JavaScript

JavaScript Destructuring: -

The destructuring assignment syntax unpack object properties into variables.

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
// Destructuring
let {firstName, lastName} = person; // John Doe
```

Destructuring is not destructive. Destructuring does not change the original object.

For potentially missing properties we can set default values.

```
let {firstName, lastName, country = "US"} = person;
```

Object Property Alias: -

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
// Destructuring
let {lastName : name} = person; // name will give Doe
```

String Destructuring: -

One use for destructuring is unpacking string characters.

```
// Create a String
let name = "W3Schools";
// Destructuring
let [a1, a2, a3, a4, a5] = name;
```

Array Destructuring: -

```
// Create an Array

const fruits = ["Bananas", "Oranges", "Apples", "Mangos"];

// Destructuring

let [fruit1, fruit2] = fruits;
```

We can skip array values using two or more commas.

```
let [fruit1,,, fruit2] = fruits;

// Create an Array
const fruits = ["Bananas", "Oranges", "Apples", "Mangos"];
// Destructuring
let {[0]:fruit1 ,[1]:fruit2} = fruits
```

The Rest Property: -

```
// Create an Array

const numbers = [10, 20, 30, 40, 50, 60, 70];

// Destructuring

const [a,b, ...rest] = numbers
```

JavaScript Regular Expressions: -

A regular expression is a sequence of characters that forms a search pattern. Regular expressions can be used to perform all types of text search and text replace operations.

Ex: - /testing/i;

The above is a regular expression.

testing is a pattern (to be used in a search).

i is a modifier (modifies the search to be case-insensitive).

In JavaScript, regular expressions are often used with the two string methods: search() and replace().

The search() method searches a string for a specified value and returns the position of the match.

```
let text = "Visit W3Schools!";
let n = text.search("W3Schools"); // 6

let text = "Visit W3Schools";
let n = text.search(/w3schools/i); // 6
```

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

```
let text = "Visit Microsoft!";
let result = text.replace("Microsoft", "W3Schools");
```

```
let text = "Visit Microsoft!";
let result = text.replace(/microsoft/i, "W3Schools"); // Visit W3Schools!
```

Regular Expression Modifiers: -

Modifiers can be used to perform case-insensitive more global searches.

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all)
m	Perform multiline matching
d	Perform start and end matching

Brackets are used to find a range of characters.

Expression	Description
[abc]	Find any of the characters between the brackets
[0-9]	Find any of the digits between the brackets
(x|y)	Find any of the alternatives separated with |

Metacharacters are characters with a special meaning.

Metacharacter	Description
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers define quantities.

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n

Using the RegExp Object: -

In JavaScript, the RegExp object is a regular expression object with predefined properties and methods.

Using test(): -

```
const pattern = /e/;
pattern.test("The best things in life are free!"); // will return true
```

Using exec(): -

It searches a string for a specified pattern, and returns the found text as an object. If no match is found, it returns an empty (null) object.

```
/e/.exec("The best things in life are free!");
```

JavaScript Operator Precedence: -

Operator precedence describes the order in which operations are performed in an arithmetic expression.

Precedence	Operator	Type	Description	Example
18	( )	Expression Grouping	Groups expressions for evaluation	(100 + 50) * 3
17	.	Member Of	Accesses properties of objects	person.name
17	[]	Member Of	Accesses properties of objects	person["name"]
17	?.	Optional Chaining (ES2020)	Safely accesses nested properties	x?.y
17	()	Function Call	Calls a function	myFunction()
17	new	New with Arguments	Creates a new object with arguments	new Date("June 5, 2022")
16	new	New without Arguments	Creates a new object without arguments	new Date()
15	++	Postfix Increment	Increments variable after usage	i++
15	--	Postfix Decrement	Decrements variable after usage	i--
14	++	Prefix Increment	Increments variable before usage	++i
14	--	Prefix Decrement	Decrements variable before usage	--i
14	!	Logical NOT	Negates a boolean value	!(x == y)

14	~	Bitwise NOT	Inverts bits in a number	~x
14	+	Unary Plus	Converts operand to a number	+x
14	-	Unary Minus	Negates a number	-x
14	typeof	Data Type	Returns the data type of an operand	typeof x
14	void	Evaluate Void	Returns undefined	void(0)
14	delete	Property Delete	Deletes a property from an object	delete myCar.color
13	**	Exponentiation (ES2016)	Raises a number to a power	10 ** 2
12	*	Multiplication	Multiplies two numbers	10 * 5
12	/	Division	Divides two numbers	10 / 5
12	%	Division Remainder	Returns the remainder of a division	10 % 5
11	+	Addition	Adds two numbers	10 + 5
11	-	Subtraction	Subtracts two numbers	10 - 5
11	+	Concatenation	Concatenates two strings	"John" + "Doe"
10	<<	Shift Left	Shifts bits left by specified number	x << 2
10	>>	Shift Right (signed)	Shifts bits right by specified number	x >> 2
10	>>>	Shift Right (unsigned)	Shifts bits right by specified number	x >>> 2
9	in	Property in Object	Checks if a property exists in an object	"PI" in Math
9	instanceof	Instance of Object	Checks if an object is an instance of a class	x instanceof Array
9	<	Less than	Compares if one value is less than another	x < y

9	<=	Less than or equal	Compares if one value is less than or equal to another	x <= y
9	>	Greater than	Compares if one value is greater than another	x > y
9	>=	Greater than or equal	Compares if one value is greater than or equal to another	x >= y
8	==	Equal	Checks if two values are equal	x == y
8	===	Strict Equal	Checks if two values are strictly equal	x === y
8	!=	Unequal	Checks if two values are not equal	x != y
8	!==	Strict Unequal	Checks if two values are strictly unequal	x !== y
7	&	Bitwise AND	Performs a bitwise AND operation	x & y
6	^	Bitwise XOR	Performs a bitwise XOR operation	x ^ y
5			Bitwise OR	Performs a bitwise OR operation
4	&&	Logical AND	Logical AND operation	x && y
3				Logical OR
3	??	Nullish Coalescing (ES2020)	Returns right operand if left is null or undefined	x ?? y
2	? :	Conditional (Ternary)	Returns one value if true, another if false	x ? "yes" : "no"
2	=	Simple Assignment	Assigns a value to a variable	x = y
2	:	Colon Assignment	Assigns a value using a shorthand	x: 5
2	+=	Addition Assignment	Adds and assigns the value	x += y

2	-=	Subtraction Assignment	Subtracts and assigns the value	x -= y
2	*=	Multiplication Assignment	Multiplies and assigns the value	x *= y
2	**=	Exponentiation Assignment	Exponentiates and assigns the value	x **= y
2	/=	Division Assignment	Divides and assigns the value	x /= y
2	%=	Remainder Assignment	Finds remainder and assigns the value	x %= y
2	<<=	Left Shift Assignment	Left shifts bits and assigns the value	x <<= y
2	>>=	Right Shift Assignment	Right shifts bits and assigns the value	x >>= y
2	>>>=	Unsigned Right Shift	Unsigned right shifts bits and assigns	x >>>= y
2	&=	Bitwise AND Assignment	Bitwise AND and assigns the value	x &= y
2		=	Bitwise OR Assignment	Bitwise OR and assigns the value
2	^=	Bitwise XOR Assignment	Bitwise XOR and assigns the value	x ^= y
2	&&=	Logical AND Assignment	Logical AND and assigns the value	x &&= y
2			=	Logical OR Assignment
2	=>	Arrow	Defines an arrow function	x => y
2	yield	Pause / Resume	Pauses a generator function	yield x
2	yield*	Delegate	Delegates to another generator function	yield* x

2	...	Spread	Expands an iterable into individual items	...x
1	,	Comma	Separates multiple expressions or variables	x , y

JavaScript Hoisting: -

Hoisting is JavaScript's default behaviour of moving declarations to the top.

In JavaScript, a variable can be declared after it has been used.

JavaScript only hoists declarations, not initializations.

Ex: -

```
 x = 5;
alert(x);
var x; // output is undefined.
```

To avoid bugs, always declare all variables at the beginning of every scope.

JS Strict Mode: -

"use strict";

The purpose of "use strict" is to indicate that the code should be executed in "strict mode". With strict mode, you can not, for example, use undeclared variables.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode)

```
"use strict";
x = 3.14;       // This will cause an error because x is not declared
```

```
"use strict";
myFunction();
function myFunction() {
  y = 3.14;   // This will also cause an error because y is not declared
}
```

Declared inside a function, it has local scope (only the code inside the function is in strict mode)

```
x = 3.14;       // This will not cause an error.

myFunction();

function myFunction() {

  "use strict";

  y = 3.14;   // This will cause an error
```

}

Strict mode makes it easier to write "secure" JavaScript. It changes previously accepted "bad syntax" into real errors.

Using an object, without declaring it, is not allowed.

Deleting a variable (or object) is not allowed.

Deleting a function is not allowed.

Duplicating a parameter name is not allowed.

Octal numeric literals are not allowed // let x = 010;

Writing to a get-only property is not allowed.

The this keyword in functions behaves differently in strict mode.

The this keyword refers to the object that called the function.

If the object is not specified, functions in strict mode will return undefined and functions in normal mode will return the global object (window).

Keywords reserved for future JavaScript versions cannot be used as variable names in strict mode.

The "use strict" directive is only recognized at the beginning of a script or a function.

JavaScript Modules: -

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain a codebase.

Modules are imported from external files with the import statement. It also rely on type="module" in the <script> tag.

```
<script type="module">
import message from "./message.js";
</script>
```

Export: -

Modules with functions or variables can be stored in any external file. There are two types of exports: Named Exports and Default Exports.

Named Exports: -

You can create named exports two ways. In-line individually, or all at once at the bottom.

```
export const name = "Jesse";
export const age = 40;
```

or

```
const name = "Jesse";
```

```
const age = 40;

export {name, age};
```

Default Exports: -

You can only have one default export in a file.

```
const message = () => {

const name = "Jesse";

const age = 40;

return name + ' is ' + age + 'years old.';

};

export default message;
```

Import: -

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports are constructed using curly braces. Default exports are not.

```
import { name, age } from "./person.js";

import message from "./message.js";
```

Modules only work with the HTTP(s) protocol.

JavaScript Debugging: -

Searching for (and fixing) errors in programming code is called code debugging.

The console.log() Method: -

you can use console.log() to display JavaScript values in the debugger window.

Setting Breakpoints: -

In the debugger window, you can set breakpoints in the JavaScript code.

At each breakpoint, JavaScript will stop executing, and let you examine JavaScript values.

After examining values, you can resume the execution of code (typically with a play button).