The console is part of the web browser and allows us to log messages, run javaScript code, and see errors and warnings.

Function used generate output to the console is :

Console.log(“coding is an art!”);

When working with numbers, quotes are not needed.

Console.log(42);

JavaScript in HTML:

we can add JavaScript code in HTML document <Script >tag .

<<body>

< <script>>

Console.log((“JavaScript in HTML”);;

</ <script>

<body>>

AlertBox:

Another way to display message is an alert box.

Alert(“I am alert box”);

Comments are explanatory statements that you can include in a program to benefit the person reading you code.

Single line comments starts with : //

Multi line comments: starts with /\* ends with \*/

Multiplication and addition have higher precedence than addition and substraction.

**Ex**

//2\*3 will be calculated first

console.log(5+2\*3);

We can control the precedence using Parentheses.

**Ex**

//5+2 will be calculated first

console.log((5+2)\*3);

Variables:

Variables are the containers for storing values. Variables are created using “let” keyword

Ex:

let name;

name=”Purushotham”;//giving value to a variable is called initialization.

In older versions of JavaScript, the “var “keyword was used instead of =let,

**Ex**

var name = 'James';

“let “ is the modern way of declaring variables and has number of advantages over “var”

Ex:

Var allows us to re-declare variables with the same name, while let results in an error, which makes the code safer.

Constants:

We can create constant variable using “const” keyword:

**Ex:**

const color = 'red';

Must have a value when declared and cannot change.

RULES FOR NAMING VARIABLES:

Variable names must begin with a letter, an underscore \_ or a dollar sign.

Variable names cannot contain spaces.

Variable names can only contain letters, underscore and dollar signs.

Variable names are case sensitive, which means that(” Name and name variables are different”).

DataTypes:

The term datatype refers to the type of values a program can work with.

we are already familiar with two types of data text and numbers:

Everything in quotes is a string.

let message=”Hello!”;

Numbers can be written with or without decimals.

let x=5;

let y=8.45;

BOOLEAN:  
JavaScript has the Boolean datatype which can only take the values true or false.

There are also numerical representation of Boolean values known that 1 means true, and 0 is false.

Operations:

**wo asterisks**\*\* are used for exponentiation.

**modulo operator**% is used to calculate the remainder of a division.

we can use ++/--**operators**to increment/decrement the value of a variable.

increment and decrement operators can be used **only with variables**

ASSIGNMENT VARIABLES:

Assignment variables assign values to variables.

We can create different variables on the same line, separating them with commas:

let x=5,y=6,z=8;

Additional Assignment operator:

let score = 100;

score+=10; //this is called additional assignment operator.

let x = 15;

x+=5; // x = x+5;

x-=5; // x = x-5;

x\*=5; // x = x\*5;

x/=5; // x = x/5;

x%=5; // x = x%5;

Strings:

 we can create a string by entering text between **two single or double quotation marks**.

EscapeCharacter:

In JS the backslash’\’ is a special character, called the escape character.

It is used to represent certain things in a string, such as new line, tabs, and other things.

**Ex**

console.log('I'm happy');//error

console.log('I\'m happy');

console.log("She said \"Yes!\"");

To create a new line we use \n.

**Ex**

console.log("One \nTwo \nThree");

TEMPLATE LITERALS:  
Template literals are another way to create strings and work with them more flexibly.

They use back-ticks(` `) rather than (“”) to define a string.

**Ex**

let temp = `Hello, user!`;

console.log(temp);

template literals allow **multiline strings**, without using \n

Template literals allow you to use variables inside the strings. You need to add a **dollar sign** **$** and enclose the variable name in **braces**{}.

**For example:**

**Ex**

let name = "John";

let text = `Welcome, ${name}`;

console.log(text);

Concatenation:

we can join two or more strings into another using the ‘+’ operator, that is concatenation.

**EX**

console.log("Java" + 'Script');

we can perform concatenation with variables as well:

let x = "Java";

let y = "Script";

let z = x +y;//"Java" + "Script"

console.log(z);

DecissionMaking:

COMPARISION OPERATORS:

Greater than(>) is a comparison operator, which compares the values and results in corresponding Boolean values.

let score = 85;

console.log(score>99);

JavaScript has a number of comparison operators:

* equal to ==
* greater than >
* not equal to !=
* greater than >
* smaller than <
* greater or equal to >=
* smaller or equal to <=

STRICT EQUALITY (===) comparison operator returns false for the values which are not of a similar type.

**Ex**

let x = 5;

let y = '5';

console.log(x == y); //true

console.log(x === y); //false

In the above example x is a number and y is a string that’s why == returns true(which compares values) and ===(strict equality) returns false(which consider the type of the value also)

BOOLEANS:  
greater than and smaller than operators can also be used to compare strings lexicographically(where the alphabetical order of word is based on the alphabetical order of their component letters).

Console.log(‘a’ < ’b’); // true

Console.log(“Bob” > “Dave”); //false

IF STATEMENTS:

we can let our program make decisions using **if statements**

the code inside of the if statement will be performed only if the condition is evaluated as true

if statements can be nested, one inside the other

ELSE and ELSE IF:

the **else** statement allows you to control the cases when the condition in the if statement is false

we can use **else if** statements to handle multiple conditions.

**Ex**

let stopLight = 'green';

if (stopLight == 'red') {

console.log('Stop!');

} else if (stopLight == 'yellow') {

console.log('Slow down.');

} else if (stopLight == 'green') {

console.log('Go!');

} else {

console.log('Unknown!');

}

SWITCH STATEMENT:  
The switch expression is evaluated once. The value of the expression is compared with the values of each case, and if there’s match, that block of code is executed.

We can achieve same result using if else statement. But switch statement is more effective and makes the code more readable.

Break statement stops the execution of other cases when there is a match.

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

let choice = 2;

switch(choice){

case 1:

console.log("Sports");

break;

case 2://match

console.log("Business");

break;//stop

case 3:

console.log("Technology");

break;

default:

console.log("Color not found.");

}

Ternary operator:

It’s an alternative to if else statement. It makes the code more shorter.

Conditional operator or ternary operator assign a value to a variable, based on some condition.

Example:

Variable = (condition)? Value1: value2

let age = 42;

let isAdult = (age < 18) ? "Too young": "Old enough";

console.log(isAdult);

Loops:

**loops** allow you to run the same code multiple times.

the for**loop** has 3 components: the initializer, the condition, and the final expression.

Both, **for** and **while** can be used to accomplish the same task.

The for loop is usually used when the number of times we need to run the loop is known.

The while loop is used when the condition of the loop is more complex and the number of times it runs is based on the expression in the loop.

the **do...while** loop executes the code block once, before checking if the condition is true, and then it repeats the loop as long as the condition is true

THE BREAK STATEMENT:  
Break statement allows us to exit a loop prematurely, based on the given condition.

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

if(i==3) {

break;

}

console.log(i)

Continue:

The continue statement is used to skip an iteration of the loop and continue from the next one.

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

if(i == 5) {

continue;

}

console.log(i);

}

Function:

A function is a block of code designed to perform a particular task. The purpose of a function is to create it once and call it multiple times.

To define a function use the function keyword, followed by a name, followed by a set of parentheses()

The code to be executed by the function placed inside the curly braces {}.

After creating a function, we can call it in our program as many times as we want.

To call a method use it’s name followed by parentheses.

Call the function:

login();

Functions can have parameters. The parameters are defined in the parentheses and can be used like variables in function.

RETURNING FROM FUNCTION:  
In some cases , we do not need to output the result but need to assign it to a variable, to work with it in our program.

Return statement ends the function execution, which means that everything after it inside the function will be ignored.

In these cases we need to return result values.

function add(x,y){

return x+y;

}

let result = add(5,6);

console.log(result);

JAVASCRIPT OBJECTS:

JavaScript variables are containers for data values. Objects are variables too , but they contain many values.

An object is a list of values that are written as “name:value” pairs, with names and values separated by colons.

Example:

Var person={

Name: “Purushotham”, age:31,

favColor:”green”, height:183

};

We can access object properties in two ways.

ObjectName.propertyName

ObjectName[‘propertyName’]

**Example:**

var person = {

name: "John", age: 31,

favColor: "green", height: 183

};

var x = person.age;

var y = person['age'];

JavaScript’s built-in “length” property is used to count the number of characters in a property or string.

**Ex**

var course = {name: "JS", lessons: 41};

document.write(course.name.length);

OBJECT METHODS:

An object method is a property that contains a function definition;

objectNAme.methodName( )

As we know that “document.write( )” outputs the data.

A” write( )” function is already a method of the document object.

**Ex**

document.write("This is some text");

The Object Constructor:

Till now we created an object using object literal(or initializer) syntax. That allows us to create a single object. But sometimes we need to set an object-type that can be used to create a number of objects of a single type.

The standard way to create an object-type is to use an object constructor function.

Example:

function person(name, age, color) {

this.name = name;

this.age = age;

this.favcolor = color;

}

The above function(person ) is an object constructor, which takes parameters and assigns them to the object properties.

CREATING OBJECXTS:  
 once we have an object constructor , we can use the new keyword to create new objects of the same type.

**Example**

function person (name, age) {

this.name = name;

this.age = age;

}

var John = new person("John", 25);

var James = new person("James", 21);

METHODS:

Methods are functions that are stored as object properties.

Syntax: To create an object method.

methodName = function( ) { code lines }

Access an object using the following syntax.

objectName.methodName( )

A method is a function, belonging to an object. It can be referenced using this keyword.

**Ex**

function person(name, age) {

this.name = name;

this.age = age;

this.changeName = function (name) {

this.name = name;

}

}

var p = new person("David", 21);

p.changeName("John");

//Now p.name equals "John"

In the above example we defined a method named changeName for our Person, which is a function that takes parameter name and assigns it to name property of object.

Arrays:

Arrays can store multiple values in a single variable.

EX:

Var courses = new Array(“HTML”, “CSS”, “JS”);

We can access array element by referring to the index number .

EX:

Var course=courses [0];

Courses[1] = “c++”; //changes the second element.

If we try to access an index outside of the array, returns the value “undefined”.

CREATING ARRAYS:

We can also create an array, tell it the number of elements it will store and add elements later.

**EX**

var courses = new Array(3);

courses[0] = "HTML";

courses[1] = "CSS";

courses[2] = "JS";

In javascript arrays are dynamic we can declare an Array( )constructor without any argument.We can then add elements dynamically.

Example:

var courses = new Array();

courses[0] = "HTML";

courses[1] = "CSS";

courses[2] = "JS";

courses[3] = "C++";

ARRAY LITERAL:

For greater simplicity, readability, and execution speed, we can also declare arrays using the array literal syntax.

EX:

Var courses = [“HTML”, “CSS”, “JS”];

JS Arrays have useful built in methods and properties.

Length property returns the number of its elements.

**Ex**

var courses = ["HTML", "CSS", "JS"];

document.write(courses.length);

COMBINING ARRAYS:

JS concat ( ) method allows us to join arrays and create an entirely new array.

**Ex**

var c1 = ["HTML", "CSS"];

var c2 = ["JS", "C++"];

var courses = c1.concat(c2);

ASSOCIATIVE ARRAYS:  
while many programming supports arrys with named indexes(text instead of numbers), called associative arrays, JS does not. However, you still can use the named array syntax, which will produce an object.

**Ex**

var person = []; //empty array

person["name"] = "John";

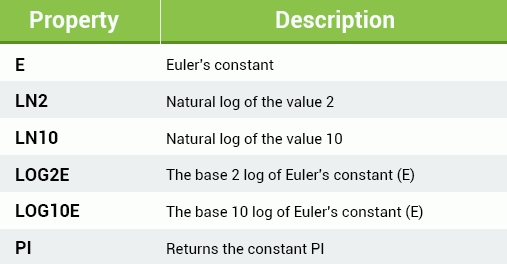
person["age"] = 46;

document.write(person["age"]);

The person is treated as object , instead of being an array. And named indexes name and age become properties of person object.

MATH OBJECT:

The math object allows us to perform mathematical tasks .

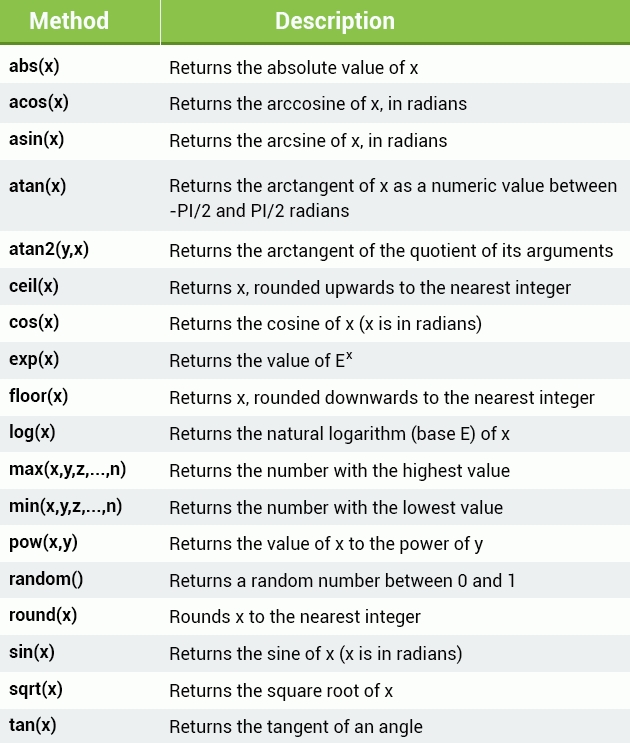


MATH OBJECT METHODS:

Math object contains number of methods that are used for calculations.Th

var number = Math.sqrt(4);

document.write(number);



SET INTERVAL:

The setInterval ( ) method calls a function or evaluates an expression at specified intervals.

It will continue calling the function until clearInterval ( ) is called .

function myAlert() {

alert("Hi");

}

setInterval(myAlert, 3000);

This will call myAlert function every 3 seconds.

THE DATE OBJECT:

The Date object enables us to work with dates.

Using “newDate ( )” create a new date object with the current date and time.

Ex:

var d= newDate( );

new Date(milliseconds)

new Date(dateString)

new Date(year, month, day, hours, minutes, seconds, milliseconds)



**Example**

function printTime() {

var d = new Date();

var hours = d.getHours();

var mins = d.getMinutes();

var secs = d.getSeconds();

document.body.innerHTML = hours+":"+mins+":"+secs;

}

setInterval(printTime, 1000);

Above example prints current time date object, and prints it to the screen.

“setInterval()” method calls the function every one second.

The” innerHTML “property sets or returns the HTML content of an element.

DOM:( document object model)

Dom stands for Document object model. It is a programming interface that allows us to create, change, or remove elements from the document. We can also add events to this elements to make our page more dynamic.

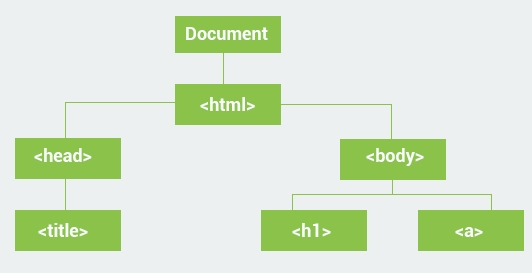
With DOM, we can easily access and manipulate tags, IDs, classes, Attributes, or elements of HTML using commands or methods provided by the document object.

The DOM of an HTML document can be represented as a nested set of boxes:

The DOM represents a document as astructure.

HTML elements become interrelated nodes in the tree.

Nodes can have child nodes. Nodes on the same tree level are called siblings.



<html> has two children (<head>, <body>);

<head> has 1 child (<title>) and 1 parent(<html>);

THE DOCUMENT OBJECT:

There is a predefined object in javascript, which can be used to access all elements on the Dom. In other words the “document” object is the owner ( root) of all objects on our web page.

So, if we want to access objects in an HTML page, we always starts with accessing the document object.

EXAMPLE:

document . body . innerHTML =”some text”;

As the body is an element of the DOM, we can access it using the “document” object and change the content of innerHTML property.

The innerHTML property can be used on almost all HTML elements to change its content.

Selecting Elements:

All HTML elements are objects. And as we know every object has properties and methods.

The “document” object has methods that allows us to select the desired HTML element.

Most commonly used 3 methods.

document . getElementById(id) // finds element by id

document . getElementByClassName(name) // finds elements by class name

document . getElementByTagName(name) // find element by tag name.

EXAMPLE:

Var elem = document.getElementById(“demo);

elem.innerHTML = “HelloWorld!”;

The above method selects the element with id=”demo” and changes its content to “Hello world!”.

The getElementsByClassName() method returns a collection of all elements in the document with specified classname.

Forexample, if our HTML page contained 3 elements with class =” demo”, The code would return all those elements as an array.

Ex:

Var arr =document . getElementByClassName(“dedmo”);

Arr[1].innerHTML =”Hi”; // accessing the second element.

Similarly, the getElementByTagName method returns all of the elements of the specified tag name as an array.

<html>

<body>

<div id ="demo">

<p>some text</p>

<p>some other text</p>

</div>

<script>

var a = document.getElementById("demo");

var arr = a.childNodes;

for(var x=0;x<arr.length;x++) {

arr[x].innerHTML = "new text";

}

</script>

</body>

</html>

Changing Attribute:

Once we have selected the elements we want work with we can change their attributes.

For example, we can change the src attribute of an image.

<<img id="myimg" src="orange.png" alt="" /> /<,

<script>>

var el =document.getElementById ("myimg");

el.src =”apple.png”;

</script>

**Ex**

<a href="http://www.example.com">Some link</a>

<script>

var el = document.getElementsByTagName("a");

el[0].href = "http://www.sololearn.com";

</script>

Changing Style:

The style of Html elements can also be changed using javascript.

<div id="demo" style="width:200px">some text</div>

<script>

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

x.style.color = "6600FF";

x.style.width = "100px";</script>

Note:

We cannot use dashes(-) in the property names these are replaced with camelCase versions, (Where the compound word begin with a capital letter).

Ex:

“background-color” property should be referred as “backgroundColor”.//( c is capital letter in color)

Creating and Removing Elements:

We can use the following methode to create new nodes:

**element.cloneNode( ) //** clones an element and returns the resulting node.

**document.createElement(element) //** creates a new element node.

**document.crerateTextNode(text) //** creates a new text node

**element.appendChild(newNode) // adds new child node to an element as the last child node.**

**element.insertBefore(node1, node2) // inserts node1 as a child before node2.**

<div id ="demo">some content</div>

<script>

//creating a new paragraph

var p = document.createElement("p");

var node = document.createTextNode("Some new text");

//adding the text to the paragraph

p.appendChild(node);

var div = document.getElementById("demo");

//adding the paragraph to the div

div.appendChild(p);

</script>

The above example creates a new paragraph and add it to the existing div element on the page.

Removing Elements

To remove an HTML element, you must select the parent of the element and use the “removeChild(node )” method.

<div id="demo">

<p id="p1">This is a paragraph.</p>

<p id="p2">This is another paragraph.</p>

</div>

<script>

var parent = document.getElementById("demo");

var child = document.getElementById("p1");

parent.removeChild(child);

</script>

This removes the paragraph with id =”p1” from the page.

An alternative way of achieving the same result would be the use of the” parentNode “ property

Ex:

Var child = document.getElementById(“p1”);

child.parentNode.removeChild(child);

Replacing Elements:

To replace an HTML element, the element.replaceChild(newNode, oldNode) method is used.

<div id="demo">

<p id="p1">This is a paragraph.</p>

<p id="p2">This is another paragraph.</p>

</div>

<script>

var p = document.createElement("p");

var node = document.createTextNode("This is new");

p.appendChild(node);

var parent = document.getElementById("demo");

var child = document.getElementById("p1");

parent.replaceChild(p, child);

</script>

The code above creates a new paragraph element that replaces the existing p1 paragraph.

Animations:

We already know how to select and change DOM elements, we can create a simple animation.

To create an aniamation, we need to change the properties of an element at small interval of time. We can achieve this by using setInterval( ) method. Which allows us to create a timer and call a function to change properties repeatedly at defined intervals.

Ex:

Vat t= setInterval(move, 500);

This code creates a timer that calls a move( ) function every 500 milliseconds.

Now we need to create a move ( ) function, that changes the position of the box.

Ex:

Var pos =0; //Starting position

Var box= document.getElementById(“box”); // our box element

Function move(){

Pos +=1;

box.style.left = pos+ ”px”; // px=pixels

}

The move function increments the left property of the box element by one each time when it is called.

<body>

    <div id ="container">

        <div id="box">

        </div>

     </div>

      <script src="second.js"></script>

  </body>

var pos = 0;

//our box element

var box = document.getElementById("box");

var t = setInterval(move, 10);

function move() {

if(pos >= 150) {

clearInterval(t);

}

else {

pos += 1;

box.style.left = pos+"px";

}

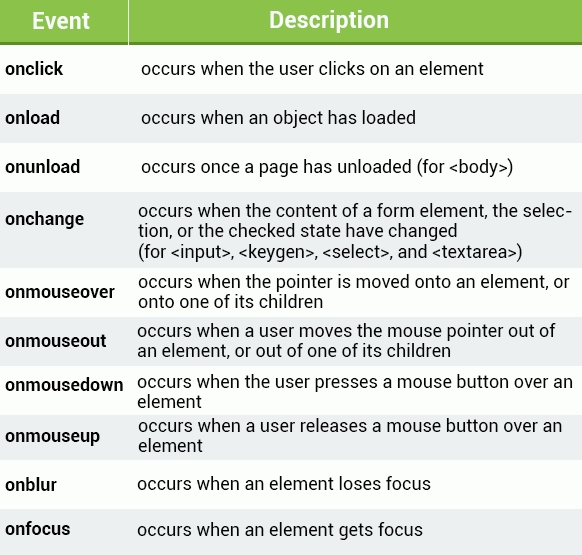
}

Handling Events:

EVENTS:

we can write JavaScript code that executes when an event occurs, such as when a user clicks an HTML element, moves the cursor, or submits a form. When an event occurs on a target element, a handler function is executed.

Common HTML EVENTS:



<button [b]onclick="show()">Click Me</button>

<script>

function show() {

alert("Hi there");

}

</script>

**Ex**

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

x.onclick = function () {

document.body.innerHTML = Date();

}

ONLOAD AND UNONLOAD:

The “onload” and “onunload” events are triggered when the user enters or leaves the page.

EX:

<body onload=”doSomething( )”>

Similarly, the window.onload event can be used to run code after the whole page is loaded.

Ex:

Window.onload=function(){

}

The “onchange” event is mostly used on textboxes. The event handler gets called when the text inside the textbox changes and focus is lost from the element.

The below example changes the text inside text box to upper case.

<input type="text" id="name" onchange="change()">

<script>

function change() {

var x = document.getElementById("name");

x.value= x.value.toUpperCase();

}

</script>

EVENT LISTENERS:

The “addEventListener( )” method attaches an event handler to an element without overwriting existing event handler. We can add many event handlers to one element.

We can also add many event handlers of the same type to one element, i.e., two “CLICK” events.

Example:

element.addEventListener(event, function, useCapture);py

The first parameter is the event type(like “click” or “mousedown”).

The second parameter is the function we want to call when the event occurs.

The third parameter is aboolean value specifying whether to use bubbling or event capturing. This is optional.(we can learn about bubbling and capturing in next page)

**NOTE:**

**We don’t use the “on “ prefix for this event use “click” instead of “onclick”.**

***Example:***

element.addEventListener("click", myFunction);

element.addEventListener("mouseover", myFunction);

function myFunction() {

alert("Hello World!");

}

We can remove one of the listener: below example removes itself after being executed:

**Ex**

<button id="demo">Start</button>

<script>

var btn = document.getElementById("demo");

btn.addEventListener("click", myFunction);

function myFunction() {

alert(Math.random());

btn.removeEventListener("click", myFunction);

}

</script>

**Event propagation:**

Event propagation allows for the definition of the element order when an event occurs.

If we have a<p> element inside a <div> element, and the user clicks on the <p> element, which element’s “click “ event should be handled first?

In “**bubbling”** the inner most element’s event is handled first and outer element’s event is handled. The <p> elements event is handled first , followed by <div> element’s click event.

In “**capturing”** the outermost element’s event is handled first and then the inner. The <div> element’s click event is handled first, followed by<p> element’s click event.

**Note:**

Capturing goes down the down the DOM. And Bubbling goes up the DOM>

**Capturing vs Bubbling:**

The ” addEventListener( ) “ allows us to specify the propagation type with the “useCapture()” parameter.

addEventListener(event, function, useCapture)

The default value is false, which means the bubbling propagation is used. When the value is set to true, the event uses the capturing propagation.

//Capturing propagation

elem1.addEventListener("click", myFunction, true);

//Bubbling propagation

elem2.addEventListener("click", myFunction, false);

**Creating Image Slider:**

Now we can create a sample image slider project. The images will be changed using “Next” and “Prev” buttons.

Example:

Create a HTML, which includes an image and two navigation buttons.

**HTML:**

<div>

<button onclick="prev()"> Prev </button>

<img id="slider"

src="http://www.sololearn.com/uploads/slider/1.jpg"

width="200px" height="100px"/>

<button onclick="next()"> Next </button>

</div>

**JavaScript:**

var images = [

"http://www.sololearn.com/uploads/slider/1.jpg",

"http://www.sololearn.com/uploads/slider/2.jpg",

"http://www.sololearn.com/uploads/slider/3.jpg"

];

var num = 0;

function next() {

var slider = document.getElementById("slider");

num++;

if(num >= images.length) {

num = 0;

}

slider.src = images[num];

}

function prev() {

var slider = document.getElementById("slider");

num--;

if(num < 0) {

num = images.length-1;

}

slider.src = images[num];

}

**Form Validation:**

HTML% adds some attributes that allow form validation. For example. The “required “ attribute can be added to an input to make it mandatory to fill in.

The form element has an “onsuubmit” event that can be handled to perform validation.

EXAMPLE:

The text in both the fields should be the same and not blank not pass the validation.<

<form onsubmit="return validate()" method="post">

Number: <input type="text" name="num1" id="num1" />

<br />

Repeat: <input type="text" name="num2" id="num2" />

<br />

<input type="submit" value="Submit" />

</form>

Now we need to define the “Validate( )” function.

function validate() {

var n1 = document.getElementById("num1");

var n2 = document.getElementById("num2");

if(n1.value != "" && n2.value != "") {

if(n1.value == n2.value) {

return true;

}

}

Alert(“The values should be equal and not blank”);

return false;

**ECMAScript:**

**European computer manufacturer’s association .**

ECMA is a standard for scripting languages such as java script, Jscript, etc., specification.

JavaScript is considered as one of the most popular implementations of ECMAscript.

The sixth edition, initially known as ECMA Script 6 (ES 6) and later renamed to ECMAScript 2015, adds significant new syntax for writing complex applications, function, binary data, typed arrays, collections(maps, sets, and weak maps), promises, number and math enhancements, reflection, and proxies.

In other words, ES6 is a superset of javascript(ES5). The reason that ES6 became so popular is that it introduced new conventions and OOP concepts such as classes.

Var and let:

In ES6 we have three ways of declaring variables:

var a = 10;

const b = 'hello';

let c = true;

The type of declaration used depends on the necessary scope. Scope is the fundamental concept in all programming languages that defines the visibility of a variable.

VAR VS LET:  
Unlike the var keyword, which defines a variable globally, or locally to an entire function regardless of block scope,

Let allows us to declare variable that are limited in scope to the block, statement, or expression in which they are used.

**Ex**

if (true) {

let name = 'Jack';

}

alert(name); //generates an error

In this case, the name variable is accessible only in the the scope of the if statement because it was declared as” let”.

To demonstrate diff between var and let

function varTest() {

var x = 1;

if (true) {

var x = 2; // same variable

console.log(x); // 2

}

console.log(x); // 2

}

function letTest() {

let x = 1;

if (true) {

let x = 2; // different variable

console.log(x); // 2

}

console.log(x); // 1

}

CONST:

Const variable have the same scope as variable declared using “let”. The difference is that const variable are immutable – they are not allowed to be reassigned .

**Ex**

const a = 'Hello';

a = 'Bye';//generate an error

Template Literals in ES6:

Template literals are a way to output variables in the string.

Prior to ES6 we had to break the string, forexample:

**Ex**

let name = 'David';

let msg = 'Welcome ' + name + '!';

console.log(msg);

ES6 introduced a new way of outputting variable value in strings. The same code above can be written as :

**EX**

let name = 'David';

let msg = `Welcome ${name}!`;

console.log(msg);

NOTE:  
The template literals are enclosed by the backtick(` `) character instead of double or single quotes.

The ${expression} is a placeholder, and can include any expression, which will get avaluated and inserted into the template literal.

Loops In ECMAScript:

In js we commonly use forloop to iterate over values in a list:

let arr = [1, 2, 3];

for (let k = 0; k < arr.length; k++) {

console.log(arr[k]);

}

The” for …..in “ loop is intended for iterating over the enumerated keys of an object .

let obj = {a: 1, b: 2, c: 3};

for (let v in obj) {

console.log(v);

}

//output:

a

b

c

ES6 introduces the new “for…..of” loop, which creates a loop iterating over iterable objects.

During each iteration the val variable is assigned the corresponding element in the list.

The “for…..of” loop works for their iterable objects as well, including strings:

**Ex**

for (let ch of "Hello") {

console.log(ch);

}

//Output:

h

E

L

o

Functions in ECMA Script:

Prior to ES6, a JavaScript function was defined like this:

function add(x, y) {

var sum = x+y;

console.log(sum);

}

ES 6 introduces a new syntax for writing functions. The same function can be written as :

const add = (x, y) => {

let sum = x + y;

console.log(sum);

}

Here, we can skip typing function and return, as well as some parentheses and braces.

const x = () => alert("Hi");

The syntax is very useful for inline functions. For example , let’s say we have an array, and for each element of the array we need to execute a function. We use the forEach method of the array to call a function for each element:

**Ex**

var arr = [2, 3, 7, 8];

arr.forEach(function(el) {

console.log(el \* 2);

});

However in ES 6 the code above can be written as following:

const arr = [2, 3, 7, 8];

arr.forEach(v => {

console.log(v \* 2);

});

Default parameters in ES6 :

In ES6, we can put the default values right in the signature of the functions:

**EX**

function test(a, b = 3, c = 42) {

return a + b + c;

}

console.log(test(5)); //50

Here is an example of an arrow function with default parameters

const test = (a, b = 3, c = 42) => {

return a + b + c;

}

console.log(test(5)); //50

Note:

Default value expressions are evaluated at function call time from left to right.

ES6 Objects:

JavaScript variables can be object data types that contain many values called properties.

An object can also have properties that are function definitions called methods for performing actions on the object.

The new method definition short hand does not require the colon( : ) or function keyword, as in the grow function of the tree object declaration:

**Ex**

let tree = {

height: 10,

color: 'green',

grow() {

this.height += 2;

}

};

tree.grow();

console.log(tree.height); // 12

# Computed property names:

With ES6, you can now use **computed property** names. Using the square bracket notation [], we can use an expression for a property name, including concatenating strings. This can be useful in cases where we want to create certain objects based on user data (e.g. id, email, and so on).

**Ex:1**

let prop = 'name';

let id = '1234';

let mobile = '08923';

let user = {

[prop]: 'Jack',

[`user\_${id}`]: `${mobile}`

};

**Ex:2**

var i = 0;

var a = {

['foo' + ++i]: i,

['foo' + ++i]: i,

['foo' + ++i]: i

};

**Ex:3**

var param = 'size';

var config = {

[param]: 12,

['mobile' + param.charAt(0).toUpperCase() +

param.slice(1)]: 4

};

console.log(config.mobileSize);

Object.assign( ) in ES6:

ES6 adds a new object method assign( ) that allows us to combine multiple sources into one target to create a single new object.

Object . assign( ) is also useful for creating duplicate of an existing object.

**Ex**

let person = {

name: 'Jack',

age: 18,

sex: 'male'

};

let student = {

name: 'Bob',

age: 20,

xp: '2'

};

let newStudent = Object.assign({}, person, student);

console.log(newStudent.name); // Bob

console.log(newStudent.age); // 20

console.log(newStudent.sex); // male

console.log(newStudent.xp); // 2

Now, let,s see how we can use assign( ) to create a duplicate object without creating a reference to the base object.

**Ex**

let person = {

name: 'Jack',

age: 18

};

let newPerson = person; // newPerson references person

newPerson.name = 'Bob';

console.log(person.name); // Bob

console.log(newPerson.name); // Bob

Array destructing in ES6:

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Example to demonstrate how to unpack values:

**Ex:**

let arr = ['1', '2', '3'];

let [one, two, three] = arr;

console.log(one); // 1

console.log(two); // 2

console.log(three); // 3

We can also destructure an array returned by a function:

**ex**

let a = () => {

return [1, 3, 2];

};

let [one, , two] = a();

console.log(one); // 1

console.log(two); // 2

The destructuring syntax also simplifies assignment and swapping values:

let a, b, c = 4, d = 8;

[a, b = 6] = [2]; // a = 2, b = 6

[c, d] = [d, c]; // c = 8, d = 4

ES6 Rest Parameters:

Prior to ES6, if we wanted to pass a variable number of arguments to a function, we could use the arguments object, an array like object, to access the parameters passed to the function.

**Ex:**

function containsAll(arr) {

for (let k = 1; k < arguments.length; k++) {

let num = arguments[k];

if (arr.indexOf(num) === -1) {

return false;

}

}

return true;

}

let x = [2, 4, 6, 7];

console.log(containsAll(x, 2, 4, 7));

console.log(containsAll(x, 6, 4, 9));

ES6 provides more readable syntax to achieve variable number of parameters by using a rest parameter.

**Ex**

function containsAll(arr, ...nums) {

for (let num of nums) {

if (arr.indexOf(num) === -1) {

return false;

}

}

return true;

}

The “…nums” parameter is called a restParameter. It takes all the “extra” arguments passed to the function. The 3 dots(…) are called Spread operator.

The spread operator is similar to the Rest Parameter, but it has another purp[ose when used in objects or arrays or functions calls(arguments).

SPREAD IN FUNCTION CALLS:

It is common to pass the elements of an array as arguments to a function. Before ES6, we used the following method.

**Ex**

function myFunction(w, x, y, z) {

console.log(w + x + y + z);

}

var args = [1, 2, 3];

myFunction.apply(null, args.concat(4));

output:

10

ES6 provides an easy way to do the example above spread operators:

const myFunction = (w, x, y, z) => {

console.log(w + x + y + z);

};

let args = [1, 2, 3];

myFunction(...args, 4);

output:

10

**Ex:2**

var dateFields = [1970, 0, 1]; // 1 Jan 1970

var date = new Date(...dateFields);

console.log(date);

### Spread in array literals

Before ES6, we used the following syntax to add an item at middle of an array:

var arr = ["One", "Two", "Five"];

arr.splice(2, 0, "Three");

arr.splice(3, 0, "Four");

console.log(arr);

You can use methods such as push, splice, and concat, for example, to achieve this in different positions of the array. However, in ES6 the spread operator lets us do this more easily:

**Ex**

let newArr = ['Three', 'Four'];

let arr = ['One', 'Two', ...newArr, 'Five'];

console.log(arr);

ES6 Classes:

We can create a class that can be used to create multiple objects of the same structure.

Example:

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

A class can be used to create multiple objects using the keyword “new”.

const square = new Rectangle(5, 5);

const poster = new Rectangle(2, 3);

We can also define a class with a class expression, where the class can be named or

un named.

A named class looks like:

var Square = class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

Un named class expression, a variable is simply assigned the class definition.

var Square = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

The constructor is a special Key word which is used for creating and initializing an object created with a class. There can be only one constructor in each class.

Class Methods in Es6:

ES6 introduced a shorthand that does not require the keyword function for a function assigned to a method’s name.

Example

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

get area() {

return this.calcArea();

}

calcArea() {

return this.height \* this.width;

}

}

const square = new Rectangle(5, 5);

console.log(square.area); // 25  
In the above code “area” is a getter, and “calcarean” is a method.

Another type of method is a static method, which cannot be called through a class instance. Static methods are often used to create utility functions for an application.

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

static distance(a, b){

const dx = a.x - b.x;

const dy = a.y - b.y;

return Math.hypot(dx, dy);

}

}

const p1 = new Point(7, 2);

const p2 = new Point(3, 8);

console.log(Point.distance(p1, p2));

The distance method called directly using the class name, without an object.

Inheritence in ES6:

The extends keyword is used in class declarations or class expressions to create a child of a class. The child inherits methods and properties of the parent.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

class Dog extends Animal {

speak() {

console.log(this.name + ' barks.');

}

}

let dog = new Dog('Rex');

dog.speak();

If there is a constructor present in the subclass, it needs to first call “super()” before using “this” . The “Super keyword is used to call parent’s methods.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

}

class Dog extends Animal {

speak() {

super.speak(); // Super

console.log(this.name + ' barks.');

}

}

let dog = new Dog('Rex');

dog.speak();

ES6 Map:

A Map object can be used to hold key/value pairs. A key or value in a map can be anything(Objects and primitive values).

The syntax “new Map([iterable]) creates a Map object where iterable is an array or any other iterable object whose elements are arrays(with a key/value pair each).

An object is similar to Map but there are important differences that makes using a map preferable in certain cases.

1)The keys can be any type including functions, objects, and any primitive.

2) You can get the size of a map.

3) you can directly iterate over map.

4) performance of the map is better in scenarios involving frequent addition and removal of key/value pairs.

**Ex**

let map = new Map([['k1', 'v1'], ['k2', 'v2']]);

console.log(map.size); // 2

**Methods**

**set(key, value)**Adds a specified key/value pair to the map. If the specified key already exists, value corresponding to it is replaced with the specified value.

**get(key)** Gets the value corresponding to a specified key in the map. If the specified key doesn't exist, undefined is returned.

**has(key)** Returns true if a specified key exists in the map and false otherwise.

**delete(key)** Deletes the key/value pair with a specified key from the map and returns true. Returns false if the element does not exist.

**clear()** Removes all key/value pairs from map.

**keys()** Returns an Iterator of keys in the map for each element.

**values()** Returns an Iterator of values in the map for each element.

entries()Returns an Iterator of array[key, value] in the map for each element.

ES6 Set:

A set object can be used to hold “unique” values (no repetitions are allowed).

A value in a set can be anything (objects and primitive values).

The syntax “new Set([iterable]) creates a set of objects where iterable is an array or any other iterable object of values.

The size property returns the number of distinct values in a set.

let set = new Set([1, 2, 4, 2, 59, 9, 4, 9, 1]);

console.log(set.size); // 5

METHODS:  
add(value): adds a new element with the given value to the set.

Delete(value): deletes a specified value from the set.

Has(value): Returns true if a specified value exists in the set and false otherwise.

Clear( ): clear the list.

Values( ): Returns an iterator of values in the set.

let set = new Set();

set.add(5).add(9).add(59).add(9);

console.log(set.has(9));

for (let v of set.values())

console.log(v);

ES6 Promises:

A promise is a better way for asynchronous programming when compared to the common way of using a setTimeout( ) type of method.

**Consider this example:**

**CODE PLAYGROUNDJS**

setTimeout(function() {

console.log("Work 1");

setTimeout(function() {

console.log("Work 2");

}, 1000);

}, 1000);

console.log("End");

It prints “End”, “Work1” and “Work2” in that order(the work is done asynchronously).But if there are more events like this, the code becomes very complex.

ES6 comes to rescue in such situations . A promise can be created as follows:

new Promise(function(resolve, reject) {

// Work

if (success)

resolve(result);

else

reject(Error("failure"));

});

Here “resolve “ is the method for success and “reject” is the method for failure.

If a method returns a promise, its calls should use the **then** method which takes two methods as input; one for success and the other for the failure.

function asyncFunc(work) {

return new Promise(function(resolve, reject) {

if (work === "")

reject(Error("Nothing"));

setTimeout(function() {

resolve(work);

}, 1000);

});

}

asyncFunc("Work 1") // Task 1

.then(function(result) {

console.log(result);

return asyncFunc("Work 2"); // Task 2

}, function(error) {

console.log(error);

})

.then(function(result) {

console.log(result);

}, function(error) {

console.log(error);

});

console.log("End");

It also prints “End”, “Work1” and “Work2”( the work is done asynchronously).But, this clearly more readable than the previous example and in more complex situations it is easier to work with.

Iterators and Generators:

Symbol.iterator is the default iterator for an object. The for…of loops are based on this type of iterator.

In this example we should implement how generator function are used.

let myIterableObj = {

[Symbol.iterator] : function\* () {

yield 1; yield 2; yield 3;

...

console.log([...myIterableObj]);

Modules:

It is a good practice to divide your related code into modules. Before ES6 there were some libraries which made this possible (e.g., RequireJS, CommonJS). ES6 is now supporting this feature natively.

**Considerations when using modules:**

The first consideration is **maintainability.** A module is independent of other modules, making improvements and expansion possible without any dependency on code in other modules.

The second consideration is **namespacing.** In an earlier lesson, we talked about variables and scope. As you know, vars are globally declared, so it's common to have namespace pollution where unrelated variables are accessible all over our code. Modules solve this problem by creating a private space for variables.

Another important consideration is **reusability.** When we write code that can be used in other projects, modules make it possible to easily reuse the code without having to rewrite it in a new project.

Let's see how we should use modules in JS files.

**For Example:**

// lib/math.js

export let sum = (x, y) => { return x + y; }

export let pi = 3.14;

// app.js

import \* as math from "lib/math"

console.log(`2p = + ${math.sum(math.pi, math.pi)}`)

**JS**Copy

Here we are exporting the sum function and the pi variable so we can use them in different files.

# Built-in Methods

ES6 also introduced new built-in methods to make several tasks easier. Here we will cover the most common ones.

### Array Element Finding

The legacy way to find the first element of an array by its value and a rule was the following:

[4, 5, 1, 8, 2, 0].filter(function (x) {

return x > 3;

})[0];

The new syntax is cleaner and more robust:

[4, 5, 1, 8, 2, 0].find(x => x > 3);

You can also get the index of the item above by using the **findIndex()** method:

[4, 5, 1, 8, 2, 0].findIndex(x => x > 3);

### Repeating Strings

Before ES6 the following syntax was the correct way to repeat a string multiple times:

console.log(Array(3 + 1).join("foo")); // foofoofoo

With the new syntax, it becomes:

console.log("foo".repeat(3)); // foofoofoo

### Searching Strings

Before ES6 we only used the indexOf() method to find the position of the text in the string. For example:

"SoloLearn".indexOf("Solo") === 0; // true

"SoloLearn".indexOf("Solo") === (4 - "Solo".length); // true

"SoloLearn".indexOf("loLe") !== -1; // true

"SoloLearn".indexOf("olo", 1) !== -1; // true

"SoloLearn".indexOf("olo", 2) !== -1; // false

ES6 has replaced this with a version that has cleaner and more simplified syntax:

"SoloLearn".startsWith("Solo", 0); // true

"SoloLearn".endsWith("Solo", 4); // true

"SoloLearn".includes("loLe"); // true

"SoloLearn".includes("olo", 1); // true

"SoloLearn".includes("olo", 2); // false

Callbacks VS Promises VS Async/Await:

A callback is a function passed as an argument to another function this technique allows a function to call another function. A call back function has run after another function has finished.

Example:

SetTimeout(( ) =>{

Console,log(“ Hello world!”);

}, 1000);

In the above “setTimeout” method we are calling first argument as a call back method i.e., “console.log” and second argument as number of milli seconds .