new Address(2, "delhi", 110094);

const address2 = new Address(2, "delhi", 110094);

let address3 = new Address(3, "delhi", 110094);

address3 = address1;

function areSame(address1, address2) {

  return address1 === address2;

}

console.log(areSame(address1, address2)); //false

console.log(areSame(address1, address3)); //true

**Blog Post Object**:

Create a ‘blog post’ object:

let post = {

  title: "a",

  body: "b",

  author: "c",

  views: 100,

  comments: [

    {

      author: "a",

      body: "b",

    },

    {

      author: "c",

      body: "d",

    },

  ],

  isLive: true,

};

Now create the same object but by using constructor function

function Post(title, body, author) {

  this.title = title;

  this.body = body;

  this.author = author;

  this.views = 0;

  this.comments = [];

  this.isLive = false;

}

let post1 = new Post("tit", "bo", "aut");

console.log(post1);

Note: We initialized only three parameters title, body and *author* in the constructor function. Because *views*, *comments* and *isLive* properties have to be created only after initialization.