**JavaScript (JS)**

**[**<https://developer.mozilla.org/en-US/docs/Web/JavaScript>, <https://www.w3schools.com/js/default.asp>**]**

JavaScript (JS) is a high-level scripting language for the Web that conforms to the ECMAScript (ES) [<https://en.wikipedia.org/wiki/ECMAScript>] specification, and is used to develop dynamic and interactive Web pages. The current version is ES2022 or ES13 (<https://codingbeautydev.com/blog/es13-javascript-features/>). It is high-level in the sense that you don’t have to deal with things like memory management etc. like you would in a lower level PL such as C.

Not only is JS used in client-side (vanilla JS, Angular.js, React.js, Vue.js) or server-side (Node.js) Web programming, it is also used in mobile application development (React Native, NativeScript, Ionic). It is even used for desktop application development (Electron JS). So it is a very popular and commonly used PL across many different platforms. As a side note, JS has nothing to do with Java, which is a compiled language used to develop desktop applications or server-side Web applications using the Spring framework.

As the name implies, JS is a scripting language, i.e., the JS code is NOT compiled into machine code, but is rather interpreted using a JS interpreter or JS engine. For example, Google Chrome uses Chrome V8 engine. Node.js [<https://nodejs.org/en/>], which is a JS runtime environment used for server-side Web development, also uses the Chrome's V8 JS engine. There are many other JS engines on the market. For a detailed list of other JS engines see <https://en.wikipedia.org/wiki/JavaScript_engine>.

**What you can do with JS & Where to place JS Code in your Web page**

Consider a Web application where you would like to change some attribute of an HTML element when a button on the page is pressed. Notice that here we would like to change the Web page after it has been loaded and rendered. To achieve this we need to run some code when the button is pressed that would make the necessary changes to the DOM. This is exactly what JS is used for.

In the following example, we have an “img” element and we change the “src” attribute of this element when a button is pressed. Specifically, when the user presses “Turn the light on” we change the “src” attribute of the image to point to the image that shows a light with the bulb on, and when the user presses “Turn the light off”, we change the “src” attribute of the image to point to the image that shows a light bulb with the bulb off.

Look at 05-JS/01-FirstEx/index1.html

When you look at this document you will see that we put the JS code directly inside the HTML element under the “onclick” event. That is, when the user presses the respective button, the JS code inside the onclick event is executed, which simply selects the image element from the document, and then changes its “src” attribute. Notice that we assign a unique identifier to the img element using the id=”myImage” attribute, and then select that image element using document.getElementById().

Although you can write JS code directly inside the HTML element as we did in the previous example, this is NOT the recommended way. Recall that we could also write CSS styles inside HTML elements with style=”” attribute, but we usually do not do this as this mixes up the document content with styling, which is not recommended. The same idea applies to the JS code. Instead of writing the JS code directly inside the HTML element, we separate it out from the HTML document and collect the JS code in a separate place. One way to do this is to put all JS code inside <script>..</script> tags, and call this JS code from within the HTML elements. Although <script></script> tags can be placed anywhere in the HTML document, it is good practice to place them at the end of the HTML document right before the end of the body tag. Thus you make sure that your Web page is loaded first before the JS code is loaded and executed. This is illustrated in the following example:

Look at 05-JS/01-FirstEx/index2.html

This example does the same code as the first example, but instead, the JS code is placed inside <script>..</script> tags, and the JS functions are registered as call-back methods for the button press events. So when a button is pressed, the corresponding JS function is called, which simply changes the “src” attribute of the image.

It is also possible to take the JS code out of the HTML file altogether and put it inside a separate JS file as we did with CSS. You can then include the JS code into the HTML file using <script src=””></script> tags. This method not only separates the JS and HTML completely, but it also allows the JS code to be included and used by many different HTML pages. The following example illustrates how this is done:

Look at 05-JS/01-FirstEx/index3.html

In the above examples, we used JS to modify an attribute of an HTML element. Specifically, we dynamically changed the “src” attribute of the “img” element. You can use JS for other purposes. For example, you can also use JS to change the CSS styles of HTML elements, or even the content of HTML elements altogether. In fact, you can modify the entire DOM of the page using JS. The following example illustrates some examples:

Look at 05-JS/01-FirstEx/index4.html

**Debugging JS Code**

As we discussed, JS code runs inside a JS Engine, which is embedded inside a Browser. So when you load a Web page that contains some JS code, that JS code is executed by the JS Engine.

To debug the JS code executed by the Browser, you can make use of the Developer Tools in your browser. In Chrome, they are under “Options”🡪“More Tools”🡪”Developer Tools”. Alternatively, you can right-click on the page and select “Inspect”.

When you open the “Developer Tools” you see many tabs starting with “Elements”, which shows the HTML elements that make up the page, “Console”, which where the JS output is sent and also where you can directly type in and execute JS code. There are 3 methods associated with the console object to send output to the console:

* console.log(‘Hello’);
* console.error(‘Hello’);
* console.warn(‘Hello’);

The only difference between these are that they are displayed in different colors on the console. Specifically, log uses black, error uses red and warn uses yellow color to display its output. For the most part we will use console.log to send some output to the console.

**JS Basics**

Now that we have seen what you can do with JS, it is time to look into how you can write code in JS. As with learning any other programming language, we will follow the outline given below:

* Basic data types (number, boolean, null, undefined), variable declaration
* Operators: Arithmetic, assignment, logical, bitwise
* Compound data types: strings, arrays, object literals (structures)
* Conditional statements (if, switch)
* Looping constructs (for, while, do/while)
* Functions & variable scope
* Higher-order array methods
* JS data structures: array, stack, queue, set, and map
* Classes and objects

**JS Syntax and Comments**

JS syntax is very similar to C. Variable naming conventions, expression syntax, block code syntax, if statements, for/while statements, functions etc. They are almost the same as in C. Even the comments are the same. You can write single line comments using //, and multiple line comments in between /\* Comment \*/

|  |
| --- |
| console.log('Hello');  console.error('Error');  console.warn('Hello');  // This is a single line comment  /\* This is a  multi-line comment  \*/ |

**Variable Declaration, Data Types**

Since JS is a scripting language, you do not specify the variable type when declaring variables like you would in strongly-typed PLs such as C, C++, C#, Java, etc. Before ES6 (ECMAScript 2015), the only way to declare a variable was using the “var” keyword. ES6 added two more ways to declare variables in JS. Now it is possible to declare variables using the keywords “var”, “let” and “const”. While the variables declared with “var” have global or function scope, variables declared with “let” and “const” have local or block scope. We will look at variable scope later in this lecture.

As for data types, JS has the usual basic data types number (both decimal and floating point) and boolean, and also complex data types such as strings and objects. Arrays are also considered to be objects.

|  |
| --- |
| let x = 3; // Decimal number  let y = 4.5; // Floating-point number  let b = true; // boolean  let u; // A variable declared without a value will have the *type* and *value* of **undefined**  let t = undefined; // “t” also has a *type* and *value* of **undefined**  let z = null; // A variable that has null value. “z” has a *type* of **object** and a *value* of **null**.  let s = "Hello"; // string: Can also use single quotes 'Hello'  let A = [4, 2, 6, 7]; // Array  let obj = {firstName: 'Ali', lastName: 'Kaya', age: 45}; // Object literal (like C structure)  let F = function(n){return 2\*n;} // Function  console.log('x: ' + x);  console.log(`y: ${y}`); // Using back-ticks  console.log(`b: ${b}`); // Using back-ticks  console.log(`u: ${u}`);  console.log(`z: ${z}`);  console.log("s: <" + s + ">, s[1]: " + s[1]); // Strings are 0-indexed as in C  console.log(A);  console.log(obj);  console.log(F); |

You can use the typeof operator to find the type of a JS variable:

|  |
| --- |
| console.log(typeof(x));  console.log(typeof(y));  console.log(typeof(b));  console.log(typeof(s));  console.log(typeof(A)); |

Look at 05-JS/02-Variables/script.js

Notice that there is no separate character type. Any value in between single quotes ‘a’ is considered to be a string of length 1. JS uses the Unicode character set. Unicode covers (almost) all the characters, punctuations, and symbols in the world. Furthermore, all JS identifiers are **case sensitive**.

**JS Operators**

JS has the usual arithmetic operators of +, -, \*, /, % and increment/decrement operators ++, --. Additionally, ES7 (ECMAscript 2016) has added an exponentiation operator \*\*:

|  |
| --- |
| let x = 2\*\*4; // 2^4 = 16 |

The usual assignment operators of =, +=, -=, \*=, /=, %= \*\*= are also available.

Finally, the logical operators are: && (AND), || (OR), ! (NOT).

As for the comparison operators, JS has the usual <, <=, >=, > and the tertiary operator ?:. But for equal and not equal, JS has two comparison operators:

|  |  |
| --- | --- |
| == | equal to |
| === | equal value and equal type |
| != | not equal |
| !=== | not equal value or not equal type |

The difference between == and === is the following. When you compare an integer with a string, if the number stored in the string evaluates to the integer being compared, == will return true, but === will return false because the types do not match. The same holds for !==.

|  |
| --- |
| If (2 == “2”) console.log(“equal”); else console.log(“false”);  If (2 === “2”) console.log(“equal”); else console.log(“false”); |

Finally, the usual bitwise operators & (AND), | (OR), ~ (NOT), ^ (XOR), << (Zero fill left shift), >> (Signed right shift), >>> (Zero fill right shift) are also available.

Look at 05-JS/03-Operators/index.html

**Compound data types: strings, arrays, structures (objects)**

JS has the usual compound data types: string, array and structure. All of these data types are in fact represented as JS objects.

A **string** is a sequence of characters in single or double quotes, indexed starting at 0.

|  |
| --- |
| const s = “Hello world”; // Can also use single quotes |

The + operator is used to concatenate two strings:

|  |
| --- |
| const s = “Hello” + “ World”; |

When you add a string with another string or number, you get a new string that is formed by concatenating the two strings:

|  |
| --- |
| const age = 48;  let s = “My ” + “age ” + “is: “ + age;  console.log(s); |

JS evaluates an expression from left to right. Therefore, the following two expressions will result in two different values:

|  |
| --- |
| let s = 16 + 4 + “Hello”; // Evaluates to “20Hello”  s = “Hello”+16+4; // Evaluates to “Hello164” |

Since strings are objects, they already have many built-in methods. For a complete JS string methods, refer to [<https://www.w3schools.com/jsref/jsref_obj_string.asp>].

Look at 05-JS/04-CompoundDataTypes/index1.html

An **array** is a container that holds multiple values. Unlike strongly typed PLs like C, C++, Java, C#, a JS array can hold values of different type. Arrays are represented as objects, and have many built-in methods.

Look at 05-JS/04-CompoundDataTypes/index2.html

An **object literal** is very similar to a structure in C and consists of key/value pairs.

Look at 05-JS/04-CompoundDataTypes/index3.html

**Conditional Statements**

JS has the usual if & switch statements to form conditional statements, and their syntax is exactly the same as in C:

|  |
| --- |
| const x = 10;  if (x === 10){  console.log(‘x is 10’);  else if (x >10) {  console.log(‘x is greater than 10’);  } else {  console.log(‘x is less than 10’);  } //end-if |

Look at 05-JS/04-CompoundDataTypes/index4.html

**Looping Constructs**

JS has the usual for, while, do/while looping constructs and they have the same syntax as in C.

Look at 05-JS/04-CompoundDataTypes/index4.html

**Functions**

JS functions also work very similar to functions in C.

|  |
| --- |
| function Add(num1, num2){return num1+num2;}  let x = Add(2, 3);  console.log(`x: ${x}`); // prints x: 5  console.log(Add(2)); // prints NaN, i.e., not a number |

ES6 introduced the arrow function, a.k.a., the lambda function:

|  |
| --- |
| // Define the function and store the function pointer in F. Below we invoke F.  const F = (num1=1, num2=2) => { return num1+num2; };  console.log(F(3, 8)); // prints 11  // If the function contains just one statement, we don’t need the curly braces.  const F2 = (num1=1, num2=2) => num1+num2;  console.log(F2(1, 4)); // Prints 5 |

Look at 05-JS/05-Functions/index.html

**Variable Scope**

As we stated before, there are 3 ways to declare variables in JS since ES6: Using “var”, “let” and “const”.

* Variables declared with “var” are globally scoped. You should avoid declaring variables with “var” unless you really have to. It is also important to note that if you directly assign a value to a variable without first declaring it, it becomes a global variable!
* Variables declared with “let” are locally scoped, i.e., block scope, and are only valid within the block in which they are declared.
* Variables declared with “const” are also locally scoped, i.e., block scope, and are only valid within the block in which they are declared. The difference to “let” is that you cannot change the assignment to the variables declared with “const”. In other words, you cannot assign a new value to the variable, but if the variable points to an object such as an array, you can change the object, but not the assignment to the object.

|  |
| --- |
| var x = 3; // Global scope  function F3(){  x = 7; // Access global variable  let sum = 0; // Block scope: Since the block is the function here, it is function scope  for (let i=0; i<10; i++){ // i is also block scope  var y = 4; // Function scope  z = 9; // z initialized without declaration and becomes globally scoped  let t = 2; // Block scope  sum += i;  } //end-for  console.log(`y: ${y}`); // Prints 4  console.log(`z: ${z}`); // Prints 9  console.log(`sum: ${sum}`); // prints 9\*10/2 = 45  // console.log(t); // This is not possible. Can’t access t because it is block scope  } // F3  console.log(`x before calling F3: ${x}`); // Prints 3  F3();  console.log(`x after calling F3: ${x}`); // Prints 7  //console.log(`y: ${y}`); // Cannot access y. It has function scope.  console.log(`z: ${z}`); // Prints 8 |

Look at 05-JS/05-Functions/index.html

**Higher Order Array Methods**

JS has some advanced methods for array manipulation, some of which was introduced with ES6.

1. **Arrays.forEach()**

In addition to using a for loop to iterate over an array, it is possible to use the forEach method for array iteration:

|  |
| --- |
| function F1(val, index, array){  if (s == "") s += val;  else s += ", " + val;  } //end-F1  s="";  A.forEach(F1);  console.log(`s: ${s}`); |

Alternatively, you can use an arrow function instead of a named function:

|  |
| --- |
| s="";  A.forEach((val) => {  if (s == "") s += val;  else s += ", " + val;  });  console.log(`s: ${s}`); |

1. **Arrays.map()**

The map() method creates a new array by performing a function on each array element. It does not execute the function for array elements without values, and does not change the original array. Instead, it creates a new array. Here is an example, where we multiply each array element by 2 and create a new array:

|  |
| --- |
| B = A.map((val, index, array) =>{return 2\*val;}); |

1. **Array.filter()**

The filter() method creates a new array with array elements that passes a test. This example creates a new array from elements with a value larger than 5:

|  |
| --- |
| B = A.filter((val, index, array) =>{return val > 5;}); |

1. **Array.reduce()**

The reduce() method runs a function on each array element to produce (reduce it to) a single value. The reduce() method works from left-to-right in the array. Here is a code that computes the sum of the elements in the array:

|  |
| --- |
| let total = A.reduce((sum=0, val, index, array) =>{sum += val; return sum;}); |

1. **Array.every()**

The every() method check if all array values pass a test. This example check if all array values are smaller than 10:

|  |
| --- |
| let allSmallerThan10 = A.every((val, index, array) =>{return val < 10;}); |

1. **Array.some()**

The some() method check if some array values pass a test. This example checks if some (at least one) array values are larger than 8:

|  |
| --- |
| let someBiggerThan8 = A.some((val, index, array) =>{return val > 8;});  console.log(`Are some array elements bigger than 8? ${someBiggerThan8}`); // prints true  let someBiggerThan10 = A.some((val, index, array) =>{return val > 10;});  console.log(`Are some array elements bigger than 10? ${someBiggerThan10}`); // prints false |

1. **Array.find()**

The find() method returns the value of the first array element that passes a test function. This example finds (returns the value of) the first element that is larger than 5:

|  |
| --- |
| let item = A.find((val)=>{return val > 5;}); |

1. **Array.findIndex()**

The findIndex() method returns the index of the first array element that passes a test function. This example finds the index of the first element that is larger than 8:

|  |
| --- |
| let index = A.findIndex((val)=>{return val > 8;}); |

1. **Streams**

It is possible to chain the above Array methods and create what is called a stream for data processing. For example, assume that we would like to filter out all elements of array bigger than 7 and add 10 to every element in the resulting array. Instead of writing two separate independent methods (filter & map) to perform this operation, we chain these two methods so that the output of the filter method is fed into the map method. This is called streaming the data.

|  |
| --- |
| let B = A.filter((val)=>{return val>7;}).map((val)=>{return val+10;}); |

Look at 05-JS/06-HigherOrderArrayMethods/script.js

**JS Data Structures**

In addition to arrays, JS also has **sets** and **maps**. Recall from your data structures course that a set is a collection of keys that has no duplicates. A map on the other hand is a collection of key/value pairs, where each key is unique. Although JS does not have Stacks and Queues, we can easily implement a Stack and a Queue using an array.

|  |
| --- |
| let S1 = new Set([3, 5, "ali", 4.5, "veli"]);  S1.add(20);  S1.add('ali'); // No duplicates in a set. So, this does not add 'ali' again.  S1.delete(4.5);  S1.delete(5);  let exist = S1.has('ali'); |

|  |
| --- |
| let M = new Map([[3, "Ali"], [5, "Veli"], ["Tree", "OK"], [4.5, 88]]);  M.set("Cem", "Smart");  if (M.has("Tree")) M.delete("Tree");  if (M.has(4.5)) console.log(`The value associated with 4.5 is: ${M.get(4.5)}`); |

Look at 05-JS/07-DataStructures/

**Object Oriented Programming in JS**

We have already seen how we can create an arbitrary object literal:

|  |
| --- |
| const person1 = {  firstName: "John",  lastName: "Doe",  dob: new Date(1980, 3, 4), // (year, month, day)  getBirthYear: function(){  return this.dob.getFullYear();  },  getFullName: function(){  return `${this.firstName} ${this.lastName}`;  }  };  console.log("Some information about person1");  console.log(typeof person1);  console.log(person1);  console.log(person1.firstName);  console.log(person1.dob.getMonth()); // Prints 3  console.log(person1.getBirthYear()); // Prints 1980  console.log(person1.getFullName()); // Prints John Doe |

You can also create an Object() using the new keyword, and then add properties to the object.

|  |
| --- |
| const person2 = new Object(); // Create an empty object  person2.firstName = 'Mary';  person2.lastName = 'Smith';  person2.dob = new Date(1970, 6, 3);  person2.getBirthYear = function(){return this.dob.getFullYear();};  person2.getFullName = function(){return `${this.firstName} ${this.lastName}`;}  console.log("Some information about person2");  console.log(typeof person2);  console.log(person2);  console.log(person2.firstName);  console.log(person2.dob.getMonth()); // Prints 6  console.log(person2.getBirthYear()); // Prints 1970  console.log(person2.getFullName()); // Prints Mary Smith |

To access the members of an object, you can either use the “.” notation as we did above, or use the following notation:

|  |
| --- |
| console.log(person2.firstName);  console.log(person2[“firstName”]);  let x = “firstName”;  console.log(person2[x]); |

It is also possible to access ALL members of an object using the JS "for .. in" loop:

|  |
| --- |
| for (let x in person2){  console.log(`person2.${x}: ${person2[x]}`);  } //end-for |

It is also possible to add new properties to an existing object by simply giving it a value. OR, it is possible to delete an existing property (member) using the "delete" keyword:

|  |
| --- |
| person2.nationality = "English";  delete person2.dob; // person2.dob is not accessible anymore |

**Object Constructor Functions (ES5 or earlier)**

The examples given above are limited. They only create single objects. In general, we need a "blueprint" for creating many objects of the same "type". The way to create an "object type", is to use an object constructor, i.e., a class. Classes were first introduced in ES6. Before the introduction to classes, JS programmers used to make use of an object constructor function to create an object. Objects of the same type are then created by calling the constructor function with the new keyword:

|  |
| --- |
| function Person(firstName, lastName, age, gender){  this.firstName = firstName;  this.lastName = lastName;  this.age = age;  this.gender = gender;    this.getFullName = function() {  return `${this.firstName} ${this.lastName}`;  } //end-getFullName  } //end-Person  person3 = new Person("Ali", "Kaya", 25, "male");  person4 = new Person("Ayse", "Kus", 15, "female"); |

We know that the value of **this**, when used in an object, is the object itself. In a constructor function however, **this** does not have a value. It is a substitute for the new object. The value of **this** will become the new object when a new object is created.

**Class prototypes:** When we look at the objects created using the above-given constructor function, we will see that the member methods appear as member functions. This means that every object will have a member variable that points to the member function, which wastes a lot of space. Ideally, we want all member methods go into the class's \_\_proto\_\_ property, which is an object of function pointers. If we now put our member methods inside the class's prototypes, we will save a lot of space. To achieve this, we can add the member methods in Person.prototype property as follows:

|  |
| --- |
| function PersonConstructor(firstName, lastName, age, gender){  this.firstName = firstName;  this.lastName = lastName;  this.age = age;  this.gender = gender;  } //end-PersonConstructor  PersonConstructor.prototype.getFullName = function() {  return `${this.firstName} ${this.lastName}`;  } //end-getFullName  const person5 = new PersonConstructor("Alper", "Can", 35, "male"); |

**Built-in constructors for native objects:** JavaScript has built-in constructors for native objects such as Number, String, Boolean etc., but there is no reason to create complex objects. Primitive values are much faster. So as a rule of thumb,

* Use object literals {} instead of new Object().
* Use string literals "" instead of new String().
* Use number literals 12345 instead of new Number().
* Use boolean literals true / false instead of new Boolean().
* Use array literals [] instead of new Array().
* Use pattern literals /()/ instead of new RegExp().
* Use function expressions () {} instead of new Function().

|  |  |
| --- | --- |
| let x1 = new Object(); // A new Object object  let x2 = new String(); // A new String object  let x3 = new Number(); // A new Number object  let x4 = new Boolean(); // A new Boolean object  let x5 = new Array(); // A new Array object  let x6 = new RegExp(); // A new RegExp object  let x7 = new Function(); // A new Function object | let x1 = {}; // new object  let x2 = ""; // new primitive string  let x3 = 0; // new primitive number  let x4 = false; // new primitive boolean  let x5 = []; // new array object  let x6 = /()/ // new regexp object  let x7 = function(){}; // new function object |

Look at 05-JS/08-OOP/oop1.js

**JS Classes (ES6 and later)**

To make pure OOP possible, ES6 added to JS the concept of **class** found in all modern OOPLs. Thus it is not possible to write the blue-print for objects in a class and create new objects of that class using the new keyword. When you have classes, you should not use object constructor functions, which was necessary in ES5 and earlier. A class is nothing but a prettier way of implementing the same thing.

To declare a class, use the “**class**” keyword, and always add the constructor() method. The constructor method is called each time the class object is initialized. Notice that the member methods do not appear now in member list when we print then with “for..in” loop:

|  |
| --- |
| class Person {  constructor(firstName, lastName, age, gender){  this.firstName = firstName;  this.lastName = lastName;  this.age = age;  this.gender = gender;  } //end-constructor  // Member function  getFullName(){  return `${this.firstName} ${this.lastName}`;  } //end-getFullName  } //end-Person  const person1 = new Person("Ali", "Kaya", 25, "male");  const person2 = new Person("Ayse", "Kus", 15, "female");  console.log(`person1.fullname: ${person1.getFullName()}`);  console.log(`person2.fullname: ${person2.getFullName()}`);  console.log("\nHere are all members of person1:");  for (let x in person1){  console.log(`person1.${x}: ${person1[x]}`);  } //end-for |

It is also possible to implement inheritance in JS. To create a class inheritance, use the “**extends**” keyword. A class created with a class inheritance inherits all the methods from another class. Furthermore, super() method refers to the parent class. By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

|  |
| --- |
| class Shape {  constructor(name){this.name = name;}  printName(){console.log(`I am a ${this.name}`);}  } //end-Shape  // Here is a rectangle class that extends the Shape class  class Rectangle extends Shape {  constructor(w, h){  super("Rectangle");  this.width = w;  this.height = h;  }  draw(){  this.printName();  for (let i=0; i<this.height; i++){  let s = "";  for (let j=0; j<this.width; j++){  s += "\*";  } //end-for  console.log(s);  } //end-for  } //end-draw  } //end-Rectangle  // Here is a square class that extends the Shape class  class Square extends Shape {  constructor(w){  super("Square");  this.width = w;  }  draw(){  this.printName();  for (let i=0; i<this.width; i++){  let s = "";  for (let j=0; j<this.width; j++){  s += "\*";  } //end-for  console.log(s);  } //end-for  } //end-draw  } //end-Rectangle  const rect = new Rectangle(10, 5);  rect.draw();  console.log();  const square = new Square(5);  square.draw(); |

Finally, as with all modern OOPLs, it is possible to declare static class member variables and member methods.

|  |
| --- |
| class Foo {  static noObjects = 0; // static variables belong to the class    constructor(x=1){ // You can also have default values for variables  this.x = x; // member variables belong to the object  Foo.noObjects++; // static variables belong to the class  } //end-constructor  // static methods belong to the class  static printNoObjects(){  console.log(`No of Foo objects created so far are: ${Foo.noObjects}`);  }  // member methods belong to the object  printX(){console.log(`The value of x is: ${this.x}`);}  }  const foo1 = new Foo(5);  const foo2 = new Foo(7);  Foo.printNoObjects();  const foo3 = new Foo();  Foo.printNoObjects();  console.log("foo1.x:"); foo1.printX();  console.log("foo2.x:"); foo2.printX();  console.log("foo3.x:"); foo3.printX(); |

Look at 05-JS/08-OOP/oop2.js