More on Javascript

Declaration using var, let, const

- Re-declare
- Access before declare
- Global Variable scope
- Block or function scope (later in the slides)

Declaration using var

EXAMPLE

```
var x=5;
console.log(x);
```

OUTPUT

```
5
```

Variables are containers for storing data (values).

Re-declaration using var

EXAMPLE

```
var a=1;  //declaration
var a=2;  //redeclaration
console.log(a);
```

OUTPUT

```
2
```

Using the keyword var you can declare and redeclare..

Redeclaration using let and const

EXAMPLE

```
let a=1;  //declaration
let a=2;  //redeclaration ERROR
console.log(a);
```

OUTPUT

```
Error
```

Using the keyword let or const you are NOT ALLOWED to redeclare..

Accessing variables before declaration

EXAMPLE

```
console.log(x);
var x=5;
console.log(x);
```

OUTPUT

```
Undefined
5
```

The undefined output is not clear, especially when you are dealing with a complex program

Accessing variables before declaration using let and const

EXAMPLE

```
console.log(a);  //uncaught ReferenceError:
let a=1;
```

OUTPUT

```
//uncaught ReferenceError
```

When declaring a variable using let or const, you cannot access a before declaration

Variable scope drama

EXAMPLE

```
var abc=1;
console.log(window.abc); // will output 1
```

OUTPUT

```
1
```

When declaring a variable using let or const, you cannot access a before declaration

Global Variable scope

EXAMPLE

```
let abc=1;
console.log(window.abc); // will output 1
```

OUTPUT

```
undefined
```

When declaring a variable using let or const, it is not placed in the global scope under the window object

Template literals

- **Template Literals** use back-ticks (``) rather than the quotes ("") to define a string:

```
let text = `Hello World!`;
```

- With **template literals**, you can use both single and double quotes inside a string:

```
let text = `He's often called "Johnny"`;
```

- **Template literals** allows multiline strings:

```
let text =
`The quick
brown fox
jumps over
the lazy dog`;
```

Template literals - interpolation

EXAMPLE

```
let a = “We love”;
let b =“Javascript”;
let c =“and”;
let d=“programming”;
console.log(a+ ”  ” +b+  ”\n”+c+” “+d);  //old way

console.log(a+” \”\” “+b+”\n”+c+” “+d);  //old way
console.log(`${a} ${b}
${c} ${d}` );
```

Use `${…}`

Template literals - Expression substitution

EXAMPLE

```
…..
let price = 10;
let VAT = 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;

document.getElementById("demo").innerHTML = total;
….
```

The toFixed() method formats a number using fixed-point notation. toFixed() returns a string representation of numObj that has exactly digits digits after the decimal place

Template literals - HTML templates

EXAMPLE

```
Let markup=`
        <div class=“card”>
                <div class=“child”>
                        <h2>Title</h2>
                        <p> paragraph</p>
                             </div>
                 </div>
                `;

document.write(markup);
```

Template literals are literals delimited with backticks (`), allowing embedded expressions called substitutions.

```
`;
```

Conditional using the ternary operator

Syntax

```
Condition ? If True : If False
```

EXAMPLE

```
Let name=“John”;
Let gender=“male”;
Let age=20;
If (gender==“male”){
console.log(“mr”);
}else{
console.log(“”Mrs”);}
```

```
`;
```

```
Let name=“John”;
Let gender=“male”;
Let age=20;
Gender==“male” ? console.log(“Mr”):console.log(“female”)
```

```
Let name=“John”;
Let gender=“male”;
Let age=20;
Let result=Gender==“male” ?“Mr” : “Mrs”
document.write(result);
console.log(Gender==“male”? “Mr” : “Mrs”);
console.log(`hello ${Gender==“male”? “Mr” : “Mrs”}
${name}`));
```

Chained ternary operator

Syntax

```
condition1
        ? statement
        : condition2
        ? statement
        : condition3
        ? statement
        : statement;
```

Example

```
theAge < 20
  ? console.log(20)
  : theAge > 20 && theAge < 60
  ? console.log("20 To 60")
  : theAge > 60
  ? console.log("Larger Than 60")
  : console.log("Unknown");
```

Additional Array methods (slice)

- The slice() method a portion of an array into a new array object selected from start to end (end not included) where start and end represent the index of items in that array.
- The original array will not be modified.

Example

```
let friends=[“”,””,””,””,””,””];
Console.log(friends);
Console.log(friends.slice());
Console.log(friends.slice(1));
Console.log(friends.slice(1,3));
Console.log(friends.slice(-3));
Console.log(friends.slice(1,-2));
Console.log(friends.slice(-4,-2));
Console.log(friends);
```

Additional Array methods (splice)

- The splice() method adds or removes (start deletion or insertion index, deletecount, add elements);
- It overwrites the original array.

Syntax

array.splice(*index*, *howmanytodelete(optional)*, *item1(optional)*, *.....*, *itemX(optional)*)

Example

```
myCourses.splice(1, 2, "Art", "Sports");
console.log(myCourses);
myCourses.splice(0,1,"Art", "Sports"); //deletes then adds
myCourses.splice(0,2,"Art", "Sports"); //deletes then adds
```

Additional Array methods (concat)

- The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

Syntax

```
array1.concat(array2, array3, ..., arrayX)
```

Example

```
Let myfrielnds=["Ahmed", "Sayed", "Ali", "Osama“]
Let mynewfriends=…
Let schoolFriends=..
Let allFriends=myfriends.conact(myNewFriend);

Let allFriends=myfriends.conact(myNewFriend,schholfriends);
Let allFriends=myfriends.conact(myNewFriend,schholfriends, “sara”);
```

Additional array methods (join)

- The join() method creates and returns a **new string** by concatenating all of the elements in an array, separated by commas or a specified separator string.
- If the array has only one item, then that item will be returned without using the separator.

Example

```
….
Console.log(allFriends.join());  //returns a string separated by comma
….. 
```

Functions rest parameters

- When you don’t know the number of arguments (Ex: skills)
- Using rest parameters, you can allow the function to receive an unknown number of parameters

```
Let result=0;
function calc(…numbers){   /*numbers is an array of arguments */
For (let i=0; i<=numbers.length;i++){
Result+=numbers[i]
}
return `final result is ${result}`;

Console.log(calc(10,33,44,55,33));
```

Anonymous Function

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

Syntax

```
function() {
    // Function Body
 }
```

Nested functions

- A nested function is **a function which is defined within another function, the enclosing function**.

EXAMPLE

```
function sayMessage(fName, lName) {
  let message = `Hello`;
  // Nested Function
  function concatMsg() {
    message = `${message} ${fName} ${lName}`;
  }
  concatMsg();
  return message;
}

console.log(sayMessage("Sara", "Tedmori"));
```

Global Scope vs Local Scope

EXAMPLE

```
var a=1;
var b=2;
Function add(){


}
```

When decalared outside a function, a and b are both global variables that can be accessed from anywhere!

```
var a=1;
var b=2;
Function add(){
var c=2;
var d=3;
}
```

When decalared inside a function, c and d are both local variables that can be accessed only from inside the function.

Block scope vs function scope

EXAMPLE

```
var x = 10;

if (10 === 10) {
  var x = 50;

  console.log(`From If Block ${x}`);
}

console.log(`From Global ${x}`);
```

```
var x = 10;

if (10 === 10) {
  let x = 50;

  console.log(`From If Block ${x}`);
}

console.log(`From Global ${x}`);
```

OUTPUT

```
From if Block 50
From Global 50
```

```
From if Block 50
From Global 10
```

EXAMPLE

```
function run() {
  var foo = "Foo";
  let bar = "Bar";

  console.log(foo, bar); // Foo Bar

  {
    var moo = "Mooo"
    let baz = "Bazz";
    console.log(moo, baz); // Mooo Bazz
  }

  console.log(moo); // Mooo
  console.log(baz); // ReferenceError
}

run();
```

EXAMPLE

```
// i IS NOT known here
// j IS NOT known here
// k IS known here, but undefined
// l IS NOT known here
function loop(arr) {

    // i IS known here, but undefined
    // j IS NOT known here
    // k IS known here, but has a value only the
         second time loop is called
    // l IS NOT known here

    for( var i = 0; i < arr.length; i++ ) {
        // i IS known here, and has a value
        // j IS NOT known here
        // k IS known here, but has a value only
          the second time loop is called
        // l IS NOT known here
    };

    // i IS known here, and has a value
    // j IS NOT known here
    // k IS known here, but has a value only the
         second time loop is called
    // l IS NOT known here
    for( let j = 0; j < arr.length; j++ ) {
        // i IS known here, and has a value
        // j IS known here, and has a value
        // k IS known here, but has a value only
           the second time loop is called
        // l IS NOT known here
    };
    // i IS known here, and has a value
    // j IS NOT known here
    // k IS known here, but has a value only the
         second time loop is called
    // l IS NOT known here
}

loop([1,2,3,4]);

for( var k = 0; k < arr.length; k++ ) {
    // i IS NOT known here
    // j IS NOT known here
    // k IS known here, and has a value
    // l IS NOT known here
};

for( let l = 0; l < arr.length; l++ ) {
    // i IS NOT known here
    // j IS NOT known here
    // k IS known here, and has a value
    // l IS known here, and has a value
};

loop([1,2,3,4]);

// i IS NOT known here
// j IS NOT known here
// k IS known here, and has a value
// l IS NOT known here
```

Lexical Scope

```
function parent() {
  let a = 10;
  function child() {
    console.log(a);
    console.log(`From Child ${b}`); //uncaught reference error..
    function grand() {
      let b = 100;
      console.log(`From Grand ${a}`);
      console.log(`From Grand ${b}`);
    }
    grand();
  }
  child();
}
parent();
```

Function sequence

```
<script>
function myDisplayer(some) {

document.getElementById("demo").innerHTML
= some;
}
function myFirst() {
  myDisplayer("Hello");
}
function mySecond() {
  myDisplayer("Goodbye");
}
myFirst();
mySecond();
```

```
Goodbye
```

```
<script>
function myDisplayer(some) {

document.getElementById("demo").innerHTML
= some;
}
function myFirst() {
  myDisplayer("Hello");
}
function mySecond() {
  myDisplayer("Goodbye");
}
mySecond();
myFirst();
```

```
hello
```

Sequence Control

```
<p id="demo"></p>
<script>
Var sum=0
function myDisplayer(some) {

document.getElementById("demo").innerHTML
= some;
}
function myCalculator(num1, num2) {
  var sum = num1 + num2;
  return sum;
}
let result = myCalculator(5
myDisplayer(result);
</script>
```

```
10
```

```
<p id="demo"></p>
<script>
Var sum=0
function myDisplayer(some) {
document.getElementById("demo").innerHTML
= some;
}
function myCalculator(num1, num2) {
 var sum = num1 + num2;
  myDisplayer(sum);
}

myCalculator(5, 5);
</script>
```

```
10
```

Javascript callbacks

- A callback is a function passed as an argument to another function
- This technique allows a function to call another function
- A callback function can run after another function has finished
- Using a callback, you could call a function with a callback, and let the calculator function run the callback after the function is finished.

Using JS callbacks

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
</script>
```

Asynchronous JavaScript

- Using setTimeOut()
- Using setInterval()

```
setTimeout(function() { myFunction(“Hello!!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
```

setTimeout()

```
setTimeout(function() { myFunction(“Hello!!!"); }, 3000);


function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
```

setInterval

```
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" +
  d.getMinutes() + ":" +
  d.getSeconds();
}
```

JavaScript Promises

- Producing code" is code that does something and can take some time. For instance, some code that loads the data over a network.
- "Consuming code" is code that must wait for the result. It wants the result of the “producing code” once it’s read
- A Promise is a JavaScript object that links producing code and consuming code
- A JavaScript Promise object contains both the producing code and calls to the consuming code:

Promise Syntax

```
// syntax for a promise object :
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise Syntax

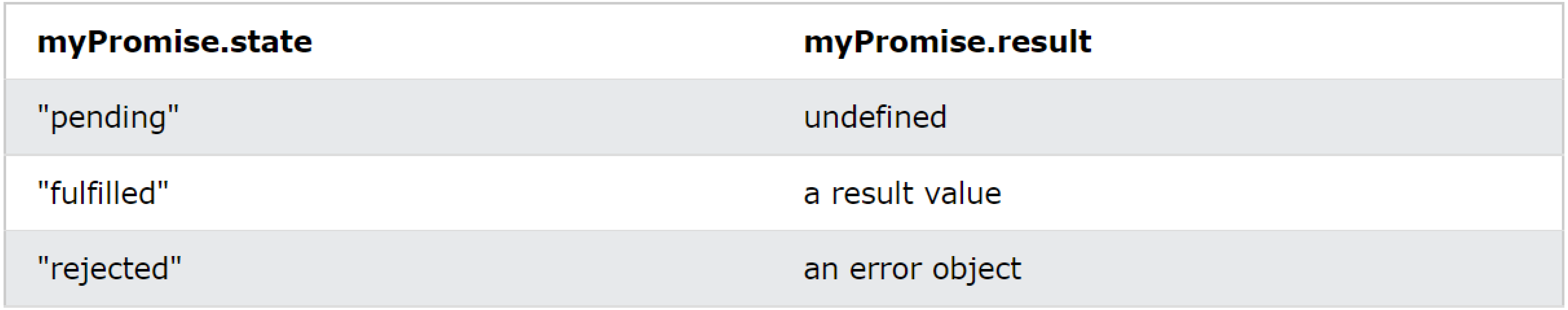
```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code)
});
```

Promise Object Properties

- A JavaScript Promise object can be:
    - Pending
    - Fulfilled
    - Rejected
- The Promise object supports two properties: **state** and **result**.
- While a Promise object is "pending" (working), the result is undefined.
- When a Promise object is "fulfilled", the result is a value.
- When a Promise object is "rejected", the result is an error object.

State and result properties

- The promise object returned by the new Promise constructor has these internal properties:
- The Promise object supports two properties: **state** and **result**.



myPromise.state	**myPromise.result**
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

Promise how to?

- Here is how to use a Promise:

```
// "Consuming Code" (Must wait for a fulfilled Promise)

myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Example

- To demonstrate the use of promises, we will use the callback examples :
- Waiting for a Timeout
- Waiting for a File (later)

Regular functions

```
/* regular functions */
function print(){
  return 10;
}

Console.log(print());
```

```
/* regular functions assigned
to variable */
Let print= function (){
  return 10;
}

Console.log(print());
```

You can assigned a function to a variable. This is called a function expression

Arrow function with no parameter

```
/* arrowfunctions */
Let print= ~~function~~ () => {
  return 10;
}

Console.log(print());
```

```
/* arrowfunctions */
Let print= () => ~~{~~
  ~~return~~ 10;
~~}~~

Console.log(print());
```

You can remove the return keyword and the {} when the body is only one statement inside the block

Cant remove the reurn and {} if more than one statement

Arrow function with one parameter

```
/* regular function */
Let print= function(num) {
  return num;
}

Console.log(print(100));
```

Regular function with one parameter

```
/* arrowfunctions */
Let print= ~~(~~num~~)~~ => num;


Console.log(print(100));
```

You can even remove the () of the parameters if you have only one parameter

Arrow function with two parameter

```
/* regular function */
Let print= function(num1,num2)
{
  return num1+num2;
}

Console.log(print(100,200));
```

```
/* arrowfunctions */
Let print= (num1,num2) =>
   num1+num2;

Console.log(print(100,200));
```

Regular function with twoparameter

You cannot remove the () of the parameters since you have two parameters

Higher order function – map()

- Higher Order Function is a function that accepts functions as parameters and/or returns a function.
- Map() function is a higher order function (an array method)
  - map() method creates a new array
  - populated with the results of calling a provided function on every element
  - in the calling array.

Syntax

```
Syntax map(callBackFunction(Element, Index, Array) { }, thisArg)
  - Element => The current element being processed in the array.
  - Index => The index of the current element being processed in the array (optional).
  - Array => The Current Array (optional)
```

Higher order practice

- Swap Cases

Example

```
let swappingCases = “pSuT";

 let sw = swappingCases
   .split("")
   .map(function (ele) {
/   // Condition ? True : False
    return ele === ele.toUpperCase() ? ele.toLowerCase() : ele.toUpperCase();
   })
   .join("");
```

Higher Order Function – filter()

- Filter function is a higher order function (an array method)
- Filter() method creates a new array with all elements that pass the test implemented by the provided function.

Syntax

```
Syntax filter(callBackFunction(Element, Index, Array) { }, thisArg)
 - Element => The current element being processed in the array.
 - Index => The index of the current element being processed in the array.
 - Array => The current Array
```

Higher Order functions-reduce()

- Reduce method executes a reducer function on each element of the array
- Reduce method results in a single output value.

Syntax reduce(callBackFunc(Accumulator, Current Val, Current Index, Source Array) { }, initialValue)
 - Accumulator => the accumulated value previously returned in the last invocation
 - Current Val => The current element being processed in the array..
 - Index => The index of the current element being processed in the array.
 ---------- Starts from index 0 if an initialValue is provided.
 ---------- Otherwise, it starts from index 1.
 - Array => The Current Array

Higher order function – forEach()

- forEach method executes a provided function once for each array element.
- It DOES NOT return a new array. It doesn’t return anything

```
, Index, Array) { }, thisArg)
  - Element => The current element being processed in the array.
  - Index => The index of the current element being processed in the array.
  - Array - The Current Array

  Note
  - Doesnt Return Anything [Undefined]
  - Break Will Not Break The Loop Syntax forEach(callBackFunction(Element
```

Object

- In JavaScript, an object is a standalone entity, with properties and type.
- An object is a collection of properties, and a property is an association between a name (or *key*) and a value.
- **A property's value can be a function, in which case the property is known as a method.**
- In addition to objects that are predefined in the browser, you can define your own objects.
- Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design, weight, a material it is made of, etc.

Object properties

- A JavaScript object has properties associated with it.
- A property of an object can be explained as a variable that is attached to the object.
- Object properties are basically the same as ordinary JavaScript variables, except for the attachment to objects.
- The properties of an object define the characteristics of the object.
- You access the properties of an object with a simple dot-notation:
- You can define a property by assigning it a value.

```
objectName.propertyName
```

Create an object with some properties example

```
//create object with new
keyword


var myCar = new Object();

//properties

myCar.make = 'Ford';

myCar.model = 'Mustang';

myCar.year = 1969;
```

```
//create object using an object
initialiser


var myCar = {

//properties

    make: 'Ford',

    model: 'Mustang',

    year: 1969,

};
```

Another Create an object example

```
//create object

let user = {
  // Properties
  theName: “John",
  theAge: 38,
  // Methods
  sayHello: function () {
    return `Hello`;
  },
};
Console.log(user.theName);
Console.log(user.theAge);
Console.log(user.sayHello());
```

Dot Notation / bracket notation

```
Let user={
theName:”Sara”,
“country of”:”Lebanon”,
};

console.log(user.theName);

Console.log(user.“county of”);
//identifier expected
```

Using the dot notation you can not access a property if its name is not a valid identifier

```
Let user={
theName:”Sara”,
“country of”:””,
};

Console.log(user{theName]);
Console.log(user“[count
of]”);
```

Using the bracket notation you can access property with valid or invalid property names

Dynamic property name

```
let myVar = "country";
//this is a property name
similar to a property in an object

let user = {
  theName: “Sara",
  country: “Lebanon",
};

console.log(user.theName);
console.log(user.country); // user.country
console.log(user.myVar); // user.country  returns undefinced..
console.log(user[myVar]); // user.country
```

Using the dot notation you can not access a property using the dynamic property name

You can do that however using the bracket notation

This Keyword

- The JavaScript this keyword refers to the object it belongs to.

- It 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 an event, this refers to the element that received the event.