CSE 220 Web Programming and Scripting

DR. RAJALAKSHMI S

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING
SRI RAMACHANDRA FACULTY OF ENGINEERING AND TECHNOLOGY

Module 2

JavaScriptAdvanced Topics

Variables life cycle

All the variables undergo three stages viz: Declaration, Initialization and Assignment.

Declaration	Register variables in the corresponding scope
Initialization	Allocate memory for the variable. The variables declared with var are automatically initialized with the value undefined
Assignment	Assign values to the initialized variable

Declaration & Initialization

This is the first stage of a variable's life cycle. Variables in JavaScript can be declared using the 'var', 'let', or 'const' keywords. The declaration process allocates memory for the variable and associates a name with it.

```
var age;            // Declaring the variable age
let name;           // Declaring the variable name
const PI = 3.14;    // Declaration of constant variable
```

Variables can be initialized during declaration or at later stage. If a variable is declared with an initial value, it enters the initialization stage and memory is allocated to store the value assigned to it

```
var age = 32;           // Initializing the variable age with the value 32
let name = "Raji" ;     // Initializing the variable name with the value Raji
```

Assignment

Variables can have their values changed or updated after initialization and this process is known as assignment.

```
var age = 32;      // Initialization of the variable 'age' with the value 32
age = 34;          // Assignment, changing the value of age to 34
```

Module2 > JS 1_variable_lifeycle.js > myfun

```
1  function myfun(a) {
2    console.log(a);
3  }
4  myfun();
5
```

Output as : Undefined. Since JavaScript automatically declares and initialize the value

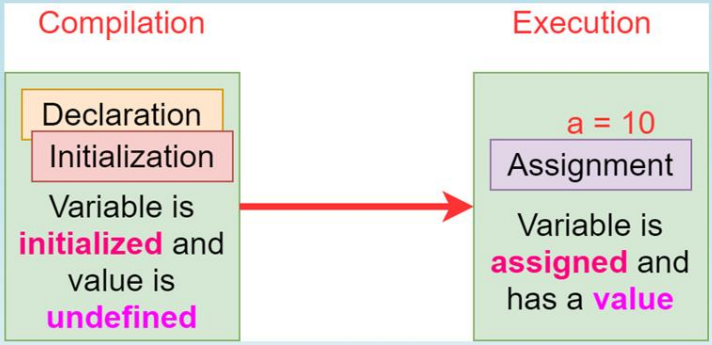
Module2 > JS 1_variable_lifeycle.js > ...

```
1  function myfun() {
2    console.log(a);
3  }
4  myfun();
5
```

Reference error: a is not defined

PREPARED BY DR.RAJALAKSHMI S 7

Lifecycle of var



PREPARED BY DR.RAJALAKSHMI S 8

Lifecycle of var - Declaration and Function Scope

Variables declared with 'var' are function – scoped, meaning they are accessible within the function where they are declared. If a variable is declared outside any function, it becomes a global variable, accessible throughout the entire program

Module2 > JS 1_var_lc1.js > ...

```
1  function myfun() {
2    var x = 10; // Declaration of 'x' within this function
3    console.log(x); // output: 10
4  }
5
6  myfun();
7  //console.log(x); // ReferenceError: x is not defined
8
```

Line 7 leads to reference error as variable x is having scope withing myfun().

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web P
10
f:\SRIHER\2023 - 2024\CSE 220 Web Programming and S
console.log(x);
            ^

ReferenceError: x is not defined
```


Lifecycle of var - Hoisting

Variables declared with 'var' are hoisted to the top of their function or global scope during the creation phase of the execution context. This means that once can use the variable before its actual declaration in the code, but it will have the value 'undefined until the declaration is reached.

Module2 > JS 1_var_lc2.js > ...

```
1  console.log(y);
2  var y = 100;
3  console.log(y);
4
```

Output: undefined as a result of hoisting
Declaration and initialization of y
Output 20, assigned value

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220
undefined
100

[Done] exited with code=0 in 1.239 seconds
```

Initialization:

'var' variables are automatically initialized with the value 'undefined; during the hoisting phase

Lifecycle of var - Redefinition and Reassignment

Variables declared with 'var' can be redeclared within the same scope without any errors, and they can also be reassigned new values.

Module2 > JS 1_var_lc3.js > myfun

```
1  var age = 30; // Declaration and initialization of 'age'
2  var age = 35; // Redefinition of 'age' (no error)
3
4  function myfun() {
5    var name = "ABC";
6    name = "Raji"; // Reassignment of name within the same function
7    var name = "DEF"; // Redefinition of 'name' (no error)
8    console.log(name); // Output:"DEF" the current value
9  }
10
11 myfun();
12
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pro
DEF

[Done] exited with code=0 in 0.148 seconds
```


Lifecycle of let

Variables declared using the 'let' keyword have a lifecycle that involves various stages including block scope and temporal dead zone. The lifecycle of variables declared with 'let' can be summarized as follows

Declaration and Block Scope:

When a variable is declared using 'let;, it is block scoped meaning it is only accessible within the block or statement where it was declared, including any nested blocks.

Module2 > JS 1_let_lc1.js > ...

```
1  {
2    let x = 10; // Declaration of 'x' within this block
3    console.log(x); // output: 10
4  }
5  console.log(x); // Error: 'x' is not defined,as it is outside the block
6
```

Lifecycle of let

Temporal Dead Zone:

Variables declared with 'let; are subject to the Temporal Dead Zone, a period between the start of the block and the actual declaration of the variable. During this time, trying to access the variable results in a Reference Error

```
console.log(x);     // Error cannot access 'x' before initialization
let x = 10;         // Declaration of 'x' comes later in the code
```

Initialization:

Unlike variables declared with 'var', 'let' variables are not automatically initialized with the value 'undefined'. As a result, they remain in the temporal dead zone until their declaration is reached during execution.

Module2 > JS 1_let_lc2.js > ...

```
1  let x; // Declaration without initialization
2  console.log(x); // Output: undefined
3  x = 20; // Initialization of 'x' with value 20
4  console.log("Now", x); // Output : 20
5
```

PREPARED BY DR.RAJALAKSHMI S 13

Lifecycle of let

Reassignment:

Variables declared with 'let' can have their values reassigned within their block scope.

Module2 > JS 1_let_lc3.js > ...

```
1  let age = 20; // Declaration and initialization of 'age'
2  age = 24; // Reassignment, changing the value of 'age' to 35
3  console.log(age); // Output: 24, updated one
4
```

No redeclaration:

Unlike variables declared with 'var', one cannot redeclare a variable with 'let' within the same scope. Attempting to do so will result in Syntax Error

Module2 > JS 1_let_lc4.js > ...

```
1  let dept = "AIML"; // Declaration and Initialization of a variable
2  let dept = "AIDA"; //SyntaxError: Identifier 'dept' has already been declared
3
```

PREPARED BY DR.RAJALAKSHMI S 14

Lifecycle of const

Variables declared using const keyword have a lifecycle that differs from both 'var' and 'let'. The 'const' keyword is used to declare constants, which are variables that cannot be reassigned once they are initialized. The lifecycle of variables declared with 'const' can be summarized as:

Declaration and Block Scope: Like variables declared with 'let', variables declared with 'const' are block scoped, meaning they are accessible within the block or statement where they were declared, including any nested blocks.

Module2 > JS 1_const_lc1.js > ...

```
1  {
2    const x = 10; // Declaration of 'x' within this block
3    console.log(x); // output: 10
4  }
5  console.log(x); // Reference Error: 'x' is not defined,as it is outside the block
6
```


Lifecycle of const

Temporal Dead Zone: Variables declared with 'const' are subject to Temporal Dead Zone, just like the variables declared with 'let'. During this time, trying to access the variable before its declaration results in a Reference Error.

Module2 > JS 1_const_lc2.js > ...

```
1  //const x; // SyntaxError: Missing initializer in const declaration
2  //console.log(x); // ReferenceError: Cannot access 'x' before initialization
3  const x = 20; // Initialization of 'x' with value 20
4  console.log("Now", x); // Output : 20
5
```

Initialization: Variables declared with 'const' must be initialized during the declaration. Once a value is assigned to a 'const' variable, it cannot be changed or reassigned.

Module2 > JS 1_const_lc3.js > ...

```
1  const PI = 3.14; // Declaration and Initialization of PI
2  console.log(PI); // Output: 3.14
3
4  //PI = 3.14159; // Error Assignment to constant variable
5
```

Lifecycle of const

No Reassignment: The most significant difference between variables declared with 'const' and those declared with 'let' is that 'const' variables cannot be reassigned or have their values changed after initialization.

```
const age = 12;      // Declaration and initialization of 'age'
age = 24;            // Error ; Assignment to constant variable
```

No Redeclaration: As with 'let; one cannot redeclare a variable with 'const' within the same scope. Attempting to do so will result in a syntax error.

```
const age = 12;     // Declaration and initialization of 'age'
const age = 24;     // Syntax Error: Identifier age has already been declared
```


ES6 Arrow Functions

Arrow functions were introduced in ECMAScript 2015 (ES6) as a concise way to define functions in JavaScript. The syntax is intuitive and compact when comparing to traditional function expressions. The basic syntax is as follows:

```
const funName = (parameters) => {
        //function body

        return value.
}
```

'const': Arrow functions are often assigned to a variable using the 'const' keyword to create a function expression.

'funName': This is the name of the function. Arrow functions can be anonymous or have a name (in case of named function expressions). If the arrow function is anonymous, one can still call it using the variable to which it is assigned

ES6 Arrow Functions

'parameters': These are the input parameters of the function, similar to regular function parameters. If there's only one parameter, the parentheses can be omitted. For zero or multiple parameters, parentheses are required.

'=>': The fat arrow '=>' separates the parameter list from the function body. It indicates that it's an arrow function

'{}': The curly braces represent the function body, where the actual code of the function resides. If the function has a single expression as its body, the curly braces can be omitted.

'return': If the function body consists of a single expression, the 'return' keyword can be omitted. The function will implicitly return the result of that expression.

PREPARED BY DR.RAJALAKSHMI S 19

ES6 Arrow Functions - Syntax

```
// arrow function with only one expression
(a,b) => a+b; // implicit return of a+b

// arrow function with one statement can be wrapped inside parenthesis with return statement
(a,b) => {
return a+b; // explicit return of a+b
}

// arrow function with only one argument
a => a*a; // omit ( ) with one parameter

// arrow function with no arguments returning constant value
() => 2; // ( ) are mandatory without any parameters

// arrow function returning object; explicit return of the object
(a,b) => {
  return {
    a1:a;
    b1:b
}

// implicit return of the object
(a,b) => (
{
                a1: a,
  b1:b
}
)
```

PREPARED BY DR.RAJALAKSHMI S

Example: Arrow function for adding two numbers

Code:

Module2 > JS 1_Af1.js > ...

```
1   const sum = (a, b) => a + b;
2
3   console.log(sum(2, 3));
4
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 2
5

[Done] exited with code=0 in 0.961 seconds
```

When trying to access the function sum before the expression results in reference error.

Module2 > JS 1_Af2.js > ...

```
1   console.log(sum(2, 3)); //ReferenceError: Cannot access 'sum' before initialization
2
3   const sum = (a, b) => a + b;
4
```

PREPARED BY DR.RAJALAKSHMI S 21

Example 2: Arrow functions to return an object explicitly

Code

Module2 > JS 1_Arf3.js > postprocess

```
1   const post = {
2     title: "JavaScript",
3     comments: 10,
4     shared: true,
5     published: true,
6     postId: 2124,
7   };
8
9   const postprocess = (post) => {
10    return {
11      title1: post.title,
12      comments1: post.comments,
13      popular: post.comments > 5 ? true : false,
14    };
15  };
16
17  console.log(postprocess(post));
18
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programmin
{ title: 'JavaScript', comments: 10, popular: true }

[Done] exited with code=0 in 0.139 seconds
```

The above example takes object as an argument and process the property of an object using ternary operator.

PREPARED BY DR.RAJALAKSHMI S 22

Example 3: Arrow functions to return an object implicitly

Code

Module2 > JS 1_Arf4.js > postprocess

```
1   const post = {
2     title: "JavaScript",
3     comments: 10,
4     shared: true,
5     published: true,
6     postId: 2124,
7   };
8
9   const postprocess = (post) => ({
10    title1: post.title,
11    comments1: post.comments,
12    popular: post.comments > 5 ? true : false,
13  });
14
15  console.log(postprocess(post));
16
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programm
{ title1: 'JavaScript', comments1: 10, popular: true }

[Done] exited with code=0 in 1.274 seconds
```

PREPARED BY DR.RAJALAKSHMI S 23

Immediately Invoked Function Expression

Immediately invoked Function Expression (IIFE)

The key to create an IIFE with arrow functions is to wrap the arrow function in parentheses and invoke it immediately.

```
() => {
  const msg = "Welcome all";
  console.log(msg);
}
```

Module2 > JS 2_Arf5.js > ...

```
1   (() => {
2     const msg = "Welcome all";
3     console.log(msg);
4   })();
5
```

The above piece of code has to be wrapped within ( ) and it is followed by (); to invoke it immediately.

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE
[Running] node "f:\SRIHER\2023 -
Welcome all

[Done] exited with code=0 in 1.25
```

PREPARED BY DR.RAJALAKSHMI S 24

IIFE with parameter

Code

Module2 > JS 2_Arf6.js

```
1  ((name) => {
2    console.log("Welcome \t" + name);
3  })("Raji");
4
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 2
Welcome     Raji

[Done] exited with code=0 in 0.134 seconds
```


Arrow Functions don't own this

Arrow functions don't have own "this". "this" in arrow functions is always statically defined by the surrounding lexical scope.

Code

Module2 > JS 2_Arf7.js > ...

```
1  const number = {
2    value: 1002,
3    fun1: function funct() {
4      console.log(this);
5      return this.value;
6    },
7  };
8
9  console.log(number.fun1());
10 //arrowfunction
11 const number1 = {
12   value: 1002,
13   funa: () => {
14     console.log("Inside arrow", this);
15     return this.value;
16   },
17 };
18
19 console.log(number1.funa());
20
```

Output

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and
{ value: 1002, fun1: [Function: funct] }
1002
Inside arrow {}
undefined

[Done] exited with code=0 in 0.127 seconds
```

Arrow Functions don't own this

In the case of arrow functions, the value of 'this' is not determined by how the function is called, but by the surrounding lexical context in which the arrow function is defined. In this example, the arrow function 'funa' is defined inside the 'number1' object. However, since its an arrow function, it does not have its own 'this' binding and instead inherits 'this' from the surrounding scope, which is the global scope (or the object that encloses 'number1 object).

Since, the call number1.funa(), the arrow function is executed in the context of the global object (E.X: 'window' in browsers), that is providing the output '{ }' as the value of 'this'. In the global context, there is no property 'value', so 'this.value' returns 'undefined'.
To access the 'value' propery of the 'number1' object correctly, one should use a regular function instead of an arrow function for 'funa' which will bind its own 'this' context based on how it is called (as a method of 'number1')

Arrow Functions don't own this

Using arrow functions within object methods is generally not recommended when need to access to the object's properties through 'this', as they won't provide the desired behavior of 'this' referring to the object itself. Regular functions are better suited for methods that rely on object properties and their context.

```
1   const num1 = {
2     value1: 100,
3     fun1: function fun1 () {   // either give
4       console.log(this);
5       return this.value1;
6     },
7   };
8
9   const num2 = {
10    value2: 200,
11  };
12
13  console.log(num1.fun1.call(num2));
14
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pro
{ value2: 200 }
undefined

[Done] exited with code=0 in 0.172 seconds
```

Arrow Functions don't own this

Code

```
Module2 > JS 2_Arf9.js > num2
1   const num1 = {
2     value1: 100,
3     fun1: () => {
4       console.log(this);
5       return this.value1;
6     },
7   };
8
9   const num2 = {
10    value2: 200,
11  };
12
13  console.log(num1.fun1.call(num2));
14
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
{}
undefined

[Done] exited with code=0 in 0.173 seconds
```

PREPARED BY DR.RAJALAKSHMI S 29

Using arrow function

```
Module2 > JS 2_Arf9.js > num2
1   const num1 = {
2     value1: 100,
3     fun1: () => {
4       console.log(this);
5       return this.value1;
6     },
7   };
8
9   const num2 = {
10    value2: 200,
11  };
12
13  console.log(num1.fun1.call(num2));
14
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Progra
{}
undefined

[Done] exited with code=0 in 0.173 seconds
```

PREPARED BY DR.RAJALAKSHMI S 30

Using arrow function

In arrow functions, this is not dynamically scoped like in regular functions. Instead, arrow functions inherit the value of this from the surrounding context, which, in this case, is the global scope (e.g., window object in a browser).

When you call num1.fun1.call(num2), the arrow function fun1 is executed within the global scope, not within the num1 object. That's why this points to the global scope (an empty object representing the global context), and this.value1 is undefined, as there is no value1 property in the global scope.

Module2 > JS 2_Af11.js > str

```
1  const str = {
2    value: "Greetings",
3    greet: function greet() {
4      const self = this;
5      setTimeout(function () {
6        console.log(self);
7        console.log(self.value);
8      }, 1000);
9    },
10 };
11
12 str.greet();
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
{ value: 'Greetings', greet: [Function: greet] }
Greetings

[Done] exited with code=0 in 1.178 seconds
```


Array Helper methods

An array helper method is a built in function in programming languages like JavaScript that is specifically designed to work with arrays. These methods provide convenient and efficient ways to perform common operations on Arrays such as iteration over array elements, filtering elements, transforming elements, searching for elements.

Array helper methods are valuable because they simplify array manipulation tasks and make the code more readable and concise. Instead of writing complex loops and conditions to work with arrays, array helper methods can be used to achieve the same results with fewer lines of code.

foreach

foreach method intends to execute a provided function once for each array element. This method allows to iterate over each element of an array and apply a callback function to each element. The syntax is given as:

```
array.forEach(callback(element, index, array), thisArg);
```

'array' : The array you want to iterate over.
'callback': A function to execute on each element of the array.
'element': The current element being processed in the array
'index' (optional): The index of the current element being processes.
'array' (optional): The array that 'forEach' is being applied to.
'thisArg' (optional): An object that will be passed as the 'this' value to the callback function. If not provided, 'undefined' will be used as the 'this' value.

foreach

The forEach method does not return anything. It iterates through the array and apploes the callback function to each element. The primary purpose of 'forEach' is to eecute some code for each item in the array without creating a new array or modifying the original one.

Module2 > JS 2_foreach.js > ...

```
1  const numbers = [10, 20, 30, 40, 50];
2
3  numbers.forEach(function (num, id) {
4    console.log("num:", num);
5  });
6
```

PROBLEMS OUTPUT DEBUG CONSOLE

```
[Running] node "f:\SRIHER\2023
num: 10
num: 20
num: 30
num: 40
num: 50
```

For each to process element in array of object

One can access the properties of the objects as well within the callback function. For example: to calculate and log the average for students who got greater than 45.

```
const student = [
{ name: "ABC", age: 18, m1: 89, m2: 93 },
{ name: "DEF", age: 20, m1: 100, m2: 99 },
{ name: "GFD", age: 21, m1: 82, m2: 44 },
];
let count = 0;
//Display all objects in array
student.forEach(function (stud, id) {
console.log("Student", id);
console.log(stud.name);
console.log(stud.age);
console.log(stud.m1);
console.log(stud.m2);
});
```

```
//Calculate average for students and display those students who got pass
student.forEach(function (stud, id) {
if (stud.m1 > 45 && stud.m2 > 45) {
let tot = stud.m1 + stud.m2;
let avg = tot / 2;
console.log("student ", id + 1);
console.log("Name=", stud.name);
console.log("Age=", stud.age);
console.log("M1", stud.m1);
console.log("M2", stud.m2);
console.log("Average", avg);
count++;
}
});


// finally print the number of students who got pass
console.log("Total Number of students who got pass", count);
```


Arrow function as callback function in forEach

One can access the properties of the objects as well within the callback function. For example: to calculate and log the average for students who got greater than 45.

```
const student = [
{ name: "ABC", age: 18, m1: 89, m2: 93 },
{ name: "DEF", age: 20, m1: 100, m2: 99 },
{ name: "GFD", age: 21, m1: 82, m2: 44 },
];


let count = 0;
//Calculate average for students and display those students who got pass
student.forEach((stud, id) => {
if (stud.m1 > 45 && stud.m2 > 45) {
let tot = stud.m1 + stud.m2;
let avg = tot / 2;
console.log("student ", id + 1);
console.log("Name=", stud.name);
```

```
console.log("Age=", stud.age);
console.log("M1", stud.m1);
console.log("M2", stud.m2);
console.log("Average", avg);
count++;
}
});

// finally print the number of students who got pass
console.log("Total Number of students who got pass", count);
```

map

The map() method is an array helper method that creates a new array by calling a provided function on each element of the original array. It returns a new array with the result of the function calls. The syntax is given as:

```
const newArr =  array.map(callback(element, index, array), thisArg);
```

'array' : The original array you want to iterate over.
'callback': A function to execute on each element of the array.
'element': The current element being processed in the array
'index' (optional): The index of the current element being processes.
'array' (optional): The array that 'mapi s being applied to.
'thisArg' (optional): An object that will be passed as the 'this' value to the callback function. If not provided, 'undefined' will be used as the 'this' value.
Example: Create an array of cube of numbers

PREPARED BY DR.RAJALAKSHMI S 41

Examples

Create an array of cube of numbers

Module2 > JS 2_map1.js > ...

```
1  const numbers = [10, 20, 30, 40, 50];
2
3  const cubeNumbers = numbers.map(function (number) {
4    return number * number;
5  });
6
7  console.log(cubeNumbers);
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 We
[ 100, 400, 900, 1600, 2500 ]

[Done] exited with code=0 in 0.822 seconds
```

map( ) is used to create new array 'names' that contains only the names of the people from the 'people' array of objects.

```
1  const people = [
2    { name: "John", age: 30 },
3    { name: "Jane", age: 25 },
4    { name: "Bob", age: 35 },
5  ];
6
7  const names = people.map(function (person) {
8    return person.name;
9  });
10
11 console.log(names);
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pr
[ 'John', 'Jane', 'Bob' ]

[Done] exited with code=0 in 0.801 seconds
```

PREPARED BY DR.RAJALAKSHMI S 42

Example

```
Module2 > JS 2_map3.js > ...
1  const celciustemperature = [10, 20, 30, 40, 50];
2
3  const FahrenheitTemp = celciustemperature.map((temp) => (temp * 9) / 5 + 32);
4
5  console.log(FahrenheitTemp);
6
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
[ 50, 68, 86, 104, 122 ]

[Done] exited with code=0 in 0.884 seconds
```

map() is a powerful method of transforming data in arrays and creating new arrays based on the values of the original array. It serves as an alternative to 'forEach( )' when there is a need to create a new array with modified values from the original one.

Implicit return: If an arrow function has only one expression, omit the curly braces '{}' and the 'return' keyword. The result of the expression will be implicitly returned.

filter

It is an array helper method that creates a new array containing all the elements of the original array that pass a specific test (provided as a callback function). It filters out elements that do not meet the criteria defined in the callback function. The syntax is given as:

```
const  newArr = array.filter(callback(element, index, array), thisArg);
```

- 'array' : The original array you want to filter
- 'callback': A function that will be called for each element in the array
  - 'element': The current element being processes in the array
  - 'index' (optional): The index of the current element being processed
  - 'array' (optional); The array that 'filter' is being applied to
- 'thisArg (Optional): An object to which 'this' will be set inside the callback function

Examples

Filter even numbers in an array

```
Module2 > JS 2_filter1.js > ...
1  const numbers = [10, 11, 13, 14, 20, 22];
2
3  const evenNumbers = numbers.filter(function (num) {
4    return num % 2 == 0;
5  });
6
7  console.log(evenNumbers);
8
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220
[ 10, 14, 20, 22 ]

[Done] exited with code=0 in 0.135 seconds
```

Find the products less than 400

```
Module2 > JS 2_filter2.js > Newprod
3    { name: "Smartphone", price: 500 },
4    { name: "Tablet", price: 350 },
5    { name: "Headphones", price: 100 },
6  ];
7
8  // Find the products which are less than 400
9  const Newprod = products.filter(function (pname) {
10   return pname.price < 400;
11 });
12
13 console.log(Newprod);
14
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and Scripti
[ { name: 'Tablet', price: 350 }, { name: 'Headphones', price: 100 } ]

[Done] exited with code=0 in 0.126 seconds
```


Example - Filter method with arrow function

```
Module2 > JS 2_filter4.js > ...
1  const students = [
2    { name: "John", age: 20, grade: "A" },
3    { name: "Alice", age: 22, grade: "B" },
4    { name: "abc", age: 18, grade: "C" },
5    { name: "def", age: 19, grade: "A" },
6  ];
7
8  // Find the students who got 'A'
9  const students1 = students.filter((stud) => stud.grade == "A");
10
11 console.log(students1);
12
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pr
[
  { name: 'John', age: 20, grade: 'A' },
  { name: 'def', age: 19, grade: 'A' }
]
```

An array of objects named ‘ students’, where each object represents a student and contains their ‘name’, ‘age’ and ‘grade’.

Using the ‘filter( )’ method along with an arrow function, a new array called ‘students1’ is created that contains only the students with a grade of ‘A’.