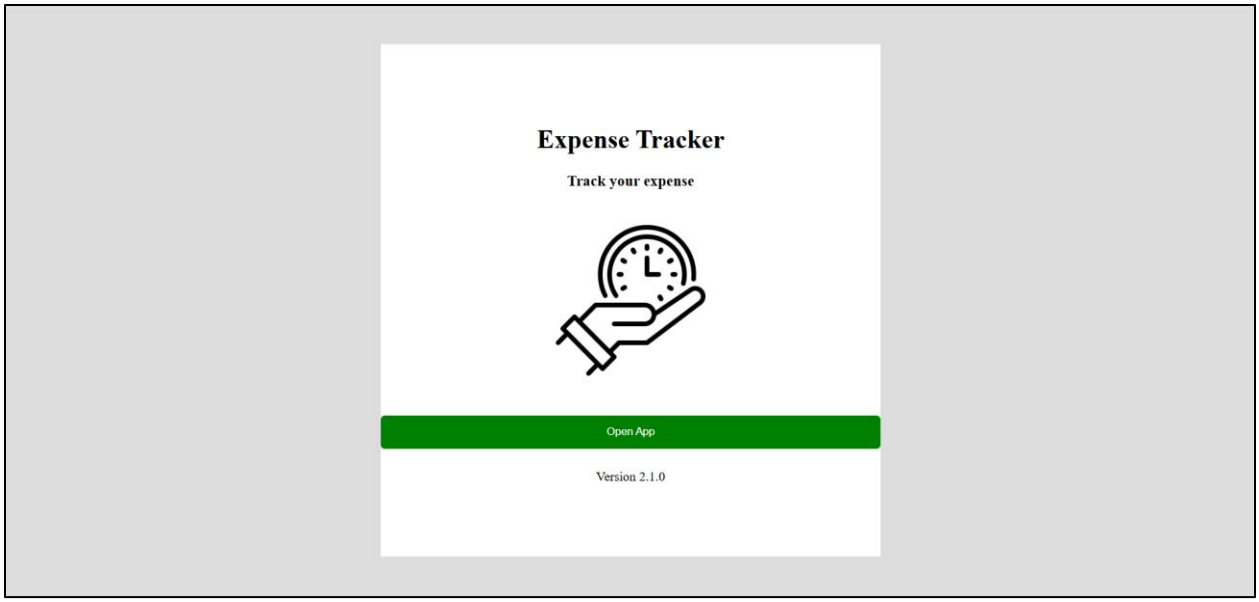
Expense Tracker With JavaScript

Author: Kabir Yusuf Bashir

Team Piccolo

2025


Expense Tracker
Track your expense
Open App
Version 2.1.0

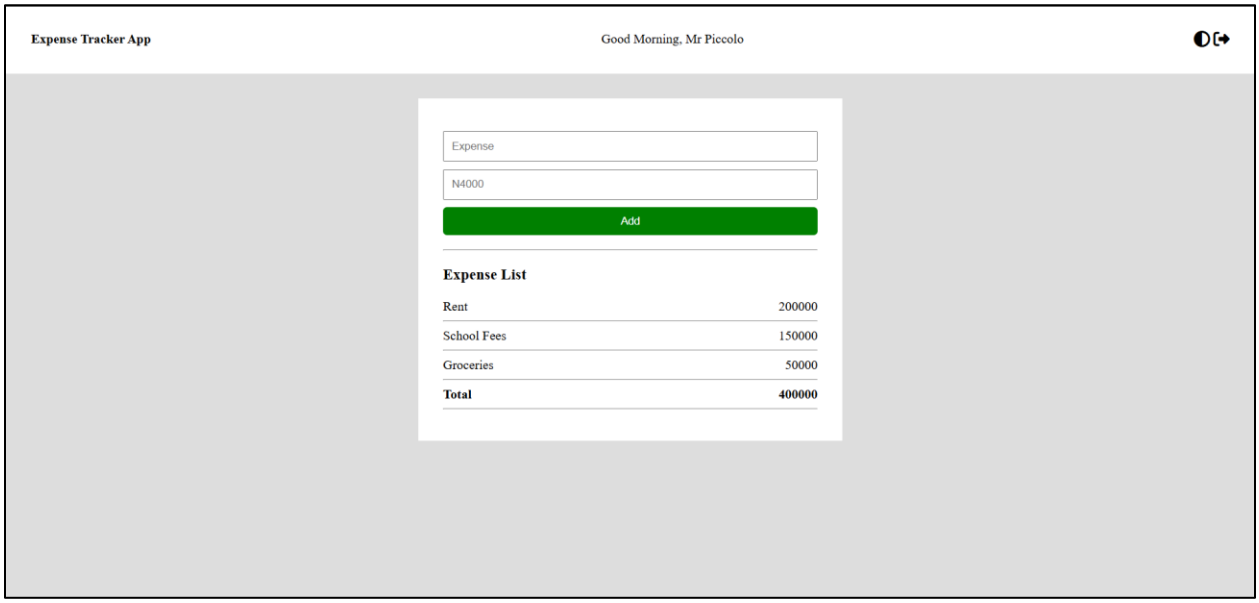

Expense Tracker App
Good Morning, Mr Piccolo
Expense
N4000
Add
Expense List
Rent 200000
School Fees 150000
Groceries 50000
Total 400000

Table of Contents

Table of Figures

Introduction to JavaScript

JavaScript is the world's most popular programming language. JavaScript is the programming language of the Web. JavaScript is easy to learn.

JavaScript and Java are completely different languages, both in concept and design. JavaScript is one of the 3 languages all web developers must learn:

- HTML to define the content of web pages.
- CSS to specify the layout of web pages.
- JavaScript to program the behavior of web pages.

JavaScript can be used to change HTML content.

- One of many JavaScript HTML methods is getElementById().

JavaScript can change HTML attributes.

JavaScript can change HTML Styles (CSS)

- document.getElementById("header").style.fontSize = "15px";

JavaScript can hide HTML Elements.

- document.getElementById("header").style.display = "none";

JavaScript can show HTML Elements.

- document.getElementById("header").style.display = "block";

Foot Note:

- You don't have to get or download JavaScript.
- JavaScript is already running in your browser on your computer, on your tablet, and on your smart-phone. Free to use for everyone.
- JavaScript accepts both double and single quotes.

Adding JavaScript to a web page

JavaScript can be added to web page using internal (inside the web page) or external (outside the web page).

Internal

JavaScript code is nestled between <script> and </script> tag as shown below:

```
<script>
    document.getElementById('header').innerHTML = 'My Apps';
</script>
```

Figure 1: Adding JavaScript to a web page

Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Apps</title>
    <script>
        function changeTitle() {
            document.getElementById('header').innerHTML = 'My Apps';
        }
    </script>
</head>
<body>
    <h2>Adding JavaScript in Head Tag</h2>

    <p id="header">My Collections</p>
    <button type="button" onclick="changeTitle()">Change Title</button>
</body>
</html>
```

Figure 2: Adding JavaScript in Head Tag

Note: Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display.

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Apps</title>
</head>
<body>
    <h2>Adding JavaScript in Body Tag</h2>

    <p id="header">My Collections</p>
    <button type="button" onclick="changeTitle()">Change Title</button>
    <script>
        function changeTitle() {
            document.getElementById('header').innerHTML = 'My Apps';
        }
    </script>
</body>
</html>
```

Figure 3: Adding JavaScript in Body Tag

External

Scripts can also be placed in external files just like how you do CSS external styling. External scripts are practical when the same code is used in many different web pages.

JavaScript files have the file extension **.js**.

You can place an external script reference in <head> or <body> as you like.

The script will behave as if it was located exactly where the <script> tag is located.

To use an external script, put the name of the script file in the src (source) attribute of a <script> tag as shown below:

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Apps</title>
</head>
<body>
    <h2>Adding JavaScript in Body Tag (External)</h2>

    <p id="header">My Collections</p>
    <button type="button" onclick="changeTitle()">Change Title</button>
    <script src="js/script.js"></script>
</body>
</html>
```

Figure 4: Adding JavaScript (External)

Placing scripts in external files has some advantages:

- It separates HTML and code
- It makes HTML and JavaScript easier to read and maintain
- Cached JavaScript files can speed up page loads

To add several script files to one page - use several script tags as shown below:

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Apps</title>
</head>
<body>
    <h2>Adding JavaScript in Body Tag (External)</h2>

    <p id="header">My Collections</p>
    <button type="button" onclick="changeTitle()">Change Title</button>
    <script src="js/script.js"></script>
    <script src="js/to-do.js"></script>
    <script src="js/calculator.js"></script>
</body>
</html>
```

Figure 5: Adding several script files

JavaScript Display Data

JavaScript can "display" data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write( ).
- Writing into an alert box, using window.alert( ).
- Writing into the browser console, using console.log( ).

innerHTML

```
document.getElementById('header').innerHTML = 'My Apps';
```

Figure 6: JS innerHTML

document.write( )

```
document.write('My Apps');
```

Figure 7: document.write( )

Note: *Using document.write( ) after an HTML document is loaded, will* ***delete all existing HTML****.*

window.alert( )

```
window.alert('My Apps');
```

Figure 8: window.alert( )

Note: *In JavaScript, the window object is the global scope object. This means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional:*

console.log( )

```
console.log('My Apps')
```

Figure 9: console.log( )

JavaScript Variables

Variables are Containers for Storing Data.

JavaScript Variables can be declared in 4 ways:

- Automatically

```
x = 10;
y = 20;
z = x + y;
```

Figure 10: JS Variable Declaration

- Using **var**

```
var x = 10;
var y = 20;
var z = x + y;
```

Figure 11: JS Variable Declaration II

- Using **let**

```
let x = 10;
let y = 20;
let z = x + y;
```

Figure 12: JS Variable Declaration III

- Using **const**

```
const x = 10;
const y = 20;
const z = x + y;
```

Figure 13: JS Variable Declaration IV

When to Use var, let, or const?

1) Always declare variables
2) Always use const if the value should not be changed
3) Always use const if the type should not be changed (Arrays and Objects)
4) Only use let if you can't use const
5) Only use var if you MUST support old browsers.

JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with $ and _ (but we will not use it in this tutorial).
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

Note:

- The var keyword was used in all JavaScript code from 1995 to 2015.
- The let and const keywords were added to JavaScript in 2015.
- The var keyword should only be used in code written for older browsers.

JavaScript let

- The let keyword was introduced in ES6 (2015)
- Variables declared with let have **Block Scope**
- Variables declared with let must be **Declared** before use
- Variables declared with let cannot be **Redeclared** in the same scope

Block Scope

- Before ES6 (2015), JavaScript did not have **Block Scope**.
- JavaScript had **Global Scope** and **Function Scope**.
- ES6 introduced the two new JavaScript keywords: let and const.
- These two keywords provided **Block Scope** in JavaScript:

```
{
    let x = 20;
}

console.log(x); // x can NOT be used here
```

Figure 14: JS Block Scope

Global Scope

- Variables declared with the var always have **Global Scope**.
- Variables declared with the var keyword can NOT have block scope:
- Variables declared with var inside a { } block can be accessed from outside the block:

```
{
    var x = 20;
}

console.log(x); // x can be used here
```

Figure 15: JS Global Scope

Cannot be Redeclared

- Variables defined with let **can not** be redeclared.
- You **can not** accidentally redeclare a variable declared with let.

```
let x = 20;

let x = 2;

// This is WRONG
```

Figure 16: JS let Declaration

But variables defined with var **can** be redeclared:

```
var x = 20;

var x = 2;

// This is CORRECT
```

Figure 17: JS var Declaration

Note:

- let and const have **block scope**.
- let and const cannot be **redeclared**.
- let and const must be **declared** before use.

JavaScript const

- The const keyword was introduced in ES6 (2015)
- Variables defined with const cannot be **Redeclared**
- Variables defined with const cannot be **Reassigned**
- Variables defined with const have **Block Scope**

Cannot be Reassigned

A variable defined with the const keyword cannot be reassigned:

```
const salary = 250000;

salary = 100000; // This is wrong

salary = salary + 50000; // This is also wrong
```

Figure 18: JS const

When to use JavaScript const?

Always declare a variable with const when you know that the value should not be changed.

Use const when you declare:

- A new Array
- A new Object

- A new Function
- A new RegExp

```
const courses = ['HTML', 'CSS', 'JavaScript'];
```

Figure 19: JS const declaring an Array

```
const courses = {
    name: 'HTML & CSS',
    duration: '4 weeks',
    cost: '150000'
}
```

Figure 20: JS const declaring an Object

JavaScript Operators

JavaScript operators are used to perform different types of mathematical and logical computations.

Examples:

- The **Assignment Operator** = assigns values
- The **Addition Operator** + adds values
- The **Multiplication Operator** * multiplies values
- The **Comparison Operator** > compares values

JavaScript Assignment

The **Assignment Operator** (=) assigns a value to a variable:

```
let salary = 150000;
```

Figure 21: JS assignment operator

JavaScript Addition

The **Addition Operator** (+) adds numbers:

```
let salary = 150000;

let allowance = 25000;

let takeHome = salary + allowance;
```

Figure 22: JS addition operator

JavaScript Multiplication

The **Multiplication Operator** (*) multiplies numbers:

```
let salary = 150000 * 12;
```

Figure 23: JS multiplication operator

Types of JavaScript Operators

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Ternary Operators
- Type Operators

JavaScript Arithmetic Operators

Arithmetic Operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Remainder)
++	Increment

--	Decrement

```
let salary = 150000;
let allowance = 25000;
let months = 12;

let takeHome = (salary + allowance) * months;
```

Figure 24: JS Arithmetic Operator

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

The **Addition Assignment Operator** (+=) adds a value to a variable.

```
let x = 10;
x += 20;
```

Figure 25: JS Assignment Operator

JavaScript Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Operator	Description
= =	Equal to
= = =	Equal value and equal type
! =	Not Equal
! = =	Not equal value and equal type
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to
?	Ternary operator

Figure 26: JS Comparison Operators

```
let age = 18;

if(age < 18){
    console.log('Too young to drive');
}
```

Figure 27: JS Comparison Operator

Note:

- When used on strings, the + operator is called the **concatenation** operator.

```
let x = "5";
let y = 2;

let z = x + y; // Out is 52 not 7
```

Figure 28: JS Concatenation Operator

JavaScript Logical Operators

Logical operators are used to determine the logic between variables or values.

Operator	Description
&&	And
||	Or
!	Not Equal

Figure 29: JS Logical Operators table

```
let age = 18;

if(age < 18 && age >= 10){
    console.log('Not eligible to drive');
}
```

Figure 30: JS Logical Operator

JavaScript Ternary Operators

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

```
let ageToDrive = 18;

(ageToDrive >= 18) ? 'Eligible to drive' : 'Not Eligible to drive';
```

Figure 31: JS Ternary Operator

JavaScript Nullish Coalescing Operator (??)

The ?? operator returns the first argument if it is not **nullish** (null or undefined).

```
let nameFetch = null;
let defaultName = 'Piccolo';

let studentName = nameFetch ?? defaultName;
```

Figure 32: JS Nullish Coalescing Operator (??)

Note: This is useful when fetching data from an API endpoint.

JavaScript Optional Chaining Operator (?.)

The ?. operator returns undefined if an object is undefined or null (instead of throwing an error).

```
let courses = {
    firstCohort: 'HTML & CSS',
    secondCohort: 'JS & Responsive Design'
}

console.log(courses?.thirdCohort); // return undefined
```

Figure 33: JS Optional Chaining Operator

JavaScript Data Type

In programming, data type is an important concept.

To be able to operate on variables, it is important to know something about the type.

JavaScript has 8 Datatypes

- String
- Number
- Bigint
- Boolean
- Undefined
- Null
- Symbol
- Object

```
// Number
let salary = 50000;
let allowance = 15000;
```

Figure 34: Number Data Type

```
//Srings
let firstName = 'Team';
let lastName = 'Piccolo';
```

Figure 35: String Data Type

```
//Booleans
let x = true;
let y = false;
```

Figure 36: Boolean Data Type

```
//Object
const tutor = {
    firstName: 'Team',
    lastName: 'Piccolo'
}
```

Figure 37: Object Data Type

```
// Array Object
const courses = ['HTML', 'CSS', 'JS', 'TailWindCSS'];
```

Figure 38: Array Object Data Type

```
// Date object
const date = new Date('2025-01-20');
```

Figure 39: Date Object Data Type

Without data types, a computer cannot safely execute the statement below:

```
let x = "Course Fee " + 150000;
```

Figure 40: Concept of Data Type

Note: When adding a number and a string, JavaScript will treat the number as a string. The statement above will be executed as shown below:

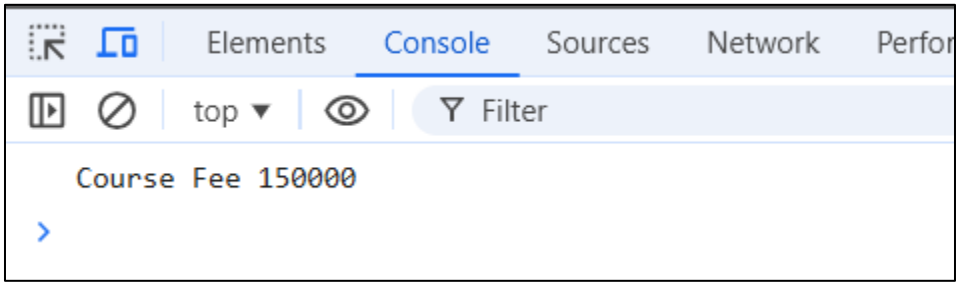


Figure 41: Adding a number and a string

JavaScript Strings

Strings are for storing text.

Strings are written with quotes.

```
//String with single quotes
let message = 'Welcome to this Course!';

//String with double quotes
let msg = "Welcome to JavaScript Course";
```

Figure 42: JS Strings

Note:

- Strings created with single or double quotes work the same.
- There is no difference between the two.

Quotes Inside Quotes

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```
// Quotes Inside Quotes
let answer1 = "It's alright";
let answer2 = "My name is 'Piccolo'";
let answer3 = 'His name is "Malamiromba"';
```

Figure 43: Quotes inside quotes

Template Strings

Templates were introduced with ES6 (JavaScript 2016).

Templates are strings enclosed in backticks as shown below:

```
// Template Strings
let message = `Welcome to JavaScript Course`;
```

Figure 44: JS Template Strings

Templates allow single and double quotes inside a string as shown below:

```
// Template Strings
let message = `Welcome 'Piccolo' to our JavaScript Course`;
```

Figure 45: JS Template strings with quotes

JavaScript String Methods

String Length

To find the length of a string, use the built-in length property as shown below:

```
// String Length
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let messageLength = message.length;
console.log(messageLength);
```

Figure 46: JS String Length

Extracting String Parts

There are 3 methods for extracting a part of a string:

- slice(*start*, *end*)
- substring(*start*, *end*)
- substr(*start*, *length*)

JavaScript String slice( )

`slice( )` extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: start position, and end position (end not included)

Note:

- JavaScript counts positions from zero.
- First position is 0.
- Second position is 1

```
// Extracting String Parts (slice)
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let extractedMessage = message.slice(9, 16)
console.log(extractedMessage);
```

Figure 47: JS String slice( )

```
// If you omit the second parameter,
// the method will slice out the rest of the string
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let extractedMessage = message.slice(9)
console.log(extractedMessage);
```

Figure 48: JS String slice( ) omitting the second parameter

```
// If a parameter is negative,
// the position is counted from the end of the string:
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let extractedMessage = message.slice(-17)
console.log(extractedMessage);
```

Figure 49: JS String slice( ) negative parameter

Converting to Upper and Lower Case

A string is converted to upper case with toUpperCase( ):

A string is converted to lower case with toLowerCase( ):

```
// JavaScript String toUpperCase()
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let messageToUpper = message.toUpperCase();
console.log(messageToUpper);
```

Figure 50: JS String toUpperCase( )

```
// JavaScript String toLowerCase()
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let messageToLower = message.toLowerCase();
console.log(messageToLower);
```

Figure 51: JS String toLowerCase( )

JavaScript String concat( )

concat( ) joins two or more strings:

```
// JavaScript String concat()
let message = `Welcome 'Piccolo'`;
let course = `to our JavaScript Course`;
let msgConc = message.concat(' ', course);
console.log(msgConc);
```

Figure 52: JS String concat( )

JavaScript String trim( )

The trim( ) method removes whitespace from both sides of a string.

```
// JavaScript String trim()
let message = `    Welcome 'Piccolo' to our JavaScript Course    `;
let msgTrim = message.trim();
console.log(msgTrim);
```

Figure 53: JS String trim( )

JavaScript String repeat( )

The repeat( ) method returns a string with a number of copies of a string.

The repeat( ) method returns a new string.

The repeat( ) method does not change the original string.

```
// JavaScript String repeat()
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let msgRepeat = message.repeat(100);
console.log(msgRepeat);
```

Figure 54: JS String repeat( )

Replacing String Content

The replace( ) method replaces a specified value with another value in a string

```
// JavaScript String replace()
let message = `Welcome 'Piccolo' to our JavaScript Course`;
let msgNew = message.replace('Piccolo', 'Malamiromba');
console.log(msgNew);
```

Figure 55: JS String replace( )

Note:

- By default, the replace( ) method is case sensitive. Writing PICCOLO (with upper-case) will not work.

JavaScript String ReplaceAll( )

In 2021, JavaScript introduced the string method `replaceAll()`:

```
// JavaScript String replaceAll()
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgNew = message.replaceAll('Piccolo', 'Malamiromba');
console.log(msgNew);
```

Figure 56: JS String replaceAll( )

Converting a String to an Array

If you want to work with a string as an array, you can convert it to an array.

JavaScript String split( )

A string can be converted to an array with the split( ) method.

```
// JavaScript String split() // Split on commas
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgSplit = message.split(',');
console.log(msgSplit);
```

Figure 57: JS String split( ) on commas

```
// JavaScript String split() // Split on spaces
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgSplit = message.split(' ');
console.log(msgSplit);
```

Figure 58: JS String split( ) on spaces

```
// JavaScript String split() // Split on pipe
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgSplit = message.split('|');
console.log(msgSplit);
```

Figure 59: JS String split( ) on pipe

JavaScript String Search

JavaScript String indexOf()

The indexOf( ) method returns the **index** (position) of the **first** occurrence of a string in a string, or it returns -1 if the string is not found:

```
// JavaScript String search indexOf()
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgIndexOf = message.indexOf(`Malamiromba`);
console.log(msgIndexOf);
```

Figure 60: JS String search indexOf( )

JavaScript String search( )

The search( ) method searches a string for a string (or a regular expression) and returns the position of the match:

```
// JavaScript String search()
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgSearch = message.search(`Piccolo`);
console.log(msgSearch);
```

Figure 61: JS String search( )

JavaScript String match( )

The match( ) method returns an array containing the results of matching a string against a string (or a regular expression).

```
// JavaScript String match()
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgMatch = message.match(`Piccolo`);
console.log(msgMatch);
```

Figure 62: JS String match( )

JavaScript String includes()

The includes( ) method returns true if a string contains a specified value. Otherwise, it returns false.

```
// JavaScript String includes()
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgIncludes = message.includes(`Malamiromba`);
console.log(msgIncludes);
```

Figure 63: JS String includes( )

JavaScript String startsWith( )

The startsWith( ) method returns true if a string begins with a specified value.

Otherwise, it returns false:

```
// JavaScript String startsWith( )
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgStartWith = message.startsWith(`'Piccolo'`);
console.log(msgStartWith);
```

Figure 64: JS Strings startsWith( )

JavaScript String endsWith( )

The endsWith( ) method returns true if a string ends with a specified value.

Otherwise, it returns false:

```
// JavaScript String endsWith( )
let message = `Welcome 'Piccolo' to our JavaScript Course! Piccolo How are you?`;
let msgEndsWith = message.endsWith(`you?`);
console.log(msgEndsWith);
```

Figure 65: JS Strings endsWith( )

JavaScript Template Strings

Back-Tics Syntax

Template Strings use back-ticks (``) rather than the quotes ("") to define a string as shown below:

```
// Back-Tics Syntax
let message = `Welcome 'Piccolo' to our JavaScript Course!`;
```

Figure 66: Back-Tics Syntax

Interpolation

Template String provide an easy way to interpolate variables and expressions into strings.

The method is called **string interpolation**.

```
// JavaScript String Interpolation
let studentName = `Piccolo`;
let message = `Welcome ${'Piccolo'} to our JavaScript Course!`;
console.log(message);
```

Figure 67: JS Template String Interpolation

JavaScript if, else, and else if

Conditional statements are used to perform different actions based on different conditions.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true.
- Use else to specify a block of code to be executed, if the same condition is false.
- Use else if to specify a new condition to test, if the first condition is false.
- Use switch to specify many alternative blocks of code to be executed

The if Statement

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

```
// JS if statement
const age = 18;
if(age >= 18){
    console.log(`Eligible to drive`)
}
```

Figure 68: JS if statement

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

```
// JS else statement
const age = 18;
if(age >= 18){
    console.log(`Eligible to drive`)
}else{
    console.log(`Not Eligible to drive`)
}
```

Figure 69: JS else statement

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

```
// JS else if statement
const time = 3;
if(time < 12){
    console.log(`Good Morning`)
}else if(time >= 12 && time < 18){
    console.log(`Good afternoon`)
}else{
    console.log(`Good evening`)
}
```

Figure 70: JS else if statement

JavaScript Switch Statement

The switch statement is used to perform different actions based on different conditions.

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed

```
// JS else if statement
let day;
switch (new Date().getDay()) {
    case 0:
        day = 'Sunday';
        console.log(day);
        break;
    case 1:
        day = 'Monday';
        console.log(day);
        break;
    case 2:
        day = 'Tuesday';
        console.log(day);
        break;
    case 3:
        day = 'Wednesday';
        console.log(day);
        break;
    case 4:
        day = 'Thursday';
        console.log(day);
        break;
    case 5:
        day = 'Friday';
        console.log(day);
        break;
    case 6:
        day = 'Saturday';
        console.log(day);
    default:
        text = "Looking forward to the Weekend";
}
```

Figure 71: JS switch statement

Note:

The break Keyword

- When JavaScript reaches a break keyword, it breaks out of the switch block.
- This will stop the execution inside the switch block.

- It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

The default Keyword

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

Common Code Blocks

Sometimes you will want different switch cases to use the same code. In the code snippet below; case 1 to case 5 share the same code block, and 6 and 0 share another code block:

```
// JS switch (common code block) statement
switch (new Date().getDay()) {
    case 1:
        day = 'Monday';
    case 2:
        day = 'Tuesday';
    case 3:
        day = 'Wednesday';
    case 4:
        day = 'Thursday';
    case 5:
        day = 'Friday';
        console.log('Is the weekday');
        break;
    case 6:
        day = 'Saturday';
    case 0:
        day = 'Sunday';
        console.log('Is weekend!');
        break;
    default:
        text = "Looking forward to the Weekend";
}
```

Figure 72: JS switch statement (common code block)

Note:

- If multiple cases match a case value, the first case is selected.
- If no matching cases are found, the program continues to the default label.

If no default label is found, the program continues to the statement(s) after the switch.

JavaScript For Loop

Loops can execute a block of code a number of times.

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Different Kinds of Loops

JavaScript supports different kinds of loops:

- for - loops through a block of code a number of times
- while - loops through a block of code while a specified condition is true
- do/while - also loops through a block of code while a specified condition is true

The For Loop

The `for` statement creates a loop with 3 optional expressions:

```
// JS for loop
let printNumbers = '';
for (let x = 0; x < 5; x++) {
    printNumbers += `The number is ${x} <br>`;
}
console.log(printNumbers)
```

Figure 73: JS for loop

Note:

- **Expression 1** sets a variable before the loop starts (let x = 0).
- **Expression 2** defines the condition for the loop to run (x must be less than 5).
- **Expression 3** increases a value (x++) each time the code block in the loop has been executed.
- **Expression 1** is used to initialize the variable(s) used in the loop (let x = 0).
- But, **expression 1** is optional.
- You can **omit expression 1** when your values are set before the loop starts:
- **Expression 2** is used to evaluate the condition of the initial variable (x < 5).
- But, **expression 2** is also optional.

- If **expression** 2 returns true, the loop will start over again. If it returns false, the loop will end.
- If you omit **expression 2**, you must provide a **break** inside the loop. Otherwise, the loop will never end. This will crash your browser.
- **Expression 3** increments the value of the initial variable (x++).
- But, **expression 3** is also optional.
- **Expression 3** can do anything like negative increment (x--), positive increment (x = x + 15), or anything else.
- **Expression 3** can also be omitted (like when you increment your values inside the loop):

```
// JS for loop
let printNumbers = '';
for (let x = 0; x < 5;) {
    printNumbers += `The number is ${x} <br>`;
    x++;
}
console.log(printNumbers)
```

Figure 74: JS for loop II

JavaScript While Loop

Loops can execute a block of code as long as a specified condition is true.

```
// JS while loop
let printNumbers = '';
let x = 0;
while (x < 10) {
    printNumbers += `The number is ${x} <br>`;
    x++;
}
console.log(printNumbers)
```

Figure 75: JS while loop

Note:

- If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The Do While Loop

The `do while` loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
// JS do while loop
let printNumbers = '';
let x = 0;
do{
    printNumbers += `The number is ${x} <br>`;
    x++;
}while (x < 10);
console.log(printNumbers)
```

Figure 76: JS do while loop

Note:

- Do not forget to increase the variable used in the condition, otherwise the loop will never end!
- while loop is much the same as a for loop, with statement 1 and statement 3 omitted.

JavaScript Break and Continue

The break statement "jumps out" of a loop.

The continue statement "jumps over" one iteration in the loop.

```
// JS break statement (loop)
let printNumbers = '';
let x = 0;
for (let x = 0; x < 10; x++) {
    if (x === 3) { break; }
    printNumbers += `The number is ${x}, `;
}
console.log(printNumbers)
```

Figure 77: JS break statement

Note:

- In the code snippet above, the break statement ends the loop ("breaks" the loop) when the loop counter (x) is 3.

The Continue Statement

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
// JS continue statement (loop)
let printNumbers = '';
let x = 0;
for (let x = 0; x < 10; i++) {
    if (x === 3) { continue; }
    printNumbers += `The number is ${x}, `;
}
console.log(printNumbers)
```

Figure 78: JS continue statement

JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

```
function name(parameter1, parameter2, parameter3) {
    // code to be executed
}
```

Figure 79: JS function syntax

- A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses **( )**.
- Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
- The parentheses may include parameter (argument) names separated by commas: ***(parameter1, parameter2, ...)***
- The code to be executed, b]y the function, is placed inside curly brackets: **{ }**

Note:

- Function **parameters (arguments)** are listed inside the parentheses ( ) in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.

When JavaScript reaches a **return** statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

```
function addNumbers(a, b) {
// Function returns the addition of a and b
  return a + b;
}

// Function is called, the return value will end up in x
let x = addNumbers(4, 3);
```

Figure 80: JavaScript Function

JavaScript Arrow Function

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

```
let addNumbers = (a, b) => a + b;

console.log(addNumbers(4, 3))
```

Figure 81: JavaScript Arrow Function

```
//Before Arrow Function
courseName = function() {
    return 'Web Development Course';
}

console.log(courseName());
```

Figure 82: Before Arrow Function

```
//With Arrow Function
courseName = () => {
    return 'Web Development Course';
}

console.log(courseName());
```

Figure 83: With Arrow Function

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the return keyword as shown below:

```
//Arrow Functions Return Value by Default
courseName = () => 'Web Development Course';

console.log(courseName());
```

Figure 84: JS Arrow Function return value by default

If you have parameters, you pass them inside the parentheses as shown below:

```
// Arrow Function With Parameters
courseName = (studentName) => studentName +', Welcome to Web Development Course ';

console.log(courseName('Piccolo'));
```

Figure 85: JS Arrow Function with parameters

In fact, if you have only one parameter, you can skip the parentheses as well as shown below:

```
// Arrow Function Without Parentheses
courseName = studentName => studentName +', Welcome to Web Development Course ';

console.log(courseName('Piccolo'));
```

Figure 86: JS Arrow function without parentheses

Expense Tracker App (Practical 1)

Step 1: Create a folder js and create a Javascript file called script.js as shown below:

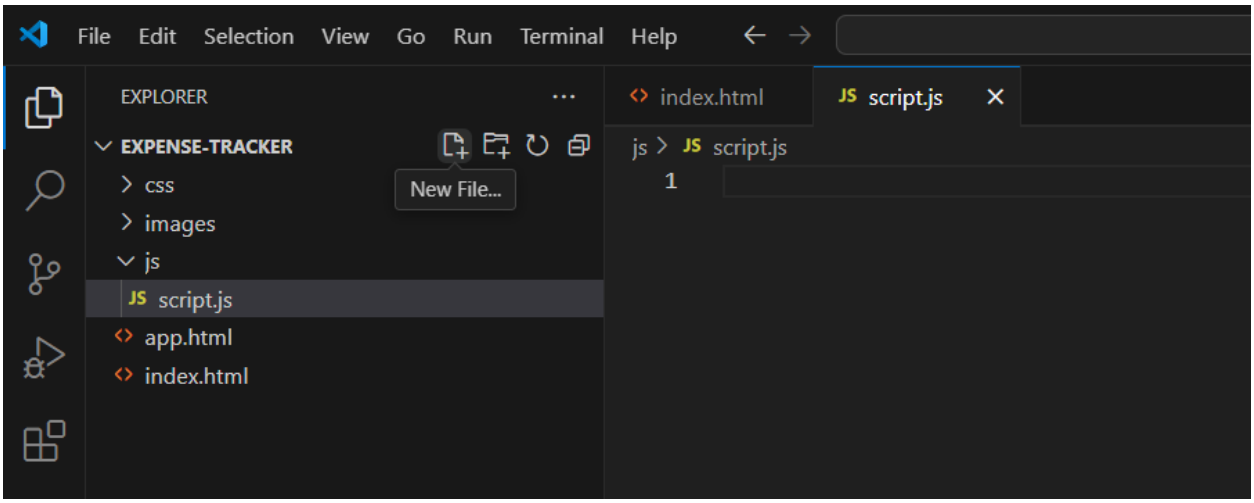


Figure 87: Creating script.js

Step 2: We will start working on the **Good Morning, Mr. Piccolo** section. We will use the date object to make it dynamic based on the time the user opens the app to response with good morning, good afternoon and good evening.

Step 3: in the app.html, add an id attribute with value of **welcomeMsg** to the div which contains the welcome address message as shown below:

```
20      <!-- Welcome Address  -->
21      <div id="welcomeMsg">
22          Good Morning, Mr Piccolo
23      </div>
24      <!-- Icons  -->
```

Figure 88: adding id attribute to welcome address div

```
80      <!-- End of App Area  -->
81      <script src="./js/script.js"></script>
82  </body>
83  </html>
```

Figure 89: Add the script.js link

Step 4: Create the **welcomeMsg** ( ) function in the script.js file as shown below:

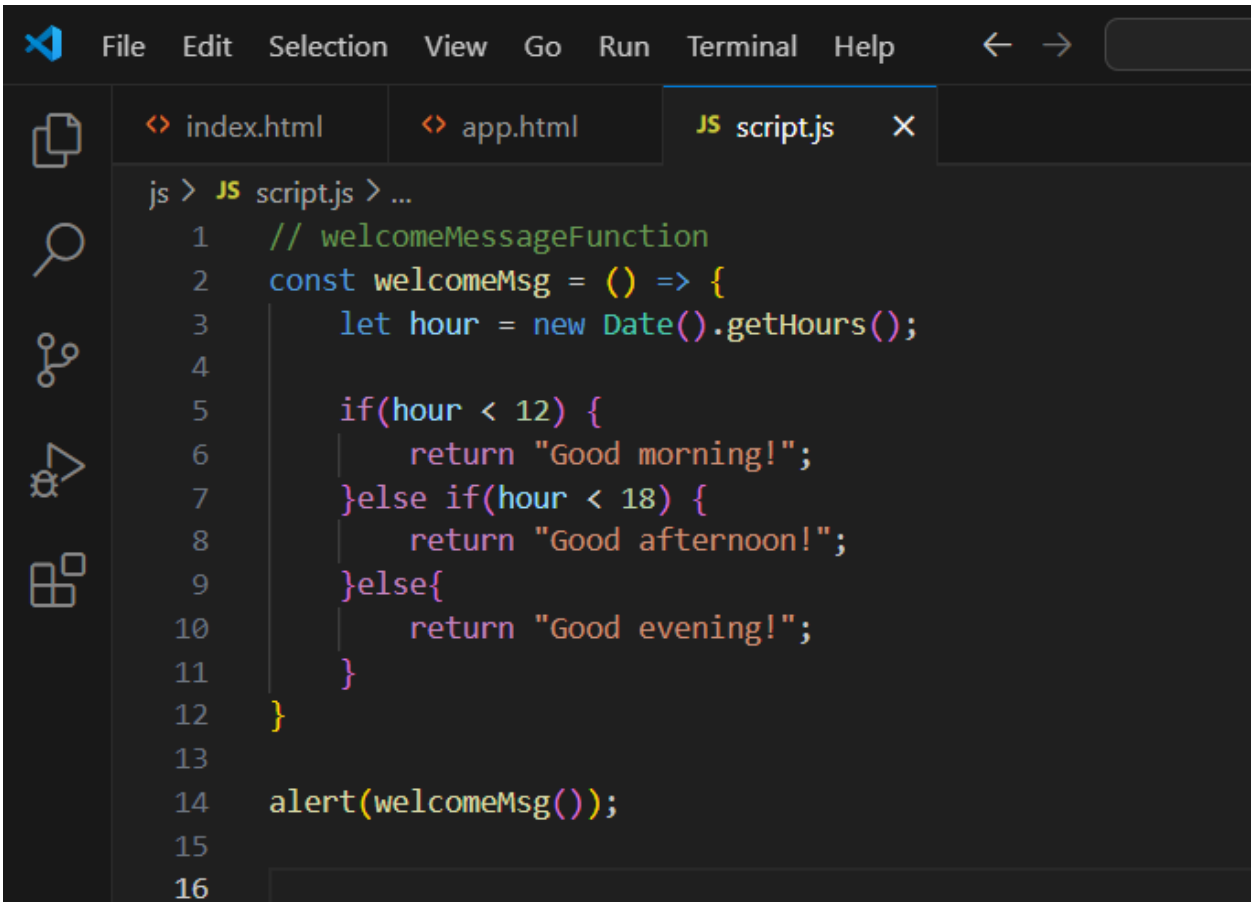


Figure 90: welcomeMsg( )

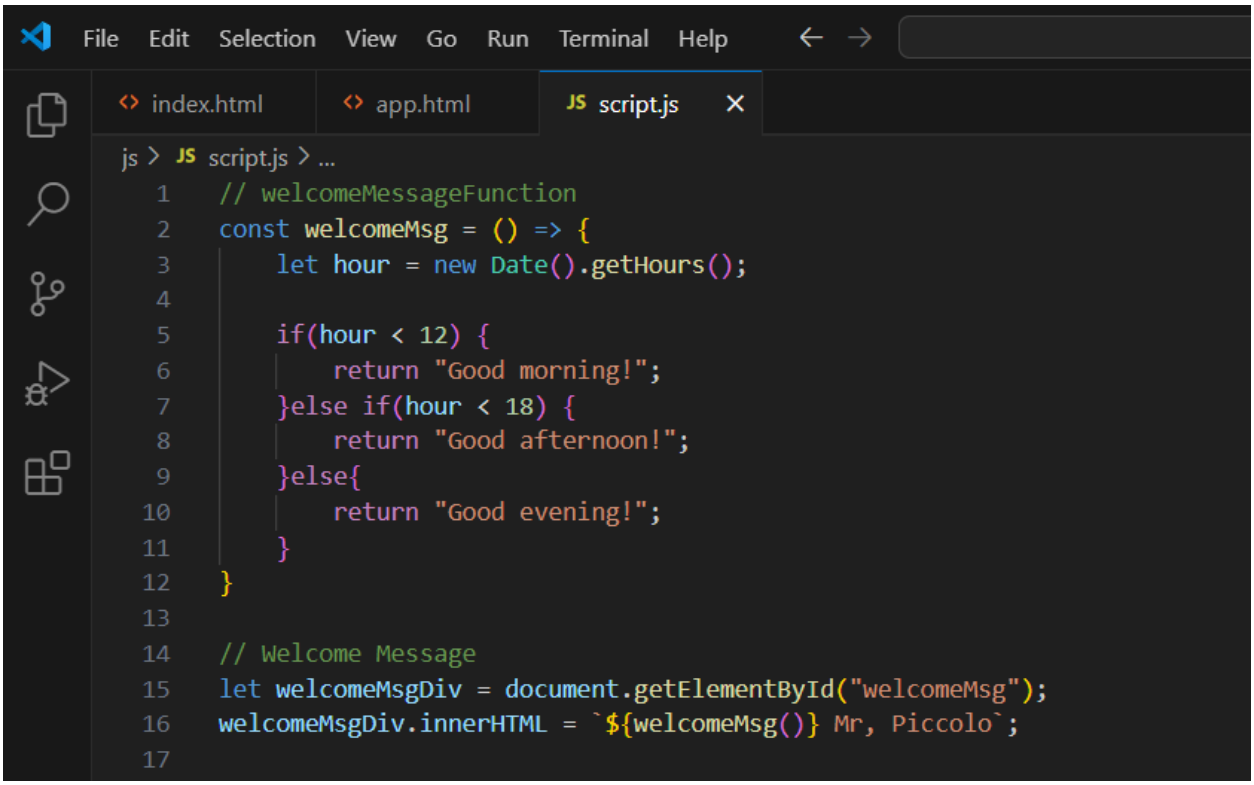


Figure 91: Welcome Message

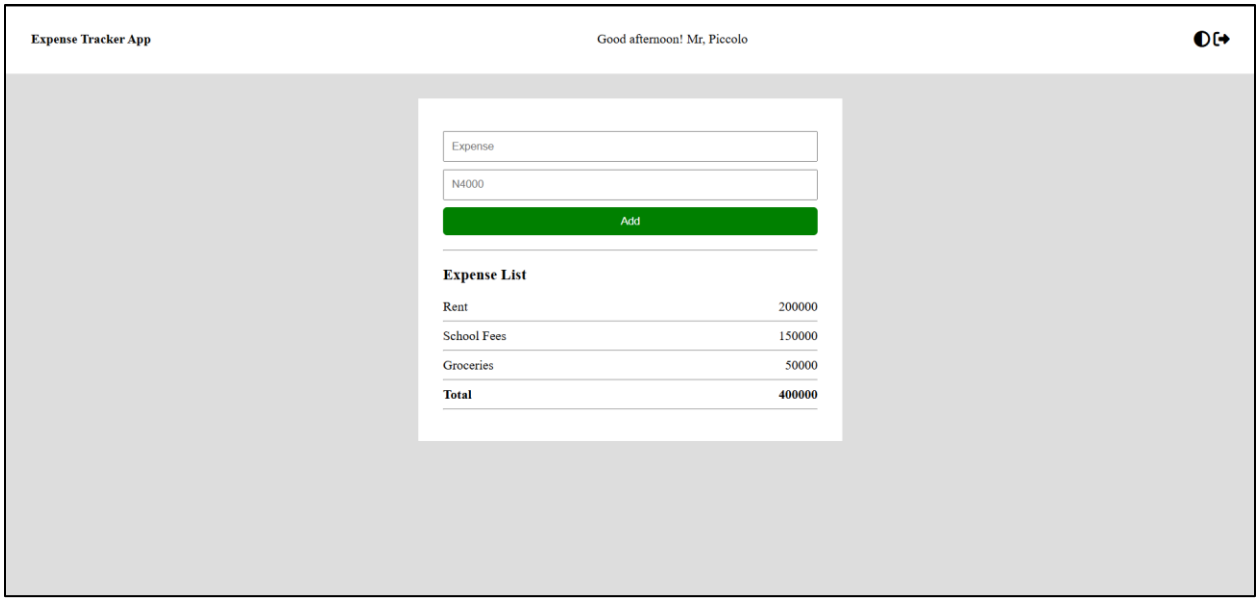


Figure 92: Welcome Message is now dynamic

JavaScript Events

HTML events are **"things"** that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

In the code snippet below, the JavaScript code changes the content of the element with id= "header".

```
<body>
    <h2>JavaScript Events</h2>

    <p id="header"></p>

    <button onclick="displayDate()">The time is?</button>

    <script src="js/script.js"></script>
</body>
```

Figure 93: JavaScript onclick event

```
//Display Date
const displayDate = () => {
    document.getElementById('header').innerHTML = Date();
};
```

Figure 94: displayDate function

Common HTML Events

Event	Description
Onchange	An HTML element has been changed
Onclick	The user clicks an HTML element
Onmouseover	The user moves the mouse over an HTML element
Onmouseout	The user moves the mouse away from an HTML element
Onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JavaScript HTML DOM EventListener

The addEventListener( ) method attaches an event handler to the specified element.

The addEventListener( ) method attaches an event handler to an element without overwriting existing event handlers.

You can add many event handlers to one element.

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

You can add event listeners to any DOM object not only HTML elements. i.e. the window object.

The addEventListener( ) method makes it easier to control how the event reacts to bubbling.

When using the addEventListener( ) method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

You can easily remove an event listener by using the removeEventListener( ) method.

```html
<body>
    <h2>JavaScript EventListener</h2>

    <button id="header">Click me to show the date</button>

    <p id="timeDiv"></p>

    <script src="js/script.js"></script>
</body>
```

Figure 95: JavaScript addEventListener( )

```javascript
let header = document.getElementById('header');
let timeDiv = document.getElementById('timeDiv');

//Display Date
const displayDate = () => {
    timeDiv.innerHTML = Date();
};


header.addEventListener("click", displayDate);
```

Figure 96: JavaScript addEventListener( ) II

Expense Tracker App (Practical II)

Step 1: Let's work on the **dark-mode.** We will create a toggle; the user can toggle to switch from light to dark mode.

```
24          <!-- Icons  -->
25          <div>
26              <!-- <i class="fa-solid fa-sun"></i> -->
27              <i class="fa-solid fa-circle-half-stroke app-icon-font-size"></i>
28              <!-- Logout  -->
29                  <a href="./index.html">
30                      <i class="fa-solid fa-right-from-bracket app-icon-font-size"></i>
31                  </a>
32              <!-- End of Logout  -->
33          </div>
```

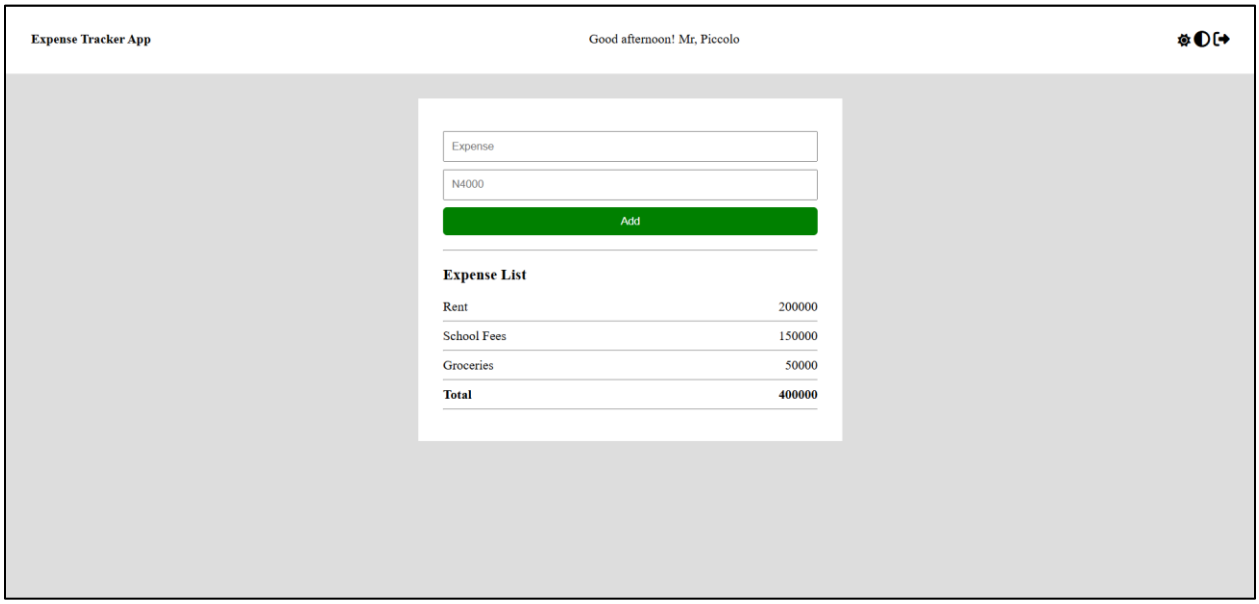
Figure 97: creating dark mode

Step 2: Uncomment the icon and add the label as shown below:

```
<!-- Icons  -->
<div>
    <!-- Light Mode  -->
    <i class="fa-solid fa-sun"></i>
    <!-- Dark Mode  -->
    <i class="fa-solid fa-circle-half-stroke app-icon-font-size"></i>
    <!-- Logout  -->
        <a href="./index.html">
            <i class="fa-solid fa-right-from-bracket app-icon-font-size"></i>
        </a>
    <!-- End of Logout  -->
</div>
```

Figure 98: Creating dark mode II

Step 3: Now, we have three icons, the light, dark and logout icons.



Step 4: add id attributes to the light and dark icon as shown below:

```
<!-- Icons  -->
<div>
    <!-- Light Mode  -->
    <i id="lightModeIcon" class="fa-solid fa-sun"></i>
    <!-- Dark Mode  -->
    <i id="darkModeIcon" class="fa-solid fa-circle-half-stroke app-icon-font-size"></i>
    <!-- Logout  -->
        <a href="./index.html">
            <i class="fa-solid fa-right-from-bracket app-icon-font-size"></i>
        </a>
    <!-- End of Logout  -->
</div>
```

Figure 99: Creating dark mode III

```
// Dark Mode Module
let lightModeIcon = document.getElementById("lightModeIcon");
let darkModeIcon = document.getElementById("darkModeIcon");
```

Figure 100: Creating dark mode IV

Step 9: lets hide the light icon.

```
// Hide light mode icon
if(lightModeIcon){
    lightModeIcon.style.display = 'none';
}
```

Figure 101: hiding the light icon

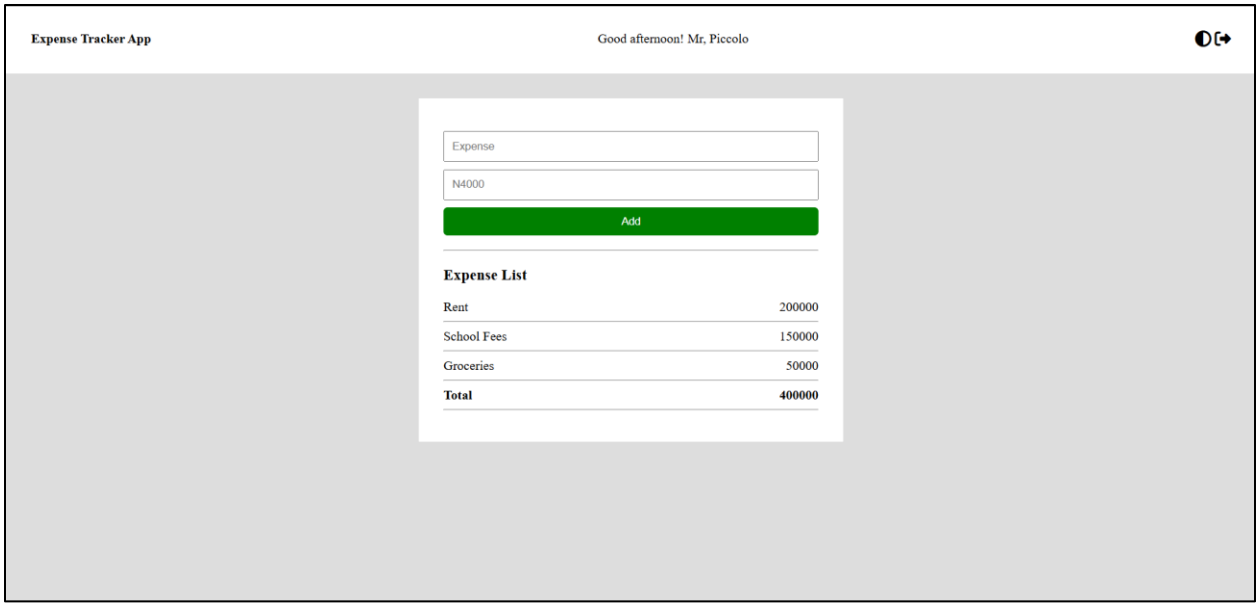


Figure 102: Light mode is hidden

Step 10: Create a function for the dark mode styling as shown below:

```
// Dark Mode Styling
const darkModeStyling = () => {
    let navSection = document.getElementById("navSection");
    let appArea = document.getElementById("appArea");
    let lightModeIcon = document.getElementById("lightModeIcon");
    let darkModeIcon = document.getElementById("darkModeIcon");
    let logoutIcon = document.getElementById("logoutIcon");

    let isDarkMode = document.body.classList.toggle("dark-mode");

    if(navSection){
        navSection.style.backgroundColor = isDarkMode ? "black" : "";
        navSection.style.color = isDarkMode ? "white" : "";
    }

    if(appArea){
        appArea.style.backgroundColor = isDarkMode ? "black" : "";
        appArea.style.color = isDarkMode ? "white" : "";
    }

    if(lightModeIcon){
        lightModeIcon.style.display = isDarkMode ? "inline-block" : "none";
        lightModeIcon.style.color = isDarkMode ? "white" : "";
    }

    if(darkModeIcon){
        darkModeIcon.style.display = isDarkMode ? "none" : "inline-block";
    }

    if(logoutIcon){
        logoutIcon.style.color = isDarkMode ? "white" : "";
    }
};


// Dark Mode Icon
let darkModeIcon = document.getElementById("darkModeIcon");

if(darkModeIcon){
    darkModeIcon.addEventListener("click", darkModeStyling);
}

if(lightModeIcon){
    lightModeIcon.addEventListener("click", darkModeStyling);
}
```

Figure 103: Dark Mode Styling function

```
72  .dark-mode{
73      color: white;
74  }
```

Figure 104: dark mode class

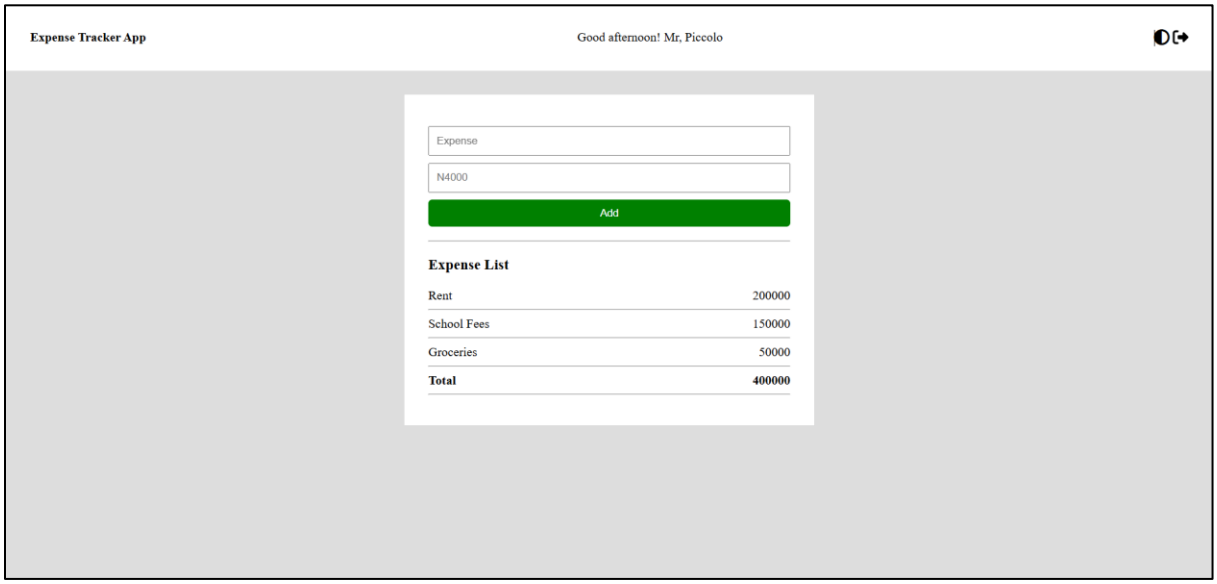


Figure 105: Light Mode

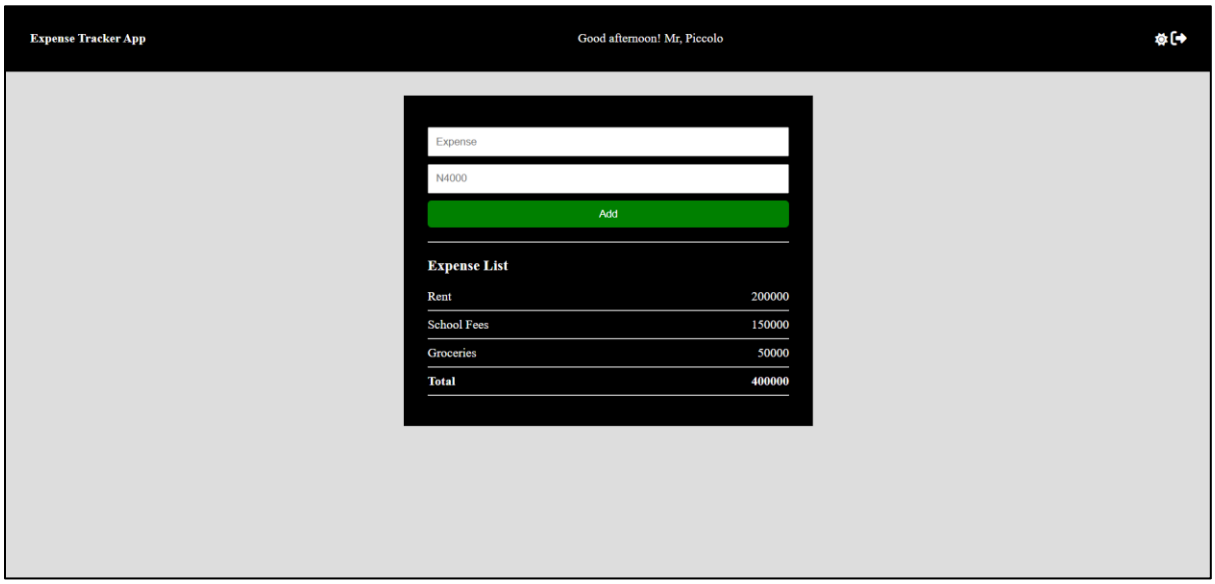


Figure 106: Dark Mode

JavaScript Objects

If you Understand Objects, you Understand JavaScript.

In real life, **objects** are things like: houses, cars, people, animals, or any other subjects.

Here is a **car object** example:

Car Object	Car Properties	Car Methods
	car.name = Honda	car.start()
	car.model = 2025	car.drive()
	car.weight = 1850kg	car.brake()
	car.color = white	car.stop()

Object Properties

A real-life car has **properties** like weight and color:

car.name = Honda, car.model = 2025, car.weight = 1850kg, car.color = white.

Car objects have the same **properties**, but the **values** differ from car to car.

Object Methods

A real-life car has **methods** like start and stop:

car.start(), car.drive(), car.brake(), car.stop().

Car objects have the same **methods**, but the methods are performed **at different times**.

Note:

Objects are variables too. But objects can contain many values.

The code below assigns **many values** (Honda, 2025, white) to an **object** named car:

```
// JS Objects
const car = {
    type:'Honda',
    model:'2025',
    color:'white'
};

console.log(car);
```

Figure 107: JS Object

Accessing Object Properties

You can access object properties in two ways as shown below:

```
// objectName.propertyName
const car = {
    type:'Honda',
    model:'2025',
    color:'white'
};

console.log(car.type);
```

Figure 108: objectName.propertyName

```
// objectName["propertyName"]
const car = {
    type:'Honda',
    model:'2025',
    color:'white'
};

console.log(car['type']);
```

Figure 109: objectName['propertyName']

JavaScript Object Methods

Methods are **actions** that can be performed on objects.

Methods are **function definitions** stored as **property values**.

```
// JavaScript Object Methods
const car = {
    type:'Honda',
    model:'2025',
    color:'white',
    carDescription : function() {
        return this.type + ' ' + this.model + ' with '+ this.color +' color';
    }
};

console.log(car.carDescription());
```

Figure 110: JS Object Methods

Note: In the example above, this refers to the **car object**:

JavaScript Object Properties

An Object is an Unordered Collection of Properties

- Properties are the most important part of JavaScript objects.
- Properties can be changed, added, deleted, and some are read only.

Adding New Properties

You can add new properties to an existing object by simply giving it a value as shown below:

```
// Adding New Properties
const car = {
    type:'Honda',
    model:'2025',
    color:'white',
    carDescription : function() {
        return this.type + ' ' + this.model + ' with '+ this.color +' color';
    }
};

car.price = '$30,000';

console.log(car.price);
```

Figure 111: Adding New property to an Object

Deleting Properties

The delete keyword deletes a property from an object as shown below:

```
// Deleting Properties
const car = {
    type:'Honda',
    model:'2025',
    color:'white',
    carDescription : function() {
        return this.type + ' ' + this.model + ' with '+ this.color +' color';
    }
};

delete car.model;

console.log(car);
```

Figure 112: Deleting property from an Object

Note:

- The delete keyword deletes both the value of the property and the property itself.
- After deletion, the property cannot be used before it is added back again.

Nested Objects

Property values in an object can be other objects.

```
// Nested Objects
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria'
};

console.log(customers.courses.course1);
```

Figure 113: Nested Object

JavaScript Object Methods

Object methods are actions that can be performed on objects.

A method is a **function definition** stored as a **property value**

```
// JavaScript Object Methods
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria',
    coursesRegistered: function(){
        return this.courses.course1 +', '+ this.courses.course2 +', '+ this.courses.course3;
    }
};

console.log(customers.coursesRegistered());
```

Figure 114: JS Object Methods

Using JavaScript Methods

JavaScript have built-in methods. The code snippet below uses the JavaScript toLowerCase( ) method to convert the coursesRegistered method to lowercase:

```
// Using JavaScript Methods
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria',
    coursesRegistered: function(){
        return (this.courses.course1 +', '+ this.courses.course2 +', '+ this.courses.course3).toLowerCase();
    }
};

console.log(customers.coursesRegistered());
```

Figure 115: Using JS methods

JavaScript Display Objects

JavaScript objects can be displayed by:

- Displaying the Object Properties by name
- Displaying the Object Properties in a Loop
- Displaying the Object using Object.values( )
- Displaying the Object using JSON.stringify( )

Displaying Object Properties

The properties of an object can be displayed as a string:

```
// Displaying Object Properties
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria',
    coursesRegistered: function(){
        return (this.courses.course1 +', '+ this.courses.course2 +', '+ this.courses.course3).toLowerCase();
    }
};

console.log('Welcome '+customers.name +', You have registered '+customers.coursesRegistered());
```

Figure 116: Displaying Object Properties

Displaying Properties in a Loop

The properties of an object can be collected in a loop:

```
// Displaying Properties in a Loop
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria'
};

// Loop the courses
let courses = "";

for(let course in customers.courses) {
    courses += customers.courses[course] + ', ';
};

console.log('Welcome '+customers.name +', You have registered '+courses);
```

Figure 117: Displaying Properties using Loop

Using Object.values( )

Object.values( ) creates an array from the property values:

```
// Using Object.values()
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria'
};

// Create an Array
const myCustomers = Object.values(customers);

console.log(myCustomers);
```

Figure 118: Using Object.values( )

Using JSON.stringify( )

JavaScript objects can be converted to a string with JSON method JSON.stringify( ).

JSON.stringify( ) is included in JavaScript and supported in all major browsers.

```
// Using JSON.stringify()
const customers = {
    name:'Piccolo',
    phone: '08068593127',
    courses: {
        course1: 'HTML',
        course2: 'CSS',
        course3: 'JS',
    },
    country: 'Nigeria'
};

// Stringify Object
let myCustomers = JSON.stringify(customers);

console.log(myCustomers);
```

Figure 119: Using JSON.stringify( )

JavaScript Arrays

An array is a special variable, which can hold more than one value.

```
// JavaScript Arrays
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
```

Figure 120: JS arrays

Why Use Arrays?

If you have a list of items (a list of countries, for example), storing the countries in single variables could look like this:

```
let country1 = `Nigeria`;
let country2 = `Ghana`;
let country3 = `Niger`;
let country4 = `Benin Republic`;
```

Figure 121: JS Arrays List of countries

However, what if you want to loop through the countries and find a specific one? And what if you had not 4 countries, but all the countries in the world?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Accessing Array Elements

You access an array element by referring to the **index number**:

Note:

- Array indexes start with 0.
- [0] is the first element. [1] is the second element.

```
// JavaScript Accessing Array Items
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
console.log(countries[2]);
```

Figure 122: JS accessing array elements

Changing an Array Element

The code snippet below changes the value of the third element in the countries array from 'Niger' to 'Gabon':

```
// JavaScript Changing an Array Element
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries[2] = 'Gabon';
console.log(countries[2]);
```

Figure 123: Changing an Array Element

Converting an Array to a String

The JavaScript method toString( ) converts an array to a string of (comma separated) array values.

```
// Converting an Array to a String
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
console.log(countries.toString());
```

Figure 124: Converting an Array to String

The length Property

The length property of an array returns the length of an array (the number of array elements).

```
// The length Property
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
console.log(countries.length);
```

Figure 125: JS array length property

Looping Array Elements

One way to loop through an array, is using a for loop:

```
// Loop through an Array
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
    let cLen = countries.length;
    let listOfCountries = `<ul>`;
        for (let i = 0; i < cLen; i++) {
            listOfCountries += `<li>${ countries[i] }</li>`;
        }
    listOfCountries += `</ul>`;
console.log(listOfCountries);
```

Figure 126: Loop through an Array

You can also use the Array.forEach( ) function

```
// Loop through an Array using forEach
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
    let listOfCountries = `<ul>`;
        const countriesVisited = (country) => {
            listOfCountries += `<li>${country}</li>`;
        };
        countries.forEach(countriesVisited);
    listOfCountries += `</ul>`;
console.log(listOfCountries);
```

Figure 127: Loop through an Array using forEach( )

Adding Array Elements

The easiest way to add a new element to an array is using the push( ) method:

```
// Adding an item to an Array using push()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.push('Mali')
console.log(countries);
```

Figure 128: JS array push( )

The Difference Between Arrays and Objects

- In JavaScript, **arrays** use **numbered indexes**.
- In JavaScript, **objects** use **named indexes**

When to Use Arrays. When to use Objects.

- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

JavaScript Array Methods

JavaScript Array at( )

ES2022 introduced the array method at( ):

```
// Array at()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
let country = countries.at(1)
console.log(country);
```

Figure 129: JS array at( ) method

JavaScript Array join( )

The join( ) method also joins all array elements into a string.

It behaves just like toString( ), but in addition you can specify the separator.

```
// Array join()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
let countriesJoined = countries.join(', ');
console.log(countriesJoined);
```

Figure 130: JS array join( ) method

When you work with arrays, it is easy to remove elements and add new elements.

This is what **popping** and **pushing** is:

Popping items **out** of an array, or **pushing** items **into** an array.

JavaScript Array pop( )

The pop( ) method removes the last element from an array:

```
// Array pop()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.pop();
console.log(countries);
```

Figure 131: JS array pop( ) method

JavaScript Array push( )

The push( ) method adds a new element to an array **(at the end)**:

```
// Array push()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.push('Mali');
console.log(countries);
```

Figure 132: JS array push( ) method

JavaScript Array shift( )

The shift( ) method removes the first array element and "shifts" all other elements to a lower index.

```
// Array shift()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.shift();
console.log(countries);
```

Figure 133: JS array shift( ) method

JavaScript Array unshift( )

The unshift( ) method adds a new element to an array **(at the beginning)**, and "unshifts" older elements:

```
// Array unshift()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.unshift('Mali');
console.log(countries);
```

Figure 134: JS array unshift( ) method

Note:

- Using delete( ) leaves undefined holes in the array.
- Use pop( ) or shift( ) instead.

JavaScript Array splice( )

The `splice( )` method can be used to add new items to an array:

```
// Array splice()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.splice(2, 0, 'Mali', 'Togo');
console.log(countries);
```

Figure 135: JS array splice( ) method

Note:

- The first parameter (**2**) defines the position **where** new elements should be **added** (spliced in).
- The second parameter (**0**) defines **how many** elements should be **removed**.
- The rest of the parameters ('Mali' , 'Togo') define the new elements to be **added**.

Using splice() to Remove Elements

With clever parameter setting, you can use `splice( )` to remove elements without leaving "holes" in the array:

```
// Array splice() Removing items
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
countries.splice(0, 1);
console.log(countries);
```

Figure 136: JS array splice( ) method to remove elements

Note:

- The first parameter (0) defines the position where new elements should be **added** (spliced in).
- The second parameter (1) defines **how many** elements should be **removed**.
- The rest of the parameters are omitted. No new elements will be added.

JavaScript Array slice( )

The `slice( )` method slices out a piece of an array into a new array:

```
// Array slice()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
const slicedCountries = countries.slice(1);
console.log(slicedCountries);
```

Figure 137: JS array slice( )

Note:

- The code snippet above sliced out a part of an array starting from array element 1 **('Nigeria')**
- The `slice( )` method creates a new array.
- The `slice( )` method does not remove any elements from the source array.

JavaScript Array Search

JavaScript Array indexOf( )

The indexOf( ) method searches an array for an element value and returns its position.

```
// Array indexOf()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
let position = countries.indexOf('Ghana');
console.log(position);
```

Figure 138: JS array indexOf( ) method

JavaScript Array includes( )

ECMAScript 2016 introduced Array.includes( ) to arrays. This allows us to check if an element is present in an array (including NaN, unlike indexOf)

```
// Array includes()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
let find = countries.includes('Niger');
console.log(find);
```

Figure 139: JS array includes( ) method

JavaScript Array find( )

The find( ) method returns the value of the first array element that passes a test function.

This example finds (returns the value of) the first element that is larger than 18

```
// Array find()
const findNumberGreaterThanEight = (value) => {
    return value > 8;
}

const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(findNumberGreaterThanEight);
console.log(first);
```

Figure 140: JS array find( ) method

JavaScript Array findIndex( )

The findIndex( ) method returns the index of the first array element that passes a test function.

The code snippet below finds the index of the first element that is larger than 8.

```
// Array findIndex()
const findNumberGreaterThanEight = (value) => {
    return value > 8;
}

const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(findNumberGreaterThanEight);
console.log(first);
```

Figure 141: JS array findIndex( ) method

JavaScript Sorting Arrays

Sorting an Array

The sort( ) method sorts an array alphabetically.

```
// Array sort()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
let sortCountries = countries.sort();
console.log(sortCountries);
```

Figure 142: JS array sort( ) method

Reversing an Array

The reverse( ) method reverses the elements in an array.

```
// Array reverse()
let countries = ['Nigeria', 'Ghana', 'Niger', 'Benin Republic'];
let reverseCountries = countries.reverse();
console.log(reverseCountries);
```

Figure 143: JS array reverse( ) method

Note:

- By combining sort( ) and reverse( ), you can sort an array in descending order

JavaScript Array toSorted( ) Method

ES2023 added the `toSorted( )` method as a safe way to sort an array without altering the original array.

The difference between `toSorted( )` and `sort( )` is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

JavaScript Array toReversed( ) Method

ES2023 added the toReversed( ) method as a safe way to reverse an array without altering the original array.

The difference between toReversed( ) and reverse( ) is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

JavaScript Array Iteration

JavaScript Array forEach( )

The forEach( ) method calls a function (a callback function) once for each array element.

```
// JS forEach array iteration
const myFunction = (value) => {
  txt += `${value}, `;
}
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);
console.log(txt);
```

Figure 144: JS array forEach( ) method

Expense Tracker App (Practical 3)

Step 1:

Next, let start working on the form, when the user adds the expense and price, it will be added to the expense list.

Step 2:

Add an id attribute to both the expense name and the price input boxes as shown below:

```
<!-- form  -->
<div>
    <div>
        <input id="expenseItem" class="input-field" type="text" placeholder="Expense">
    </div>
    <div>
        <input id="expensePrice" class="input-field" type="amount" placeholder="N4000">
    </div>
    <div>
        <input id="expenseAdd" class="app-btn app-margin-y-2" type="submit" value="Add">
    </div>
</div>
```

Figure 145: Adding id to form input boxes

Step 3:

Write the following script to check if everything is set

```
// Expense Form
let expenseItem = document.getElementById("expenseItem");

if(expenseItem){
    expenseItem.addEventListener("keyup", function() {
        alert(expenseItem.value);
    });
}
```

Figure 146: keyup EventListener

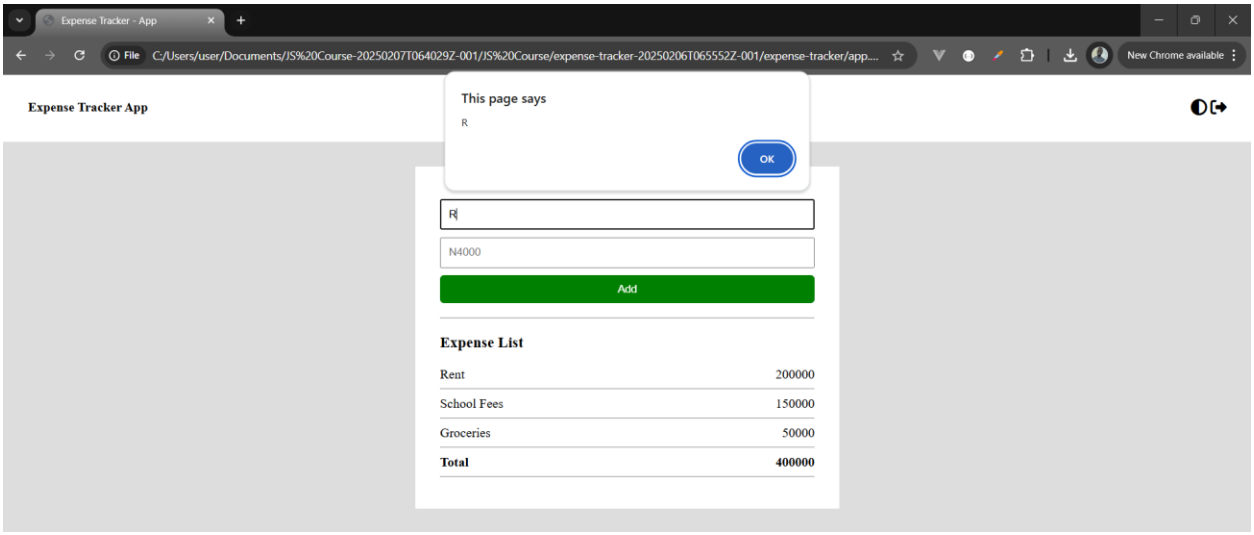


Figure 147: Working perfectly

Step 4: Let's store the values in a variable as shown below:

```
// Expense Form
let expenseItem = document.getElementById("expenseItem");
let expensePrice = document.getElementById("expensePrice");
let expenseAdd = document.getElementById("expenseAdd");
```

Figure 148: Storing the variables

Step 5: Let's print the values of the variables when the user clicks on the add button

```
// Expense Form
let expenseItem = document.getElementById("expenseItem");
let expensePrice = document.getElementById("expensePrice");
let expenseAdd = document.getElementById("expenseAdd");

if(expenseAdd){
    expenseAdd.addEventListener("click", () => {
        alert(`${expenseItem.value} with a price of ${expensePrice.value}`);
    });
}
```

Figure 149: Printing the variables

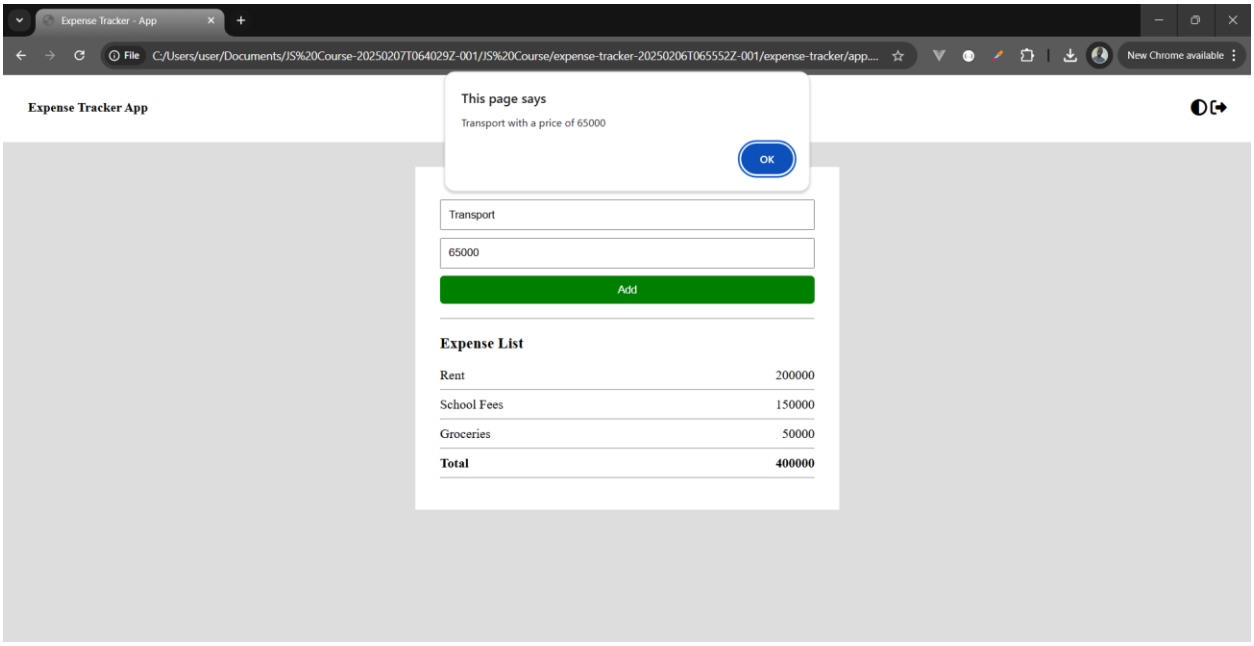


Figure 150: Printing the variables II

Step 6: Let's create a default object and add the expense list as shown below:

```
// Expenses Data
let expenses = [
    {
        item: 'Rent',
        price: 200000
    },
    {
        item: 'School Fees',
        price: 150000
    },
    {
        item: 'Groceries',
        price: 50000
    }
];
```

Figure 151: Default expenses data

Step 7: Let's remove the hardcoded expenses list we have in our app.html file as shown below:

```
<!-- List  -->
<div>
    <h3>Expense List</h3>
    <!-- item 1  -->
    <div class="expense-item">
        <span>Rent</span>
        <span>200000</span>
    </div>
    <hr>
    <!-- item 2  -->
    <div class="expense-item">
        <span>School Fees</span>
        <span>150000</span>
    </div>
    <hr>
    <!-- item 3  -->
    <div class="expense-item">
        <span>Groceries</span>
        <span>50000</span>
    </div>
    <hr>
    <!-- item 4  -->
    <div class="expense-item">
        <span><b>Total</b></span>
        <span><b>400000</b></span>
    </div>
    <hr>
</div>
```

Figure 152: Hardcoded data

```
<!-- List  -->
<div>
    <h3>Expense List</h3>
    <div id="expenseList">

    </div>
</div>
```

Figure 153: New div created

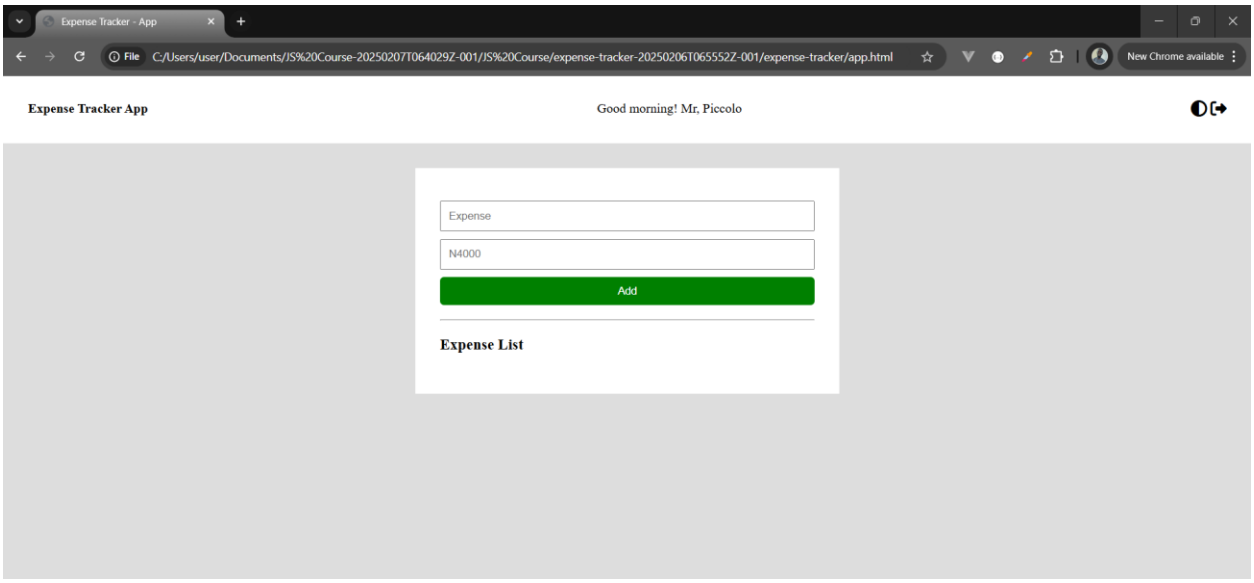


Figure 154: Items removed

```
let expenseList = document.getElementById("expenseList");

// Looping each expense
expenses.forEach(expense => {
    expenseList.innerHTML +=
        `<div class="expense-item">
            <span>${expense.item}</span>
            <span>${expense.price}</span>
        </div>
        <hr>`
});
```

Figure 155: Loop and display the expenses data

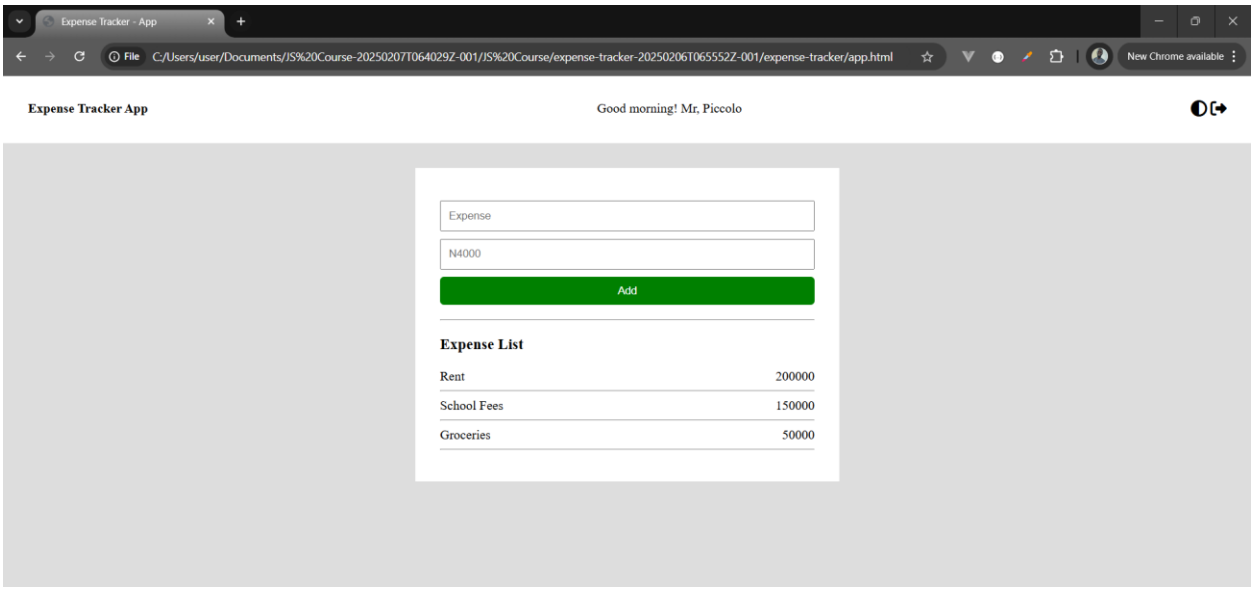


Figure 156: The items are displayed

Step 8: Now let's create the total.

```
// Getting Total Sum
let total = expenses.reduce((sum, expense) => sum + expense.price, 0);
// Add it to the list
    expenseList.innerHTML +=
        `<div class="expense-item">
            <span><b>Total</b></span>
            <span><b>${total}</b></span>
        </div>
        <hr>`
```

Figure 157: Sum the total Expense

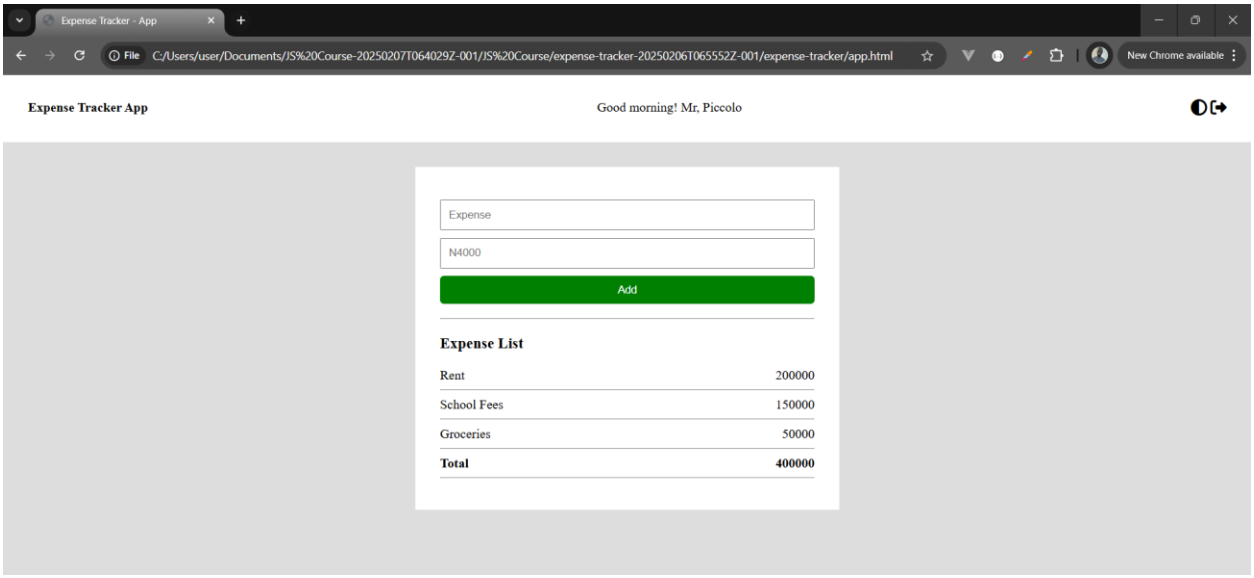


Figure 158: Showing total

Step 9: Let's make it dynamic, once we click on the add button, the items will be added to the expenses data.

```
// Add Item to the expense data
if(expenseAdd){
    expenseAdd.addEventListener("click", () => {
        if(expenseItem.value && expensePrice.value) {
            let expense = {
                item: expenseItem.value,
                price: parseFloat(expensePrice.value)
            };
            expenses.push(expense);
            console.log(expenses)
        }
    });
}
```

Figure 159: adding new item to the expense data

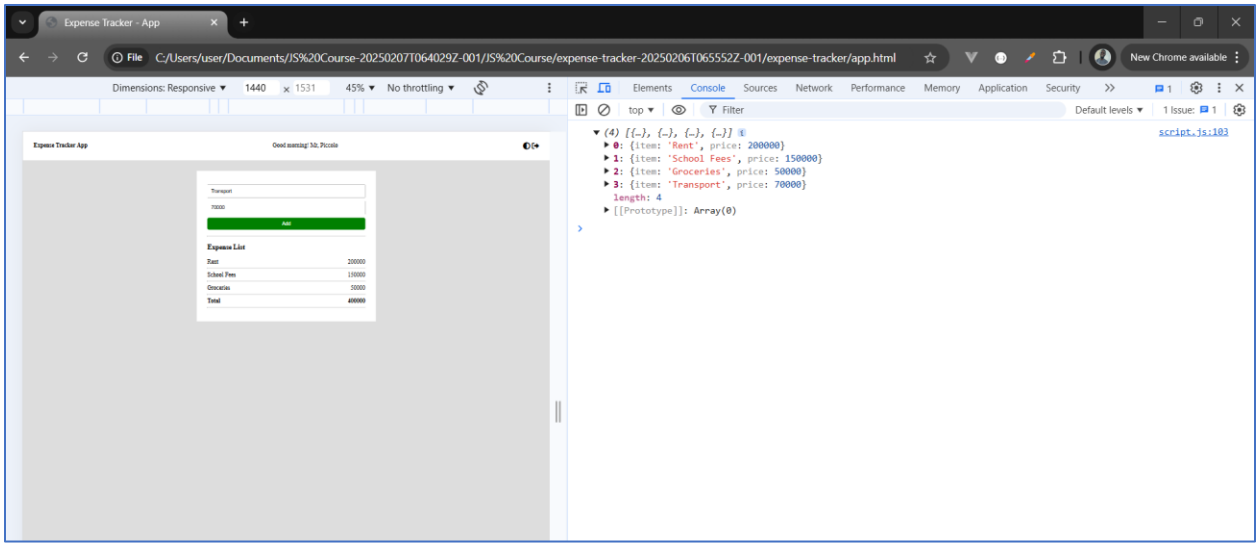


Figure 160: the item has been added

Step 10: Let's update our UI, to make it render dynamic

```
// Expenses Data
let expenses = [
    { item: 'Rent', price: 200000 },
    { item: 'School Fees', price: 150000 },
    { item: 'Groceries', price: 50000 }
];

let expenseItem = document.getElementById("expenseItem");
let expensePrice = document.getElementById("expensePrice");
let expenseAdd = document.getElementById("expenseAdd");
let expenseList = document.getElementById("expenseList");

// Function to render expenses
const renderExpenses = () => {
    expenseList.innerHTML = ""; // Clear previous list

    expenses.forEach(expense => {
        expenseList.innerHTML +=
            `<div class="expense-item">
                <span>${expense.item}</span>
                <span>${expense.price}</span>
            </div>
            <hr>`;
    });

    // Calculate total
    let total = expenses.reduce((sum, expense) => sum + expense.price, 0);
```

```
98
99      // Calculate total
100     let total = expenses.reduce((sum, expense) => sum + expense.price, 0);
101
102     // Add total to the list
103     expenseList.innerHTML +=
104         `<div class="expense-item">
105             <span><b>Total</b></span>
106             <span><b>${total}</b></span>
107         </div>
108         <hr>`;
109 };
110
111 // Initial rendering
112 renderExpenses();
113
114 // Add Item to the expense data
115 if (expenseAdd) {
116     expenseAdd.addEventListener("click", () => {
117         if (expenseItem.value && expensePrice.value) {
118             let expense = {
119                 item: expenseItem.value,
120                 price: parseFloat(expensePrice.value)
121             };
122             expenses.push(expense);
123
124             renderExpenses(); // Update UI with new expense
125
126             // Clear input fields
127             expenseItem.value = "";
128             expensePrice.value = "";
129         }
130     });
131 }
```

Figure 161: Updated Script to render the UI

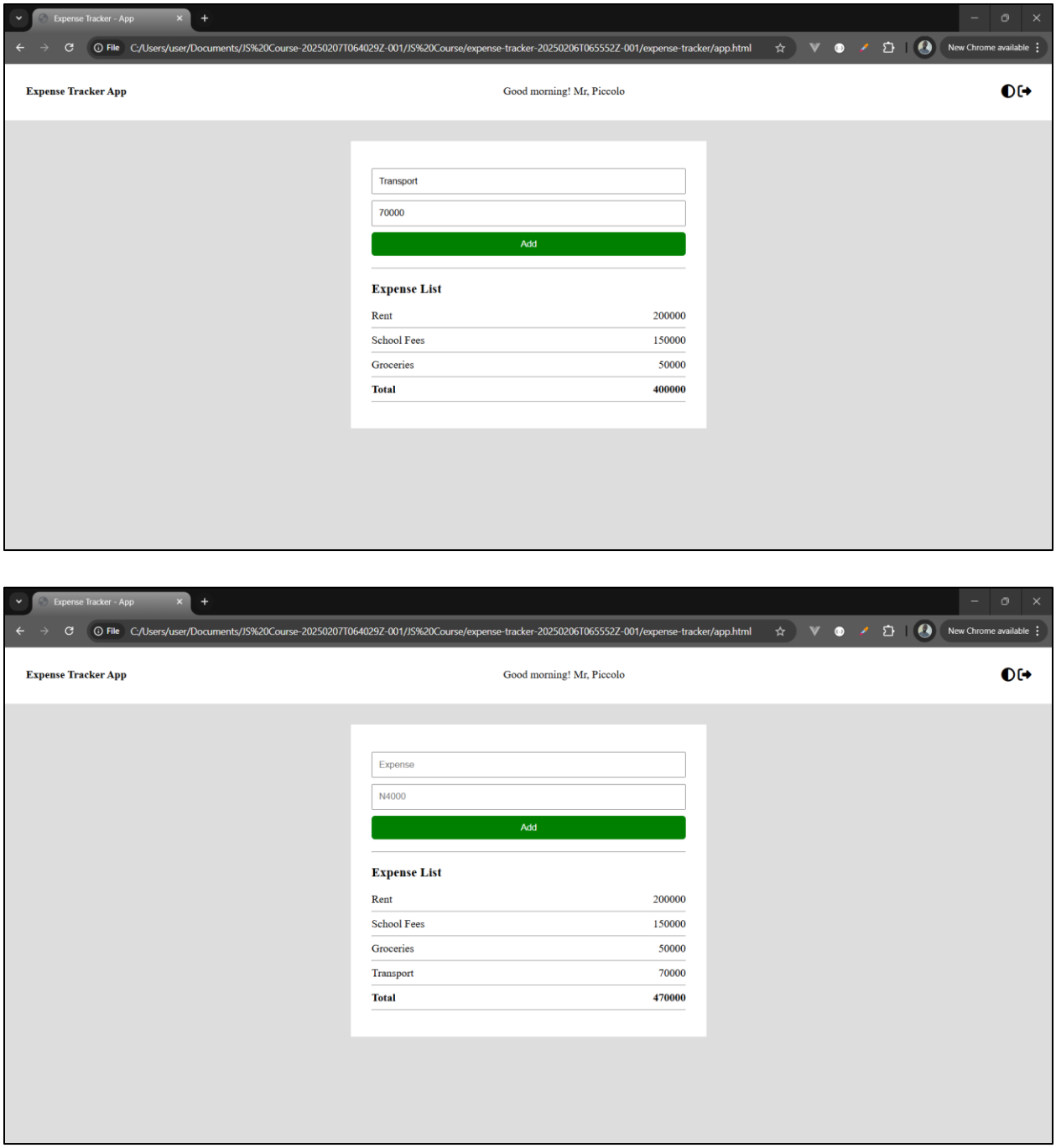


Figure 162: New item added and the UI have been updated

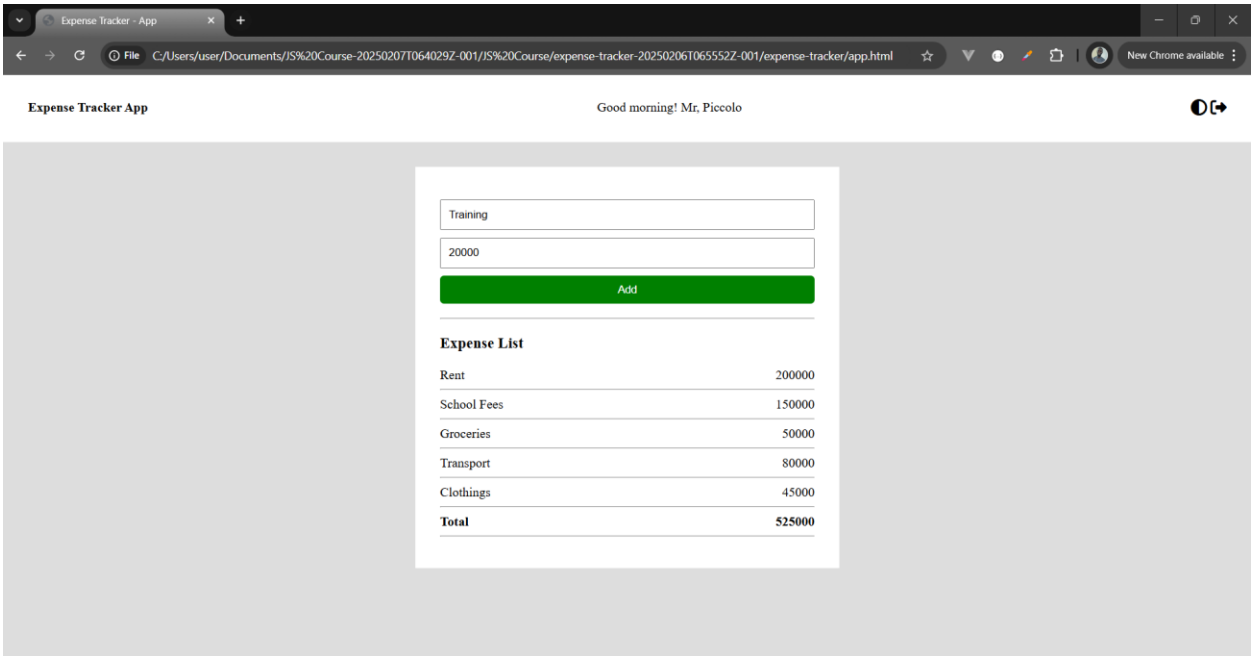
Step 11: If you notice, once we reloaded the page, the new data are deleted. To resolve this, we will use JavaScript **localStorage.**

```
// Expenses Data
let expenses = JSON.parse(localStorage.getItem("expenses")) || [
    { item: 'Rent', price: 200000 },
    { item: 'School Fees', price: 150000 },
    { item: 'Groceries', price: 50000 }
];

let expenseItem = document.getElementById("expenseItem");
let expensePrice = document.getElementById("expensePrice");
let expenseAdd = document.getElementById("expenseAdd");
let expenseList = document.getElementById("expenseList");

// Function to render expenses
const renderExpenses = () => {
    expenseList.innerHTML = ""; // Clear previous list

    expenses.forEach(expense => {
        expenseList.innerHTML +=
            `<div class="expense-item">
                <span>${expense.item}</span>
                <span>${expense.price}</span>
            </div>
            <hr>`;
    });

    // Calculate total
    let total = expenses.reduce((sum, expense) => sum + expense.price, 0);
```

```
    // Calculate total
    let total = expenses.reduce((sum, expense) => sum + expense.price, 0);

    // Add total to the list
    expenseList.innerHTML +=
        `<div class="expense-item">
            <span><b>Total</b></span>
            <span><b>${total}</b></span>
        </div>
        <hr>`;
};

// Save to localStorage
const saveExpenses = () => {
    localStorage.setItem("expenses", JSON.stringify(expenses));
};

// Initial rendering from localStorage
renderExpenses();

// Add Item to the expense data
if (expenseAdd) {
    expenseAdd.addEventListener("click", () => {
        if (expenseItem.value && expensePrice.value) {
            let expense = {
                item: expenseItem.value,
                price: parseFloat(expensePrice.value)
            };
            expenses.push(expense);

            saveExpenses(); // Save new expenses to localStorage
            renderExpenses(); // Update UI with new expense

```

```
119  // Add Item to the expense data
120  if (expenseAdd) {
121      expenseAdd.addEventListener("click", () => {
122          if (expenseItem.value && expensePrice.value) {
123              let expense = {
124                  item: expenseItem.value,
125                  price: parseFloat(expensePrice.value)
126              };
127              expenses.push(expense);
128
129              saveExpenses(); // Save new expenses to localStorage
130              renderExpenses(); // Update UI with new expense
131
132              // Clear input fields
133              expenseItem.value = "";
134              expensePrice.value = "";
135          }
136      });
137  }
```

Figure 163: Updated Code



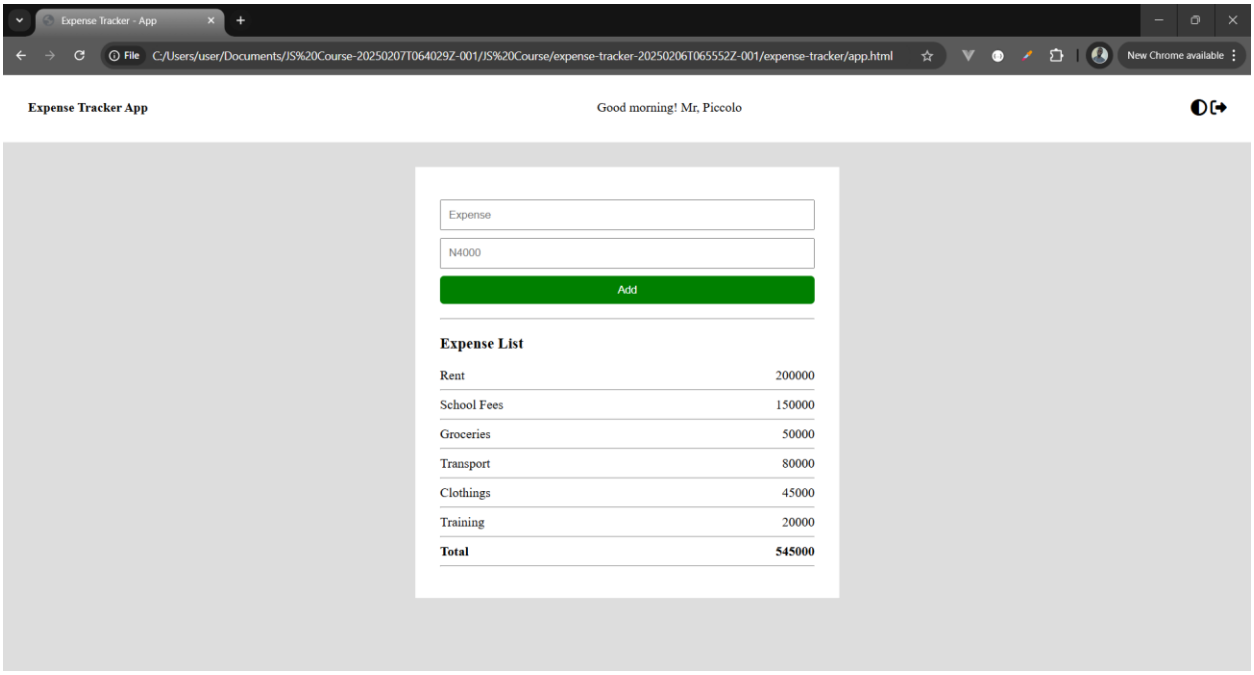


Figure 164: Now the data are stored

Step 12: Remove an item from the list

```
// Expenses Data
let expenses = JSON.parse(localStorage.getItem("expenses")) || [
    { item: 'Rent', price: 200000 },
    { item: 'School Fees', price: 150000 },
    { item: 'Groceries', price: 50000 }
];

let expenseItem = document.getElementById("expenseItem");
let expensePrice = document.getElementById("expensePrice");
let expenseAdd = document.getElementById("expenseAdd");
let expenseList = document.getElementById("expenseList");

// Function to render expenses
const renderExpenses = () => {
    expenseList.innerHTML = ""; // Clear previous list

    expenses.forEach((expense, index) => {
        expenseList.innerHTML +=
            `<div class="expense-item">
                <span>${expense.item}</span>
                <span>${expense.price}</span>
                <button onclick="deleteExpense(${index})"><i class="fa-solid fa-trash"></i></button>
            </div>
            <hr>`;
    });

    // Calculate total
    let total = expenses.reduce((sum, expense) => sum + expense.price, 0);

```

```
            <hr>`;
        });

        // Calculate total
        let total = expenses.reduce((sum, expense) => sum + expense.price, 0);

        // Add total to the list
        expenseList.innerHTML +=
            `<div class="expense-item">
                <span><b>Total</b></span>
                <span><b>${total}</b></span>
            </div>
            <hr>`;
    };

    // Save to localStorage
    const saveExpenses = () => {
        localStorage.setItem("expenses", JSON.stringify(expenses));
    };

    // Function to delete an expense
    const deleteExpense = (index) => {
        expenses.splice(index, 1); // Remove the expense
        saveExpenses(); // Update localStorage
        renderExpenses(); // Update UI
    };

    // Initial rendering from localStorage
    renderExpenses();
```

```
123
124   // Initial rendering from localStorage
125   renderExpenses();
126
127   // Add Item to the expense data
128   if (expenseAdd) {
129       expenseAdd.addEventListener("click", () => {
130           if (expenseItem.value && expensePrice.value) {
131               let expense = {
132                   item: expenseItem.value,
133                   price: parseFloat(expensePrice.value)
134               };
135               expenses.push(expense);
136
137               saveExpenses(); // Save new expenses to localStorage
138               renderExpenses(); // Update UI with new expense
139
140               // Clear input fields
141               expenseItem.value = "";
142               expensePrice.value = "";
143           }
144       });
145   }
```

Figure 165: Add delete

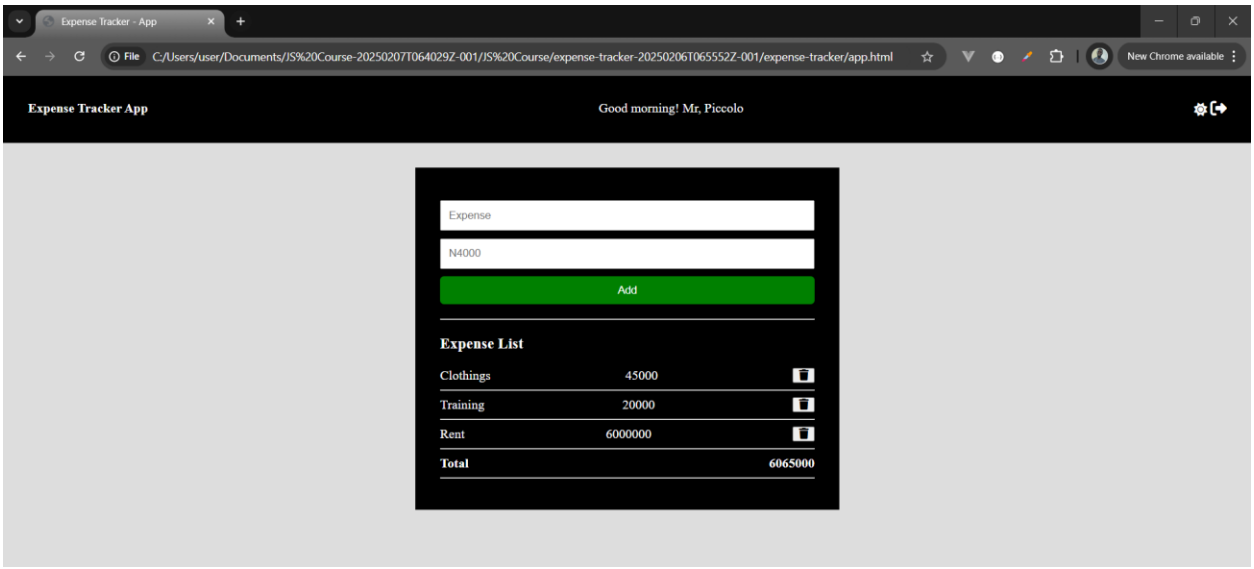


Figure 166: Complete Expense Tracker App

JavaScript Array map( )

The map( ) method creates a new array by performing a function on each array element.

The map( ) method does not execute the function for array elements without values.

The map( ) method does not change the original array.

This example multiplies each array value by 2:

```
// JS map() array iteration
const numbers1 = [45, 4, 9, 16, 25];
const myFunction = (value, index, array)  => {
    return value * 2;
}
const numbers2 = numbers1.map(myFunction);
console.log(numbers2);
```

Figure 167 JS array map( ) method

Note: When a callback function uses only the value parameter, the index and array parameters can be omitted.

JavaScript Array filter( )

The filter( ) method creates a new array with array elements that pass a test.

This example creates a new array from elements with a value less than 18

```
// JS filter() array iteration
const numbers = [45, 4, 9, 16, 25];
const myFunction = (value) =>{
    return value < 18;
}
const lessThan18 = numbers.filter(myFunction);
console.log(lessThan18)
```

Figure 168: JS array filter( ) method

JavaScript Array Spread (...)

The ... operator expands an iterable (like an array) into more elements:

```
// JS ... spread
const q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const q3 = ["Jul", "Aug", "Sep"];
const q4 = ["Oct", "Nov", "Dec"];
const year = [...q1, ...q2, ...q3, ...q4];
console.log(year);
```

Figure 169: JS array spread (...)

JavaScript Maps

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

How to Create a Map

You can create a JavaScript Map by:

- Passing an Array to new Map( )
- Create a Map and use Map.set( )

The new Map( ) Method

You can create a Map by passing an Array to the `new Map()` constructor:

```
// JS Create a Map
const cars = new Map([
    ["Honda", 50000],
    ["Toyota", 30000],
    ["Mercedes Benz", 70000]
]);
```

Figure 170: JS new Map( ) constructor

The set( ) Method

You can add elements to a Map with the set( ) method:

```
// Set Map Values
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);
console.log(cars);
```

Figure 171: JS map set( ) method

The get( ) Method

The get( ) method gets the value of a key in a Map:

```
// Map get() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);
console.log(cars.get("Toyota"));
```

Figure 172: JS map get( ) method

Map.size

The size property returns the number of elements in a map:

```
// Map size property
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);
console.log(cars.size);
```

Figure 173: JS map size property

Map.delete( )

The delete( ) method removes a map element:

```
// Map delete() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);
console.log(cars.delete("Honda"));
```

Figure 174: JS map delete( ) method

Map.clear( )

The clear( ) method removes all the elements from a map:

```
// Map clear() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);
console.log(cars.clear());
```

Figure 175: JS map clear( ) method

Map.has( )

The has( ) method returns true if a key exists in a map:

```
// Map has() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);
console.log(cars.has("Honda"));
```

Figure 176: JS map has( ) method

Map.forEach( )

The forEach( ) method invokes a callback for each key/value pair in a map:

```
// Map forEach() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);

// Loop through the map
let listOfCars = "";
cars.forEach (function(value, key) {
    listOfCars += `${key} price is ${value}, `;
})

// Print the list of cars
console.log(listOfCars);
```

Figure 177: JS Map forEach( ) method

Map.entries( )

The entries( ) method returns an iterator object with the [key,values] in a map:

```
// Map entries() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);

// Loop through the map
let listOfCars = "";
    for(const x of cars.entries()) {
        listOfCars += x;
    }

// Print the list of cars
console.log(listOfCars);
```

Figure 178: JS Map entries( ) method

Map.keys( )

The keys( ) method returns an iterator object with the keys in a map:

```
// Map keys() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);

// Loop through the map
let listOfCars = "";
    for(const x of cars.keys()) {
        listOfCars += x;
    }

// Print the list of cars
console.log(listOfCars);
```

Figure 179: JS array keys( ) method

Map.values( )

The values( ) method returns an iterator object with the values in a map:

```
// Map values() method
const cars = new Map();
cars.set("Honda", 50000);
cars.set("Toyota", 30000);
cars.set("Mercedes Benz", 70000);

// Loop through the map
let listOfCars = "";
    for(const x of cars.values()) {
        listOfCars += x;
    }

// Print the list of cars
console.log(listOfCars);
```

Figure 180: JS array values( ) method

References

1) https://www.w3schools.com/js/default.asp
2) https://www.w3schools.com/js/js_whereto.asp
3) https://www.w3schools.com/js/js_output.asp
4) https://www.w3schools.com/js/js_variables.asp
5) https://www.w3schools.com/js/js_operators.asp
6) https://www.w3schools.com/js/js_comparisons.asp
7) https://www.w3schools.com/js/js_datatypes.asp
8) https://www.w3schools.com/js/js_functions.asp
9) https://www.w3schools.com/js/js_arrow_function.asp
10) https://www.w3schools.com/js/js_objects.asp
11) https://www.w3schools.com/js/js_object_property.asp
12) https://www.w3schools.com/js/js_object_method.asp
13) https://www.w3schools.com/js/js_object_display.asp
14) https://www.w3schools.com/js/js_object_constructors.asp
15) https://www.w3schools.com/js/js_events.asp
16) https://www.w3schools.com/js/js_htmldom_eventlistener.asp

17) https://www.w3schools.com/js/js_strings.asp
18) https://www.w3schools.com/js/js_string_methods.asp
19) https://www.w3schools.com/js/js_string_search.asp
20) https://www.w3schools.com/js/js_string_templates.asp
21) https://www.w3schools.com/js/js_arrays.asp
22) https://www.w3schools.com/js/js_array_methods.asp
23) https://www.w3schools.com/js/js_array_search.asp
24) https://www.w3schools.com/js/js_array_sort.asp
25) https://www.w3schools.com/js/js_if_else.asp
26) https://www.w3schools.com/js/js_switch.asp
27) https://www.w3schools.com/js/js_loop_for.asp
28) https://www.w3schools.com/js/js_loop_while.asp
29) https://www.w3schools.com/js/js_break.asp
30) https://www.w3schools.com/js/js_maps.asp