reduce

The reduce( ) method reduces the elements of an array to a single value. It executes a provided callback function for each element of the array (from left to right) and accumulates the results into a single value. The syntax is given as:

```
const result= array.reduce(callback(collector, currentValue, index, array), initialValue);
```

'array': The original array that you want to reduce

'callback': A function that will be created for each element in the array

'collector': This parameter stores the accumulated value after each iteration. It starts with the 'initial value' if provided, otherwise with the first element of the array. The value returned by the callback in each iteration becomes the new value of the collector.

'current value': This parameter represents the current element being processes in the array

reduce with function

'index': This parameter (optional) is the index of the 'currentValue' in the array.

'array': This parameter (optional) is the orignial array that 'reduce' is being applied to. It's useful if you need to access the original array from within the callback.

'initialValue': This parameter(optional) is the initial value of the 'collector'. It's not required, but if provided, it will be starting value of the 'collector' in the first iteration

```
1  const student = [
2    { name: "John", score: 85 },
3    { name: "ABC", score: 82 },
4    { name: "Bbb", score: 90 },
5    { name: "ccc", score: 80 },
6  ];
7
8  function TotalScore(collect, student) {
9    return collect + student.score;
10 }
11
12 const ts = student.reduce(TotalScore, 0);
13 console.log("Total Score:",ts);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
Total Score: 337

[Done] exited with code=0 in 0.12 seconds
```

Example reduce with arrow

```
const students = [
  { name: "John", score: 85 },
  { name: "ABC", score: 82 },
  { name: "Bbb", score: 90 },
  { name: "ccc", score: 80 },
];

// Calculate the average score of all students
const totalScore = students.reduce(
  (collect, student) => collect + student.score,
  0
);
const averageScore = totalScore / students.length;

console.log("AverageScore = ",averageScore);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 2
AverageScore =  84.25

[Done] exited with code=0 in 0.136 seconds
```

The arrow function (collect, student) => collect + student.score is used as a callback. Here, collect takes on the role of an intermediate variable that holds the cumulative collect of the scores as we iterate through each student.

After the reduce() operation, the collect variable contains the total score of all students.

Here's how the process works:

Iteration 1: collect = 0, student.score = 85

collect after iteration: 0 + 85

Iteration 2: collect = 85, student.score = 82

collect after iteration: 167

Iteration 3: collect = 167, student.score = 90

collect after iteration: 257

Iteration 4: collect = 257, student.score = 80

collect after iteration: 337

PREPARED BY DR.RAJALAKSHMI S 51

Example: reduce with function expression

```
const students = [
  { name: "John", score: 85 },
  { name: "ABC", score: 82 },
  { name: "Bbb", score: 90 },
  { name: "ccc", score: 80 },
];

// Define a function expression for the callback
const TotalScore = function (collect, student) {
  return collect + student.score;
};
// Calculate the average score of all students
const ts = students.reduce(TotalScore, 0);
const averageScore = ts / students.length;
console.log("Average Score =", averageScore);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web P
Average Score = 84.25

[Done] exited with code=0 in 0.275 seconds
```

PREPARED BY DR.RAJALAKSHMI S 52

some

The function some() checks if atleast one element in the array passes the test implemented by the provided callback function. It returns a Boolean value (true if atleast one element passes the test, otherwise false.

```
const result= array. some(callback(element, index, array),thisArg);
```

- 'array' : The array need to check
- 'callback': a function that will be called for each element in the array until the condition is satisfied
- 'element': The current element being processed in the array
- 'index' (optional): The index of the current element being processed
- 'array' (optional): The array that 'some' is being applied to
- 'thisArg' (optional): An object to which 'this' will be set inside the callback function

Some - Example

The some() method is particularly useful when you need to determine if a certain condition is met by at least one element in the array. It stops iterating through the array as soon as it finds the first element that satisfies the condition, which can provide a performance advantage for large arrays.

```
1  const numbers = [10, 25, 7, 14, 30];
2
3  const res = numbers.some((number) => number > 20);
4
5  console.log(res);
6
```

```
[Running] node "f:\SRIHER\2
true

[Done] exited with code=0 i
```

Example

```
const students = [
  { name: "John", score: 85 },
  { name: "ABC", score: 82 },
  { name: "Bbb", score: 89 },
  { name: "ccc", score: 80 },
];

const highScore = students.some((student) => student.score >= 90);
if (highScore) {
  console.log("Atleast one student has mark greater than 90");
} else {
  console.log("No student has a high score");
}
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2
Scripting\Programs\Module2\2_some.
No student has a high score
```

PREPARED BY DR.RAJALAKSHMI S 55

every

`every()` method checks if all the elements in the array pass the test implemented by the provided callback function. It returns a Boolean value (true if all elements pass the test, otherwise false)

```
const result= array.every (callback(element, index, array),thisArg);
```

'array' : The array to check

'callback' : A function that will be called for each element in the array until the condition is not satisfied

'element' : The current element being processed in the array

'index' (optional): The index of the current element being processed

'array' (optional): The arrat that 'every is being applied to

'thisArg' (optional): An object to which 'this' will be set inside the callback function

PREPARED BY DR.RAJALAKSHMI S 56

Every - example

Create an array called students with 4 objects. The properties for each object are name and score which takes the value as specified below:

name	score
'JJJ'	85
'AAA'	92
'BBB'	78
'CCC'	95

Check whether all the student score is greater than or equal to 70

```
const students = [
  { name: "JJJ", score: 85 },
  { name: "AAA", score: 92 },
  { name: "BBB", score: 78 },
  { name: "CCC", score: 95 },
];

const allPassing = students.every((student) => student.score >= 70);

if (allPassing) {
  console.log("All students have passing scores.");
} else {
  console.log("Not all students have passing scores.");
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 W
All students have passing scores.

[Done] exited with code=0 in 0.138 seconds
```


every

Check whether elements in the array are positive

```
var numbers = [2, 0, 12, 14, -2];

var result = numbers.every((num) => num > 0);

if (result) {
  console.log("All numbers are positive");
} else {
  console.log("Not All numbers are positive");
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 2
Not All numbers are positive

[Done] exited with code=0 in 0.842 seconds
```

every() method is used to check if all elements in the array satisfy the specified condition. The callback function returns true if the condition is satisfied for all elements, and false if at least one element doesn't meet the condition.

find

The 'find()' method is used to search for an element in an array based on a specified condition. It returns the first element in the array that satisfies the given condition. If no element satisfies the condition, 'undefined' is returned. The syntax is given as:

```
const result= array.find (callback(element, index, array), thisArg);
```

'array' : The array to search
'callback': A function that will be called for each element in the array until a matching element is found
'element' : The current element being processed in the array
'index' (optional): The index of the current element being processed
'array' (optional)" The array that 'find' is being applied to
'thisArg'(optional) : An object to which 'this' will be set inside the callback function.

find - example

Find the first even number

```
1  const arr = [3, 4, 2, 5, 9];
2
3  const res = arr.find((n) => n % 2 == 0);
4
5  if (res) {
6    console.log("First Even number:", res);
7  } else {
8    console.log("no number found:", res);
9  }
10
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
First Even number: 4

[Done] exited with code=0 in 0.14 seconds
```

find - example

```
const fruits = ["apple", "banana", "cherry", "kiwi"];

const result = fruits.find((fruit) => fruit === "cherry");

if (result) {
  console.log("Found:", result);
} else {
  console.log("Not found");
}
```

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

[Done] exited with code=0 in 0.184 seconds
```

The find() method is used to search for an element in an array based on a provided condition. It returns the first element that satisfies the condition. If no element satisfies the condition, it returns undefined.

The find() method is used to find the first occurrence of the string "cherry" in the fruits array. The callback function checks if the current element is equal to "cherry", and when it finds the first element that matches, it returns that element. If no match is found, it returns undefined.

Keep in mind that find() stops searching as soon as it finds the first matching element. If you want to find multiple elements that match the condition, you can use the filter() method instead.

PREPARED BY DR.RAJALAKSHMI S 61

find - example

Create an array called students with 4 objects. The properties for each object are name and score which takes the value as specified below:

name	score
'JJJ'	85
'AAA'	92
'BBB'	78
'CCC'	95

Find the student with highest score

```
const students = [
  { name: "JJJ", score: 85 },
  { name: "AAA", score: 92 },
  { name: "BBB", score: 78 },
  { name: "CCC", score: 95 },
];

const highScorer = students.find((student) => student.score >= 90);

if (highScorer) {
  console.log("High scorer:", highScorer.name);
} else {
  console.log("No high scorer found.");
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 We
High scorer: AAA

[Done] exited with code=0 in 0.133 seconds
```

The find() method stops searching as soon as a matching element is found, and it returns that element. If no match is found, it returns undefined.

PREPARED BY DR.RAJALAKSHMI S 62

indexOf

'indexOf()' method can be used to find the index of the first occurrence of a specified element in an array. IF the element is not found in the array, it returns -1. The syntax is:

```
const result= array.indexOf(searchElement, fromIndex);
```

'array': The array to search in
'searchElement': The element to find in the array
'fromIndex' (optional): The index at which to start searching. If not specified, the search starts from the beginning of the array.

- indexOf() method doesn't work with arrow functions for comparison because it relies on reference equality.
- indexOf() only finds the index of the first occurrence. If you need to find the index of subsequent occurrences, you may need to use indexOf() with a custom fromIndex, or consider using other methods like findIndex() for more complex scenarios.

indexOf - example

```
1  const fruits = ["apple", "banana", "cherry", "apple", "kiwi"];
2
3  const index1 = fruits.indexOf("apple");
4  const index2 = fruits.indexOf("cherry");
5  const index3 = fruits.indexOf("pear");
6
7  console.log(`Index of apple : ${index1}`);
8  console.log(`Index of cherry : ${index2}`);
9  console.log(`Index of pear : ${index3}`);
10
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220
Index of apple : 0
Index of cherry : 2
Index of pear : -1

[Done] exited with code=0 in 0.165 seconds
```

findIndex

The findIndex() method is a built in JavaScript array method that searches for an element in an array based on a provided condition and returns the index of the first element that satisfies the condition.

```
const result= array.findIndex(callback(element, index, array), thisArg);
```

'array': The array one needs to search
'callback': A function that will be called for each element in the array until a matching element is found
'element': The current element being processed in the array
'index' (optional): The index of the current element being processed.
'array' (optional): The array that 'findIndex' is being applied to
'thisArg' (optional): An object to which 'this' will be set inside the callback function.

findIndex - example

Find the index of first number which is greater than 4.

```
1  const numbers = [2, 5, 8, 1, 4];
2
3  const index = numbers.findIndex((number) => number > 4);
4
5  if (index !== -1) {
6    console.log(`First number greater than 4 found at index ${index}`);
7  } else {
8    console.log("No number greater than 4 found.");
9  }
10
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
First number greater than 4 found at index 1

[Done] exited with code=0 in 0.144 seconds
```

findIndex - example

Find the index of the first occurrence of student whose name is ABC and score is 82

```
const students = [
  { name: "John", score: 85 },
  { name: "ABC", score: 82 },
  { name: "Bbb", score: 89 },
  { name: "ABC", score: 82 },
];
const searchStud = { name: "ABC", score: 82 };

const index1 = students.findIndex(
  (student) =>
    student.name === searchStud.name && student.score === searchStud.score
);
if (index1 != -1) {
  console.log(`Found at index ${index1}`);
} else {
  console.log(`Not found`);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
Found at index 1

[Done] exited with code=0 in 0.277 seconds
```

findIndex() method uses an arrow function as its argument. The arrow function compares both the name and score properties of each student object in the array with the searchStud object's properties.

lastIndexOf

The 'lastIndexOf() method is used to find the index of the last occurrence of a specified element in an array. If the element is not found in the array, it returns -1.

```
const result= array.lastIndexOf(searchElement, fromIndex);
```

'array' : The array to search in

'searchElement' : The element to find in the array

'fromIndex' (optional): The index at which to start searching from the end of the array. If not specified, the search starts from the last element.

lastIndexOf - example

To find the last occurrence

```
1  const fruits = ["apple", "banana", "cherry", "apple", "kiwi"];
2
3  const index1 = fruits.lastIndexOf("apple");
4  const index2 = fruits.lastIndexOf("cherry");
5  const index3 = fruits.lastIndexOf("pear");
6
7  console.log(`Index of apple : ${index1}`);
8  console.log(`Index of cherry : ${index2}`);
9  console.log(`Index of pear : ${index3}`);
10
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 22
Index of apple : 3
Index of cherry : 2
Index of pear : -1

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


Template Literals

A template literal, also known as a template string, is a feature in JavaScript that allows to embed expressions and variables directly into strings. This provides a more concise and flexible way to create strings, especially when need to include dynamic content or multiline text

Template literals are enclosed by backticks (``) instead of single or double quotes, which are used for regular strings. Inside template literals, placeholders can be used to include values dynamically

Example: `${expression}`

When the template literal is evaluated, these placeholders are replaced by the actual values of the expressions. Some of the advantages are: String Interpolation, Multi-line String, Expression Evaluation, Escaping Special Characters and Tagged template literals

String Interpolation

Variables and expressions can be directly embedded within a string

```
1   const name = "ABC";
2   let mark = 90;
3
4   // using string concatenation
5   let cs = "My name is :" + name + " and my mark is : " + mark;
6   console.log(cs);
7
8   // using Template Literal - String Interpolation
9   let ts = `My name is : ${name}  and my mark is : ${mark}`;
10  console.log(ts);
11
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
My name is :ABC and my mark is : 90
My name is : ABC  and my mark is : 90

[Done] exited with code=0 in 0.146 seconds
```

- In this example, we have a name variable and mark variable. We want to create a sentence that includes these values. Using string concatenation, we need to use the + operator to join the strings and variables.
- With template literals and string interpolation, we can directly include the variables within ${} placeholders in the string.

Multiline Strings

Template literals can span multiple lines without the need for concatenation or escaping line breaks.

```
1   const multilineString = `this is multi-
2   line string prepared using Template literal.
3   It does not use line break and
4   + for concatenation.`;
5
6   console.log(multilineString);
7   console.log("\n");
8   const stringliteral =
9     "This is normal string. \n" +
10    "It requires line break \n and " +
11    " + for concatenation.";
12
13  console.log(stringliteral);
14
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web
this is multi-
line string prepared using Template literal.
It does not use line break and
+ for concatenation.


This is normal string.
It requires line break
 and  + for concatenation.

[Done] exited with code=0 in 0.15 seconds
```

- **In Template literal:** the string content is enclosed within backticks (`), and it spans multiple lines naturally. The line breaks are preserved in the output without any additional effort.
- **In String Literal:** Each line of the string is separated by the + operator, and line breaks are indicated using the escape sequence \n. This approach is less readable and can become cumbersome when dealing with longer strings or more complex content.
- Template literals are recommended for creating multiline strings, as they provide a more concise and readable way to achieve the same result

Expression Evaluation

Expressions inside `${}` are evaluated and replaced with their results

```
1  const x = 5;
2  const y = 10;
3
4  console.log(`sum of ${x} and ${y} is ${x + y}`);
5
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 W
sum of 5 and 10 is 15

[Done] exited with code=0 in 0.143 seconds
```


Escaping Special Characters

To escape characters inside template literals, use the backslash '\' character before the character to escape.

Example: To escape backticks, dollar signs, backslashes, The escaped characters are treated as regular characters in the resulting string

```
1  const msg = `This is backticks \` and this is \$`;
2  const msg1 = `Line 1 \n Line 2 \n Indented line \t Line 3`;
3  console.log(msg);
4  console.log(msg1);
5
```

Template literals also support escape sequences like \n for new lines and \t for tabs

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 W
This is backticks ` and this is $
Line 1
 Line 2
 Indented line    Line 3

[Done] exited with code=0 in 0.143 seconds
```

Tagged Template Literals

A tagged template is an advanced feature of template literals in JavaScript. It allows to customize the way template literals are processed by using a function, called a tag function, that is applied to the template literal's parts and expressions. This enables to create powerful and flexible string manipulation functions. The syntax is given as

```
tagFunction `template literal parts ${expressions}`
```

`tagFunction` : A function that is applied to the template literal parts and expressions

'template literal parts': The parts of the template literal separated by expressions

'expressions': The values of expressions inside '${}' placeholders

Tagged Template Literals - Example

Template literals can span multiple lines without the need for concatenation or escaping line breaks.

```
function emphasize(strings, ...values) {
  console.log("Strings : ", strings); // Array of template literal parts
  console.log("values : ", values); // Array of expression values
}

const name = "ABC";
const age = 30;

emphasize`Hello, my name is ${name} and I am ${age} years old.`;

function highlight(strings, ...values) {
  let result = "";
  strings.forEach((string, index) => {
    result += string;
    if (index < values.length) {
      result += values[index].toString().toUpperCase();
    }
  });
  return result;
}

const ht = highlight`Hello, my name is ${name} and I am ${age} years old.`;

console.log(ht);
```

Tagged Template Literals - Example

Output

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Programming and
Strings :  [ 'Hello, my name is ', ' and I am ', ' years old.' ]
values :  [ 'abc', 30 ]
Hello, my name is ABC and I am 30 years old.

[Done] exited with code=0 in 0.143 seconds
```

- In this part of the code, tag function called highlight is defined. This function processes the template literal and creates a modified string. It iterates over the strings array and concatenates the string part with the corresponding value from the values array. If there is a corresponding value, it converts it to uppercase using .toUpperCase(). Then, it returns the modified string.
- The emphasize function shows the raw breakdown of template parts and values, while the highlight function demonstrates a more advanced use case of processing and modifying the template literal's content.

Rest Operators

The introduction of the rest parameter in JavaScript enables the capturing of multiple function arguments as an array. This proves valuable when dealing with a varying number of arguments and the need to handle them as a unified collection.

Syntax:

The rest parameter is denoted by three dots (`...`) followed by a parameter name. This parameter gathers all the remaining arguments passed to a function as an array

```
function myFunction(...args) {
  // 'args' is an array containing all arguments passed
}
```

- 'myFunction' is the name of the function
- '...args' is the rest parameter.
  - Inside the function, ...args is treated as like array that holds all the additional arguments passed to the function.

PREPARED BY DR.RAJALAKSHMI S 79

Rest Operators – Example - Sum of the elements

- The rest parameter can be used like any other array within the function. It contains all the arguments beyond the ones explicitly defined as individual parameters.

Restrictions:

- More than one rest parameter is not allowed
- A rest parameter must always come last.

In the below example, the parameters are parsed and resolved via rest functionality. This outputs,

```
function sum(...numbers) {
  let total = 0;
  for (let number of numbers) {
    total += number;
  }
  return total;
}

console.log(sum(1, 2, 3));
console.log(sum(10, 20, 30, 40));
```

```
PROBLEMS 2  OUTPUT  D
[Running] node "f:\SRIHER\2
6
100

[Done] exited with code=0 i
```

PREPARED BY DR.RAJALAKSHMI S 80

Combining Rest Parameter with Regular Parameter

- In a function declaration, regular parameters can be used alongside the rest parameter, but it's essential to ensure that the rest parameter appears after all the regular parameters.
- A rest parameter gathers all parameters following the first one and stores them in an array. Subsequently, each value in this array is multiplied by the initial parameter, and the resulting array is returned.

```
function multiply(multiplier, ...arg) {
  return arg.map((element) => multiplier * element);
}

const result = multiply(2, 5, 10, 15);
console.log(result);
```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONS
[Running] node "t:
Scripting\Programs
[ 10, 20, 30 ]

[Done] exited with
```

PREPARED BY DR.RAJALAKSHMI S 81

Enforcing a certain number of parameters via a rest parameter

Rest parameter can be used to enforce a certain number of arguments. Consider, the below example,

```
function sum_req_args(...numbers) {
  const req_arg_count = 3; // Change this to the number of req

  if (numbers.length !== req_arg_count) {
    throw new Error(`Exactly ${req_arg_count} arguments are re
  }

  const result = numbers.reduce((total, num) => total + num, 0
  return result;
}

try {
  const result = sum_req_args(2, 3, 4); // This will work with
  console.log("Result:", result);
} catch (error) {
  console.error("Error:", error.message);
}

try {
  const result = sum_req_args(2, 3); // This will throw an err
  console.log("Result:", result);
} catch (error) {
  console.error("Error:", error.message);
}
```

- A function named sum_req_args that uses a rest parameter ...numbers to accept any number of arguments as an array is defined. The purpose of this function is to calculate the sum of these numbers.
- Declare a constant variable named req_arg_count and assign it the value of 3. This value can be adjusted to specify the desired number of required arguments for this function.

PREPARED BY DR.RAJALAKSHMI S 82

Enforcing a certain number of parameters via a rest parameter

- The code assesses whether the length of the `numbers` array, which holds the provided arguments, doesn't correspond to the value stored in `req_arg_count`. When there's a mismatch, it signifies that the function was invoked with an incorrect number of arguments.
- If this condition is met (indicating an argument count that doesn't align with expectations), an error is thrown, featuring an error message specifying the precise count of necessary arguments.
- A try...catch block is used to call the sum_req_args function with different sets of arguments.
- `throw`: This is a JavaScript keyword used to explicitly throw an exception. When `throw` is used, it stops the normal execution of the code and transfers control to the nearest enclosing `catch` block, or if there isn't one, it terminates the script.
- new Error()`: This part of the statement creates a new instance of the built-in `Error` object. The `Error` object is a standard JavaScript object that represents an error or an exception. This is used to create instances of `Error` to capture and describe errors in the code.

Enforcing a certain number of parameters via a rest parameter

- `Exactly ${req_arg_count} arguments are required.``: This is a template literal. Template literals are enclosed in backticks (`` ` ``) and allow to embed expressions within `${}`. In this case, `${req_arg_count}` is an expression that evaluates to the value of the `req_arg_count` variable.

when this line of code is executed:

- It creates a new `Error` object with a custom error message that says "Exactly X arguments are required," where `X` is the value of the `req_arg_count` variable.
- Then, it throws this error, which can be caught and handled by a surrounding `catch` block or, if not caught, will result in the script terminating with an error message.
- The purpose of this line is to provide informative error messages when the function is called with an incorrect number of arguments, making it easier to understand what went wrong in the code.

Empty Rest Parameter

If no arguments are passed to a function that uses a rest parameter, the rest parameter will simply be an empty array.

```
1  function myFunction(...args) {
2    console.log(args);
3  }
4
5  myFunction();
6
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERM
[Running] node "f:\SRIH
Scripting\Programs\Modu
[]

[Done] exited with code=
```


Rest Parameter vs Arguments Object

The rest parameter is similar to the `arguments` object, but it has some advantages. Rest parameters are actual arrays, so you can use array methods and properties directly. The `arguments` object is an array-like object but array methods cannot be applied on that.

```
1  function example(...args) {
2    console.log(args);
3    console.log(arguments);
4  }
5
6  example("Cat", "Dog", "Rat");
7
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
Scripting\Programs\Module2\2_rest_array.js
[ 'Cat', 'Dog', 'Rat' ]
[Arguments] { '0': 'Cat', '1': 'Dog', '2': 'Rat' }

[Done] exited with code=0 in 0.987 seconds
```

Rest Parameter with arrays - Example

```
1  const numbers = [1, 2, 3, 4, 5];
2  const [first, second, ...rest] = numbers;
3
4  console.log(first); // 1
5  console.log(second); // 2
6  console.log(rest); // [3, 4, 5]
7
```

Output

Rest Parameter with array of objects

The rest parameter with objects allows to collect all the remaining properties of an object into a new object. It provides a convenient way to destructure objects while still retaining access to the properties didn't explicitly extract. The working is given as:

- **Destructuring:** When object destructuring is used, one must specify which properties to extract from an object and assign them to variables. Any properties not explicitly mentioned are typically left untouched.
- **Rest Parameter**: By using the rest parameter syntax (usually denoted by three dots `...` followed by a variable name), one can collect all the properties that were not explicitly destructured into a new object. This object contains the remaining properties.
- **Usage:** The rest parameter is especially useful to access and work with some properties immediately while keeping others for later use or for further processing.

Rest Parameter with objects

```
const person = { name: "Rajalakshmi", age: 30, city: "Chennai" };
const { name, ...info } = person;

console.log(name); // 'Rajalakshmi'
console.log(info); // { age: 30, city: 'Chennai' }

```

Output

```
Rajalakshmi
{ age: 30, city: 'Chennai' }
```


Spread Operators - Spread Operator for Arrays

The spread operator is another powerful feature introduced in ES6, and it works in a way as related to rest parameters. It's denoted by three dots (`...`) as well, but its purpose is to split an array or object into individual elements or properties.

When used with arrays, the spread operator can be used to create a new array by spreading the element of an existing array. The spread operator can be used to split an array into individual elements.

```
const numbers = [1, 2, 3];
const expandedNumbers = [...numbers, 4, 5];

console.log(expandedNumbers); // Outputs: [1, 2, 3, 4, 5]

```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220
[ 1, 2, 3, 4, 5 ]

[Done] exited with code=0 in 1.327 seconds
```

Spread Operators - Meging the array (concatenation)

The spread operator can be used to merge multiple arrays into one

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const mergedArray = [...arr1, ...arr2];

console.log(mergedArray); // Outputs: [1, 2, 3, 4, 5, 6]

```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMI
[Running] node "f:\SRIHER\2023 - 2024\CSE
[ 1, 2, 3, 4, 5, 6 ]

[Done] exited with code=0 in 1.169 second
```

PREPARED BY DR.RAJALAKSHMI S 91

Spread Operators - Clone the array

```
const arr1 = [10, 20, 30];
const arr2 = [...arr1];

console.log("After copying  arr2:", arr2);
arr2.push(40);
console.log("After Inserting 40 in arr2 : ", arr2);
console.log("Original array remains same arr1 : ", arr1);

```

Output

```
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Progr
After copying  arr2: [ 10, 20, 30 ]
After Inserting 40 in arr2 :  [ 10, 20, 30, 40 ]
Original array remains same arr1 :  [ 10, 20, 30 ]

[Done] exited with code=0 in 0.137 seconds
```

PREPARED BY DR.RAJALAKSHMI S 92

Spread Operators - Using spread in function calls

The spread operator can be used to pass the elements of an array as individual arguments to a function.

```
function sum(a, b, c) {
  return a + b + c;
}

const values = [10, 20, 30];
const result = sum(...values);

console.log("Result = ", result); // Outputs: 60
```

Math Object does not work on single array. With the help of spread operator, the array is broken into several individual elements.

```
Module2 > JS 2_spread_array.js > ...
let numbers = [11, 12, 800, -3, 0, -900, -100];
//console.log(Math.min(numbers)); //output NaN
console.log(Math.min(...numbers)); // Output -900
```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TER
[Running] node "f:\SRIHER\2023 - 2024\(
Result =  60

[Done] exited with code=0 in 1.118 sec
```

```
[Running] node "f:\SRIHER\2023 - 2024\CSE
-900

[Done] exited with code=0 in 0.149 second
```


Spread operator with Objects

The spread operator can also be used to spread the properties of an object into another object. It's useful for creating new objects based on existing ones.

```
Module2 > JS 2_spread_obj.js > ...
const person = { name: "Rajalakshmi", age: 30 };
const updatedPerson = { ...person, age: 31, location: "Chennai" };

console.log(updatedPerson); // Outputs: { name: 'Rajalakshmi', age: 31, location: 'Chennai' }
```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Progr
{ name: 'Rajalakshmi', age: 31, location: 'Chennai' }

[Done] exited with code=0 in 0.143 seconds
```

Concatenate or merge objects using spread parameter.

It is also possible to concatenate or merge objects using spread parameter.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const merged = { ...obj1, ...obj2 };

console.log(merged); // { a: 1, b: 2, c: 3, d: 4 }

```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 22
{ a: 1, b: 2, c: 3, d: 4 }

[Done] exited with code=0 in 0.152 seconds
```


Combining Rest with Spread

The spread operator can be combined with rest parameters to split and gather elements simultaneously.

```
function example(first, ...rest) {
  console.log("First Number: ", first); // First element
  console.log("Rest of the numbers", rest); // Array of remaining elements
}

const numbers1 = [1, 2, 3, 4, 5];
example(...numbers1); // Outputs: 1, [2, 3, 4, 5]

```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 2
First Number:  1
Rest of the numbers [ 2, 3, 4, 5 ]

[Done] exited with code=0 in 0.13 seconds
```

Default Function Parameters

In JavaScript, default function parameters allow to specify default values for parameters directly in the function's parameter list. This can make the code more concise and improve the readability of the functions.

Working of default parameters

Declaration with default values:

It is possible to declare default parameter values directly within the function's parameter list using the assignment operator (`=`). When the function is called, if a value is not provided for that parameter or if `undefined` is passed, the default value will be used instead.

```
function sayHello(name = "Rajalakshmi", greeting = "Welcome") {
  console.log(`${greeting}, ${name}!`);
}
```


Declaration with default values - Usage

When a function with default parameters is called, it is possible to omit those parameters or explicitly pass `undefined` to indicate that the default value should be used.

```
function sayHello(name = "Rajalakshmi", greeting = "Welcome") {
  console.log(`${greeting}, ${name}!`);
}

sayHello(); //empty function call displays Welcome, Rajalakshmi!
sayHello("Priya"); //Single param displays Welcome, Priya!
sayHello("Sree", "Hi"); //Override and displays Hi, Sree!
sayHello("Kumar", ""); //Empty string displays , Kumar!

```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 22
Welcome, Rajalakshmi!
Welcome, Priya!
Hi, Sree!
, Kumar!

[Done] exited with code=0 in 0.138 seconds
```

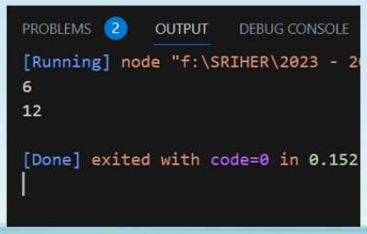
Evaluation of Default values

Default parameter values are evaluated at the time the function is called, not when the function is defined. This means that any expressions used as default values are evaluated each time the function is invoked.

Example

```
function increment(value, step = 1) {
  return value + step;
}

console.log(increment(5)); // Outputs: 6
console.log(increment(10, 2)); // Outputs: 12

```

Output



PREPARED BY DR.RAJALAKSHMI S 99

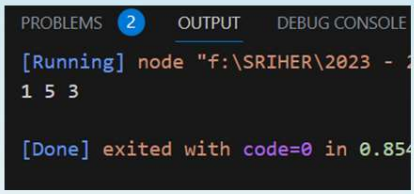
Interaction with other Parameters

Default parameters can interact with other parameters in the function. Parameters with default values must come after parameters without default values in the parameter list.

Example

```
function example(a, b = 5, c) {
  console.log(a, b, c);
}

example(1, undefined, 3); // Outputs: 1 5 3

```

Output



PREPARED BY DR.RAJALAKSHMI S 100

Enhanced Object Literals

An object literal is a way to define a new object using a concise syntax directly within the code. It allows to create an object with properties and values within the curly braces. Object literals are a fundamental part of JavaScript and are commonly used for creating data structures, defining configuration settings, and more. Object is created with key: value separated by comma. Enhanced object literals are a set of features introduced in ES6 (ECMAScript 2015) that provide more concise and flexible ways to define and work with objects in JavaScript. These features enhance the readability and expressiveness of object literals. The characteristics are listed as:

- Shorthand property names
- Computed Property names
- Method definitions
- Object Property Assignment
- Object property names in method definitions
- Object Property shortcuts in Methods

PREPARED BY DR.RAJALAKSHMI S 102

Shorthand property names

When the property name and variable name are the same, you can omit the duplicate declaration.

```
1   //Shorthand Property Names
2   const name = "Rajalakshmi";
3   const age = 30;
4
```

The traditional way of representing the object literal is given as:

```
// Traditional object literal
const person = {
    name: name,
    age: age
};
```

Enhanced object literal is represented as:

```
// Enhanced object literal
const enhancedPerson = {
    name,
    age
};
```


Computed Property names

Square brackets `[]` can be used to compute property names dynamically.

```
//Computed Property Names
const propName = "language";

// Traditional object literal
const programming = {};
programming[propName] = "JavaScript";

// Enhanced object literal
const enhancedProgramming = {
  [propName]: "JavaScript",
};
```

Shorthand notation for Method Definitions

The syntax for creating the method in traditional (ES5) and ES6 is shown as:

Traditional way of creating object method:

```
const calculator = {
  add: function (a, b) {
    return a + b;
  },
};
console.log(calculator.add(5, 2));
```

In enhanced object literal notation, the object method is represented as:

```
// Enhanced object literal
const enhancedCalculator = {
  add(a, b) {
    return a + b;
  },
};
console.log(enhancedCalculator.add(7, 7));
```


Object Property Assignment

The properties from one object to another can be copied is represented below.

```
const defaults = { theme: "light", fontSize: 16 };
const userSettings = { fontSize: 18 };

// Traditional object literal
const mergedSettings = Object.assign({}, defaults, userSettings);

// Enhanced object literal
const enhancedMergedSettings = { ...defaults, ...userSettings };
```

Object Computed Property Names in Method Definitions

The syntax for specifying computed property names in method definitions is specified as:

```
const propName1 = "Action2"; // this should appear before using propName1
const actions = {
  action1() {
    console.log("Action 1");
  },
  [propName1]() {
    console.log(`Action for ${propName1}`);
  },
};

actions.action1();
actions[propName1]();
```


Object Computed Property Names in Method Definitions - Output

A constant variable named `propName1` is declared with the value `"Action2"`. It's important that `propName1` is defined before it is used as a property name in the `actions` object.

```
Action 1
Action for Action2
```

An object named `actions` is created with two methods:

- `action1()`: This method logs the message "Action 1" when it's called.
- `[propName1]()`: This is a method with a dynamic property name, which is determined by the value of `propName1`. When this method is called, it logs a message using the value stored in `propName1`.

To call the `action1` method, you use the syntax `actions.action1();`. This results in the message "Action 1" being displayed in the console.

To call the method with the dynamic property name, square bracket notation is used: `actions[propName1]();`. Since `propName1` contains the value "Action2," this will display "Action for Action2" in the console.

Object property names in method

The object properties can be referenced in the method as stated below:

```
// Define a msg variable
var msg = "Welcome";

// Create an object with properties and a method
var greeter = {
  msg: msg,

  greetings: function (name) {
    console.log(this.msg + ", " + name);
  },
};
// Call the greetings method
greeter.greetings("SREE"); // Output: "Welcome, SREE"
```

- An object named `greeter` is created.
- One of the properties is `msg`, which is expected to hold the value "Welcome"
- The object also has a method named `greetings`. This method takes a single parameter called `name`.
- When the `greetings` method is called with the argument "SREE", it logs a message to the console. The message is constructed by concatenating the value stored in the `msg` property with the `name` parameter.

Object property names in method

The object properties can be referenced in the method as stated below:

```
// Create an object using enhanced object literal notation
const Egreeter = {
  msg, // Property shorthand: Equivalent to msg:msg

  greetings(name) {
    console.log(`${this.msg}, ${name}`);
  },
};
// Call the greetings method
Egreeter.greetings("ABC"); // Output: "Welcome, ABC"
```

- An object called `Egreeter` is created using enhanced object literal notation.
- Inside the `Egreeter` object, there's a property named `msg`. This property uses property shorthand, which means that it is equivalent to `msg: msg`
- The `Egreeter` object also contains a method named `greetings`, which takes a parameter called `name`.
- When the `greetings` method is called with the argument "ABC", it logs a message to the console. The message is constructed using a template string, combining the value stored in the `msg` property with the provided `name` parameter.

Object property names in method - Example

In the context of an online store application, an `onlineStore` object has been created with specific functionalities:

- The `addProduct` method enables the addition of products to the shopping cart. It requires two arguments: `productName` and `productPrice`.
- The `calculateSubtotal` method computes the total cost of all products in the shopping cart.
- The `applyCoupon` method applies discounts to the total based on provided coupon codes. Valid codes include `SALE10` for a 10% discount, `SPECIAL20` for a 20% discount, and `HALFPRICE` for a 50% discount. In case of an invalid code, it returns an "Invalid coupon code" message.

Three products have been added to the cart: a laptop priced at $800, headphones priced at $100, and a smartphone priced at $600. The objective is to apply the "SPECIAL20" coupon code and determine the discounted total.

1. Compute and present the subtotal of the shopping cart before any coupon is applied.
2. Implement the "SPECIAL20" coupon code and reveal the final cost after applying the discount.
3. In the event that an invalid coupon code is provided, display the message "Invalid coupon code."

Object property names in method - Example

Provide the code to execute these tasks and specify the expected output for each task.

```
const onlineStore = {
  products: [],

  addProduct(productName, productPrice) {
    const product = { name: productName, price: productPrice };
    this.products.push(product);
  },

  calculateSubtotal() {
    let subtotal = 0;
    for (const product of this.products) {
      subtotal += product.price;
    }
    return subtotal;
  },
```

```
applyCoupon(code) {
  const couponCodes = {
    SALE10: 10, // 10% discount
    SPECIAL20: 20, // 20% discount
    HALFPRICE: 50, // 50% discount
  };

  if (couponCodes.hasOwnProperty(code)) {
    const discountPercentage = couponCodes[code];
    //console.log(code);
    //console.log(couponCodes[code]);
    const subtotal = this.calculateSubtotal();
    const discountAmount = (discountPercentage / 100) * subtotal;
    const discountedTotal = subtotal - discountAmount;
    return discountedTotal;
  } else {
    return "Invalid coupon code";
  }
},
```

Object property names in method - Example

Provide the code to execute these tasks and specify the expected output for each task.

```
};

onlineStore.addProduct("Laptop", 800);
onlineStore.addProduct("Headphones", 100);
onlineStore.addProduct("Smartphone", 600);

const couponCode = "SPECIAL20";
const discountedTotal = onlineStore.applyCoupon(couponCode);

console.log("Subtotal before coupon:", onlineStore.calculateSubtotal());
if (typeof discountedTotal === "number") {
  console.log(
    `Total cost after applying ${couponCode} coupon:`,
    discountedTotal
  );
} else {
  console.log(discountedTotal); // Invalid coupon code message
}
```

Output

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL
[Running] node "f:\SRIHER\2023 - 2024\CSE 220 Web Pr
Subtotal before coupon: 1500
Total cost after applying SPECIAL20 coupon: 1200
```


ES6 Aray and Object Destructuring - Creating Arrays

Using Array Literal Notation (Square Brackets):

This is the most common and straightforward way to create an array.

```
const fruits = ['apple', 'banana', 'cherry'];
```

Using the `Array` Constructor:

```
const fruits = new Array('apple', 'banana', 'cherry');
```

Creating an Empty Array:

```
const emptyArray = [];
```

Creating Arrays with a Specific Length:

```
const arrayWithLength = new Array(3); // Creates an array with 3 undefined elements
```


ES6 Aray and Object Destructuring - Accessing Arrays

An array called fruits is created with three elements apple, banana and cherry.

```
const fruits = ["apple", "banana", "cherry"];
```

Using array indexing, it is possible to access the element of the array

```
// Access the first element (index 0)
const firstFruit = fruits[0]; // 'apple'

// Access the second element (index 1)
const secondFruit = fruits[1]; // 'banana'

// Access the third element (index 2)
const thirdFruit = fruits[2]; // 'cherry'

// Accessing elements out of bounds will return undefined
const outOfBounds = fruits[3]; // undefined

// using variables or expressions as indices
const index = 1;
const selectedFruit = fruits[index]; // 'banana'
```

```
// Access the last element using negative indexing
const lastFruit = fruits[-1]; // 'cherry'

// Access the second-to-last element using negative indexing
const secondToLastFruit = fruits[-2]; // 'banana'

// Access the first element using negative indexing (equivalent to [0])
const firstFruit = fruits[-fruits.length]; // 'apple'
```

Negative indexing can be useful when you want to access elements from the end of the array without knowing the exact length of the array.

Modify array elements

Array elements can be modified by using the assignment operator (=) to assign a new value to a specific index of the array.

```
// Modify the second element (index 1)
fruits[1] = "grape";

console.log(fruits); // ['apple', 'grape', 'cherry']
```

The second element of the array has been changed to grape from banana.

It is also possible to modify the array elements using any valid expression which is specified as:

```
const numbers = [1, 2, 3, 4];

// Double the value of the third element (index 2)
numbers[2] = numbers[2] * 2;

console.log(numbers); // [1, 2, 6, 4]
```


Length of the array

To find the length of the array, one can use length property of the array.

```
// Length of the array
const Arr = [1, 2, 3, 4, 5];
const arrayLength = Arr.length;

console.log("The length of the array is: " + arrayLength); //outputs 5
```

arr.length returns the number of elements present in the array, and it is stored in the arrayLength variable.

Add elements to the array

`push`: To add an element to the end of the array

```
const myArray = [1, 2, 3];
myArray.push(4); // Adds 4 to the end of the array

console.log("After inserting 4 : ", myArray);
```

`unshift`: To add an element to the beginning of the array, one can use the 'unshift' method

```
const myArray_us = [2, 3, 4];
myArray_us.unshift(1); // Adds 1 to the beginning of the array

console.log("After inserting 1 : ", myArray_us);
```

`concat`: To add elements from another array to the end of the array

```
const myArray_c = [1, 2, 3];
const newArray_c = myArray_c.concat([4, 5]); // Adds 4 and 5 to the end of myArray

console.log("After inserting one array at the end of other : ", newArray_c);
```


Deleting elements in array

`delete`: To delete an element use the `delete` keyword followed by the array element to delete. However, `delete` keyword doesn't actually remove the element; it sets it to `<1 empty item> `, and it leaves a gap in the array.

```
const myArray_d = [1, 2, 3, 4, 5];
// Delete the first element (index 0)
delete myArray_d[0];
console.log(myArray_d); // The array will have a gap at index 0: [<1 empty item>, 2, 3, 4, 5]
console.log("After deleting first element : ", myArray_d);
```

In this example, `**delete myArray[0]**` removes the element at index 0 (which is 1), but it leaves `**<1 empty item>** ` in its place, and the array still retains the same length. To remove an element and also remove the gap, one should use the `**splice**` method

Deleting elements in array

`splice`: The `splice` method in JavaScript is used to change the contents of an array by removing or replacing existing elements and/or adding new elements in place. Its syntax is as follows:

```
array.splice(start, deleteCount, item1, item2, ...);
```

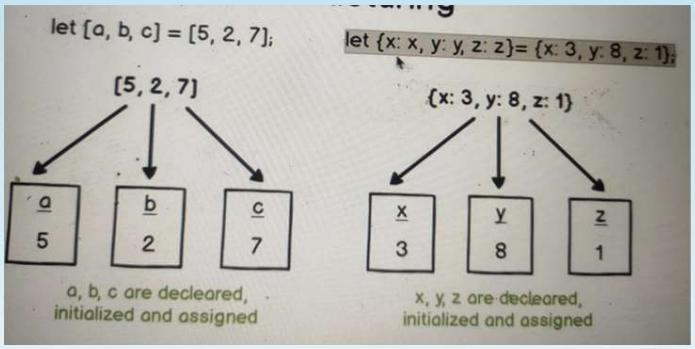
- `array`: The array to modify.
- `start`: The index at which to start changing the array. If negative, it counts from the end of the array. For example, -1 refers to the last element.
- `deleteCount`: An optional integer that specifies the number of elements to remove from the `start` index. If omitted, all elements from `start` to the end of the array are removed.
- `item1`, `item2`, ...: Optional items to add to the array, starting from the `start` index.

Deleting elements in array - Example

```
const myArray_sp = [1, 2, 3, 4, 5];

// Remove elements starting from index 2 (inclusive) and remove 2 elements
myArray_sp.splice(2, 2); // Resulting array: [1, 2, 5]
console.log(myArray_sp);

// Add elements starting from index 2 (inclusive) without removing any
myArray_sp.splice(2, 0, 6, 7); // Resulting array: [1, 2, 6, 7, 3, 4, 5]
console.log(myArray_sp);

// Replace elements starting from index 2 (inclusive) and remove 2 elements
myArray_sp.splice(2, 2, 8, 9); // Resulting array: [ 1, 2, 8, 9, 5 ]
console.log(myArray_sp);
```

Array Destructuring

Destructuring the array in JavaScript simply means extracting multiple values from data stored in objects and arrays. The destructing assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.



Array Destructuring - Examples

Example 1 - Declaration and assignment using array destructuring

```
const myArray = [1, 2, 3];
/* let a, b, c;
a = Myarray[0];
b = Myarray[1];
c = Myarray[2];
 */
let [a, b, c] = myArray;
console.log(a, b, c); // Output: 1 2 3
[b, a, c] = myArray; // order can be changed
console.log(a, b, c); //Output:2 1 3
```

Example 2 - Assignment using array destructuring

```
const myarray = [1, 2, 3];
let a1, b1, c1;
[a1, b1, c1] = myarray;

const aarray = [4, 5, 6];

[a1, b1, c1] = aarray;

console.log(a1, b1, c1); //Output:4 5 6
```

Array Destructuring - Examples

Example 3 - More variables than elements in the array

```
const myArray1 = ["a"];
const [a2, b2, c2] = myArray1;
console.log(a2, b2, c2); //Output:a undefined
undefined
```

Example 4 - Default values

```
const myArray4 = ["a"];
const [a4, b4, c4 = "c"] = myArray4;
console.log(a4, b4, c4); //Output:a undefined c
```

Example 5 - Skip element during destructuring

```
const myArray5 = [1, 2, 3, 4, 5];
const [, , a5, , b5] = myArray5;
console.log(a5, b5); //Output:3 5
const [, , a6, , b6] = myArray;
console.log(a6, b6); //Output:3 undefined
```

PREPARED BY DR.RAJALAKSHMI S 125

Array Destructuring - Examples

Example 6 - Rest Operator in array destructuring

```
const myArray6 = [1, 2, 3, 4, 5];
const [a7, b7, ...rest7] = myArray6; // Rest Operator
console.log(a7, b7, rest7); //Output: 1 2 [ 3, 4, 5 ]
const d = [...rest7]; // spread operator
console.log(d); //Output: [ 3, 4, 5 ]
```

Example 7 - Delete first element

```
let myArray7 = [1, 2, 3];
const [, ...aa7] = myArray7;
console.log(aa7); //Output: [ 2, 3 ]
```

Example 8 - Swap values

```
let x = 5,
y = 10;
[y, x] = [x, y];
console.log(x, y); // Output:10,5
```

Example 9 - Destructuring in the function

```
const myPosts = [
 ["Post1", 10],
 ["Post2", 20],
];
myPosts.forEach((title, likes) => console.log(`${title}
has ${likes} likes`));
```

PREPARED BY DR.RAJALAKSHMI S 126

Array Destructuring - Examples

Example 10 - Nested array Destructuring

```
const myArray9 = [1, 2, [3, 4]];
const [a9, b9, [c9, d9, e9]] = myArray9;
console.log(a9, b9, c9, d9, e9); //Output: 1 2 3 4
undefined

const processQuantitites  = (/*parameters*/) => {
 console.log(minQty);  // 8
 console.log(maxQty);  // 29
 console.log(defaultQty);  //0
 return maxQty - minQty; // returns 21
};
```

```
const qtyRange = [8, 29];
const sol = processQuantities(qtyRange);
console.log(sol);
```


Exercise 2

Modify parameters section in the ProcessQuantities function to match console.log outputs

```
const processQuantitites = (/*parameters*/) => {
  console.log(minQty); // 8
  console.log(maxQty); // 29
  console.log(defaultQty); //0
  return maxQty - minQty; // returns 21
};
const qtyRange = [8, 29];
const sol = processQuantities(qtyRange);
console.log(sol);
```

Console output

```
8
29
0
21
```

Object De-structuring

Example 1 - Declaration and assignmnet using object destructuring

```
const myObject = {
  a: 10,
  b: true,
};
// const a = myObject.a;
// const b = myObject.b;

const { a: a, b: b } = myObject; // lhs is property of object rhs is variable name

console.log(a, b); // 10 true
```

Example 2 - Declaration and assignment using object destructuring and shorthand property names

```
const myObject1 = {
  an: 10,
  bn: true,
};
const { an, bn } = myObject1;
console.log(an, bn); // Output: 10 true
```


Object De-structuring

Example 3: Assignment using Object destructuring and shorthand property names

```
let a1, b1;

//{a1,b1} = myObject1; // Error Declaration or statement expected
/*
Because javaScript considers { } as block rather than object literal
*/
// solution is wrap it inside ()
({ a1, b1 } = myObject1);
console.log(a1, b1); //Output: 10 true
```

Object De-structuring

Example 4 - Destructure non - object value

```
//const {a2,b2} = null; // Uncaught TypeError: cannot destructure property 'a' of undefined or null

const val = undefined;
const { a2, b2 } = val || {};
console.log(a2, b2); // Output: undefined undefined

const myArray = [1, 2, 30, 40, 50];
const { length, d } = myArray; // Arrays are objects
// length is the default property of array, it says length of the array
// {0:1, 1:2, 2:30, 3:40, 4:50}
console.log(length, d); //output: 5 undefined
```


Object De-structuring

Example 5 - Change the order of properties

```
const myObject5 = {
  a5: 10,
  b5: true,
};
// Order of the property is not important while destructuring
const { b5, a5 } = myObject5;
console.log(a5, b5); // Output: 10 true
```

Example 6 - Rest operator in Object Destructuring

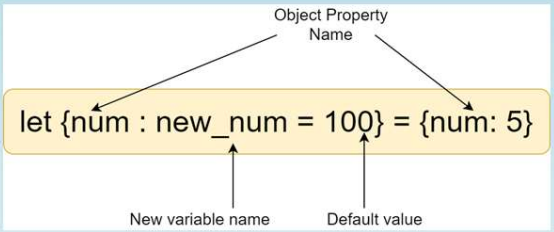
```
const myObjectR = {
  aR: 10,
  bR: true,
  c: [],
  d: "abc",
  e: 20,
};
console.log(myObjectR);
const { aR, bR, ...rest } = myObjectR;
console.log(aR, bR, rest); // 10 true { c: [], d: 'abc', e: 20 }
```

Example 7 - Destructuring Non - existing properties

```
const myObjectD = {
  aM: 10,
  bM: true,
};
// Since cM is not the property of myObjectD it prints undefined
// const { aM, bM, cM } = myObjectD;
// console.log(aM, bM, cM);  // 10 true undefined
// it is possible to specify default value for cM
const { aM, bM, cM = 100 } = myObjectD;
console.log(aM, bM, cM); // 10 true 100
```

Renaming variables

The extracted values are assigned to variables with different names using a colon (`:`) followed by the new variable name.



- It looks for the property name "num" in the object literal on the right – hand side of the assignment
- If property "num" is present, take its "value"
- If property "num" is absent, "value" will be "undefined"
- If "value" is "undefined" and default value is present for the property name "num" on the left-hand side of the assignment, assign default value 100 to the "value"
- New variable "newNum" will be declared, initialized and assigned value "value"

Object De-structuring

Example 8 - Default values and new variable names

```
const myObjectDV = {
  x: 10,
  y: true,
};
const { x: newX, y: newY, z: newZ = "default" } = myObjectDV;
console.log(newX, newY, newZ); //10 true default
```

Example 9: Nested object destructuring

```
const obj1 = {
  u: 1,
  v: 2,
  nestedobj: {
    w: 3,
    w1: 4,
  },
};
// Solution - 1
// const { u: u, v: v, nestedobj: nestedobj } = obj1;
// const { w: w, w1: w1 } = nestedobj;
// Alternatively, Solution - 2
const {
  u: u,
  v: v,
  nestedobj: { w: w, w1: w1 },
} = obj1;
console.log(u, v, w, w1); // 1 2 3 4
```

Exercise 1

- Modify the "personInfo" function to match console.log output.
- Object that is returned by "personInfo" function must contain only shorthand property names. Use the concept of Object destructuring wherever required

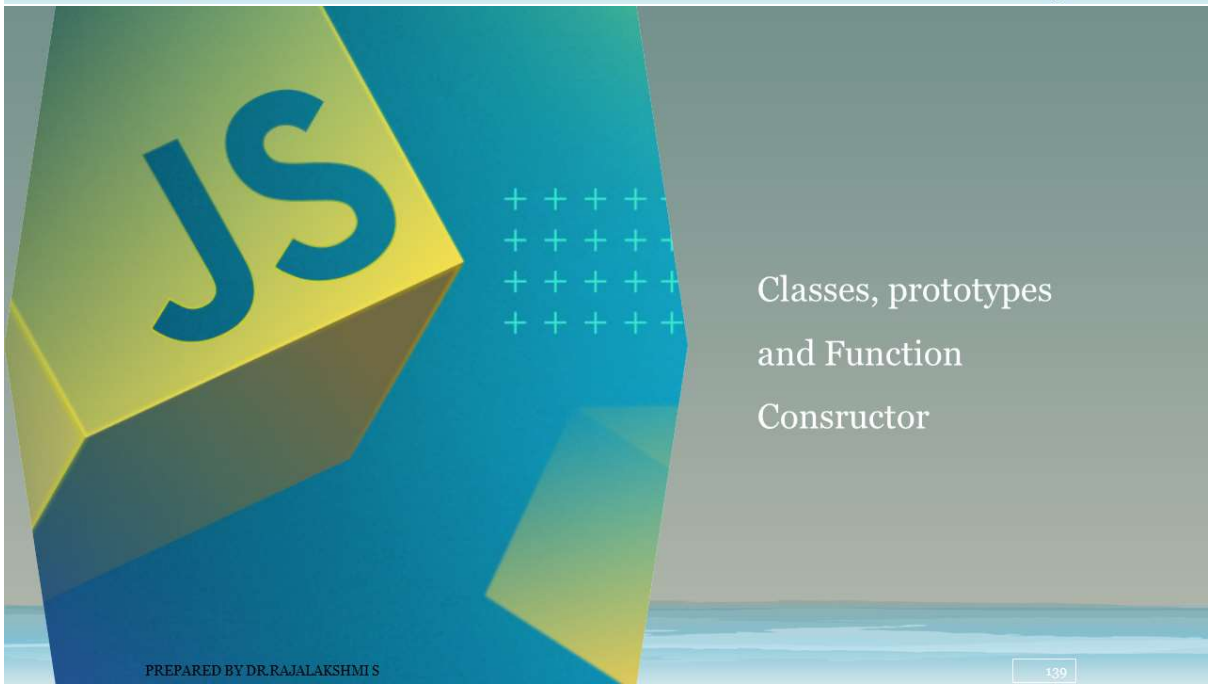
```
const personInfo = (/*parameters*/) => {
  /* return */
};

const person = {
  name: "ABC",
  age: 19,
  location: {
    country: "India",
    city: "Chennai",
  },
};
console.log(personInfo(person));
```

```
{
  name: 'ABC',
  personAge: 19,
  sourceCountry: 'India',
  homecity: 'Chennai',
  interestedIn: 'Learning',
  yearOfStudy: 2
}
```

Solution

```
const personInfo = ({
  name: name,
  age: personAge,
  location: { country: sourceCountry, city: homecity },
  interestedIn = "Learning",
  yearOfStudy = 2,
}) => {
  /* return */
  return {
    name,
    personAge,
    sourceCountry,
    homecity,
    interestedIn,
    yearOfStudy,
  };
};
```

PREPARED BY DR.RAJALAKSHMI S 136



PREPARED BY DR.RAJALAKSHMI S 139

Classes, prototypes and Function Consructor

Class based

C++
Java

Prototype based

JavaScript

ES6 Classes

- ES6 introduced the concept of classes to JavaScript, providing a more structured and object-oriented way to define and create objects.
- Classes allow you to define blueprints for creating objects with shared properties and methods, making your code more organized and easier to maintain.
- They provide a cleaner and more intuitive way to work with object-oriented programming concepts and provides support for inheritance and encapsulation.

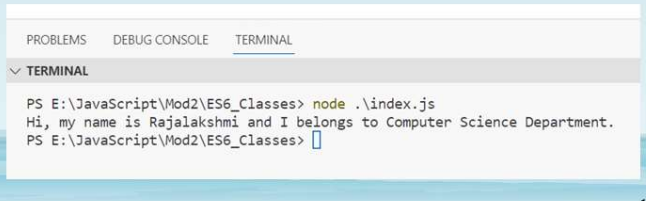
Syntax

```
class <class name> {
  constructor (a, b) {
  }
  function() {
  }
}

const obj = new <class name>('a', 'b');
obj.function();
```

ES6 Classes - Example

```
class Students {
  constructor(name, dep) {
    this.name = name;
    this.dep = dep;
  }

  studDetail() {
    console.log(`Hi, my name is ${this.name} and I belongs to
${this.dep} Department.`);
  }
}

const student1 = new Students('Rajalakshmi', 'Computer Science');
student1.studDetail(); // Output: Hi, my name is Rajalakshmi and I
belongs to Computer Science Department.
```

- The `class` keyword is used to define the `Students` class.
- The `constructor` method is a special method that gets called when an object of the class is created. It initializes the object's properties.
- The `studDetail` method is a regular method defined in the class. Instances of the `Students` class will inherit this method.
- An instance of the `Students` class is created using the `new` keyword.



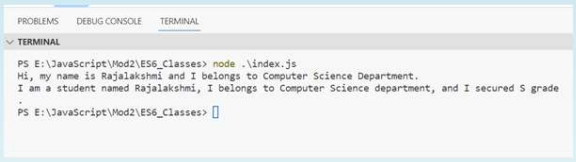
PREPARED BY DR.RAJALAKSHMI S 142

ES6 Classes - Inheritance

ES6 classes also support inheritance. You can create a subclass that inherits properties and methods from a parent class:

```
class Exam extends Students {
   constructor(name, dep, grade) {
     super(name, dep);
     this.grade = grade;
   }

   result() {
     console.log(`I am a student named ${this.name}, I
belongs to ${this.dep} department, and I secured
${this.grade} grade.`);
   }
 }

const exam1 = new Exam('Rajalakshmi', 'Computer
Science', 'S');
exam1.result(); // Output: I am a student named
Rajalakshmi, I belongs to Computer Science department,
and I secured S grade.
```

The 'Exam' class extends the 'Students' class using the 'extends' keyword. The 'super()' function is used to call the constructor of the parent class.



PREPARED BY DR.RAJALAKSHMI S 143

ES6 Prototypes

- In JavaScript, ES6 prototypes refer to a mechanism that allows objects to inherit properties and methods from other objects.
- This forms the basis of object-oriented programming in JavaScript and helps create a relationship between objects in a more memory-efficient way compared to traditional class-based inheritance.
- In JavaScript, every object has an internal property known as its prototype.
- Prototypes are used to implement inheritance in a more dynamic manner compared to classical inheritance found in languages like Java or C++.
- When a property or method is accessed on an object, JavaScript looks up the prototype chain to find the property or method if it's not directly present on the object itself.

<FunctionName>.prototype.<prototype function name>

PREPARED BY DR.RAJALAKSHMI S 144

ES6 Prototypes - Example

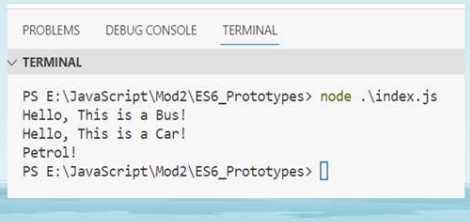
```
// Define a base constructor function
function Vehicle(name) {
  this.name = name;
}

// Add a method to the prototype of the base constructor
Vehicle.prototype.vehType = function() {
  console.log(`Hello, This is a ${this.name}!`);
};

// Create an instance of Vehicle
const vehicle = new Vehicle('Bus');
vehicle.vehType();  // Output: Hello, This is a Bus!

// Define a derived constructor function
function Car(name, brand) {
  // Call the base constructor using 'call' to set 'this'
  Vehicle.call(this, name);
  this.brand = brand;
}
```

```
// Set up prototype inheritance
Car.prototype = Object.create(Vehicle.prototype);
Car.prototype.constructor = Car;

// Add a method specific to Car
Car.prototype.fuel = function() {
  console.log('Petrol!');
};

// Create an instance of Car
const car = new Car('Car', 'BMW');
car.vehType();  // Output: Hello, This is a Car!
car.fuel();     // Output: Petrol!
```

PREPARED BY DR.RAJALAKSHMI S 145

ES6 Prototypes - Example

- 'Vehicle' is a constructor function that sets the 'name' property and defines the 'vehType' method on its prototype.
- 'Car' is a constructor function that inherits from 'Vehicle'. It calls 'Vehicle' using `call` to set its own properties and sets up the prototype chain using `Object.create`.
- Instances of 'Vehicle' and 'Car' can access properties and methods from both their own prototypes and their ancestor prototypes.
- The `vehType` method is defined on `Vehicle`'s prototype, and the `fuel` method is defined on `Car`'s prototype.



Class based vs. Prototype based

Class Based	Prototype based
Properties are inherited based on the class chain	Properties are inherited based on the prototype chain
Class chain – subclass > class	Prototype chain – object >object
Class and instance of the class are distinct elements	Any object can inherit from any other object
All properties specified in the class are fixed and cannot be changed dynamically during execution	Prototype includes initial set of properties. New properties can be added dynamically

JavaScript is prototype based language and each object in JavaScript can be prototype for other objects

ES6 Function Constructors

- ES6 function constructors, often referred to as "classes," are a modern way to define constructor functions and create objects with prototype-based inheritance in JavaScript.
- They provide a more structured and familiar syntax for creating objects and defining methods compared to the traditional constructor functions.
- ES6 classes are essentially syntactic sugar over JavaScript's existing prototype-based inheritance model.

Key Features of ES6 Function Constructors (Classes):

- **Constructor Method:** The 'constructor' method is used to create and initialize objects created from the class.
- **Methods:** You can define methods directly within the class, making it more organized and concise.
- **Inheritance:** Classes can extend other classes to inherit properties and methods.

PREPARED BY DR.RAJALAKSHMI S 148

ES6 Function Constructors - Syntax

```
function ConstructorName(parameter1, parameter2, ...) {
  this.property1 = parameter1;
 this.property2 = parameter2;

 this.method1 = function() {
   // Method logic
 };


}
```

→ Constructor function

PREPARED BY DR.RAJALAKSHMI S 149

ES6 Function Constructors - Example

```
function Vehicle(name) {
  this.name = name;
  this.vehType = function() {
    return `Hello, This is a ${this.name}!`;
  };
}

const V = new Vehicle("Car");

// accessing properties
console.log(V.name); // "Car"
console.log(V.vehType()); // Hello, This is a Car!
```

In the example,

- This constructor function Vehicle takes one parameter name.
- Inside the constructor function, it assigns the name parameter to the name property of the newly created object (using this.name = name;).
- It also assigns a method called vehType to the object. This method returns a string that includes the name property.
- V.name retrieves the name property of the V object, which is "Car," and logs it to the console.
- V.vehType() calls the vehType method of the V object, which returns a string that includes the name property. It logs "Hello, This is a Car!" to the console.

Function Constructors – Inheritance Example

```
// Base constructor function
function Vehicle(make, model) {
  this.make = make;
  this.model = model;
}
// Adding a method to the prototype of Vehicle
Vehicle.prototype.getDetails = function() {
  return `Make: ${this.make}, Model: ${this.model}`;
};

// Derived constructor function
function Car(make, model, year) {
  // Call the base constructor to initialize make and model
  Vehicle.call(this, make, model);

  // Additional property specific to Car
  this.year = year;
}
```

```
// Inherit the methods from Vehicle's prototype
Car.prototype = Object.create(Vehicle.prototype);

// Set the constructor property correctly for Car
Car.prototype.constructor = Car;

// Adding a method specific to Car
Car.prototype.getCarDetails = function() {
  return `Make: ${this.make}, Model: ${this.model}, Year: ${this.year}`;
};

// Create instances of Car
const myCar = new Car("Toyota", "Camry", 2022);

// Access methods from both Vehicle and Car
console.log(myCar.getDetails()); // Output: Make: Toyota, Model: Camry
console.log(myCar.getCarDetails()); // Output: Make: Toyota, Model: Camry, Year: 2022
```

Node Package Manager – NPM

What is NPM?

- NPM is a package manager for JavaScript and the default package manager for Node.js runtime. It allows developers to easily install, manage, and share code modules and libraries, known as "packages" or "dependencies." NPM is a crucial tool in the JavaScript ecosystem, enabling efficient development by providing a centralized repository for code reuse.

- Accessing NPM is free and you could install all applications available in repository without any registration. It has powerful CLI that shall allow to install software across system.

Installing NPM:

In general, NPM installed along with Node.js. So, in order to install NPM, you require to install Node.js. You could install from official website https://nodejs.org/en

PREPARED BY DR.RAJALAKSHMI S 153

How does it works?

The npm packages are defined within a file called package.json. This file will be of below format,

```
{
 "name": "npm",
 "version": "1.0.0",
 "description": "NPM software distribution for Node",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "Dr.S.Rajalakshmi",
 "license": "ISC"
}
```

When any dependent package has been installed, the package.json shall be updated to reflect that as dependencies. You could use different commands like,

npm install <package>

npm install <package>@<version>

npm install <package>@<tag>

The installed packages are used by modules by import command (require('package'))

To install the package as development dependency,

npm install <package> --save-dev

How does it works? - Example

Run the command to install nodemon - npm install nodemon

```
PS E:\JavaScript\Mod2\NPM> npm install nodemon

added 33 packages, and audited 34 packages in 3s

3 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

This updates the package.json as below,

```
{
 "name": "npm",
 "version": "1.0.0",
 "description": "NPM software distribution for Node",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "Dr.S.Rajalakshmi",
 "license": "ISC",
 "dependencies": {
  "nodemon": "^3.0.1"
 }
}
```

Custom Script

The package.json file shall also be used for specifying the starting point for your application. You could mention "Start" or "Build" or "Debug" or any script commands under the keyword "Scripts"

```
"scripts": {
  "start": "node index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Once specified, the you could simply run the application as **npm run start**

This implies the same behavior as we call node index.js from Terminal.

NPM Operations

Update Package:

In order to update the package, you could use

npm update <package>

Installed Package:

To list down installed package, you could use the command,

npm ls

Uninstall Package:

To uninstall existing package you could use below command,

pm uninstall <package>

Features of NPM

- **Package Management:** NPM enables developers to install, update, and remove packages from their projects. It manages package versions and their dependencies.
- **Versioning:** NPM allows specifying version ranges for packages, helping manage compatibility and updates.
- **Dependency Management:** NPM automatically manages the dependencies of installed packages, simplifying the process of installing and maintaining packages.
- **Centralized Repository:** NPM provides access to a vast collection of open-source JavaScript packages through the npm registry.
- **Scripts:** NPM supports defining custom scripts in the `package.json` file, making it easy to run tasks like building, testing, and deployment.
- **Global and Local Installation:** Packages can be installed globally (system-wide) or locally (project-specific), depending on their usage.
- **Semantic Versioning:** Packages follow semantic versioning conventions (e.g., MAJOR.MINOR.PATCH) to convey compatibility and changes.

PREPARED BY DR.RAJALAKSHMI S 158

PREPARED BY DR.RAJALAKSHMI S 159

Babel in JavaScript

- Babel is a widely used JavaScript compiler that allows developers to write modern JavaScript code (ES6+ and beyond) and transform it into an older version of JavaScript (usually ES5) that is compatible with most browsers and environments.
- This is particularly useful for ensuring cross-browser compatibility and supporting older systems.

Installation

Some of the related packages for babel are *@babel/core, @babel/preset-env and babel-loader*

- @babel/core ➔ This is the core babel compiler
- @babel/preset-env ➔ This enables Babel to transform modern JavaScript syntax to an older version
- babel-loader ➔ A loader that integrates Babel with webpack

Above packages shall be installed from npm package manager.

Babel in JavaScript – Configuration and Example

Create a `**.babelrc**` file in your project root directory to configure Babel.

```
{
  "presets": ["@babel/preset-env"]
}
```

So, the above keyword represents to enable the transformation process.

Assume we have example code of ES6 format.

```
const greet = (name) => {
   console.log(‘Hello, ${name}!’);
  };

greet('Rajalakshmi');
```

The presence of bable converts the above code into ES5 format,

```
var greet = function greet(name) {
  console.log("Hello, " + name + "!");
};

greet('Rajalakshmi');
```

You could observe the => has been transformed as function and the variable substitution is appearing as older concatenate fashion.

Babel in JavaScript – Using Babel with Webpack

To use Babel with webpack, you need to set up the `babel-loader` in your webpack configuration

```
module.exports = {
  // other webpack configuration principles
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
    ],
  },
};
```

In the example, the 'babel-loader' processes all '.js' files except those in the 'node_modules' directory.

Babel in JavaScript – Customizing Babel

You can customize Babel's behavior by adding additional plugins and presets to your '.babelrc' file. For instance, you can add plugins to handle class properties, decorators, and more:

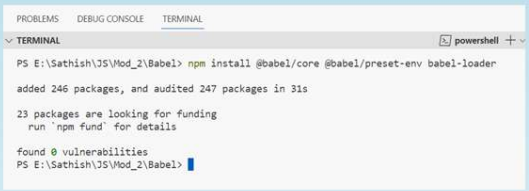
```
{
  "presets": ["@babel/preset-env"],
  "plugins": [
    "@babel/plugin-proposal-class-properties",
    "@babel/plugin-proposal-decorators"
  ]
}
```

These plugins extend Babel's capabilities beyond the default transformations provided by the preset

Using Babel, you can write modern JavaScript code while ensuring compatibility with older environments. It's an essential tool for modern web development, especially when targeting a wide range of browsers and platforms.

Babel in JavaScript – Example Code

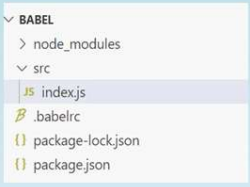
Step 1: Let we initialize repository and install require babel packages.



Step 2: This requires babel-cli package also. You have to install the latest version of this via command npm install –save-dev @babel/cli

```
PS E:\Sathish\JS\Mod_2\Babel> npm install --save-dev @babel/cli

added 33 packages, and audited 288 packages in 5s
```

Step 3: Now create files '.babelrc' and 'src/index.js' files and require directory.



Step 4: Specify babelrc file to consider transform the format

```
{
  "presets": ["@babel/preset-env"]
}
```

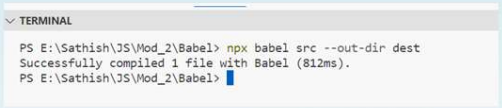

Babel in JavaScript – Example Code

Step 5: Now create a script of ES6 format

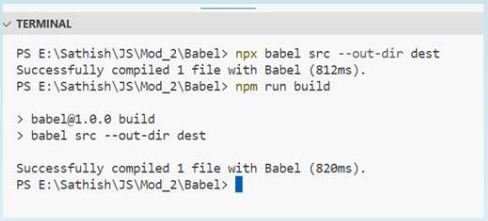
```
const greet = (name) => {
  console.log(`Hello, ${name}!`);
};

greet('Rajalakshmi');
```

Step 6: Now execute the code to by below command,

```
npx babel src --out-dir dest
```

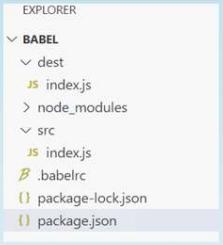


Step 7: You could also add ***"build": "babel src --out-dir dest"*** in package.json file under script and could execute as "*npm run build*" command too.



Babel in JavaScript – Example Code

Step 8: Now you could see index.js file generated automatically at dest folder



Step 9: This contains the transpiled content of ES5 format as below

```
"use strict";

var greet = function greet(name) {
  console.log("Hello, ".concat(name,
"!"));
};
greet('Rajalakshmi');
```


Webpacks in JavaScript

- Webpack is a *static module bundler* for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph (eg., images, CSS, JavaScript which are dependency for application) from one or more *entry points* and then combines every module your project (code maintained in separate files) needs into one or more *bundles*

- Webpack separates the code based on how it is used in your app, and with this modular breakdown of responsibilities, it becomes much easier to manage, debug, verify, and test your code.

Core Principles of Webpack

Webpack operates based on several core principles that guide its behavior. Some of the principles are,

- Entry Points
- Output
- Loaders
- Plugins
- Mode
- Code Splitting or Optimization
- Dev Server
- Resolve

Start with Webpack

Step 1: Create a folder and initialize repository via npm init

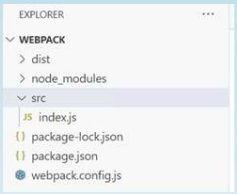
Step 2: We require the dependency webpack and hence shall install the same via npm install webpack. Additional package might be needed for some principle components that shall be added while overcome the modules.

```
TERMINAL
PS E:\Sathish\JS\Mod_2\Webpack> npm install webpack

added 77 packages, and audited 78 packages in 11s

9 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\Sathish\JS\Mod_2\Webpack>
```

Step 3: Let we create a folder structure as mentioned below for our example,

Folders => src and dist

Files => src/index.js and webpack.config.js



Step 4: Now within the index.js file, just print some output message

```
const message = "Hello, Welcome to Webpack!";
console.log(message);
```


Entry Points

- Entry points are the starting modules or files where webpack begins building the dependency graph.
- Now let we configure webpack.config.js file for all operations. We shall use the module export object.
- This module export object in Node.js holds the exported values and functions from that module. The entry shall be defined here.
- So, our entry here is index.js which is available within src folder

```
module.exports = {
  entry: './src/index.js',
};
```

Output

It defines where and how webpack should emit the bundles it creates. So here, the generate output shall be bundle.js file and the path where it place the generated file shall be "dist" folder path.

```
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

For dynamic file name provide as **[name]** instead of bundle.

PREPARED BY DR.RAJALAKSHMI S 172

Loaders

Loaders allow webpack to process different file types before they are added to the bundle. They are defined in the `module.rules` configuration. This is array of rules which are matched to request when modules are created. Here we specify rules for validate and exclude.

Syntax

```
module: {
   rules: {
      test: RegExp;
      exclude: RegExp;
      use: string;
   }[];
}
```

Example

```
module: {
   rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: 'babel-loader',
    },
   ],
 },
```

Babel loader is used to convert the code written in modern flavors of JavaScript to plain old JavaScript that supports older browser.

PREPARED BY DR.RAJALAKSHMI S 173

Plugins

Plugins are used to perform tasks that loaders can't handle, such as bundle optimization, asset management, and more. For instance html file which is available in src folder. Let the HTML shall be as below which calls the generated bundle file,

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello Webpack in JS</title>
</head>
<body>
  <div id="app"></div>
  <script src="dist/bundle.js"></script>
</body>
</html>
```

Now the JS code shall be as below,

```
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html',
  }),
```

This HtmlWebpackPlugin is the object created out of html-webpack-plugin (requires import)

Mode and Code Splitting

Mode

It defines the webpack mode (development, production, or none), which sets predefined optimizations.

```
module.exports = {
  mode: 'development', // or 'production'
};
```

Code Splitting

Allows breaking down the bundle into smaller chunks to optimize initial loading.

```
import('./module').then(module => {
  module.doSomething();
});
```

Dev Server and Resolve

Dev Server

Webpack Dev Server provides a live development server that supports Hot Module Replacement (HMR) and serves files from memory rather than the filesystem. The HMR basically exchanges, adds or removes modules while an application is running without full reload.

```
devServer: {
    static: {
    directory: path.join(__dirname, 'dist'),
    },
    compress: true,
    port: 9000,
  },
```

Resolve

This helps webpack locate modules using specific rules, like file extensions to be considered or directories to search in.

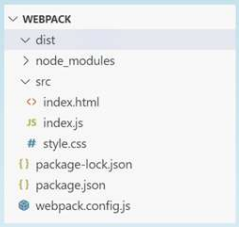
```
resolve: {
      extensions: ['.js', '.json', '.jsx'],
    },
```


Example - Webpack

For this example, we have below packages as dependencies,

- Webpack
- Webpack-cli
- Webpack-dev-server
- html-webpack-plugin
- mini-css-extract-plugin
- css-loader
- babel-loader

Step 1: Let we start with below folder structure and file structure creation.



Example - Webpack

Step 2: The style sheet shall be as below,

```
body {
  font-family: Arial, sans-serif;
}

h1 {
  color: blue;
}
```

Step 3: Let index.js shall contain the content that requires to be rendered in HTML and ofcourse it have the dependency of **style.css** which was defined before. So let we shall have simple example,

```
import './style.css';

const element = document.createElement('h1');  //Of type H1 style
element.textContent = 'Hello, Welcome to Webpack Demo!'; // Message
document.getElementById('app').appendChild(element); //ID defined in HTML
console.log(message);
```

PREPARED BY DR.RAJALAKSHMI S 178

Example - Webpack

Step 4: The HTML webpage that shown at front end that connects to JS file generated at Dist location.

```
<!DOCTYPE html>
<html>
<head>
   <title>Hello Webpack in JS</title>
</head>
<body>
   <div id="app"></div>
   <script src="dist/bundle.js"></script>
</body>
</html>
```

Step 5: Let we start with webpack.config.js file. Import the require modules, as we look into the principals of plugins, rules, output and more, the dependency modules are imported. Install those with npm install command

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
```

PREPARED BY DR.RAJALAKSHMI S 179

Example - Webpack

Step 6: Let we use Module export object and add up,

a) Entry to index.js
b) Generate bundle.js at output dist path
c) Apply rule of including JS and CSS files and exclude node_modules folder
d) Specify to use bable loader and css loader for rendering JS and CSS files
e) Use plugin to utilize HTML web page designed and CSS file created
f) Optimize by means of splitting into small chunks
g) Launch the final application as development mode at port 9000
h) Resolve only JS, JSON and JSX files

Example - Webpack

Step 7: So the code for webpack.config.js shall be as below, specify filename dynamically as [name], else you will get error as "Conflict: Multiple chunks emit assets to the same filename"

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: '[name].js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader,
          'css-loader',
        ],
      },
    ],
  },
```

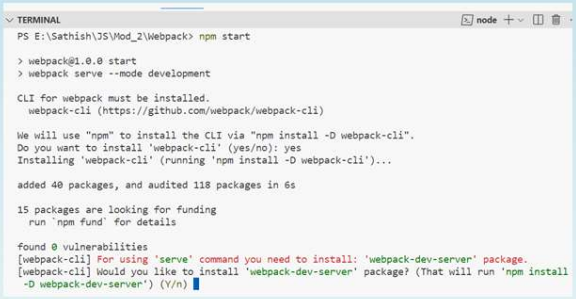
```
plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
    new MiniCssExtractPlugin({
      filename: '[name].css',
    }),
  ],
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
  },
  devServer: {
    static: {
directory: path.join(__dirname, 'dist'),
    },
    compress: true,
    port: 9000,
  },
  resolve: {
    extensions: ['.js', '.json', '.jsx'],
  },
};
```

Example - Webpack

Step 8: Now update the package.json file to add webpack start and build mode under start. It defines to start at Dev mode

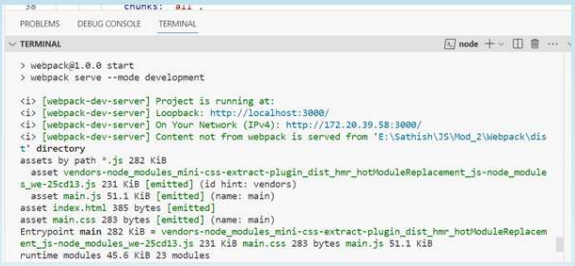
```
"scripts": {
  "start": "webpack serve --mode development",
  "build": "webpack --mode production",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Step 9: Now execute the command npm start from the Terminal. If any dependencies require for start doesn't installed it will prompt for installation and continue to give yes

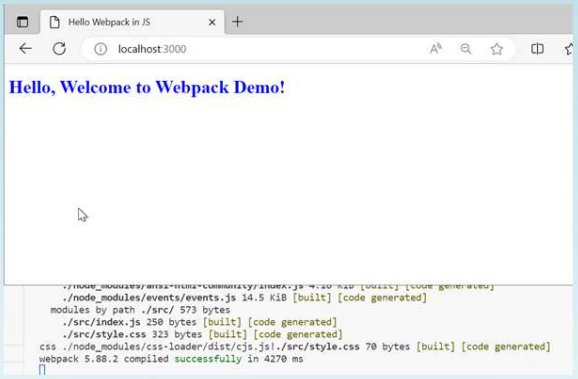


Example - Webpack

For other dependencies imported in code and if its not installed, then it will prompt error and you could install the same.



Step 10: Now after successful build, launch the browser with http://localhost:3000/ and you could see below,



Webpack – Advantages and Disadvantage

Advantages:

- Efficient bundle creation and optimizing file sizes for production.
- Support for modern JavaScript features, CSS preprocessors, and more.
- Active community and extensive plugin ecosystem.

Disadvantage:

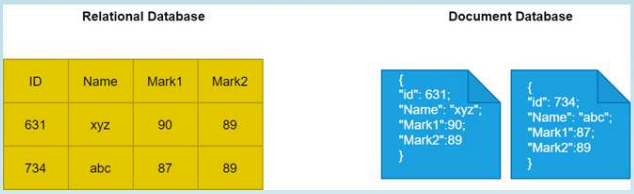
- The configuration is little bit complex and often raise confusion
- Setting up a configuration might require more effort for smaller projects.

Introduction to the MongoDB

- MongoDB is extremely popular database, and it can be used in different kinds of applications such as desktop applications, mobile applications or only in the backend
- Most often MongoDB is used in full stack applications
- Each full stack web application usually utilizes several technologies, and this set is called stack.
- There are two popular stacks: Mern and Mean
  - **Mern** includes, MongoDB (Database), Express (web server), React (front end framework), Node.js (Java Script Technology)
  - **Mean** includes, MongoDb, Express, Angular, Node.js
- MongoDB uses Mozilla's Spider Monkey JavaScript Engine
- Two common types of database:
  - Realtional DB (SQL database)
    - Relational databases are suited for structured data, complex queries and data integrity.
    - Example: MySql and Oracle
  - Document DB (NoSql Database)
    - Document database are suited for storing semi – structured, unstructured data and nested data structures such as JSON or XML documents.

Relational Vs Document Database

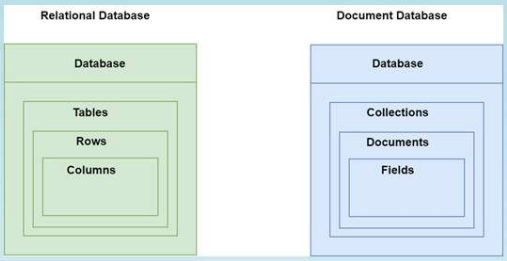
The representation of Relational and document database is shown below,



Structured Query Language (SQL) is a standardized programming language that is used to manage **relational databases**. SQL normalizes data as schemas and tables, and every table has a fixed structure. This means, it holds the attributes in columns and items are created as rows. Even though the item doesn't require certain attribute, it always reside as entry for each items.

NoSQL (Not only Structured Query Language) is used as an alternative to traditional relational databases. NoSQL databases are quite useful for working with large sets of distributed data. The data are stored as Documents

Relational Vs Document Database



- Documents are composed of field and value pairs. The documents are similar to JavaScript Object Notation (JSON) but use a variant called Binary JSON (BSON).
- The benefit of using BSON is that it accommodates more **data types**.
- The fields in these documents are like the **columns** in a relational database.
- Values contained can be a variety of data types, including other documents, arrays and arrays of documents. Documents will also incorporate a primary key as a unique identifier. A document's structure is changed by adding or deleting new or existing fields.
- Sets of documents are called collections, which function as the equivalent of relational database tables.
- Collections can contain any type of data, but the restriction is the data in a collection cannot be spread across different databases.
- Users of MongoDB can create multiple databases with multiple collections.

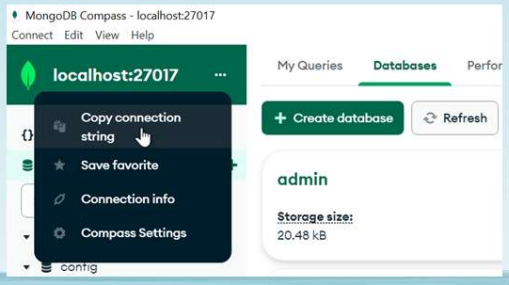
PREPARED BY DR.RAJALAKSHMI S 188

Working with MongoDB using JS

Before you begin, make sure MongoDB is installed and running in your machine. If not get it from https://www.mongodb.com/try/download/community. The default host information shall be mongodb://localhost and if installed in other port than 27017, then specify that as well along with host. If you face trouble with localhost then use 127.0.0.1

You could see the host information from your MongoDB application as well

Optional



After installation make sure to set the installed location BIN path to the environment variable PATH

The mongo shell shall also be installed **if required** from https://www.mongodb.com/try/download/shell

Download and extract and add the bin PATH to environment for easy access. Shell is required only if you want to access DB over Terminal than GUI

PREPARED BY DR.RAJALAKSHMI S 189

Connect to MongoDB

Step1: Let we initialize the directory with npm init

```
{
 "name": "mongodb",
 "version": "1.0.0",
 "description": "Connect MongoDB using JS",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "Dr.Rajalakshmi S",
 "license": "ISC"
}
```

Step 2: Now we need to install dependencies for our application. Main dependency is mongodb which is required.

npm install mongodb

```
PS E:\Sathish\JS\MongoDB> npm install mongodb
>>

added 13 packages, and audited 14 packages in 12s

found 0 vulnerabilities
```

Step 3: Now lets create a file connect.js and import the mongodb client as below,

```
const MongoClient  = require('mongodb').MongoClient;
```


Connect to MongoDB

Step 4: Specify the connection URL. This could be a localhost or IP address,

```
const url = 'mongodb://127.0.0.1:27017';
```

If you use password then

```
mongodb://username:password@127.0.0.1
```

Step 5: Initializing client for connection

```
const client = new MongoClient(url);
```

Step 6: Create a connection via connect() command and handle it within try catch for any exceptions. The await or asychronize helps to run the operation in background. Finally you could close the connection.

```
try {
  // Connect to the server
  await client.connect();
  console.log('Connected  successfully to MongoDB');
 } catch (err) {
  console.error('Error connecting to MongoDB:', err);
 } finally {
  // Close the connection  when you're done
  await client.close();
 }
```

Connect to MongoDB

Step 4: Specify the connection URL. This could be a localhost or IP address,

```
const url = 'mongodb://127.0.0.1:27017';
```

If you use password then

```
mongodb://username:password@127.0.0.1
```

Step 5: Initializing client for connection

```
const client = new MongoClient(url);
```

Step 6: Create a connection via connect() command and handle it within try catch for any exceptions. The await or asychronize helps to run the operation in background. Finally you could close the connection.

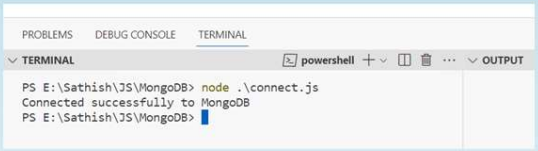
```
try {
  // Connect to the server
  await client.connect();
  console.log('Connected successfully to MongoDB');
 } catch (err) {
  console.error('Error connecting to MongoDB:', err);
 } finally {
  // Close the connection when you're done
  await client.close();
 }
```


Connect to MongoDB

Step 7: Now the code shall be as below,

```
const { MongoClient } = require('mongodb');

// MongoDB connection URL
const url = 'mongodb://127.0.0.1:27017'; // Change this to your MongoDB server URL

// Create a new MongoClient
const client = new MongoClient(url);

async function main() {
 try {
  // Connect to the server
  await client.connect();
  console.log('Connected successfully to MongoDB');
 } catch (err) {
  console.error('Error connecting to MongoDB:', err);
 } finally {
  // Close the connection when you're done
  await client.close();
 }
}

main();
```

Step 8: The output shall be,



Create Database and Collection

Step 1: Now lets create a database with the name "interviews", so declared here

```
const dbName = 'interviews';
```

Step 2: Now after the connection is successful, we shall initialize the database name

```
const db = client.db(dbName);
```

Step 3: Now we need to define the collections within the database. Here, I just define as "companies" in my collection

```
const companiesCollection = db.collection('companies');
```

Step 4: Now we have DB and Collection available. So we are ready for data set. Let we define the dataset as BSON format as below,

```
const newCcompanies = {
    companyname: 'Microsoft',
    website: 'www.microsoft.com',
    location: 'USA',
};
```

Step 5: Let we call the insert operation to feed our newly defined data,

```
const result = await
companiesCollection.insertOne(newCompanies);
```

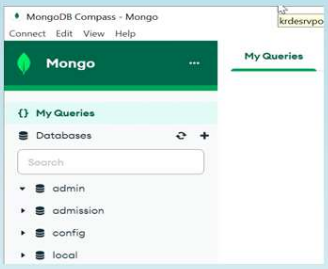

Create Database and Collection

Step 6: Handle these operations in try catch to capture error and hence code shall now be like this,

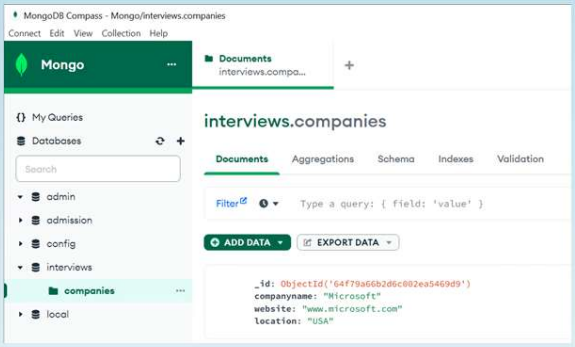
```
const { MongoClient } = require('mongodb');

// MongoDB connection URL
const url = 'mongodb://127.0.0.1:27017'; // Change this to your MongoDB server URL

// Database Name
const dbName = 'interviews'; // Change this to your database name

// Create a new MongoClient
const client = new MongoClient(url);

async function main() {
  try {
    // Connect to the server
    await client.connect();

    console.log('Connected successfully to MongoDB');

    // Reference to the database
    const db = client.db(dbName);

    // Reference to the collection
    const companiesCollection = db.collection('companies'); // Change 'users' to your collection name

    // Data to be inserted (a new user document)
    const newCompanies = {
      companyname: 'Microsoft',
      website: 'www.microsoft.com',
      location: 'USA',
    };

// Insert the new document into the collection
    const result = await companiesCollection.insertOne(newCompanies);

    console.log(`Inserted ${result.insertedCount} document into the collection`);
    console.log('Inserted document ID:', result.insertedId);
  } catch (err) {
    console.error('Error:', err);
  } finally {
    // Close the connection when you're done
    await client.close();
  }
}

main();
```

Create Database and Collection

Step 7: Before the DB Creation



Now Refresh the Database



Execution

```
TERMINAL
PS E:\Sathish\JS\MongoDB> node .\create.js
Connected successfully to MongoDB
Inserted undefined document into the collection
Inserted document ID: new ObjectId("64f79a66b2d6c002ea5469
d9")
PS E:\Sathish\JS\MongoDB>
```


Read data and display in Console

Step 1: For read and display the data available in database "Find" operation shall be used.

```
const cursor = companiesCollection.find();
```

Step 2: For iterating all items within the document, a for loop shall be used.

```
await cursor.forEach(document => {
    console.log(document);
}
```

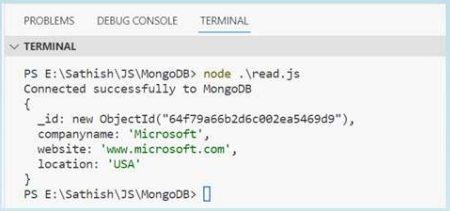
Step 3: The overall code shall be like this,

```
const { MongoClient } = require('mongodb');

// MongoDB connection URL
const url = 'mongodb://127.0.0.1:27017'; // Change this to your
MongoDB server URL

// Database Name
const dbName = 'interviews'; // Change this to your database
name

// Create a new MongoClient
const client = new MongoClient(url);

async function main() {
 try {
  // Connect to the server
  await client.connect();

  console.log('Connected successfully to MongoDB');

  // Reference to the database
  const db = client.db(dbName);
```

Read data and display in Console

```
// Reference to the collection
  const companiesCollection = db.collection('companies'); //
Change 'companies' to your collection name

  // Find all documents in the collection
  const cursor = companiesCollection.find();

  // Iterate over the documents and print them to the console
  await cursor.forEach(document => {
   console.log(document);
  });
 } catch (err) {
  console.error('Error:', err);
 } finally {
  // Close the connection when you're done
  await client.close();
 }
}

main();
```

Step 4: On execution, the output shall be as below,

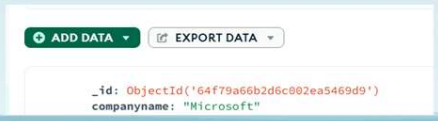


Update Database Entry

Step 1: For replace operation, you have to start with the find filter that holds the _id which is a kind of primary key. We already retrieved this ID when we execute Read functionality. So our id in this case is "*64f79a66b2d6c002ea5469d9*" as below,

```
PS E:\Sathish\JS\MongoDB> node .\read.js
Connected successfully to MongoDB
{
  _id: new ObjectId("64f79a66b2d6c002ea5469d9"),
  companyname: 'Microsoft',
```

Or obtained from the document itself,



Step 2: Now with this value, we shall define the filter,

```
const filter = { _id: new
ObjectId('64f79a66b2d6c002ea5469d9') };
```

Step 3: Now define the values or attribute entry that requires to be updated. Same BSON format,

```
const update = {
   $set: {
    companyname: 'Microsoft Corporation',
    // Add more fields to update as needed
   },
  };
```

In above I am updating Microsoft as Microsoft Corporation

Update Database Entry

Step 4: Finally the update functionality,

```
const result = await companiesCollection.updateOne(filter, update);
```

Step 5: So the code shall look as below,

```
const { MongoClient, ObjectId } = require('mongodb');

// MongoDB connection URL
const url = 'mongodb://127.0.0.1:27017'; // Change this to your MongoDB server URL

// Database Name
const dbName = 'interviews'; // Change this to your database name

// Create a new MongoClient
const client = new MongoClient(url);

async function main() {
 try {
  // Connect to the server
  await client.connect();

  console.log('Connected successfully to MongoDB');

 // Reference to the database
  const db = client.db(dbName);

  // Reference to the collection
  const companiesCollection = db.collection('companies');

  // Define the filter to find the document you want to update
  const filter = { _id: new ObjectId('64f79a66b2d6c002ea5469d9') };
 // Replace with the actual document ID

  // Define the update operation
  const update = {
   $set: {
    companyname: 'Microsoft Corporation',
    // Add more fields to update as needed
   },
  };
```


Update Database Entry

```
  // Update the document using updateOne
    const result = await companiesCollection.updateOne(filter, update);

    console.log(`Matched ${result.matchedCount} document(s) and modified ${result.modifiedCount} document(s)`);
  } catch (err) {
    console.error('Error:', err);
  } finally {
    // Close the connection when you're done
    await client.close();
  }
}

main();
```

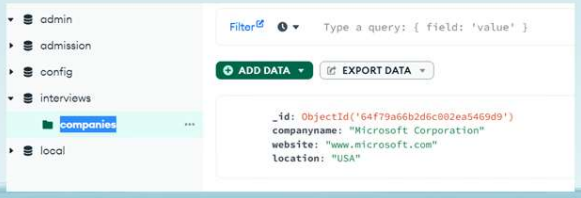
Step 6: Now lets execute to check the output,

```
PS E:\Sathish\JS\MongoDB> node .\update.js
Connected successfully to MongoDB
Matched 1 document(s) and modified 1 document(s)
```

Step 7: Its matched and updated, lets see the entry,

```
PS E:\Sathish\JS\MongoDB> node .\read.js
Connected successfully to MongoDB
{
  _id: new ObjectId("64f79a66b2d6c002ea5469d9"),
  companyname: 'Microsoft Corporation',
  website: 'www.microsoft.com',
  location: 'USA'
}
PS E:\Sathish\JS\MongoDB>
```

You could also check in document view, (you need to close the query window and reopen again)



Drop the Database

Step 1: As like update, here also we will use filter to find based on _id and then shall trigger for drop.

```
const filter = { _id: new ObjectId('64f79a66b2d6c002ea5469d9') };
```

The above filter is for one item deletion, if you need to delete multiple items then shall chose the filter for multiple items

Eg., const filter = {location: 'USA'}; //delete all of location USA

Step 2: Now let use deleteOne function,

```
const result = await companiesCollection.deleteOne(filter, update);
```

in case of multiple item delete,

```
const result = await companiesCollection.deleteMany(filter, update);
```


Drop the Database

Step 3: So the code shall be as below,

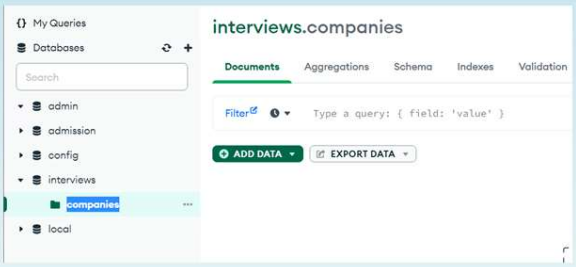
```
const { MongoClient, ObjectId } = require('mongodb');

// MongoDB connection URL
const url = 'mongodb://127.0.0.1:27017'; // Change this to your MongoDB server URL

// Database Name
const dbName = 'interviews'; // Change this to your database name

// Create a new MongoClient
const client = new MongoClient(url);

async function main() {
  try {
    // Connect to the server
    await client.connect();

    console.log('Connected successfully to MongoDB');

    // Reference to the database
    const db = client.db(dbName);

    // Reference to the collection
    const companiesCollection = db.collection('companies');

    // Define the filter to find the document you want to delete
    const filter = { _id: new ObjectId('64f79a66b2d6c002ea5469d9') };
// Replace with the actual document ID

    // Delete the document using deleteOne
    const result = await companiesCollection.deleteOne(filter);

    console.log(`Deleted ${result.deletedCount} document(s)`);
  } catch (err) {
    console.error('Error:', err);
  } finally {
    // Close the connection when you're done
    await client.close();
  }
}

main();
```

Drop the Database

Step 4: The execution shall return,

```
TERMINAL
PS E:\Sathish\JS\MongoDB> node .\drop.js
Connected successfully to MongoDB
Deleted 1 document(s)
PS E:\Sathish\JS\MongoDB>
```

(close the query window and open again)



Sign Off