Java Script

Content

- Variables
- Functions
- Objects
- HTML DOM

Javascript

- JavaScript is a prototype-based scripting language with dynamic typing and first-class function support. JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl, and Python. JavaScript is case-sensitive and white space-agnostic.

Variable Declaration

- If you declare a variable without assigning a value to it, its type is undefined by default.
- if you don't declare your variable with the var keyword, they become implicit globals.
- You should always declare a variable with the var keyword unless you know what you are doing

Variable Declaration

- var a;
- //declares a variable but its undefined
- var b = 0;
- console.log(b);
- //0
- console.log(a);
- //undefined
- console.log(a+b);
- //NaN

Datatypes

- Number
- String
- Boolean
- Symbol (new in ECMAScript 6)
- Object: Function, Array, Date, RegExp
- Null
- Undefined

Wrapper Classes

- Number.MAX_VALUE

NaN (Not a Number).

- In general, this occurs when conversion from another type (String, Boolean, and so on) fails.
- isNaN(123)
- //false
- isNaN(-1.23)
- //false
- isNaN(5-2)
- //false
- isNaN(0)
- //false
- isNaN('123')
- //false
- isNaN('Hello')
- //true
- isNaN('2005/12/12') //true
- isNaN('')
- //false
- isNaN(true)
- //false
- isNaN(undefined)
- //true
- isNaN('NaN')
- //true
- isNaN(NaN)
- //true
- isNaN(0 / 0)
- //true

String

- Strings are a sequence of Unicode characters (each character takes 16 bits).
- Each character in the string can be accessed by its index.
- The first character index is zero.
- Strings are enclosed inside " or ' —both are valid ways to represent strings.

Wrapper objects

- var s = new String("dummy"); //Creates a String object
- console.log(s); //"dummy"
- console.log(typeof s); //"object"
- var nonObject = "1" + "2";
- //Create a String primitive console.log(typeof nonObject); //"string"

- //Helper functions console.log("Hello".length); //5 console.log("Hello".charAt(0)); //"H" console.log("Hello".charAt(1)); //"e" console.log("Hello".indexOf("e")); //1 console.log("Hello".lastIndexOf("l")); //3 console.log("Hello".startsWith("H")); //true console.log("Hello".endsWith("o")); //true console.log("Hello".includes("X")); //false var splitStringByWords = "Hello World".split(" "); console.log(splitStringByWords); //["Hello", "World"] var splitStringByChars = "Hello World".split(""); console.log(splitStringByChars); //["H", "e", "l", "l", "o", " ", "W", "o", "r", "l", "d"] console.log("lowercasestring".toUpperCase()); //"LOWERCASESTRING" console.log("UPPPERCASESTRING".toLowerCase()); //"upppercasestring" console.log("There are no spaces in the end ".trim()); //"There are no spaces in the end"

Undefined

- Indicates an absence of meaningful values by two special values—null, when the non-value is deliberate, and undefined, when the value is not assigned to the variable yet.
- Example:
- var xl;
- console.log(typeof xl); undefined
- console.log(null==undefined);
- true

Booleans

- You can create primitive Booleans by assigning a true or false literal to a variable.
- Consider the following example:
  - var pBooleanTrue = true;
  - var pBooleanFalse = false
- Use the Boolean() function; this is an ordinary function that returns a primitive
  - Boolean: var fBooleanTrue = Boolean(true);
  - var fBooleanFalse = Boolean(false);

The instanceof operator

- One of the problems with using reference types to store values has been the use of the typeof operator, which returns object no matter what type of object is being referenced.
- The solution is instanceof operator.
  - var aStringObject = new String("string");
  - console.log(typeof aStringObject);
  - //"object"
  - console.log(aStringObject instanceof String);
  - //true
  - var aString = "This is a string";
  - console.log(aString instanceof String);
  - //false

The + operator

- var b="70";
- console.log(typeof b);
- //string
- b=+b;
- //converts string to number
- console.log(b);
- //70
- console.log(typeof b);
- //number

three logical Boolean operators

- Logical AND(&&): If the first operand object is falsy , it returns that object. If its truthy , the second operand object is returned: console.log (0 && "Foo");
- //First operand is falsy - return it
- console.log ("Foo" && "Bar");
- //First operand is truthy, return the second operand •
- Logical OR(||): If the first operand is truthy , it's returned. Otherwise, the second operand is returned
- The typical use of a logical OR is to assign a default value to a variable:

```
function greeting(name){
    name = name || "John";
    console.log("Hello " + name);
}
greeting("Johnson"); // alerts "Hi Johnson";
greeting(); //alerts "Hello John"
```

Logical operators

==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	true
		x !== 8	true

Equality

- A strict equality check is performed by === while a loose equality check is performed by ==.

Strict equality using ===

- Strict equality compares two values without any implicit type conversions.
- If the values are of a different type, they are unequal.
- For non-numerical values of the same type, they are equal if their values are the same.
- For primitive numbers, strict equality works for values.
- If the values are the same, === results in true .
- However, a NaN doesn't equal to any number and NaN===<a number> would be a false .

Strict Equality

- Condition "" === "0"
- 0 === ""
- 0 === "0"
- false === "false"
- false === "0"
- false === undefined
- false === null
- null === undefined
- Output false false false false false false false false

Comparing objects

- Condition {} === {};
- new String('bah') === 'bah’;
- new Number(1) === 1;
- var bar = {};
- bar === bar;
- Output false false false true

Variable and Value

- In JavaScript, values have types, variables don't. Due to the dynamic nature of the language, variables can hold any value at any time.
- JavaScript doesn't does not enforce types, which means that the language doesn't insist that a variable always hold values of the same initial type that it starts out with.
- A variable can hold a String, and in the next assignment, hold a number, and so on:

```
var a = 1; typeof a; // "number"
a = false; typeof a; // "boolean"
```

- The typeof operator always returns a String:

```
typeof typeof 1; // "string"
```

Semicolon-less

- JavaScript is not a semicolon-less language.
- A JavaScript language parser needs the semicolons in order to understand the source code.
- the JavaScript parser automatically inserts them whenever it encounters a parse error due to a missing semicolon. A
- Automatic semicolon insertion (ASI) will only take effect in the presence of a newline (also known as a line break).
- Semicolons are not inserted in the middle of a line.
- ASI dictates that a statement also ends in the following cases:
  - A line terminator (for example, a newline) is followed by an illegal token
  - A closing brace is encountered
  - The end of the file has been reached

The eval() method

- It takes a String containing JavaScript code, compiles it and runs it
- It is one of the most misused methods in JavaScript.
- There are a few situations where you will find yourself using eval() , for example, when you are building an expression based on the user input.
- It's slow, unwieldy, and tends to magnify the damage when you make a mistake.
- There is probably a better way.

```
console.log(typeof eval(new String("1+1"))); // "object"
console.log(eval(new String("1+1")));        //1+1
console.log(eval("1+1"));                    // 2
console.log(typeof eval("1+1"));             // returns "number"
var expression = new String("1+1");
console.log(eval(expression.toString()));    //2
```

strict mode

- Eliminates some JavaScript silent errors by changing them to throw errors.
- Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
- Prohibits some syntax likely to be defined in future versions of ECMAScript.

strict mode

- Typing the following line first in your JavaScript file or in your <script> element:
- 'use strict';
- After ECMAScript 5
- If you want to switch on the strict mode per function,

```
function foo() {     'use strict';
}
```

Simplifying variable uses

```
function strictFunc() {
  'use strict';
  strictVar = 123;
}
strictFunc();
```

Functions

- The most important fact about functions is that in JavaScript, functions are first-class objects.
- They are treated like any other JavaScript object. Just like other JavaScript data types,
  - they can be referenced by variables,
  - declared with literals,
  - even passed as function parameters.
  - returned as values from functions
  - possess properties that can be dynamically created and assigned

A function literal

- Functions are the pieces where you will wrap all your code, hence they will give your programs a structure. JavaScript functions are declared using a function literal.
- Function literals are composed of the following four parts:
  - The function keyword.
  - An optional name that, if specified, must be a valid JavaScript identifier.
  - A list of parameter names enclosed in parentheses. If there are no parameters to the function, you need to provide empty parentheses.
  - The body of the function as a series of JavaScript statements enclosed in braces.
- A function declaration The following is a very trivial example to demonstrate all the components of a function declaration:

```
 function myFunction(a, b) {
  return a * b;
}
```

function expressions

- Function Expressions
- A JavaScript function can also be defined using an expression.

- A function expression can be stored in a variable:

- Example

```
var x = function (a, b) {return a * b};
var z = x(4, 3);
```

function expressions

- One problem with this style of function declaration is that we cannot have recursive calls to this kind of function.
- You can use named function expressions to solve this limitation.

```
var facto = function factorial(n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
};
console.log(facto(3));  //prints 6
```

self-invoking function

```
function () {
  var x = "Hello!!";  // I will invoke myself
})();
```

Arrow Function

- var anon = function (a, b) { return a + b };
- In ES6 we have arrow functions with a more flexible syntax that has some bonus features and gotchas.

- // we could write the above example as:
- var anon = (a, b) => a + b;
- // or
- var anon = (a, b) => { return a + b };
- // if we only have one parameter we can loose the parentheses
- var anon = a => a;
- // and without parameters
- var () => {} // noop

- // this looks pretty nice when you change something like:
- [1,2,3,4].filter(function (value) {return value % 2 === 0});
- // to:
- [1,2,3,4].filter(value => value % 2 === 0);

https://www.vinta.com.br/blog/2015/javascript-lambda-and-arrow-functions/

Array

- Var arabalar=[‘Honda’,’Reno’,’Mercedes’];

a function as a parameter

- Once defined, a function can be called in other JavaScript functions.
- After the function body is executed, the caller code (that executed the function) continues to execute.
- You can also pass a function as a parameter to another function:

```
function changeCase(val) {
    return val.toUpperCase();
}
function demofunc(a, passfunction) {
        console.log(passfunction(a));
}
demofunc("smallcase", changeCase);
```

Scope

- variable's scope is the context in which the variable exists.
- The scope specifies from where you can access a variable and whether you have access to the variable in that context.
- Scopes can be globally or locally defined.

Global Scope

- Any variable that you declare is by default defined in global scope.
- This is one of the most annoying language design decisions taken in JavaScript.
- As a global variable is visible in all other scopes, a global variable can be modified by any scope.
- Global variables make it harder to run loosely coupled subprograms in the same program/module.
- If the subprograms happen to have global variables that share the same names, then they will interfere with each other and likely fail, usually in difficult-to-diagnose ways.
- This is sometimes known as namespace clash.

Global variable

- Create a global variable in two ways:
- The first way is to place a var statement outside any function. Essentially, any variable declared outside a function is defined in the global scope.
- The second way is to omit the var statement while declaring a variable (also called implied globals).

Global Scope Example

- var a = 1;
- function scopeTest() {
  - a = 2; //Overwrites global variable 2, you omit 'var'
- console.log(a);
- }
- console.log(a); //prints 1
- scopeTest(); //prints 2
- console.log(a); //prints 2 (global value is overwritten)

Local scope

- Unlike most programming languages, JavaScript does not have block-level scope (variables scoped to surrounding curly brackets); instead, JavaScript has function level scope.
- Variables declared within a function are local variables and are only accessible within that function or by functions inside that function.

Local Scope example

```
var scope_name = "Global";
function showScopeName () {
    // local variable; only accessible in this function   var
    scope_name = "Local";
     console.log (scope_name); // Local
}
 console.log (scope_name);     //prints - Global
showScopeName();              //prints – Local
```

Question ?

```
function foo() {
    function bar(a) {
        var i = 2; // changing the 'i' in the enclosing scope's for-loop
        console.log(a+i);
    }
    for (var i=0; i<10; i++) {
        bar(i);
    }
}
foo();
```

IIFE - Immediately Invoked Function Expression

- Several programmers omit the function name when they use IIFE.
- As the primary use of IIFE is to introduce function-level scope, naming the function is not really required.
- We can write the earlier example as follows:

```
var a = 1;
(function() {
   var a = 2;
    console.log( a ); // 2
})();
console.log( a ); // 1
```

Block scopes

- ECMAScript 6 (ES6) introduces the let keyword to introduce traditional block scope.
- This is so incredibly convenient that if you are sure your environment is going to support ES6, you should always use the let keyword.
- See the following code:

```
var foo = true;
if (foo) {
    let bar = 42;
    console.log( bar );
}
console.log( bar ); // ReferenceError
```

Hoisting

- Variable and function declarations are moved up to the top of the code during compilation phase—this is also popularly known as hoisting .
- It is very important to remember that only the declarations themselves are hoisted, while any assignments or other executable logic are left in place.
- The following snippet shows you how function declarations are hoisted:

```
foo();
function foo() {
    console.log(a); // undefined
    var a = 1;
}
```

Hoisting

- a = 1;
- var a;
- console.log(a);
- What should be the output of the preceding code? It is natural to expect undefined
- However, the output will be 1 .
- When you see var a = 1 , JavaScript splits it into two statements: var a and a = 1 .
- The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left in place for the execution phase.

Hoisting

- The declaration of the foo() function is hoisted such that we are able to execute the function before defining it.
- One important aspect of hoisting is that it works per scope.
- Within the foo() function, declaration of the a variable will be hoisted to the top of the foo() function, and not to the top of the program.

Hoisting

- Function declarations are hoisted but function expressions are not.

Hoisting

```
//Function expression
functionOne(); //Error //"TypeError: functionOne is not a function
var functionOne = function() {
    console.log("functionOne");
};
//Function declaration
functionTwo(); //No error //Prints - functionTwo function
functionTwo() {
     console.log("functionTwo");
}
```

Hoisting

- A function declaration is processed when execution enters the context in which it appears before any step-by-step code is executed.
- However, functionOne() is an anonymous function expression, evaluated when it's reached in the step-by-step execution of the code (also called runtime execution); we have to declare it before we can invoke it.
- Both function declarations and variable declarations are hoisted but functions are hoisted first, and then variables.

function declarations conditionally

- One thing to remember is that you should never use function declarations conditionally.
- This behavior is non-standardized and can behave differently across platforms.
- Conditional code is not guaranteed to work across all browsers and can result in unpredictable results: // Never do this - different browsers will behave differently

```
if (true) {
function sayMoo() {
 return 'trueMoo';
 }
}
else {
function sayMoo() {
return 'falseMoo';
}
} foo();
```

function declarations conditionally

- it's perfectly safe and, in fact, smart to do the same with function expressions:

```
var sayHello;
if (true) {   sayHello = function() {
                return 'trueHello';
  };
}
else {
   sayHello = function() {
   return 'false Hello';
};
}
foo();
```

The arguments parameter

- The arguments parameter is a collection of all the arguments passed to the function.
- The collection has a property named length that contains the count of arguments, and the individual argument values can be obtained using an array indexing notation.
- The arguments parameter is not a JavaScript array, and if you try to use array methods on arguments, you'll fail miserably.
- You can think of arguments as an array-like structure. This makes it possible to write functions that take an unspecified number of parameters.

The arguments parameter example

```
x = findMax(1, 123, 500, 115, 44, 88);
function findMax() {
    var i;
    var max = -Infinity;
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
      }
   }
    return max;
}
console.log(x);
```

this parameter

- Whenever a function is invoked, in addition to the parameters that represent the explicit arguments that were provided on the function call, an implicit parameter named this is also passed to the function.
- It refers to an object that's implicitly associated with the function invocation, termed as a function context .
- If you have coded in Java, the this keyword will be familiar to you; like Java, this points to an instance of the class in which the method is defined.

This parameter

- When a function is invoked with this pattern, this is bound to the global object.
- It seem a bad design choice.
- It is natural to assume that this would be bound to the parent context.
- When you are in a situation such as this, you can capture the value of this in another variable. We will focus on this pattern later.

Lab:

- Create a function that returns this
- Call and display the result
- Assing the function to a variable.
- Call function using varianle and display the result

This parameter

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>This test</title>
    <script type="text/javascript">
        function testF() {
            return this;
        }
        console.log(testF());
        var testFCopy = testF;
        console.log(testFCopy());
        var testObj = { testObjFunc: testF };
        console.log(testObj.testObjFunc());
    </script>
</head>
<body> </body>
</html>
```

This has different values depending on where it is used

- In a method, this refers to the owner 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(), and apply() can refer this to any object.

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

Anonymous functions while creating an object

- var santa = { say: function () {
-    console.log("hello");
-   }
- }
- santa.say();

this in a Function (Strict)

-
  ```
  "use strict";
  function myFunction() {
    return this;
  }
  ```

- JavaScript strict mode does not allow default binding.
- So, when used in a function, in strict mode, this is undefined.

this in Event Handlers

- In HTML event handlers, this refers to the HTML element that received the event:

- Example
- <button onclick="this.style.display='none'">
-   Click to Remove Me!
- </button>

this in an object

```
function bike() {
  console.log(this.name);
}

var name = "Ninja";
var obj1 = { name: "Pulsar", bike: bike };
var obj2 = { name: "Gixxer", bike: bike };

bike();            // "Ninja"
obj1.bike();       // "Pulsar"
obj2.bike();       // "Gixxer"
```

https://codeburst.io/all-about-this-and-new-keywords-in-javascript-38039f71780c

Precedence of “this” keyword bindings

- First it checks whether the function is called with new keyword.
- Second it checks whether the function is called with call or apply method means explicit binding.
- Third it checks if the function called via context object (implicit binding).
- Default global object (undefined in case of strict mode).

The new keyword and *this*

- The new keyword in front of any function turns the function call into constructor call and below things occurred when new keyword put in front of function
- A brand new empty object gets created
- new empty object gets linked to prototype property of that function
- same new empty object gets bound as this keyword for execution context of that function call
- if that function does not return anything then it implicit returns this object.

The new keyword and *this*

```
function bike() {
  var name = "Ninja";
  this.maker = "Kawasaki";
  console.log(this.name + " " + maker);
}

var name = "Pulsar";
var maker = "Bajaj";

obj = new bike();
console.log(obj.maker);
```

Closures

- Implify complex operations
- In a nutshell, closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to this function.
- In other words, closures allow a function to access all the variables, as well as other functions, that are in scope when the function itself is declared.
- It makes it possible for a function to have "**private**" variables.

Closures Example

```
var outer = 'I am outer'; //Define a value in global scope
function outerFn() {
//Declare a a function in global scope   console.log(outer);
}
outerFn(); //prints - I am outer
```

Closure Example

```
var outer = 'Outer'; //Variable declared in global scope
var copy;
function outerFn() { //Function declared in global scope
    var inner = 'Inner'; //Variable has function scope only, can not be accessed from outside
    function innerFn() {
        //Inner function within Outer function,  //both global context and outer
        //context are available hence can access  //'outer' and 'inner'
        console.log(outer);
        console.log(inner);
    }
copy = innerFn;
//Store reference to inner function, because 'copy' itself is declared
//in global context, it will be available outside also
}
outerFn();
copy(); //Cant invoke innerFn() directly but can invoke via a variable declared in global scope
```

□

Closure Example

- What phenomenon allows the inner variable to still be available when we execute the inner function, long after the scope in which it was created has gone away?
- When we declared innerFn() in outerFn() , not only was the function declaration defined,
- But a closure was also created that encompasses not only the function declaration,
- But also all the variables that are in scope at the point of the declaration.

A closure can be used as a static content

```
// Initiate counter
var counter = 0;

// Function to increment counter
function add() {
  var counter = 0;
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 0
```

A closure can be used as a static content

```
var add = (function () {
  var counter = 0;
  return function () {
      counter += 1;
      return counter
 }
})();

add();
add();
add();
```

A closure can be used as a static content

- The variable add is assigned the return value of a self-invoking function.
- The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.
- This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.
- This is called a JavaScript closure. It makes it possible for a function to have "private" variables.
- The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

Another Example Usage

```
var data = [
  { 'id': 'name', 'value': 'Steve' },
  { 'id': 'greeting', 'value': 'Hello!' }
];
for (var i = 0; i < data.length; i++) {
  $('#'+data[i].id).on('click', function(e) {
    alert(data[i].value);
  });
}
```

Another Example Usage

- this code does not work.
- Clicking on either of the buttons and triggering either of the two click-handlers results in undefined being the thing that is alerted.
- This is because i gets overwritten in the process of going through the for loop.
- By the time the event-handler is called, i has been set to 2 (the condition that caused it to exit the for loop), and there is no record in the data array at index 2.

Another Example Usage

```
var data = [
  { 'id': 'name', 'value': 'Steve' },
  { 'id': 'greeting', 'value': 'Hello!' }
];
function wrapIt(value) {
  return function() {
    alert(value);
  }
}
for (var i = 0; i < data.length; i++) {
  $('#'+data[i].id).on('click', wrapIt(data[i].value));
}
```

Another Example

- var call_back_array = []
- for (var i = 0; i <= 2; i++) {
-     call_back_array[i] = function () { return i * 2 }
- }
-
- call_back_array[0]()
- <output nedir?>
-
- call_back_array[1]()
- <output nedir?>
-
- call_back_array[2]()
- <output nedir?>

Another Example

```
<body>
   <p id="help">Helpful notes will appear here</p>
   <p>E-mail: <input type="text" id="email" name="email"></p>
   <p>Name: <input type="text" id="name" name="name"></p>
   <p>Age: <input type="text" id="age" name="age"></p>
</body>

<script>
   function showHelp(help) {  document.getElementById('help').innerHTML = help;   }
   function makeHelpCallback(help) {
      return function () { showHelp(help);    };
   }
   function setupHelp() {
      var helpText = [
         { 'id': 'email', 'help': 'Your e-mail address' },
         { 'id': 'name', 'help': 'Your full name' },
         { 'id': 'age', 'help': 'Your age (you must be over 16)' }
      ];

      for (var i = 0; i < helpText.length; i++) {
         var item = helpText[i];
         document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
      }
   }
   setupHelp();
</script>
```

Practical closures

- Closures are useful because they let you associate some data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow us to associate some data (the object's properties) with one or more methods.
- Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Function factories

- One powerful use of closures is to use the outer function as a factory for creating functions that are somehow related.

```
function dwightJob(title) {
    return function(prefix) {
        return prefix + ' ' + title;
    };
}
```

- var sales = dwightJob('Salesman');
- var manager = dwightJob('Manager');

- alert(sales('Top')); // Top Salesman
- alert(manager('Assistant to the Regional')); // Assistant to the Regional Manager
- alert(manager('Regional')); // Regional Manager

Modules

- Used to mimic classes and focus on public and private access to variables and functions.
- Modules help in reducing the global scope pollution. Effective use of modules can reduce name collisions across a large code base.
- A typical format that this pattern takes is as follows:
- Var moduleName=function() {
-   //private state
-   //private functions
-     return {   //public state   //public variables   }
- }
- There are two requirements to implement this pattern in the preceding format:
  - There must be an outer enclosing function that needs to be executed at least once.
  - This enclosing function must return at least one inner function. This is necessary to create a closure over the private state—without this, you can't access the private state at all.

Module Example

```
var superModule = (function (){
    var secret = 'supersecretkey';
    var passcode = 'nuke';
    function getSecret() {
      console.log( secret );
    }
    function getPassCode() {
      console.log( passcode );
    }
    return {     getSecret: getSecret,
                   getPassCode: getPassCode   };
  })();
superModule.getSecret();
superModule.getPassCode();
```

Another Library Approach

- İmport and export keywords

The export statement

- is used when creating JavaScript modules to export functions, objects, or primitive values from the module so they can be used by other programs with the import statement.

- Exported modules are in strict mode whether you declare them as such or not. The export statement cannot be used in embedded scripts.

Export syntax

- / Exporting individual features
- export let name1, name2, …, nameN; // also var, const
- export let name1 = …, name2 = …, …, nameN; // also var, const
- export function functionName(){...}
- export class ClassName {...}

- // Export list
- export { name1, name2, …, nameN };

- // Renaming exports
- export { variable1 as name1, variable2 as name2, …, nameN };

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

Import

- The static import statement is used to import bindings which are exported by another module. Imported modules are in strict mode whether you declare them as such or not. The import statement cannot be used in embedded scripts unless such script has a type="module".

- There is also a function-like dynamic import(), which does not require scripts of type="module".

Import Syntax

- import defaultExport from "module-name";
- import * as name from "module-name";
- import { export1 } from "module-name";
- import { export1 as alias1 } from "module-name";
- import { export1 , export2 } from "module-name";
- import { foo , bar } from "module-name/path/to/specific/un-exported/file";
- import { export1 , export2 as alias2 , [...] } from "module-name";
- import defaultExport, { export1 [, [...]] } from "module-name";
- import defaultExport, * as name from "module-name";
- import "module-name";
- var promise = import("module-name");

Export inport not ES6 module syntax

- NodeJS uses CommonJS Module syntax (module.exports) not ES6 module syntax (export keyword).
- Use babel npm package to transpile your ES6 to a commonjs target

Kaynaklar

- JavaScript Primer