Rebase
1st Batch 2022/23

**Teacher: Wadleo**
**Email: info@wadingaleonard.com**

# Programming in JS

—

### Course Outline

---

## Chapter 1: Introduction:

1. Getting Started With JavaScript
2. JS Variables and Constants
3. JS console.log()
4. JavaScript Data Types
5. JavaScript Operators
6. JavaScript Comments
7. JavaScript Type Conversions
8. JavasScript Reserved Words
9. Exercises & Assignments

## Chapter 2: Control Flow

- Comparison and Logical Operators
- JavaScript if...else Statement
- JavaScript for Loop
- JavaScript while Loop
- JavaScript break Statement
- JavaScript continue Statement
- JavaScript switch Statement
- Exercises & Assignments

## Chapter 3: Functions

- JavaScript Functions and Expressions

- JavaScript Variable Scope
- JavaScript Hoisting
- JavaScript Recursion
- Exercises & Assignments

## Chapter 4: Objects

- JavaScript Objects
- JavaScript Methods
- JavaScript Constructor Function
- JavaScript Getters and Setters
- JavaScript Prototype
- Exercises & Assignments

## Chapter 5: Types

- JavaScript Arrays
- JavaScript Multidimensional Arrays
- JavaScript Strings
- JavaScript for…in Loop
- JavaScript Numbers
- JavaScript Symbols
- Exercises & Assignments

## Chapter 6: Exceptions & Modules

- JavaScript try…catch…finally
- JavaScript throw Statement
- JavaScript Modules
- Exercises & Assignments

## Chapter 7: ES6

- JavaScript ES6
- JavaScript Arrow Function
- JavaScript Default Parameters
- JavaScript Template Literals
- JavaScript Spread Operator
- JavaScript Map
- JavaScript Set

- JS Destructuring Assignment
- JavaScript Classes
- JavaScript Inheritance
- JavaScript for...of Loop
- JavaScript Proxies
- Exercises & Assignments

# Chapter 8: Asynchronous

- JavaScript setTimeout()
- JavaScript CallBack
- JavaScript Promises
- JavaScript async/await
- JavaScript setInterval()
- Exercises & Assignments

# Chapter 9: Miscellaneous

- JavaScript JSON
- JavaScript Date and Time
- JavaScript Closures
- JavaScript this
- JavaScript 'use strict'
- JS Iterators and Iterables
- JavaScript Generators
- JavaScript Regex
- JavaScript Browser Debugging
- Uses of JavaScript
- Exercises & Assignments

### Exercises

---

# Chapter 1: Introduction

1. JavaScript Program To Print Hello World
2. JavaScript Program to Add Two Numbers
3. 
4. JavaScript Program to Find the Square Root

5. JavaScript Program to Calculate the Area of a Triangle
6. JavaScript Program to Swap Two Variables
7. JavaScript Program to Convert Kilometers to Miles
8. Javascript Program to Convert Celsius to Fahrenheit
9. JavaScript Program To Work With Constants
10. JavaScript Program to Write to Console

# Chapter 2: Control Flow

10. Javascript Program to Solve Quadratic Equations
11. Javascript Program to Check if a number is Positive, Negative, or Zero
12. Javascript Program to Check if a Number is Odd or Even
13. JavaScript Program to Find the Largest Among Three Numbers
14. JavaScript Program to Check Prime Number
15. JavaScript Program to Print All Prime Numbers in an Interval
16. JavaScript Program to Find the Factorial of a Number
17. JavaScript Program to Display the Multiplication Table
18. JavaScript Program to Print the Fibonacci Sequence
19. JavaScript Program to Check Armstrong Number
20. JavaScript Program to Find Armstrong Number in an Interval
21. JavaScript Program to Make a Simple Calculator
22. JavaScript Program to Find the Sum of Natural Numbers
23. JavaScript Program to Check if the Numbers Have the Same Last Digit
24. JavaScript Program to Find HCF or GCD
25. JavaScript Program to Find LCM
26. JavaScript Program to Find the Factors of a Number
27. JavaScript Program to Display Fibonacci Sequence Using Recursion

# Chapter 3: Functions

28. Javascript Program to Generate a Random Number
29. JavaScript Program to Find Sum of Natural Numbers Using Recursion
30. JavaScript Program to Guess a Random Number
31. JavaScript Program to Find Factorial of Numbers Using Recursion
32. JavaScript Program to Convert Decimal to Binary
33. JavaScript Program to Find ASCII Value of Character
34. JavaScript Program to Set a Default Parameter Value For a Function
35. JavaScript Program to Check If a Variable is of Function Type
36. JavaScript Program to Pass Parameter to a setTimeout() Function

# Chapter 4: Arrays & Objects



# Chapter 5: Strings

# Chapter 6: Exceptions & Modules



**Chapter 7: ES6**



# Chapter 8: Asynchronous



# Chapter 9: Miscellaneous

98. JavaScript Program to Include a JS file in Another JS file
99. Javascript Program to Generate a Random Number Between Two Numbers
100. JavaScript Program To Get The Current URL
101. JavaScript Program to Validate An Email Address
102. JavaScript Program to Implement a Stack
103. JavaScript Program to Implement a Queue
104. JavaScript Program to Check if a Number is a Float or Integer
105. JavaScript Program to Get the Dimensions of an Image
106. JavaScript Program to Convert Date to Number

## Resources

1. MDN Web Docs - It provides one of the best resources to learn JavaScript from basics to advance. Visit MDN-JavaScript Basics.
2. FreeCodeCamp - One of the best self-guided web development sites on the internet.
3. W3Schools - Another excellent study tool with plenty of exercises to do in the browser.
4. Can I Use - Check your browser version with js versions for compactibility.

# Lecture Notes

## Chapter 1: Introduction

### *Lecture 1: Introduction*

JavaScript is a popular programming language that has a wide range of applications.

JavaScript was previously used mainly for making webpages interactive such as form validation, animation, etc. Nowadays, JavaScript is also used in many other areas such as server-side development, mobile app development, etc.

Because of its wide range of applications, you can run JavaScript in several ways:

1. Using the console tab of web browsers
2. Using Node.js
3. By creating web pages

### *Lecture 2: JS Variables and Constants*

In programming, a variable is a container (storage area) to hold data. For example,
***let num = 5;***

In JavaScript, we use either var or let keyword to declare variables. For example,
***var x;***
***let y;***

**JavaScript var Vs let**

Both var and let are used to declare variables. However, there are some differences between them.

| var | let |
| --- | --- |
| `var` is used in the older versions of JavaScript | `let` is the new way of declaring variables starting **ES6 (ES2015)**. |
| `var` is function scoped (will be discussed in later tutorials). | `let` is block scoped (will be discussed in later tutorials). |
| For example, `var x;` | For example, `let y;` |

**Note**: It is recommended we use let instead of var. However, there are a few browsers that do not support let. Visit JavaScript let browser support to learn more.

### JavaScript Initialize Variables

We use the assignment operator = to assign a value to a variable.

```
let x;
x = 5;
```

You can also initialize variables during its declaration.

```
let x = 5;
let y = 6;
```

In JavaScript, it's possible to declare variables in a single statement.

```
let x = 5, y = 6, z = 7;
```

If you use a variable without initializing it, it will have an undefined value.

```
let x;  // x is the name of the variable
console.log(x); // undefined
```

Here x is the variable name and since it does not contain any value, it will be undefined.

### Change the Value of Variables

```
// 5 is assigned to variable x
let x = 5;
console.log(x); // 5

// vaue of variable x is changed
x = 3;
console.log(x); // 3
```

### Rules for Naming JavaScript Variables

The rules for naming variables are:

1. Variable names must start with either a letter, an underscore _, or the dollar sign $. For example,

```
//valid
let a = 'hello';
let _a = 'hello';
let $a = 'hello';
```

2. Variable names cannot start with numbers. For example,

```
//invalid
Let 1a = 'hello'; // this gives an error
```

3. JavaScript is case-sensitive. So y and Y are different variables. For example,

```
let y = "hi";
let Y = 5;

console.log(y); // hi
console.log(Y); // 5
```

4. Keywords cannot be used as variable names. For example,

```
//invalid
let new = 5; // Error! new is a keyword.
```

**Notes**:

Though you can name variables in any way you want, it's a good practice to give a descriptive variable name. If you are using a variable to store the number of apples, it is better to use apples or numberOfApples rather than x or n.

In JavaScript, the variable names are generally written in camelCase if it has multiple words. For example, firstName, annualSalary, etc.

**JavaScript Constants**

The const keyword was also introduced in the ES6(ES2015) version to create constants. For example,

```
const x = 5;
```

Once a constant is initialized, we cannot change its value.

```
const x = 5;
x = 10;  // Error! constant cannot be changed.
console.log(x)
```

Simply, a constant is a type of variable whose value cannot be changed.

Also, you cannot declare a constant without initializing it. For example,

```
const x;  // Error! Missing initializer in const declaration.
x = 5;
```

***console.log(x)***

**Note**: If you are sure that the value of a variable won't change throughout the program, it's recommended to use const. However, there are a few browsers that do not support const. Visit JavaScript const browser support to learn more.

## *Lecture 3: JS console.log()*

All modern browsers have a web console for debugging. The console.log() method is used to write messages to these consoles. For example,

```
let sum = 44;
console.log(sum);   // 44
```

When you run the above code, 44 is printed on the console.

**console.log() Syntax**

Its syntax is:

```
console.log(message);
```

Here, the message refers to either a variable or a value.

**Example 1: Print a Sentence**

```
// program to print a sentence
// passing string
console.log("I love JS");
```

**Example 2: Print Values Stored in Variables**

```
// program to print variables values
// storing values
const greet = 'Hello';
const name = 'Jack';
console.log(greet + '  ' + name);
console.log(greet, name, ' ', 66, 'leo', 'wadleo');
```

As you can see from these examples, console.log() makes it easier to see the value inside a variable. That's why it's commonly used for testing/debugging code.

## *Lecture 4: JS Data Types*

There are different types of data that we can use in a JavaScript program. For example,

```
const x = 5;
const y = "Hello";
```

Here, 5 is an integer data.

"Hello" is a string data.

**JavaScript Data Types**

There are eight basic data types in JavaScript. They are:

| Data Types | Description | Example |
|---|---|---|
| `String` | represents textual data | `'hello'`, `"hello world!"` etc |
| `Number` | an integer or a floating-point number | `3`, `3.234`, `3e-2` etc. |
| `BigInt` | an integer with arbitrary precision | `900719925124740999n`, `1n` etc. |
| `Boolean` | Any of two values: true or false | `true` and `false` |
| `undefined` | a data type whose variable is not initialized | `let a;` |
| `null` | denotes a `null` value | `let a = null;` |
| `Symbol` | data type whose instances are unique and immutable | `let value = Symbol('hello');` |
| `Object` | key-value pairs of collection of data | `let student = { };` |

Here, all data types except Object are primitive data types, whereas Object is non-primitive.

**Note**: The Object data type (non-primitive type) can store collections of data, whereas primitive data type can only store a single data.

**JavaScript String**

String is used to store text. In JavaScript, strings are surrounded by quotes:

Single quotes: 'Hello'

Double quotes: "Hello"

Backticks: `Hello`

For example,

```
//strings example
const name = 'ram';
const name1 = "hari";
const result = `The names are ${name} and ${name1}`;
```

Single quotes and double quotes are practically the same and you can use either of them.

Backticks are generally used when you need to include variables or expressions into a string. This is done by wrapping variables or expressions with ${variable or expression} as shown above.

**JavaScript Number**

Number represents integer and floating numbers (decimals and exponentials). For example,

```
const number1 = 3;
const number2 = 3.433;
const number3 = 3e5 // 3 * 10^5
```

A number type can also be +Infinity, -Infinity, and NaN (not a number). For example,

```
const number1 = 3/0;
console.log(number1); // Infinity

const number2 = -3/0;
console.log(number2); // -Infinity

// strings can't be divided by numbers
const number3 = "abc"/3;
console.log(number3);  // NaN
```

**JavaScript BigInt**

In JavaScript, Number type can only represent numbers less than (2^53 - 1) and more than -(2^53 - 1). However, if you need to use a larger number than that, you can use the BigInt data type.

A BigInt number is created by appending n to the end of an integer. For example,

```
// BigInt value
const value1 = 900719925124740998n;

// Adding two big integers
const result1 = value1 + 1n;
console.log(result1); // "900719925124740999n"

const value2 = 900719925124740998n;
```

***// Error! BitInt and number cannot be added***
***const result2 = value2 + 1;***
***console.log(result2);***

**Output**

***9007199251247409999n***
***Uncaught TypeError: Cannot mix BigInt and other types***

**Note**: BigInt was introduced in the newer version of JavaScript and is not supported by many browsers including Safari. Visit JavaScript BigInt support to learn more.

**JavaScript Boolean**

This data type represents logical entities. Boolean represents one of two values: true or false. It is easier to think of it as a yes/no switch. For example,

***const dataChecked = true;***
***const valueCounted = false;***

**JavaScript undefined**

The undefined data type represents a value that is not assigned. If a variable is declared but the value is not assigned, then the value of that variable will be undefined. For example,

***let name;***
***console.log(name); // undefined***

It is also possible to explicitly assign a variable value undefined. For example,

***let name = undefined;***
***console.log(name); // undefined***

**Note**: It is recommended not to explicitly assign undefined to a variable. Usually, null is used to assign an 'unknown' or 'empty' value to a variable.

**JavaScript null**

In JavaScript, null is a special value that represents an empty or unknown value. For example,

***const number = null;***

The code above suggests that the number variable is empty.

**Note**: null is not the same as NULL or Null.

**JavaScript Symbol**

This data type was introduced in a newer version of JavaScript (from ES2015).

A value having the data type Symbol can be referred to as a symbol value. Symbol is an immutable primitive value that is unique. For example,

***// two symbols with the same description***

```
const value1 = Symbol('hello');
const value2 = Symbol('hello');
```

Though value1 and value2 both contain 'hello', they are different as they are of the Symbol type.

**JavaScript Object**

An object is a complex data type that allows us to store collections of data. For example,

```
const student = {
   firstName: 'ram',
   lastName: null,
   class: 10
};
```

**JavaScript Type**

JavaScript is a dynamically typed (loosely typed) language. JavaScript automatically determines the variables' data type for you.

It also means that a variable can be of one data type and later it can be changed to another data type. For example,

```
// data is of undefined type
let data;

// data is of integer type
data = 5;

// data is of string type
data = "JavaScript Programming";
```

**JavaScript typeof**

To find the type of a variable, you can use the typeof operator. For example,

```
const name = 'ram';
typeof(name); // returns "string"

const number = 4;
typeof(number); //returns "number"

const valueChecked = true;
typeof(valueChecked); //returns "boolean"

const a = null;
typeof(a); // returns "object"
```

**Note**: Notice that typeof returned "object" for the null type. This is a known issue in JavaScript since its first release.

## *Lecture 5: JS Operators*

### What is an Operator?

In JavaScript, an operator is a special symbol used to perform operations on operands (values and variables). For example,
***2 + 3; // 5***

Here + is an operator that performs addition, and 2 and 3 are operands.

### JavaScript Operator Types

Here is a list of different operators;

- **Assignment Operators**
- **Arithmetic Operators**
- **Comparison Operators**
- **Logical Operators**
- **Bitwise Operators**
- **String Operators**
- **Other Operators**

### JavaScript Assignment Operators

Assignment operators are used to assign values to variables. For example,
***const x = 5;***

Here, the = operator is used to assign value 5 to variable x.

Here's a list of commonly used assignment operators:

| Operator | Name | Example |
| --- | --- | --- |
| `=` | Assignment operator | `a = 7; // 7` |
| `+=` | Addition assignment | `a += 5; // a = a + 5` |
| `-=` | Subtraction Assignment | `a -= 2; // a = a - 2` |
| `*=` | Multiplication Assignment | `a *= 3; // a = a * 3` |
| `/=` | Division Assignment | `a /= 2; // a = a / 2` |
| `%=` | Remainder Assignment | `a %= 2; // a = a % 2` |
| `**=` | Exponentiation Assignment | `a **= 2; // a = a**2` |

**Note**: The commonly used assignment operator is =. You will understand other assignment operators such as +=, -=, *= etc. once we learn arithmetic operators.

**JavaScript Arithmetic Operators**

Arithmetic operators are used to perform arithmetic calculations. For example,
***const number = 3 + 5; // 8***

Here, the + operator is used to add two operands.

| Operator | Name | Example |
| --- | --- | --- |
| `+` | Addition | `x + y` |
| `-` | Subtraction | `x - y` |
| `*` | Multiplication | `x * y` |
| `/` | Division | `x / y` |
| `%` | Remainder | `x % y` |
| `++` | Increment (increments by 1) | `++x` or `x++` |
| `--` | Decrement (decrements by 1) | `--x` or `x--` |
| `**` | Exponentiation (Power) | `x ** y` |

**Example 1: Arithmetic operators in JavaScript**

```
let x = 5;
let y = 3;

// addition
console.log('x + y = ', x + y);  // 8

// subtraction
console.log('x - y = ', x - y);  // 2

// multiplication
console.log('x * y = ', x * y);  // 15

// division
console.log('x / y = ', x / y);  // 1.6666666666666667

// remainder
console.log('x % y = ', x % y);   // 2

// increment
console.log('++x = ', ++x); // x is now 6
console.log('x++ = ', x++); // prints 6 and then increased to 7
console.log('x = ', x);     // 7

// decrement
console.log('--x = ', --x); // x is now 6
console.log('x-- = ', x--); // prints 6 and then decreased to 5
```

***console.log('x = ', x);  // 5***

***//exponentiation***
***console.log('x ** y =', x ** y);***

**Note**: The ** operator was introduced in ECMAScript 2016 and some browsers may not support them. To learn more, visit JavaScript exponentiation browser support.

**JavaScript Comparison Operators**

Comparison operators compare two values and return a boolean value, either true or false. For example,

***const a = 3, b = 2;***
***console.log(a > b); // true***

Here, the comparison operator > is used to compare whether a is greater than b.

| Operator | Description | Example |
|---|---|---|
| `==` | **Equal to**: returns `true` if the operands are equal | `x == y` |
| `!=` | **Not equal to**: returns `true` if the operands are not equal | `x != y` |
| `===` | **Strict equal to**: `true` if the operands are equal and of the same type | `x === y` |
| `!==` | **Strict not equal to**: `true` if the operands are equal but of different type or not equal at all | `x !== y` |
| `>` | **Greater than**: `true` if left operand is greater than the right operand | `x > y` |
| `>=` | **Greater than or equal to**: `true` if left operand is greater than or equal to the right operand | `x >= y` |
| `<` | **Less than**: `true` if the left operand is less than the right operand | `x < y` |
| `<=` | **Less than or equal to**: `true` if the left operand is less than or equal to the right operand | `x <= y` |

Example 2: Comparison operators in JavaScript

***// equal operator***
***console.log(2 == 2); // true***
***console.log(2 == '2'); // true***

```
// not equal operator
console.log(3 != 2); // true
console.log('hello' != 'Hello'); // true

// strict equal operator
console.log(2 === 2); // true
console.log(2 === '2'); // false

// strict not equal operator
console.log(2 !== '2'); // true
console.log(2 !== 2); // false
```

Comparison operators are used in decision-making and loops.

### JavaScript Logical Operators

Logical operators perform logical operations and return a boolean value, either true or false. For example,

```
const x = 5, y = 3;
(x < 6) && (y < 5); // true
```

Here, && is the logical operator AND. Since both x < 6 and y < 5 are true, the result is true.

| Operator | Description | Example |
|---|---|---|
| `&&` | **Logical AND**: `true` if both the operands are `true`, else returns `false` | `x && y` |
| `\|\|` | **Logical OR**: `true` if either of the operands is `true`; returns false if both are `false` | `x \|\| y` |
| `!` | **Logical NOT**: `true` if the operand is `false` and vice-versa. | `!x` |

### Example 3: Logical Operators in JavaScript

```
// logical AND
console.log(true && true); // true
console.log(true && false); // false

// logical OR
console.log(true || false); // true

// logical NOT
console.log(!true); // false
```

Logical operators are used in decision making and loops.

### JavaScript Bitwise Operators

Bitwise operators perform operations on binary representations of numbers.

| Operator | Description |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise XOR |
| ~ | Bitwise NOT |
| << | Left shift |
| >> | Sign-propagating right shift |
| >>> | Zero-fill right shift |

Bitwise operators are rarely used in everyday programming. If you are interested, visit JavaScript Bitwise Operators to learn more.

### JavaScript String Operators

In JavaScript, you can also use the + operator to concatenate (join) two or more strings.

**Example 4: String operators in JavaScript**

```
// concatenation operator
console.log('hello' + 'world');

let a = 'JavaScript';

a += ' tutorial';  // a = a + ' tutorial';
console.log(a);
```

**Note**: When + is used with strings, it performs concatenation. However, when + is used with numbers, it performs addition.

### Other JavaScript Operators

Here's a list of other operators available in JavaScript.

| Operator | Description | Example |
| --- | --- | --- |
| `,` | evaluates multiple operands and returns the value of the last operand. | `let a = (1, 3 , 4); // 4` |
| `?:` | returns value based on the condition | `(5 > 3) ? 'success' : 'error'; // "success"` |
| `delete` | deletes an object's property, or an element of an array | `delete x` |
| `typeof` | returns a string indicating the data type | `typeof 3; // "number"` |
| `void` | discards the expression's return value | `void(x)` |
| `in` | returns `true` if the specified property is in the object | `prop in object` |
| `instanceof` | returns `true` if the specified object is of of the specified object type | `object instanceof object_type` |

## *Lecture 6: JS Comments*

JavaScript comments are hints that a programmer can add to make their code easier to read and understand. They are completely ignored by JavaScript engines.

There are two ways to add comments to code:
***// - Single Line Comments***
***/* */ -Multi-line Comments***

**Single Line Comments**

In JavaScript, any line that starts with // is a single line comment. For example,

```
name = "Jack";
// printing name on the console
console.log("Hello " + name);
```

Here, // printing name on the console is a comment.

You can also use single line comment like this:

```
name = "Jack";
```

***console.log("Hello " + name);  // printing name on the console***

**Multi-line Comments**

In Javascript, any text between /* and */ is a multi-line comment. For example,

***/* The following program contains the source code for a game called Baghchal.***
***Baghchal is a popular board game in Nepal where two players choose either sheep or tiger. It is played on a 5x5 grid.***

***For the player controlling the tiger to win, they must capture all the sheep. There are altogether 4 tigers on the board.***

***For the sheep to win, all tigers must be surrounded and cornered so that they cannot move.***
***The player controlling the sheep has 20 sheep at his disposal.***
*******/*

Since the rest of the source code will be used to implement the rules of the game, the comment above is a good example where you might use a multi-line comment.

**Using Comments for Debugging**

Comments can also be used to disable code to prevent it from being executed. For example,

***console.log("some code");***
***console.log("Error code);***
***console.log("other code");***

If you get an error while running the program, instead of removing the error-prone code, you can use comments to disable it from being executed; this can be a valuable debugging tool.

***console.log("some code");***
***// console.log("Error code);***
***console.log("other code");***

**Pro Tip**: Remember the shortcut for using comments; it can be really helpful. For most code editors, it's Ctrl + / for Windows and Cmd + / for Mac.

**Make Code Easier to Understand**

As a JavaScript developer, you will not only write code but may also have to modify code written by other developers.

If you write comments on your code, it will be easier for you to understand the code in the future. Also, it will be easier for your fellow developers to understand the code.

As a general rule of thumb, use comments to explain why you did something rather than how you did something, and you are good.

**Note**: Comments shouldn't be the substitute for a way to explain poorly written code in English. You should always write well-structured and self-explanatory code. And, then use comments.

## *Lecture 7: JS Type Conversions*

In programming, type conversion is the process of converting data of one type to another. For example: converting String data to Number.

There are two types of type conversion in JavaScript.

- **Implicit Conversion - automatic type conversion**
- **Explicit Conversion - manual type conversion**

**JavaScript Implicit Conversion**

In certain situations, JavaScript automatically converts one data type to another (to the right type). This is known as implicit conversion.

**Example 1: Implicit Conversion to String**

```
// numeric string used with + gives string type
let result;

result = '3' + 2;
console.log(result) // "32"

result = '3' + true;
console.log(result); // "3true"

result = '3' + undefined;
console.log(result); // "3undefined"

result = '3' + null;
console.log(result); // "3null"
```

**Note**: When a number is added to a string, JavaScript converts the number to a string before concatenation.

**Example 2: Implicit Conversion to Number**

```
// numeric string used with - , / , * results number type

let result;

result = '4' - '2';
console.log(result); // 2

result = '4' - 2;
console.log(result); // 2
```

```
result = '4' * 2;
console.log(result); // 8

result = '4' / 2;
console.log(result); // 2
```

**Example 3: Non-numeric String Results to NaN**

```
// non-numeric string used with - , / , * results to NaN

let result;

result = 'hello' - 'world';
console.log(result); // NaN

result = '4' - 'hello';
console.log(result); // NaN
```

**Example 4: Implicit Boolean Conversion to Number**

```
// if boolean is used, true is 1, false is 0

let result;

result = '4' - true;
console.log(result); // 3

result = 4 + true;
console.log(result); // 5

result = 4 + false;
console.log(result); // 4
```

**Note**: JavaScript considers 0 as false and all non-zero numbers as true. And, if true is converted to a number, the result is always 1.

**Example 5: null Conversion to Number**

```
// null is 0 when used with number
let result;

result = 4 + null;
console.log(result);  // 4

result = 4 - null;
console.log(result);  // 4
```

**Example 6: undefined used with number, boolean or null**

```
// Arithmetic operation of undefined with number, boolean or null gives NaN

let result;
```

```
result = 4 + undefined;
console.log(result);  // NaN

result = 4 - undefined;
console.log(result);  // NaN

result = true + undefined;
console.log(result);  // NaN

result = null + undefined;
console.log(result);  // NaN
```

**JavaScript Explicit Conversion**

You can also convert one data type to another as per your needs. The type conversion that you do manually is known as explicit type conversion.

In JavaScript, explicit type conversions are done using built-in methods.

Here are some common methods of explicit conversions.

**1. Convert to Number Explicitly**

To convert numeric strings and boolean values to numbers, you can use Number(). For example,

```
let result;

// string to number
result = Number('324');
console.log(result); // 324

result = Number('324e-1')
console.log(result); // 32.4

// boolean to number
result = Number(true);
console.log(result); // 1

result = Number(false);
console.log(result); // 0
```

**In JavaScript, empty strings and null values return 0. For example,**

```
let result;
result = Number(null);
console.log(result);  // 0

let result = Number('')
console.log(result);  // 0
```

If a string is an invalid number, the result will be NaN. For example,

```
let result;
result = Number('hello');
console.log(result); // NaN

result = Number(undefined);
console.log(result); // NaN

result = Number(NaN);
console.log(result); // NaN
```

**Note**: You can also generate numbers from strings using parseInt(), parseFloat(), unary operator + and Math.floor(). For example,

```
let result;
result = parseInt('20.01');
console.log(result); // 20

result = parseFloat('20.01');
console.log(result); // 20.01

result = +'20.01';
console.log(result); // 20.01

result = Math.floor('20.01');
console.log(result); // 20
```

**2. Convert to String Explicitly**

To convert other data types to strings, you can use either String() or toString(). For example,

```
//number to string
let result;
result = String(324);
console.log(result);  // "324"

result = String(2 + 4);
console.log(result); // "6"

//other data types to string
result = String(null);
console.log(result); // "null"

result = String(undefined);
console.log(result); // "undefined"

result = String(NaN);
console.log(result); // "NaN"

result = String(true);
console.log(result); // "true"
```

```
result = String(false);
console.log(result); // "false"

// using toString()
result = (324).toString();
console.log(result); // "324"

result = true.toString();
console.log(result); // "true"
```

**Note**: String() takes null and undefined and converts them to string. However, toString() gives error when null are passed.

**3. Convert to Boolean Explicitly**

To convert other data types to a boolean, you can use Boolean().

In JavaScript, undefined, null, 0, NaN, '' converts to false. For example,

```
let result;
result = Boolean('');
console.log(result); // false

result = Boolean(0);
console.log(result); // false

result = Boolean(undefined);
console.log(result); // false

result = Boolean(null);
console.log(result); // false

result = Boolean(NaN);
console.log(result); // false
```

**All other values are true. For example,**

```
result = Boolean(324);
console.log(result); // true

result = Boolean('hello');
console.log(result); // true

result = Boolean(' ');
console.log(result); // true
```

**JavaScript Type Conversion Table**

The table shows the conversion of different values to String, Number, and Boolean in JavaScript.

| Value | String Conversion | Number Conversion | Boolean Conversion |
| --- | --- | --- | --- |
| 1 | "1" | 1 | `true` |
| 0 | "0" | 0 | `false` |
| "1" | "1" | 1 | `true` |
| "0" | `"0"` | 0 | `true` |
| "ten" | `"ten"` | `NaN` | `true` |
| `true` | "true" | 1 | `true` |
| `false` | "false" | 0 | `false` |
| `null` | "null" | 0 | `false` |
| `undefined` | "undefined" | `NaN` | `false` |
| '' | "" | 0 | `false` |
| ' ' | " " | 0 | `true` |

## *Lecture 8: JS  Reserved Words*

In JavaScript you cannot use these reserved words as variables, labels, or function names:

| | | | |
|---|---|---|---|
| abstract | arguments | await* | boolean |
| break | byte | case | catch |
| char | class* | const | continue |
| debugger | default | delete | do |
| double | else | enum* | eval |
| export* | extends* | false | final |
| finally | float | for | function |
| goto | if | implements | import* |
| in | instanceof | int | interface |
| let* | long | native | new |
| null | package | private | protected |
| public | return | short | static |
| super* | switch | synchronized | this |
| throw | throws | transient | true |
| try | typeof | var | void |
| volatile | while | with | yield |

**Note**: Words marked with* are new in ECMAScript 5 and 6.

### Removed Reserved Words

The following reserved words have been removed from the ECMAScript 5/6 standard:

| | | | |
|---|---|---|---|
| abstract | boolean | byte | char |
| double | final | float | goto |
| int | long | native | short |
| synchronized | throws | transient | volatile |

# Chapter 2: Control Flow

## *Lecture 1: JS Comparison and Logical Operators*

### JavaScript Comparison Operators

Comparison operators compare two values and give back a boolean value: either true or false. Comparison operators are used in decision-making and loops.

| Operator | Description | Example |
|---|---|---|
| `==` | **Equal to:** `true` if the operands are equal | `5==5; //true` |
| `!=` | **Not equal to:** `true` if the operands are not equal | `5!=5; //false` |
| `===` | **Strict equal to:** `true` if the operands are equal and of the same type | `5==='5'; //false` |
| `!==` | **Strict not equal to:** `true` if the operands are equal but of different type or not equal at all | `5!=='5'; //true` |
| `>` | **Greater than:** `true` if the left operand is greater than the right operand | `3>2; //true` |
| `>=` | **Greater than or equal to:** `true` if the left operand is greater than or equal to the right operand | `3>=3; //true` |
| `<` | **Less than:** `true` if the left operand is less than the right operand | `3<2; //false` |
| `<=` | **Less than or equal to:** `true` if the left operand is less than or equal to the right operand | `2<=2; //true` |

**Example 1: Equal to Operator**

```
const a = 5, b = 2, c = 'hello';
// equal to operator
console.log(a == 5);     // true
console.log(b == '2');   // true
console.log(c == 'Hello');  // false
```

== evaluates to true if the operands are equal.

**Note**: In JavaScript, == is a comparison operator, whereas = is an assignment operator. If you mistakenly use = instead of ==, you might get unwanted results.

**Example 2: Not Equal to Operator**

```
const a = 3, b = 'hello';
// not equal operator
console.log(a != 2); // true
console.log(b != 'Hello'); // true
```

!= evaluates to true if the operands are not equal.

**Example 3: Strict Equal to Operator**

```
const a = 2;
// strict equal operator
console.log(a === 2); // true
console.log(a === '2'); // false
```

=== evaluates to true if the operands are equal and of the same type. Here 2 and '2' are the same numbers but the data type is different. And === also checks for the data type while comparing.

**Note**: The difference between == and === is that:

== evaluates to true if the operands are equal, however, === evaluates to true only if the operands are equal and of the same type

**Example 4: Strict Not Equal to Operator**

```
const a = 2, b = 'hello';
// strict not equal operator
console.log(a !== 2); // false
console.log(a !== '2'); // true
console.log(b !== 'Hello'); // true
```

!== evaluates to true if the operands are strictly not equal. It's the complete opposite of strictly equal ===.

In the above example, 2 !== '2' gives true. It's because their types are different even though they have the same value.

**Example 5: Greater than Operator**

```
const a = 3;
// greater than operator
console.log(a > 2); // true
```

> evaluates to true if the left operand is greater than the right operand.

**Example 6: Greater than or Equal to Operator**

```
const a = 3;
```

***// greater than or equal operator***
***console.log(a >= 3); //true***

>= evaluates to true if the left operand is greater than or equal to the right operand.

**Example 7: Less than Operator**

***const a = 3, b = 2;***
***// less than operator***
***console.log(a < 2); // false***
***console.log(b < 3); // true***

< evaluates to true if the left operand is less than the right operand.

**Example 8: Less than or Equal to Operator**

***const a = 2;***
***// less than or equal operator***
***console.log(a <= 3) // true***
***console.log(a <= 2); // true***

<= evaluates to true if the left operand is less than or equal to the right operand.

**JavaScript Logical Operators**

Logical operators perform logical operations: AND, OR and NOT.

| Operator | Description | Example |
|---|---|---|
| `&&` | **Logical AND**: `true` if both the operands/boolean values are true, else evaluates to `false` | `true && false; // false` |
| `\|\|` | **Logical OR**: `true` if either of the operands/boolean values is `true`. evaluates to `false` if both are `false` | `true \|\| false; // true` |
| `!` | **Logical NOT**: `true` if the operand is `false` and vice-versa. | `!true; // false` |

**Example 9: Logical AND Operator**

**const a = true, b = false;**
**const c = 4;**
**// logical AND**
**console.log(a && a); // true**
**console.log(a && b);  // false**
***console.log((c > 2) && (c < 2)); // false***

&& evaluates to true if both the operands are true, else evaluates to false.

**Note**: You can also use logical operators with numbers. In JavaScript, 0 is false and all non-zero values are true.

**Example 10: Logical OR Operator**

```
const a = true, b = false, c = 4;
// logical OR
console.log(a || b); // true
console.log(b || b); // false
console.log((c>2) || (c<2)); // true
```

|| evaluates to true if either of the operands is true. If both operands are false, the result is false.

**Example 11: Logical NOT Operator**

```
const a = true, b = false;
// logical NOT
console.log(!a); // false
console.log(!b); // true
```

! evaluates to true if the operand is false and vice-versa.

## *Lecture 2: JS If and Else*

In computer programming, there may arise situations where you have to run a block of code among more than one alternatives. For example, assigning grades A, B or C based on marks obtained by a student.

In such situations, you can use the JavaScript if...else statement to create a program that can make decisions.

In JavaScript, there are three forms of the if...else statement.

- ***if statement***
- ***if...else statement***
- ***if...else if...else statement***

**JavaScript if Statement**

The syntax of the if statement is:

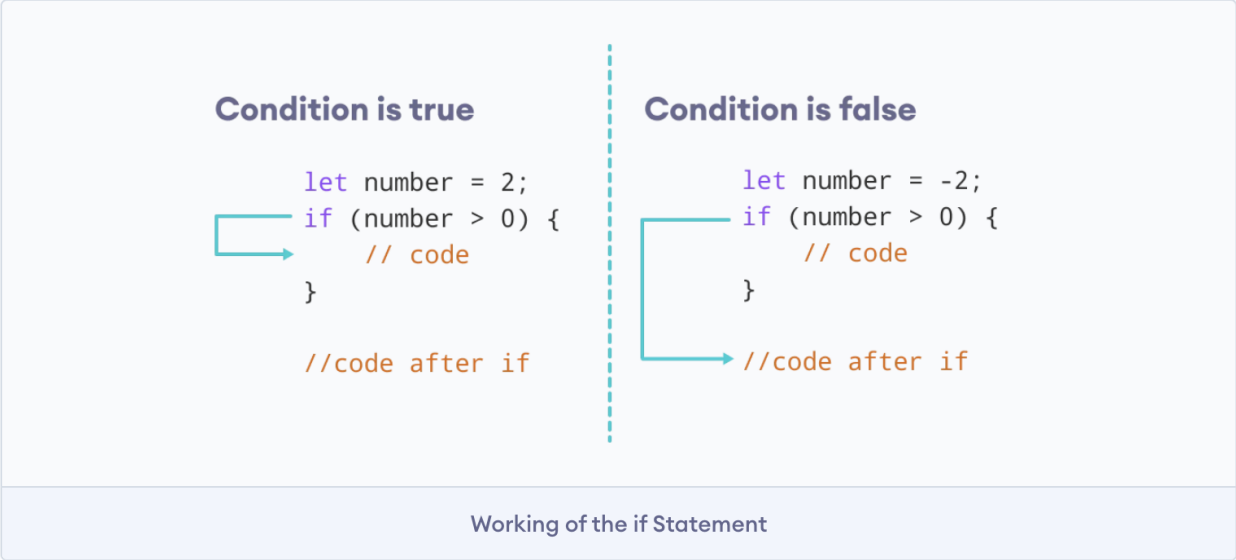
```
if(condition) {
   // the body of if
}
```

The if statement evaluates the condition inside the parenthesis ().

If the condition is evaluated to true, the code inside the body of if is executed.

If the condition is evaluated to false, the code inside the body of if is skipped.

**Note**: The code inside { } is the body of the if statement.




Working of the if Statement

**Example 1: if Statement**

```
// check if the number is positive
const number = prompt("Enter a number: ");
// check if number is greater than 0
if (number > 0) {
 // the body of the if statement
  console.log("The number is positive");
}
console.log("The if statement is easy");
```

**Output 1**

```
Enter a number: 2
The number is positive
The if statement is easy
```

Suppose the user entered 2. In this case, the condition number > 0 evaluates to true. And, the body of the if statement is executed.

**Output 2**

```
Enter a number: -1
The if statement is easy
```

Suppose the user entered -1. In this case, the condition number > 0 evaluates to false. Hence, the body of the if statement is skipped.

Since console.log("The if statement is easy"); is outside the body of the if statement, it is always executed.

**JavaScript if...else statement**

An if statement can have an optional else clause. The syntax of the if...else statement is:

```
if (condition) {
   // block of code if condition is true
} else {
  // block of code if condition is false
}
```

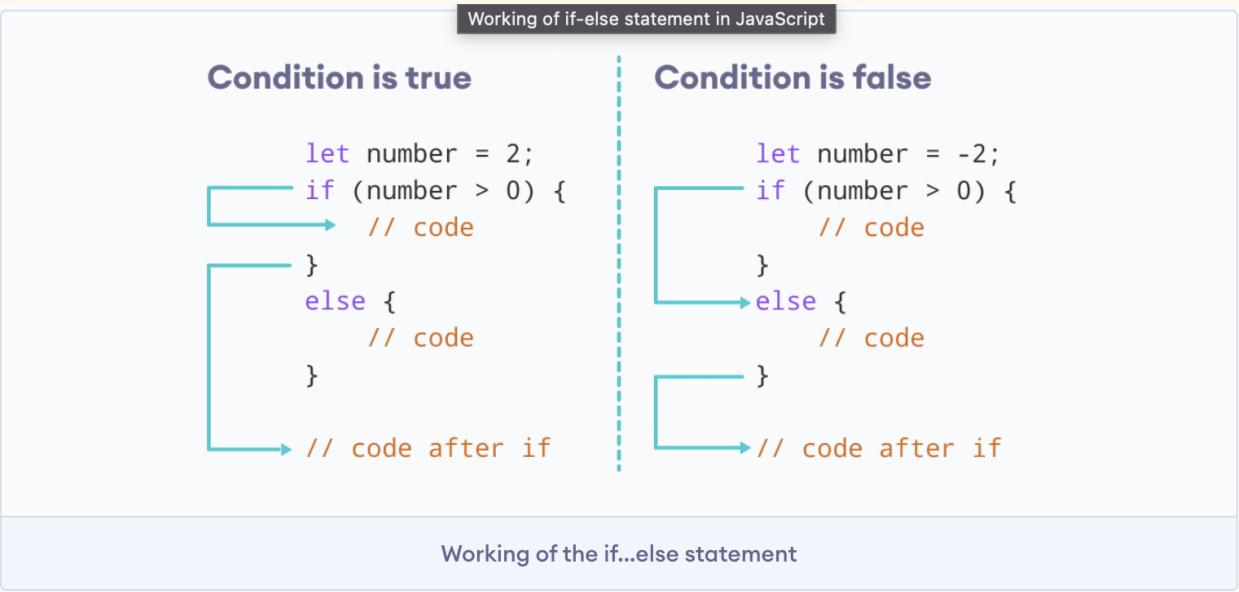
The if..else statement evaluates the condition inside the parenthesis.

If the condition is evaluated to true,

1. the code inside the body of if is executed
2. the code inside the body of else is skipped from execution

If the condition is evaluated to false,

3. the code inside the body of else is executed
4. the code inside the body of if is skipped from execution



Working of the if...else statement

**Example 2: if...else Statement**

```
// check if the number is positive or negative/zero
const number = prompt("Enter a number: ");
// check if number is greater than 0
if (number > 0) {
  console.log("The number is positive");
```

```
}
// if number is not greater than 0
else {
  console.log("The number is either a negative number or 0");
}

console.log("The if...else statement is easy");
```

**Output 1**

```
Enter a number: 2
The number is positive
The if...else statement is easy
```

Suppose the user entered 2. In this case, the condition number > 0 evaluates to true. Hence, the body of the if statement is executed and the body of the else statement is skipped.

**Output 2**

```
Enter a number: -1
The number is either a negative number or 0
The if...else statement is easy
```

Suppose the user entered -1. In this case, the condition number > 0 evaluates to false. Hence, the body of the else statement is executed and the body of the if statement is skipped.

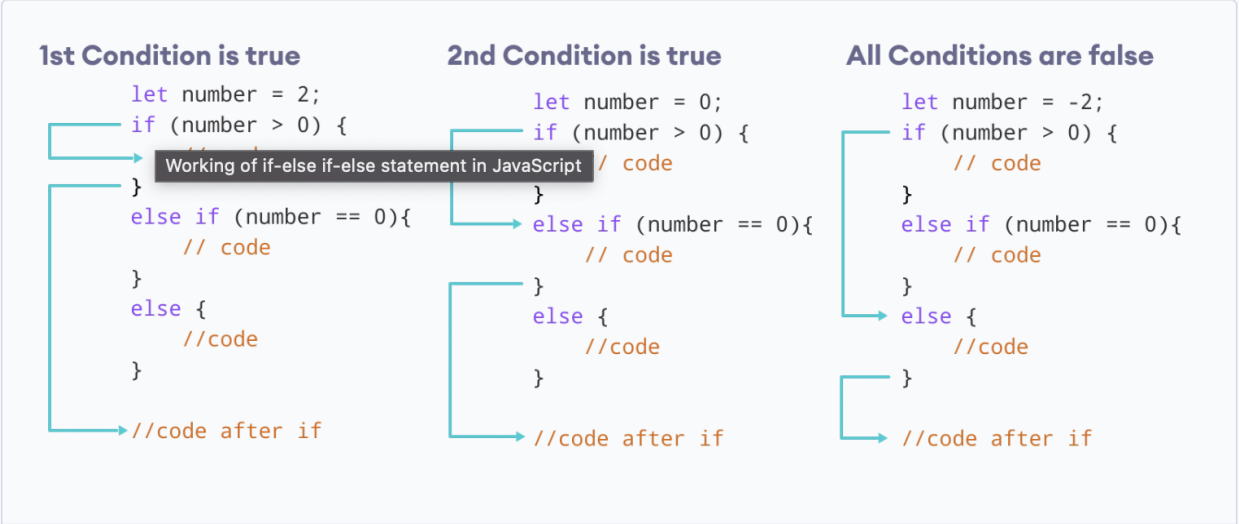
**JavaScript if...else if statement**

The if...else statement is used to execute a block of code among two alternatives. However, if you need to make a choice between more than two alternatives, if...else if...else can be used.

The syntax of the if...else if...else statement is:

```
if (condition1) {
   // code block 1
} else if (condition2){
   // code block 2
} else if (condition3){
   // code block 3
} else {
   // code block 4
}
```

- If condition1 evaluates to true, the code block 1 is executed.
- If condition1 evaluates to false, then condition2 is evaluated.
    - If the condition2 is true, the code block 2 is executed.
    - If the condition2 is false, the code block 3 is executed.




Working of the if...else if...else statement

**Example 3: if...else if Statement**

```
// check if the number if positive, negative or zero
const number = prompt("Enter a number: ");
// check if number is greater than 0
if (number > 0) {
    console.log("The number is positive");
}
// check if number is 0
else if (number == 0) {
  console.log("The number is 0");
}
// if number is neither greater than 0, nor zero
else {
    console.log("The number is negative");
}

console.log("The if...else if...else statement is easy");
```

**Output**

```
Enter a number: 0
The number is 0
The if...else if...else statement is easy
```

Suppose the user entered 0, then the first test condition number > 0 evaluates to false. Then, the second test condition number == 0 evaluates to true and its corresponding block is executed.

**Nested if...else Statement**

You can also use and if...else statement inside of an if...else statement. This is known as the nested if...else statement.

**Example 4: Nested if...else Statement**

```
// check if the number is positive, negative or zero
const number = prompt("Enter a number: ");
if (number >= 0) {
    if (number == 0) {
        console.log("You entered number 0");
    } else {
        console.log("You entered a positive number");
    }
} else {
    console.log("You entered a negative number");
}
```

**Output**

```
Enter a number: 5
You entered a positive number
```

Suppose the user entered 5. In this case, the condition number >= 0 evaluates to true, and the control of the program goes inside the outer if statement.

Then, the test condition, number == 0, of the inner if statement is evaluated. Since it's false, the else clause of the inner if statement is executed.

**Note**: As you can see, nested if...else makes our logic complicated and we should try to avoid using nested if...else whenever possible.

**Body of if...else With Only One Statement**

If the body of if...else has only one statement, we can omit { } in our programs. For example, you can replace

```
const number = 2;
if (number > 0) {
    console.log("The number is positive.");
} else {
   console.log("The number is negative or zero.");
}
```

with

```
const number = 2;
if (number > 0)
    console.log("The number is positive.");
 else
   console.log("The number is negative or zero.");
```

**Output**

***The number is positive.***

## *Lecture 3: JS  for loop*

In programming, loops are used to repeat a block of code.

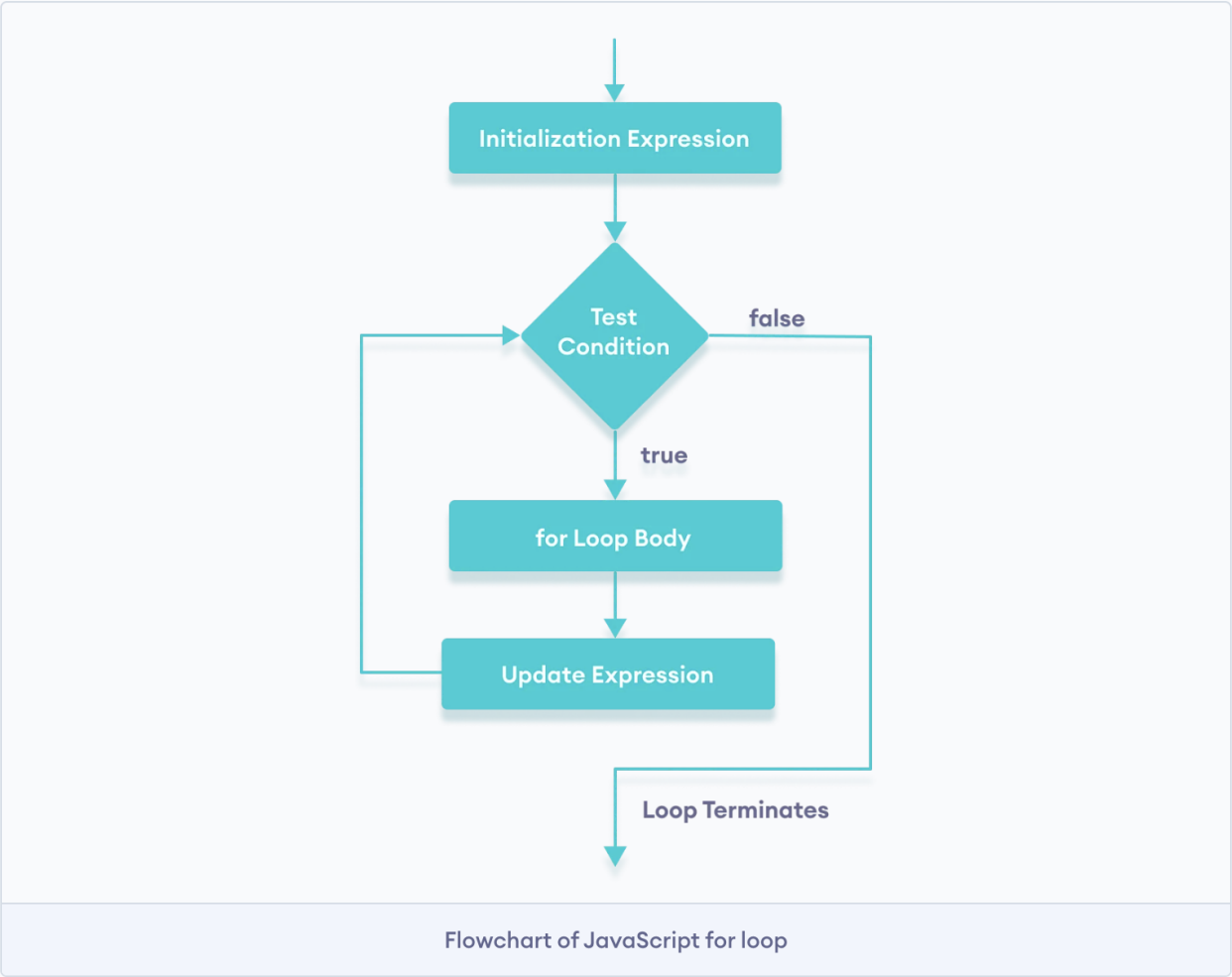
For example, if you want to show a message 100 times, then you can use a loop. It's just a simple example; you can achieve much more with loops.

The syntax of the for loop is:

```
for (initialExpression; condition; updateExpression) {
    // for loop body
}
```

Here,

1. The initialExpression initializes and/or declares variables and executes only once.
2. The condition is evaluated.
    a. If the condition is false, the for loop is terminated.
    b. If the condition is true, the block of code inside of the for loop is executed.
3. The updateExpression updates the value of initialExpression when the condition is true.
4. The condition is evaluated again. This process continues until the condition is false.




Flowchart of JavaScript for loop

**Example 1: Display a Text Five Times**

```
// program to display text 5 times
const n = 5;
// looping from i = 1 to 5
for (let i = 1; i <= n; i++) {
    console.log(`I love JavaScript.`);
}
```

**Output**

```
I love JavaScript.
I love JavaScript.
I love JavaScript.
I love JavaScript.
I love JavaScript.
```

Here is how this program works.

| Iteration | Variable | Condition: i <= n | Action |
|---|---|---|---|
| 1st | `i = 1`<br>`n = 5` | `true` | `I love JavaScript.` is printed.<br>`i` is increased to **2**. |
| 2nd | `i = 2`<br>`n = 5` | `true` | `I love JavaScript.` is printed.<br>`i` is increased to **3**. |
| 3rd | `i = 3`<br>`n = 5` | `true` | `I love JavaScript.` is printed.<br>`i` is increased to **4**. |
| 4th | `i = 4`<br>`n = 5` | `true` | `I love JavaScript.` is printed.<br>`i` is increased to **5**. |
| 5th | `i = 5`<br>`n = 5` | `true` | `I love JavaScript.` is printed.<br>`i` is increased to **6**. |
| 6th | `i = 6`<br>`n = 5` | `false` | The loop is terminated. |

**Example 2: Display Numbers from 1 to 5**

```
// program to display numbers from 1 to 5
const n = 5;
// looping from i = 1 to 5
// in each iteration, i is increased by 1
for (let i = 1; i <= n; i++) {
    console.log(i);     // printing the value of i
}
```

**Output**

```
1
2
3
4
5
```

Here is how this program works.

| Iteration | Variable | Condition: i <= n | Action |
|---|---|---|---|
| 1st | `i = 1`<br>`n = 5` | `true` | `1` is printed.<br>`i` is increased to **2**. |
| 2nd | `i = 2`<br>`n = 5` | `true` | `2` is printed.<br>`i` is increased to **3**. |
| 3rd | `i = 3`<br>`n = 5` | `true` | `3` is printed.<br>`i` is increased to **4**. |
| 4th | `i = 4`<br>`n = 5` | `true` | `4` is printed.<br>`i` is increased to **5**. |
| 5th | `i = 5`<br>`n = 5` | `true` | `5` is printed.<br>`i` is increased to **6**. |
| 6th | `i = 6`<br>`n = 5` | `false` | The loop is terminated. |

**Example 3: Display Sum of n Natural Numbers**

```
// program to display the sum of natural numbers
let sum = 0;
const n = 100

// looping from i = 1 to n
// in each iteration, i is increased by 1
for (let i = 1; i <= n; i++) {
    sum += i;  // sum = sum + i
}

console.log('sum:', sum);
```

**Output**

```
sum: 5050
```

Here, the value of sum is 0 initially. Then, a for loop is iterated from i = 1 to 100. In each iteration, i is added to the sum and its value is increased by 1.

When i becomes 101, the test condition is false and the sum will be equal to 0 + 1 + 2 + ... + 100.

The above program to add sum of natural numbers can also be written as

```
// program to display the sum of n natural numbers
let sum = 0;
const n = 100;
```

```
// looping from i = n to 1
// in each iteration, i is decreased by 1
for(let i = n; i >= 1; i-- ) {
    // adding i to sum in each iteration
    sum += i; // sum = sum + i
}

console.log('sum:',sum);
```

This program also gives the same output as Example 3. You can accomplish the same task in many different ways in programming; programming is all about logic.

Although both ways are correct, you should try to make your code more readable.

**JavaScript Infinite for loop**

If the test condition in a for loop is always true, it runs forever (until memory is full). For example,

```
// infinite for loop
for(let i = 1; i > 0; i++) {
    // block of code
}
```

In the above program, the condition is always true which will then run the code for infinite times.

## *Lecture 4: JS while Loop*

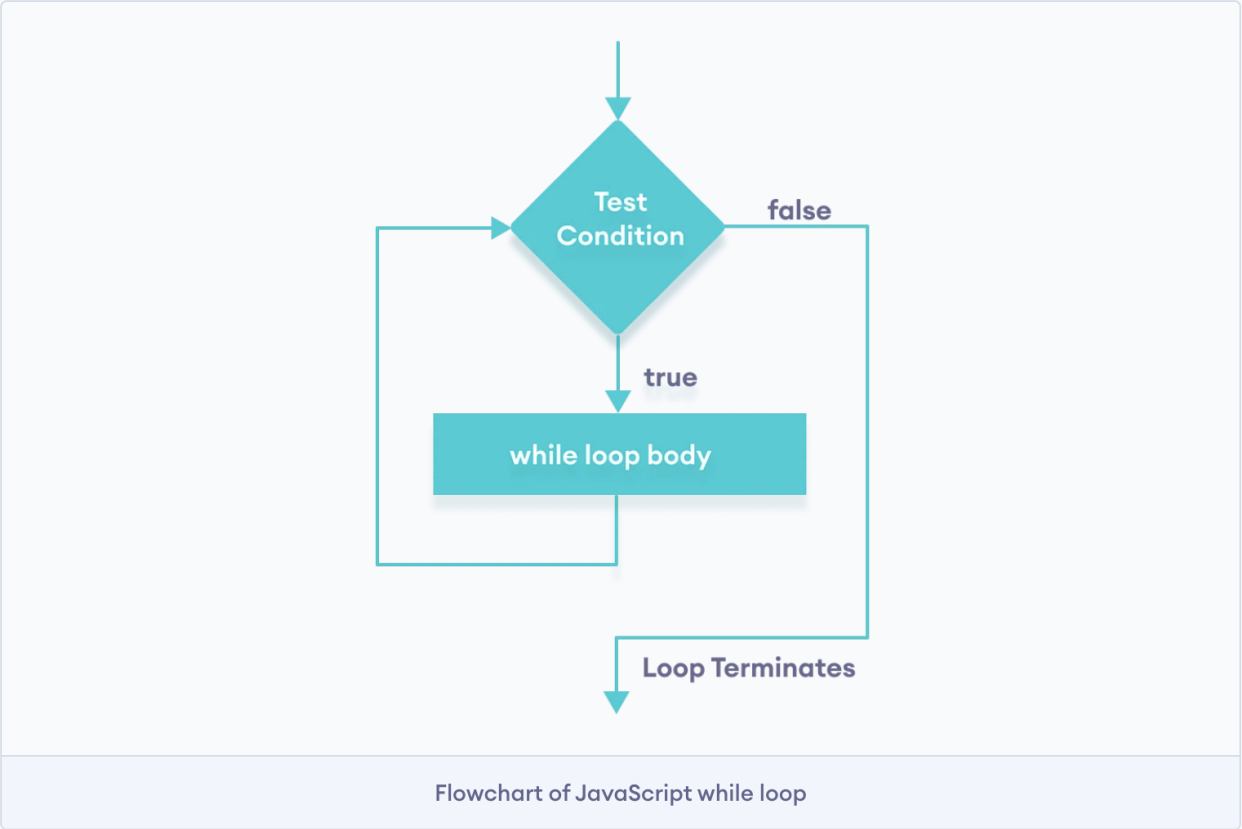
The syntax of the while loop is:

```
while (condition) {
    // body of loop
}
```

Here,

1. A while loop evaluates the condition inside the parenthesis ().
2. If the condition evaluates to true, the code inside the while loop is executed.
3. The condition is evaluated again.
4. This process continues until the condition is false.
5. When the condition evaluates to false, the loop stops.

**Flowchart of while Loop**




Flowchart of JavaScript while loop

**Example 1: Display Numbers from 1 to 5**

```
// program to display numbers from 1 to 5
// initialize the variable
let i = 1, n = 5;

// while loop from i = 1 to 5
while (i <= n) {
    console.log(i);
    i += 1;
}
```

**Output**

```
1
2
3
4
5
```

Here is how this program works

| Iteration | Variable | Condition: i <= n | Action |
|---|---|---|---|
| 1st | `i = 1`<br>`n = 5` | `true` | `1` is printed. `i` is increased to **2**. |
| 2nd | `i = 2`<br>`n = 5` | `true` | `2` is printed. `i` is increased to **3**. |
| 3rd | `i = 3`<br>`n = 5` | `true` | `3` is printed. `i` is increased to **4**. |
| 4th | `i = 4`<br>`n = 5` | `true` | `4` is printed. `i` is increased to **5**. |
| 5th | `i = 5`<br>`n = 5` | `true` | `5` is printed. `i` is increased to **6**. |
| 6th | `i = 6`<br>`n = 5` | `false` | The loop is terminated |

**Example 2: Sum of Positive Numbers Only**

```
// program to find the sum of positive numbers
// if the user enters a negative numbers, the loop ends
// the negative number entered is not added to sum

let sum = 0;

// take input from the user
let number = parseInt(prompt('Enter a number: '));

while(number >= 0) {

    // add all positive numbers
    sum += number;

    // take input again if the number is positive
    number = parseInt(prompt('Enter a number: '));
}

// display the sum
console.log(`The sum is ${sum}.`);
```

**Output**

```
Enter a number: 2
Enter a number: 5
```

***Enter a number: 7***
***Enter a number: 0***
***Enter a number: -3***
***The sum is 14.***

In the above program, the user is prompted to enter a number.

Here, parseInt() is used because prompt() takes input from the user as a string. And when numeric strings are added, it behaves as a string. For example, '2' + '3' = '23'. So parseInt() converts a numeric string to a number.

The while loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the sum variable.

When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

**JavaScript do...while Loop**

The syntax of do...while loop is:

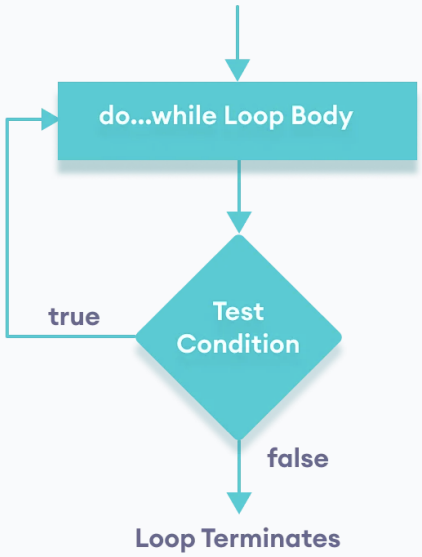
```
do {
    // body of loop
} while(condition)
```

Here,

1. The body of the loop is executed at first. Then the condition is evaluated.
2. If the condition evaluates to true, the body of the loop inside the do statement is executed again.
3. The condition is evaluated once again.
4. If the condition evaluates to true, the body of the loop inside the do statement is executed again.
5. This process continues until the condition evaluates to false. Then the loop stops.

**Note**: do...while loop is similar to the while loop. The only difference is that in do...while loop, the body of loop is executed at least once.

**Flowchart of do...while Loop**




Flowchart of JavaScript do...while loop

Let's see the working of do...while loop.

**Example 3: Display Numbers from 1 to 5**

```
// program to display numbers
let i = 1;
const n = 5;

// do...while loop from 1 to 5
do {
    console.log(i);
    i++;
} while(i <= n);
```

**Output**

```
1
2
3
4
5
```

Here is how this program works.

| Iteration | Variable | Condition: i <= n | Action |
|---|---|---|---|
| | `i = 1`<br>`n = 5` | not checked | `1` is printed. `i` is increased to **2**. |
| 1st | `i = 2`<br>`n = 5` | `true` | `2` is printed. `i` is increased to **3**. |
| 2nd | `i = 3`<br>`n = 5` | `true` | `3` is printed. `i` is increased to **4**. |
| 3rd | `i = 4`<br>`n = 5` | `true` | `4` is printed. `i` is increased to **5**. |
| 4th | `i = 5`<br>`n = 5` | `true` | `5` is printed. `i` is increased to **6**. |
| 5th | `i = 6`<br>`n = 5` | `false` | The loop is terminated |

**Example 4: Sum of Positive Numbers**

```
// to find the sum of positive numbers
// if the user enters a negative number, the loop terminates
// negative number is not added to sum

let sum = 0;
let number = 0;

do {
    sum += number;
    number = parseInt(prompt('Enter a number: '));
} while(number >= 0)

console.log(`The sum is ${sum}.`);
```

**Output 1**

```
Enter a number: 2
Enter a number: 4
Enter a number: -500
The sum is 6.
```

Here, the do...while loop continues until the user enters a negative number. When the number is negative, the loop terminates; the negative number is not added to the sum variable.

**Output 2**

***Enter a number: -80***
***The sum is 0.***

The body of the do...while loop runs only once if the user enters a negative number.

**Infinite while Loop**

If the condition of a loop is always true, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loop
while(true){
    // body of loop
}
```

Here is an example of an infinite do...while loop.

```
// infinite do...while loop
const count = 1;
do {
   // body of loop
} while(count == 1)
```

In the above programs, the condition is always true. Hence, the loop body will run for infinite times.

**for Vs while Loop**

A for loop is usually used when the number of iterations is known. For example,

```
// this loop is iterated 5 times
for (let i = 1; i <=5; ++i) {
   // body of loop
}
```

And while and do...while loops are usually used when the number of iterations are unknown. For example,

```
while (condition) {
    // body of loop
}
```

## *Lecture 5: JS break*

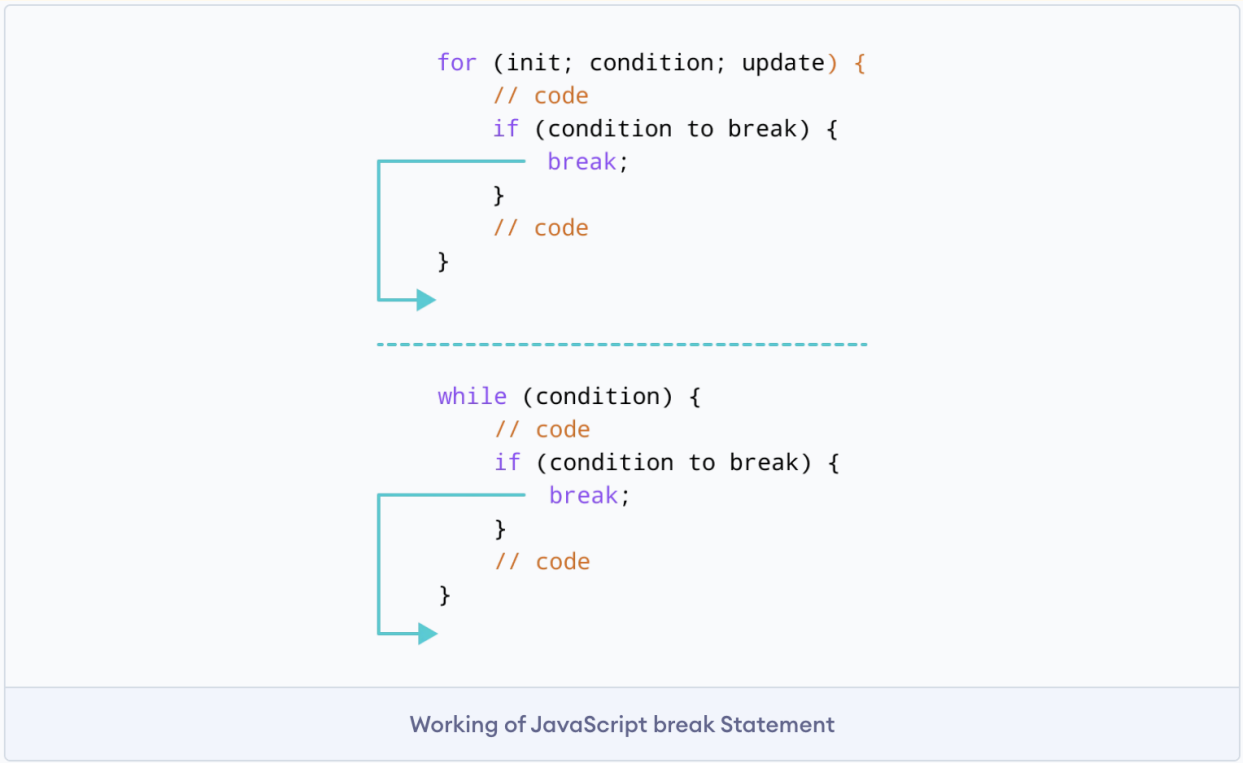
The break statement is used to terminate the loop immediately when it is encountered.

The syntax of the break statement is:

```
break [label];
```

**Note**: label is optional and rarely used.

### Working of JavaScript break Statement




Working of JavaScript break Statement

### Example 1: break with for Loop

```
// program to print the value of i
for (let i = 1; i <= 5; i++) {
   // break condition
   if (i == 3) {
      break;
   }
   console.log(i);
}
```

### Output

```
1
2
```

In the above program, the for loop is used to print the value of i in each iteration. The break statement is used as:

```
if(i == 3) {
   break;
}
```

This means, when i is equal to 3, the break statement terminates the loop. Hence, the output doesn't include values greater than or equal to 3.

**Note**: The break statement is almost always used with decision-making statements. To learn more, visit JavaScript if...else Statement.

**Example 2: break with while Loop**

```
// program to find the sum of positive numbers
// if the user enters a negative number, break ends the loop
// the negative number entered is not added to sum
let sum = 0, number;
while(true) {
    // take input again if the number is positive
    number = parseInt(prompt('Enter a number: '));
    // break condition
    if(number < 0) {
        break;
    }
    // add all positive numbers
    sum += number;
}
// display the sum
console.log(`The sum is ${sum}.`);
```

**Output**

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: -5
The sum is 6.
```

In the above program, the user enters a number. The while loop is used to print the total sum of numbers entered by the user.

Here the break statement is used as:

```
if(number < 0) {
    break;
}
```

When the user enters a negative number, here -5, the break statement terminates the loop and the control flow of the program goes outside the loop.

Thus, the while loop continues until the user enters a negative number.

**break with Nested Loop**

When the break is used inside of two nested loops, the break terminates the inner loop. For example,

```
// nested for loops
// first loop
for (let i = 1; i <= 3; i++) {
   // second loop
   for (let j = 1; j <= 3; j++) {
      if (i == 2) {
        break;
      }
      console.log(`i = ${i}, j = ${j}`);
   }
}
```

**Output**

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

In the above program, when i == 2, the break statement executes. It terminates the inner loop and control flow of the program moves to the outer loop.

Hence, the value of i = 2 is never displayed in the output.

## *Lecture 6: JS continue*

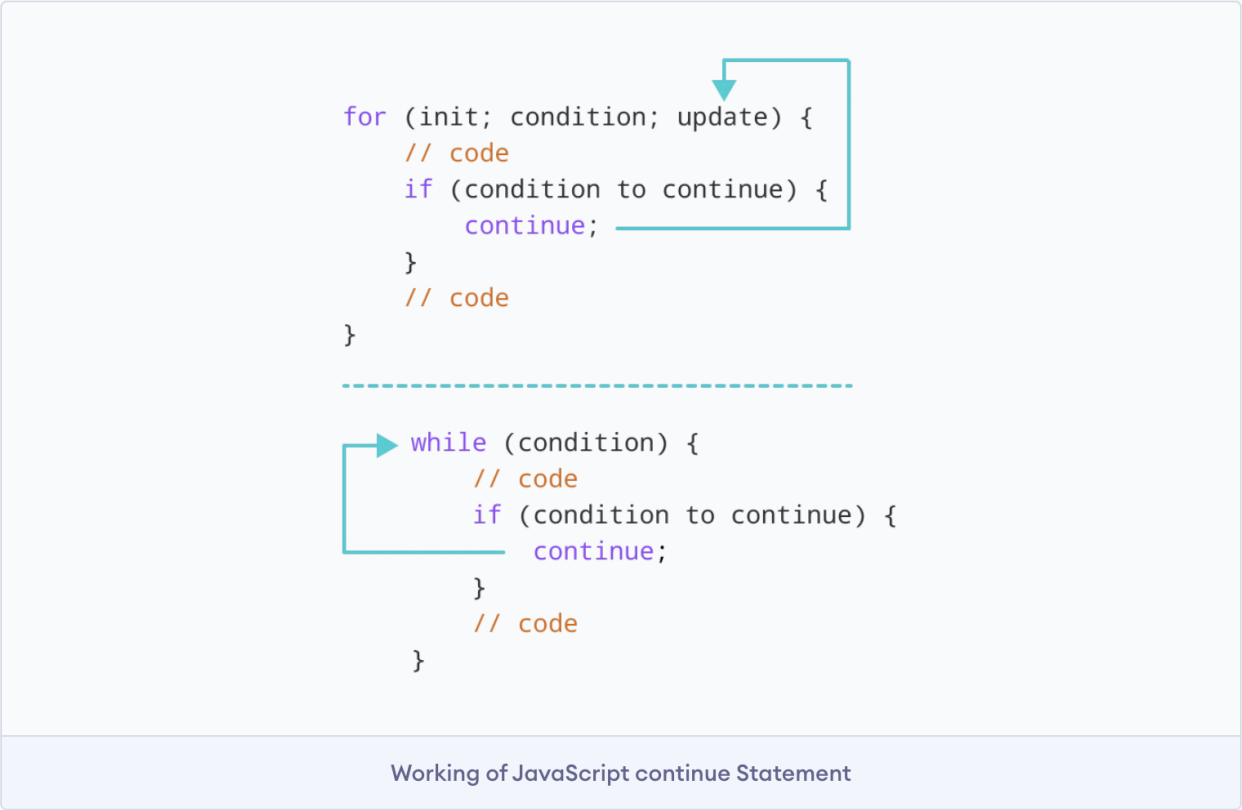
The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

The syntax of the continue statement is:

```
continue [label];
```

**Note**: label is optional and rarely used.

**Working of JavaScript continue Statement**




Working of JavaScript continue Statement

**continue with for Loop**

In a for loop, continue skips the current iteration, and control flow jumps to the updateExpression.

**Example 1: Print the Value of i**

```
// program to print the value of i
for (let i = 1; i <= 5; i++) {
   // condition to continue
   if (i == 3) {
      continue;
   }
   console.log(i);
}
```

**Output**

```
1
2
4
5
```

In the above program, a for loop is used to print the value of i in each iteration.

Notice the continue statement inside the loop.

```
if(i == 3) {
   continue;
}
```

This means

- When i is equal to 3, the continue statement skips the third iteration.
- Then, i becomes 4 and the test condition and continue statement is evaluated again.
- Hence, 4 and 5 are printed in the next two iterations.

**Note**: The continue statement is almost always used with decision-making statements. To learn more, visit JavaScript if...else Statement.

**Note**: The break statement terminates the loop entirely. However, the continue statement only skips the current iteration.

**continue with while Loop**

In a while loop, continue skips the current iteration and control flow of the program jumps back to the while condition.

The continue statement works in the same way for while and do...while loops.

**Example 2: Calculate Positive Number**

```
// program to calculate positive numbers only
// if the user enters a negative number, that number is skipped from calculation
// negative number -> loop terminate
// non-numeric character -> skip iteration
let sum = 0;
let number = 0;
while (number >= 0) {
   // add all positive numbers
   sum += number;
   // take input from the user
   number = parseInt(prompt('Enter a number: '));
   // continue condition
   if (isNaN(number)) {
      console.log('You entered a string.');
      number = 0; // the value of number is made 0 again
      continue;
   }
}
// display the sum
console.log(`The sum is ${sum}.`);
```

**Output**

```
Enter a number: 1
Enter a number: 2
```

***Enter a number: hello***
***You entered a string.***
***Enter a number: 5***
***Enter a number: -2***
***The sum is 8.***

In the above program, the user enters a number. The while loop is used to print the total sum of positive numbers entered by the user.

Notice the use of the continue statement.

```
if (isNaN(number)) {
    continue;
}
```

When the user enters a non-numeric number/string, the continue statement skips the current iteration. Then the control flow of the program goes to the condition of the while loop.

When the user enters a number less than 0, the loop terminates.

In the above program, isNaN() is used to check if the value entered by a user is a number or not.

**continue with Nested Loop**

When continue is used inside of two nested loops, continue skips the current iteration of the inner loop. For example,

```
// nested for loops
// first loop
for (let i = 1; i <= 3; i++) {
    // second loop
    for (let j = 1; j <= 3; j++) {
        if (j == 2) {
          continue;
        }
        console.log(`i = ${i}, j = ${j}`);
    }
}
```

**Output**

```
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 3
```

In the above program, when the continue statement executes, it skips the current iteration in the inner loop and control flow of the program moves to the updateExpression of the inner loop.

Hence, the value of j = 2 is never displayed in the output.

## *Lecture 7: JS switch*

The JavaScript switch statement is used in decision-making.

The switch statement evaluates an expression and executes the corresponding body that matches the expression's result.

The syntax of the switch statement is:

```
switch(variable/expression) {
   case value1:
      // body of case 1
      break;
   case value2:
      // body of case 2
      break;
   case valueN:
      // body of case N
      break;
   default:
      // body of default
}
```

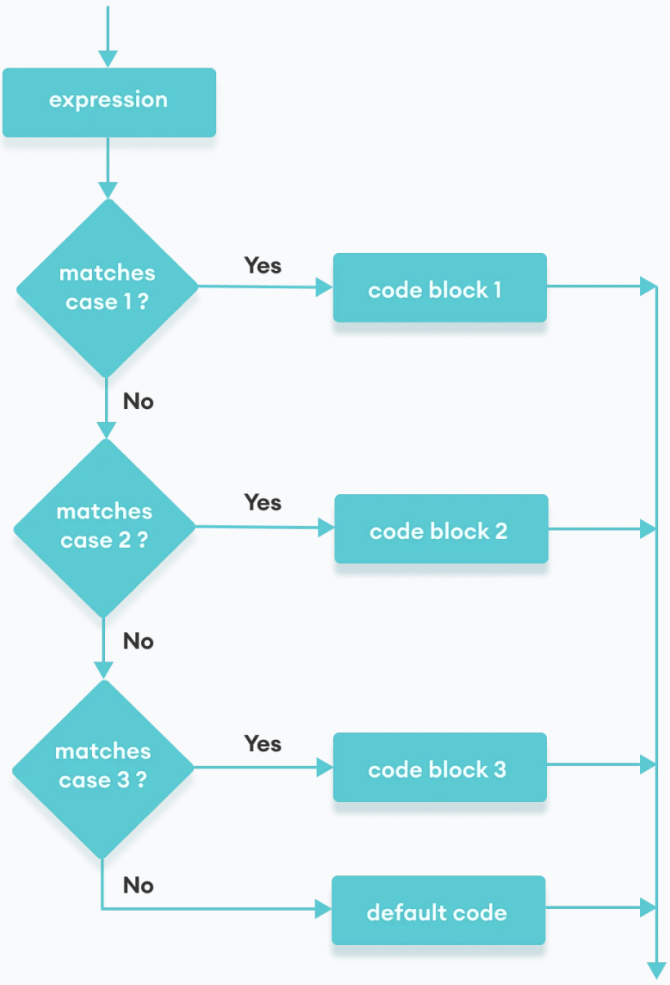
The switch statement evaluates a variable/expression inside parentheses ().

- If the result of the expression is equal to value1, its body is executed.
- If the result of the expression is equal to value2, its body is executed.
- This process goes on. If there is no matching case, the default body executes.

**Notes**:

- The break statement is optional. If the break statement is encountered, the switch statement ends.
- If the break statement is not used, the cases after the matching case are also executed.
- The default clause is also optional.

**Flowchart of switch Statement**




Flowchart of JavaScript switch statement

**Example 1: Simple Program Using switch Statement**

```
// program using switch statement
let a = 2;
switch (a) {
    case 1:
        a = 'one';
        break;
    case 2:
        a = 'two';
        break;
    default:
        a = 'not found';
        break;
}
console.log(`The value is ${a}`);
```

**Output**

***The value is two.***

In the above program, an expression a = 2 is evaluated with a switch statement.

The expression's result is evaluated with case 1 which results in false.

Then the switch statement goes to the second case. Here, the expression's result matches with case 2. So The value is two is displayed.

The break statement terminates the block and control flow of the program jumps to outside of the switch block.

**Example 2: Type Checking in switch Statement**

```
// program using switch statement
let a = 1;
switch (a) {
   case "1":
      a = 1;
      break;
   case 1:
      a = 'one';
      break;
   case 2:
      a = 'two';
      break;

   default:
      a = 'not found';
      break;
}
console.log(`The value is ${a}`);
```

**Output**

***The value is one.***

In the above program, an expression a = 1 is evaluated with a switch statement.

In JavaScript, the switch statement checks the value strictly. So the expression's result does not match with case "1".

Then the switch statement goes to the second case. Here, the expression's result matches with case 1. So The value is one is displayed.

The break statement terminates the block and control flow of the program jumps to outside of the switch block.

**Note**: In JavaScript, the switch statement checks the cases strictly (should be of the same data type) with the expression's result. Notice in the above example, 1 does not match with "1".

Let's write a program to make a simple calculator with the switch statement.

**Example 3: Simple Calculator**

```
// program for a simple calculator
let result;
// take the operator input
const operator = prompt(`Enter operator ( either +, -, * or / ): `);
// take the operand input
const number1 = parseFloat(prompt(`Enter first number: `));
const number2 = parseFloat(prompt(`Enter second number: `));
switch(operator) {
   case '+':
      result = number1 + number2;
      console.log(`${number1} + ${number2} = ${result}`);
      break;
   case '-':
      result = number1 - number2;
      console.log(`${number1} - ${number2} = ${result}`);
      break;
   case '*':
      result = number1 * number2;
      console.log(`${number1} * ${number2} = ${result}`);
      break;
   case '/':
      result = number1 / number2;
      console.log(`${number1} / ${number2} = ${result}`);
      break;
   default:
      console.log('Invalid operator');
      break;
}
```

**Output**

```
Enter operator: +
Enter first number: 4
Enter second number: 5
4 + 5 = 9
```

In above program, the user is asked to enter either +, -, * or /, and two operands. Then, the switch statement executes cases based on the user input.

**JavaScript switch With Multiple Case**

In a JavaScript switch statement, cases can be grouped to share the same code.

**Example 4: switch With Multiple Case**

```
// multiple case switch program
let fruit = 'apple';
switch(fruit) {
   case 'apple':
   case 'mango':
   case 'pineapple':
      console.log(`${fruit} is a fruit.`);
      break;
   default:
      console.log(`${fruit} is not a fruit.`);
      break;
}
```

**Output**

```
apple is a fruit.
```

In the above program, multiple cases are grouped. All the grouped cases share the same code.

If the value of the fruit variable had value mango or pineapple, the output would have been the same.

# Chapter 3: JS Functions

## *Lecture 1: JS Function and Function Expressions*

**JavaScript Function**

A function is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

JavaScript also has a huge number of inbuilt functions. For example, Math.sqrt() is a function to calculate the square root of a number.

**Declaring a Function**

The syntax to declare a function is:

```
function nameOfFunction () {
   // function body
}
```

- A function is declared using the function keyword.
- The basic rules of naming a function are similar to naming a variable. It is better to write a descriptive name for your function. For example, if a function is used to add two numbers, you could name the function add or addNumbers.
- The body of function is written within {}.

For example,

```
// declaring a function named greet()
function greet() {
   console.log("Hello there");
}
```

**Calling a Function**

In the above program, we have declared a function named greet(). To use that function, we need to call it.

Here's how you can call the above greet() function.

```
// function call
greet();
```


Working of a Function in JavaScript

**Example 1: Display a Text**

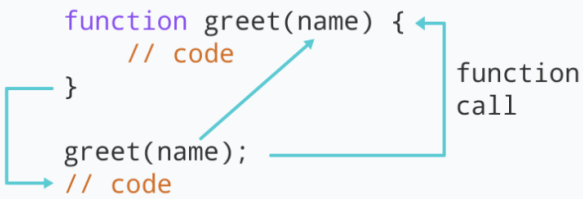
```
// program to print a text
// declaring a function
function greet() {
    console.log("Hello there!");
}
// calling the function
greet();
```

**Output**

```
Hello there!
```

**Function Parameters**

A function can also be declared with parameters. A parameter is a value that is passed when declaring a function.




Working of JavaScript Function with parameter

**Example 2: Function with Parameters**

```
// program to print the text
// declaring a function
function greet(name) {
    console.log("Hello " + name + ":)");
}
```

```
// variable name can be different
let name = prompt("Enter a name: ");

// calling function
greet(name);
```

**Output**

```
Enter a name: Simon
Hello Simon :)
```

In the above program, the greet function is declared with a name parameter. The user is prompted to enter a name. Then when the function is called, an argument is passed into the function.

**Note**: When a value is passed when declaring a function, it is called parameter. And when the function is called, the value passed is called argument.

**Example 3: Add Two Numbers**

```
// program to add two numbers using a function
// declaring a function
function add(a, b) {
    console.log(a + b);
}

// calling functions
add(3,4);
add(2,9);
```

**Output**

```
7
11
```

In the above program, the add function is used to find the sum of two numbers.

The function is declared with two parameters a and b.

The function is called using its name and passing two arguments 3 and 4 in one and 2 and 9 in another.

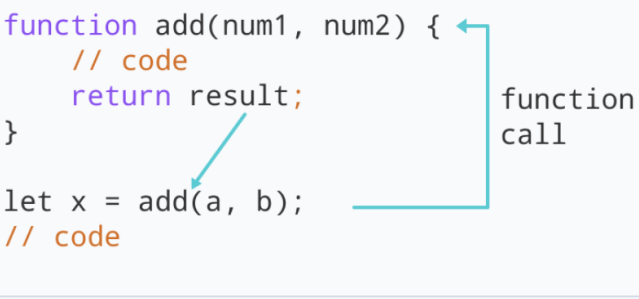
Notice that you can call a function as many times as you want. You can write one function and then call it multiple times with different arguments.

**Function Return**

The return statement can be used to return the value to a function call.

The return statement denotes that the function has ended. Any code after return is not executed.

If nothing is returned, the function returns an undefined value.




Working of JavaScript Function with return statement

**Example 4: Sum of Two Numbers**

```
// program to add two numbers
// declaring a function
function add(a, b) {
   return a + b;
}

// take input from the user
let number1 = parseFloat(prompt("Enter first number: "));
let number2 = parseFloat(prompt("Enter second number: "));

// calling function
let result = add(number1,number2);

// display the result
console.log("The sum is " + result);
```

**Output**

```
Enter first number: 3.4
Enter second number: 4
The sum is 7.4
```

In the above program, the sum of the numbers is returned by the function using the return statement. And that value is stored in the result variable.

**Benefits of Using a Function**

- Function makes the code reusable. You can declare it once and use it multiple times.
- Function makes the program easier as each small task is divided into a function.
- Function increases readability.

**Function Expressions**

*In Javascript, functions can also be defined as expressions. For example,*

```
// program to find the square of a number
// function is declared inside the variable
let x = function (num) { return num * num };
console.log(x(4));

// can be used as variable value for other variables
let y = x(3);
console.log(y);
```

**Output**

```
16
9
```

In the above program, variable x is used to store the function. Here the function is treated as an expression. And the function is called using the variable name.

The function above is called an anonymous function.

**Note**: In ES2015, JavaScript expressions are written as arrow functions.

```
const square = (side) => {
        return side*side;
}
console.log(square(4));
```

## *Lecture 2: JS Variable Scope*

Scope refers to the availability of variables and functions in certain parts of the code.

In JavaScript, a variable has two types of scope:

- **Global Scope**
- **Local Scope**

**Global Scope**

A variable declared at the top of a program or outside of a function is considered a global scope variable.

Let's see an example of a global scope variable.

```
// program to print a text
let a = "hello";
function greet () {
```

```
    console.log(a);
}
greet(); // hello
```

In the above program, variable a is declared at the top of a program and is a global variable. It means the variable a can be used anywhere in the program.

The value of a global variable can be changed inside a function. For example,

```
// program to show the change in global variable
let a = "hello";
function greet() {
    a = 3;
}
// before the function call
console.log(a);
//after the function call
greet();
console.log(a); // 3
```

In the above program, variable a is a global variable. The value of a is hello. Then the variable a is accessed inside a function and the value changes to 3.

Hence, the value of a changes after changing it inside the function.

**Note**: It is a good practice to avoid using global variables because the value of a global variable can change in different areas in the program. It can introduce unknown results in the program.

In JavaScript, a variable can also be used without declaring it. If a variable is used without declaring it, that variable automatically becomes a global variable.

For example,

```
function greet() {
    a = "hello"
}
greet();
console.log(a); // hello
```

In the above program, variable a is a global variable.

If the variable was declared using let a = "hello", the program would throw an error.

**Note**: In JavaScript, there is "strict mode"; in which a variable cannot be used without declaring it. To learn more about strict, visit JavaScript Strict.

**Local Scope**

A variable can also have a local scope, i.e it can only be accessed within a function.

**Example 1: Local Scope Variable**

```
// program showing local scope of a variable
let a = "hello";
function greet() {
   let b = "World"
   console.log(a + b);
}
greet();
console.log(a + b); // error
```

**Output**

```
helloWorld
Uncaught ReferenceError: b is not defined
```

In the above program, variable a is a global variable and variable b is a local variable. The variable b can be accessed only inside the function greet. Hence, when we try to access variable b outside of the function, an error occurs.

let is Block Scoped

The let keyword is block-scoped (variable can be accessed only in the immediate block).

**Example 2: block-scoped Variable**

```
// program showing block-scoped concept
// global variable
let a = 'Hello';
function greet() {
   // local variable
   let b = 'World';
   console.log(a + ' ' + b);
   if (b == 'World') {
      // block-scoped variable
      let c = 'hello';
      console.log(a + ' ' + b + ' ' + c);
   }
   // variable c cannot be accessed here
   console.log(a + ' ' + b + ' ' + c);
}
greet();
```

**Output**

```
Hello World
Hello World hello
Uncaught ReferenceError: c is not defined
```

**In the above program, variable a is a global variable. It can be accessed anywhere in the program.**

b is a local variable. It can be accessed only inside the function greet.

c is a block-scoped variable. It can be accessed only inside the if statement block.

Hence, in the above program, the first two console.log() work without any issue.

However, we are trying to access the block-scoped variable c outside of the block in the third console.log(). This will throw an error.

**Note**: In JavaScript, var is function scoped and let is block-scoped. If you try to use var c = 'hello'; inside the if statement in the above program, the whole program works, as c is treated as a local variable.

## *Lecture 3: JS Hoisting*

Hoisting in JavaScript is a behavior in which a function or a variable can be used before declaration. For example,

```
// using test before declaring
console.log(test);   // undefined
var test;
```

The above program works and the output will be undefined. The above program behaves as

```
// using test before declaring
var test;
console.log(test); // undefined
```

Since the variable test is only declared and has no value, undefined value is assigned to it.

**Note**: In hoisting, though it seems that the declaration has moved up in the program, the actual thing that happens is that the function and variable declarations are added to memory during the compile phase.

**Variable Hoisting**

In terms of variables and constants, keyword var is hoisted and let and const does not allow hoisting.

For example,

```
// program to display value
a = 5;
console.log(a);
var a; // 5
```

In the above example, variable a is used before declaring it. And the program works and displays the output 5. The program behaves as:

```
// program to display value
var a;
a = 5;
console.log(a); // 5
```

However in JavaScript, initializations are not hoisted. For example,

```
// program to display value
console.log(a);
var a = 5;
```

**Output**

```
undefined
```

The above program behaves as:

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

Only the declaration is moved to the memory in the compile phase. Hence, the value of variable a is undefined because a is printed without initializing it.

Also, when the variable is used inside the function, the variable is hoisted only to the top of the function. For example,

```
// program to display value
var a = 4;
function greet() {
    b = 'hello';
    console.log(b); // hello
    var b;
}
greet(); // hello
console.log(b);
```

**Output**

```
hello
Uncaught ReferenceError: b is not defined
```

In the above example, variable b is hoisted to the top of the function greet and becomes a local variable. Hence b is only accessible inside the function. b does not become a global variable.

**Note**: In hoisting, the variable declaration is only accessible to the immediate scope.

If a variable is used with the let keyword, that variable is not hoisted. For example,

```
// program to display value
a = 5;
console.log(a);
```

***let a; // error***

**Output**

***Uncaught ReferenceError: Cannot access 'a' before initialization***

While using let, the variable must be declared first.

**Function Hoisting**

A function can be called before declaring it. For example,

```
// program to print the text
greet();

function greet() {
    console.log('Hi, there.');
}
```

**Output**

***Hi, there***

In the above program, the function greet is called before declaring it and the program shows the output. This is due to hoisting.

However, when a function is used as an expression, an error occurs because only declarations are hoisted. For example;

```
// program to print the text
greet();

let greet = function() {
    console.log('Hi, there.');
}
```

**Output**

***Uncaught ReferenceError: greet is not defined***

If var was used in the above program, the error would be:
*Uncaught TypeError: greet is not a function*

**Note**: Generally, hoisting is not performed in other programming languages like Python, C, C++, Java.

Hoisting can cause undesirable outcomes in your program. And it is best to declare variables and functions first before using them and avoid hoisting.

In the case of variables, it is better to use let than var.

## *Lecture 4: JS Recursion*

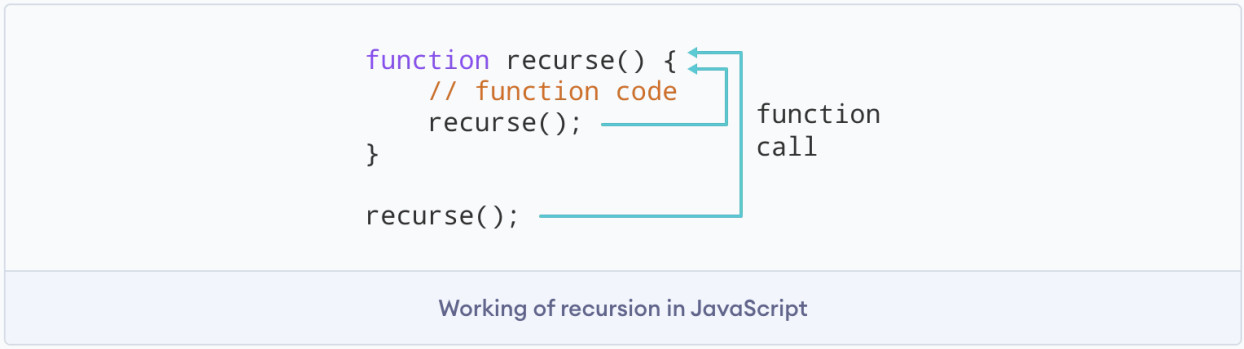
Recursion is a process of calling itself. A function that calls itself is called a recursive function.

The syntax for recursive function is:

```
function recurse() {
   // function code
   recurse();
   // function code
}

recurse();
```

Here, the recurse() function is a recursive function. It is calling itself inside the function.




Working of recursion in JavaScript

A recursive function must have a condition to stop calling itself. Otherwise, the function is called indefinitely.

Once the condition is met, the function stops calling itself. This is called a base condition.

To prevent infinite recursion, you can use the if...else statement (or similar approach) where one branch makes the recursive call, and the other doesn't.

So, it generally looks like this.

```
function recurse() {
   if(condition) {
      recurse();
   }
   else {
      // stop calling recurse()
   }
}

recurse();
```

A simple example of a recursive function would be to count down the value to 1.

**Example 1: Print Numbers**

```
// program to count down numbers to 1
function countDown(number) {
    // display the number
    console.log(number);
    // decrease the number value
    const newNumber = number - 1;
    // base case
    if (newNumber > 0) {
        countDown(newNumber);
    }
}

countDown(4);
```

**Output**

```
4
3
2
1
```

In the above program, the user passes a number as an argument when calling a function.

In each iteration, the number value is decreased by 1 and function countDown() is called until the number is positive. Here, newNumber > 0 is the base condition.

This recursive call can be explained in the following steps:

```
countDown(4) prints 4 and calls countDown(3)
countDown(3) prints 3 and calls countDown(2)
countDown(2) prints 2 and calls countDown(1)
countDown(1) prints 1 and calls countDown(0)
```

When the number reaches 0, the base condition is met, and the function is not called anymore.

**Example 2: Find Factorial**

```
// program to find the factorial of a number
function factorial(x) {
    // if number is 0
    if (x === 0) {
        return 1;
    }
    // if number is positive
    else {
        return x * factorial(x - 1);
    }
}
const num = 3;
// calling factorial() if num is non-negative
```

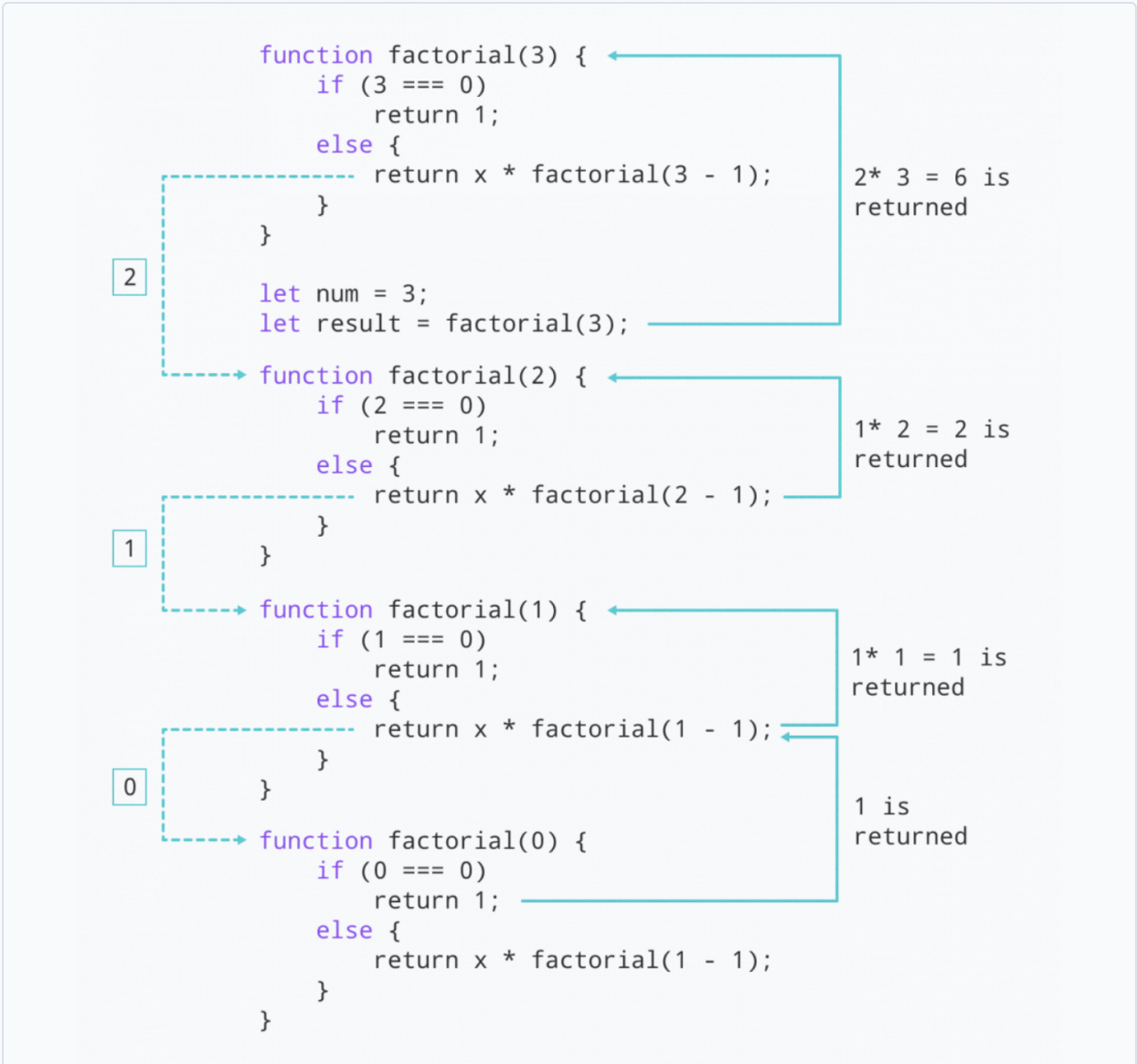
```
if (num > 0) {
    let result = factorial(num);
    console.log(`The factorial of ${num} is ${result}`);
}
```

**Output**

***The factorial of 3 is 6***

When you call the function factorial() with a positive integer, it will recursively call itself by decreasing the number.

This process continues until the number becomes 1. Then when the number reaches 0, 1 is returned.




Computing Factorial Using Recursion

This recursive call can be explained in the following steps:

**factorial(3) returns 3 * factorial(2)**
**factorial(2) returns 3 * 2 * factorial(1)**
**factorial(1) returns 3 * 2 * 1 * factorial(0)**
**factorial(0) returns 3 * 2 * 1 * 1**

# Chapter 4: Objects

## *Lecture 1: JS Objects*

JavaScript object is a non-primitive data-type that allows you to store multiple collections of data.

**Note**: If you are familiar with other programming languages, JavaScript objects are a bit different. You do not need to create classes in order to create objects.

Here is an example of a JavaScript object.

```
// object
const student = {
   firstName: 'ram',
   class: 10
};
```

Here, student is an object that stores values such as strings and numbers.

**JavaScript Object Declaration**

The syntax to declare an object is:

```
const object_name = {
  key1: value1,
  key2: value2
}
```

Here, an object object_name is defined. Each member of an object is a key: value pair separated by commas and enclosed in curly braces {}.

For example,

```
// object creation
const person = {
   name: 'John',
   age: 20
};
console.log(typeof person); // object
```

You can also define an object in a single line.

```
const person = { name: 'John', age: 20 };
```

In the above example, name and age are keys, and John and 20 are values respectively.

There are other ways to declare an object in JavaScript.

**JavaScript Object Properties**

In JavaScript, "key: value" pairs are called properties. For example,

```
let person = {
   name: 'John',
   age: 20
};
```

Here, name: 'John' and age: 20 are properties.


JavaScript object properties

### Accessing Object Properties

You can access the value of a property by using its key.

#### 1. Using dot Notation

Here's the syntax of the dot notation.

objectName.key

For example,

```
const person = {
   name: 'John',
   age: 20,
};
// accessing property
console.log(person.name); // John
```

#### 2. Using bracket Notation

Here is the syntax of the bracket notation.

```
objectName["propertyName"]
```

For example,

```
const person = {
   name: 'John',
   age: 20,
};
// accessing property
console.log(person["name"]); // John
```

### JavaScript Nested Objects

An object can also contain another object. For example,

```
// nested object
const student = {
   name: 'John',
   age: 20,
   marks: {
      science: 70,
      math: 75
   }
}
// accessing property of student object
console.log(student.marks); // {science: 70, math: 75}

// accessing property of marks object
console.log(student.marks.science); // 70
```

In the above example, an object student contains an object value in the marks property.

**JavaScript Object Methods**

In JavaScript, an object can also contain a function. For example,

```
const person = {
   name: 'Sam',
   age: 30,
   // using function as a value
   greet: function() { console.log('hello') }
}

person.greet(); // hello
```

Here, a function is used as a value for the greet key. That's why we need to use person.greet() instead of person.greet to call the function inside the object.

A JavaScript method is a property containing a function declaration.

## *Lecture 2: JS Methods*

In JavaScript, objects can also contain functions. For example,

```
// object containing method
const person = {
   name: 'John',
   greet: function() { console.log('hello'); }
};
```

In the above example, a person object has two keys (name and greet), which have a string value and a function value, respectively.

Hence basically, the JavaScript method is an object property that has a function value.

**Accessing Object Methods**

You can access an object method using a dot notation. The syntax is:

***objectName.methodKey()***

You can access property by calling an objectName and a key. You can access a method by calling an objectName and a key for that method along with (). For example,

```
// accessing method and property
const person = {
   name: 'John',
   greet: function() { console.log('hello'); }
};

// accessing property
person.name; // John

// accessing method
person.greet(); // hello
```

Here, the greet method is accessed as person.greet() instead of person.greet.

If you try to access the method with only person.greet, it will give you a function definition.

```
person.greet; // f () { console.log('hello'); }
```

**JavaScript Built-In Methods**

In JavaScript, there are many built-in methods. For example,

```
let number = '23.32';
let result = parseInt(number);

console.log(result); // 23
```

Here, the parseInt() method of Number object is used to convert numeric string value to an integer value.

**Adding a Method to a JavaScript Object**

You can also add a method in an object. For example,

```
// creating an object
let student = { };

// adding a property
student.name = 'John';
```

```
// adding a method
student.greet = function() {
   console.log('hello');
}

// accessing a method
student.greet(); // hello
```

In the above example, an empty student object is created. Then, the name property is added. Similarly, the greet method is also added. In this way, you can add a method as well as property to an object.

**JavaScript this Keyword**

To access a property of an object from within a method of the same object, you need to use the this keyword. Let's consider an example.

```
const person = {
   name: 'John',
   age: 30,

   // accessing name property by using this.name
   greet: function() { console.log('The name is' + ' ' + this.name); }
};

person.greet();
```

**Output**

```
The name is John
```

In the above example, a person object is created. It contains properties (name and age) and a method greet.

In the method greet, while accessing a property of an object, this keyword is used.

In order to access the properties of an object, this keyword is used following by . and key.

**Note**: In JavaScript, this keyword when used with the object's method refers to the object. this is bound to an object.

However, the function inside of an object can access its variable in a similar way as a normal function would. For example,

```
const person = {
   name: 'John',
   age: 30,
   greet: function() {
      let surname = 'Doe';
      console.log('The name is' + ' ' + this.name + ' ' + surname); }
```

```
};

person.greet();
```

Output

***The name is John Doe***

## *Lecture 3: JS Constructor Function*

In JavaScript, a constructor function is used to create objects. For example,

```
// constructor function
function Person () {
   this.name = 'John',
   this.age = 23
}

// create an object
const person = new Person();
```

In the above example, function Person() is an object constructor function.

To create an object from a constructor function, we use the new keyword.

**Note**: It is considered a good practice to capitalize the first letter of your constructor function.

Create Multiple Objects with Constructor Function

In JavaScript, you can create multiple objects from a constructor function. For example,

```
// constructor function
function Person () {
   this.name = 'John',
   this.age = 23,

    this.greet = function () {
       console.log('hello');
   }
}

// create objects
const person1 = new Person();
const person2 = new Person();
```

```
// access properties
console.log(person1.name);  // John
console.log(person2.name);  // John
```

In the above program, two objects are created using the same constructor function.

**JavaScript this Keyword**

In JavaScript, when this keyword is used in a constructor function, this refers to the object when the object is created. For example,

```
// constructor function
function Person () {
    this.name = 'John',
}

// create object
const person1 = new Person();

// access properties
console.log(person1.name);  // John
```

Hence, when an object accesses the properties, it can directly access the property as person1.name.

**JavaScript Constructor Function Parameters**

You can also create a constructor function with parameters. For example,

```
// constructor function
function Person (person_name, person_age, person_gender) {

   // assigning  parameter values to the calling object
    this.name = person_name,
    this.age = person_age,
    this.gender = person_gender,

    this.greet = function () {
        return ('Hi' + ' ' + this.name);
    }
}
```

```
// creating objects
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'female');

// accessing properties
console.log(person1.name); // "John"
console.log(person2.name); // "Sam"
```

In the above example, we have passed arguments to the constructor function during the creation of the object.

```
const person1 = new Person('John', 23, 'male');

const person2 = new Person('Sam', 25, 'male');
```

This allows each object to have different properties. As shown above,

```
console.log(person1.name); gives John

console.log(person2.name); gives Sam
```

**Create Objects: Constructor Function Vs Object Literal**

Object Literal is generally used to create a single object. The constructor function is useful if you want to create multiple objects. For example,

```
// using object literal
let person = {
   name: 'Sam'
}
// using constructor function
function Person () {
   this.name = 'Sam'
}

let person1 = new Person();
let person2 = new Person();
```

Each object created from the constructor function is unique. You can have the same properties as the constructor function or add a new property to one particular object. For example,

```
// using constructor function
function Person () {
   this.name = 'Sam'
}
```

```
let person1 = new Person();
let person2 = new Person();

// adding new property to person1
person1.age = 20;
```

Now this age property is unique to person1 object and is not available to person2 object.

However, if an object is created with an object literal, and if a variable is defined with that object value, any changes in variable value will change the original object. For example,

```
// using object lateral
let person = {
   name: 'Sam'
}

console.log(person.name); // Sam

let student = person;

// changes the property of an object
student.name = 'John';

// changes the origins object property
console.log(person.name); // John
```

When an object is created with an object literal, any object variable derived from that object will act as a clone of the original object. Hence, any change you make in one object will also reflect in the other object.

**Adding Properties And Methods in an Object**

You can add properties or methods in an object like this:

```
// constructor function
function Person () {
   this.name = 'John',
   this.age = 23
}

// creating objects
let person1 = new Person();
```

```
let person2 = new Person();

// adding property to person1 object
person1.gender = 'male';

// adding method to person1 object
person1.greet = function () {
   console.log('hello');
}

person1.greet();   // hello

// Error code
// person2 doesn't have greet() method
person2.greet();
```

**Output**

```
hello
Uncaught TypeError: person2.greet is not a function
```

In the above example, a new property gender and a new method greet() is added to the person1 object.

However, this new property and method is only added to person1. You cannot access gender or greet() from person2. Hence the program gives error when we try to access person2.greet();

**JavaScript Object Prototype**

You can also add properties and methods to a constructor function using a prototype. For example,

```
// constructor function
function Person () {
   this.name = 'John',
   this.age = 23
}

// creating objects
let person1 = new Person();
let person2 = new Person();

// adding new property to constructor function
```

***Person.prototype.gender = 'Male';***

***console.log(person1.gender); // Male***
***console.log(person2.gender); // Male***

**JavaScript Built-in Constructors**

JavaScript also has built-in constructors. Some of them are:
***let a = new Object();  // A new Object object***
***let b = new String();  // A new String object***
***let c = new Number();  // A new Number object***
***let d = new Boolean();  // A new Boolean object***

In JavaScript, strings can be created as objects by:
***const name = new String ('John');***
***console.log(name); // "John"***

In JavaScript, numbers can be created as objects by:
***const number = new Number (57);***
***console.log(number); // 57***

In JavaScript, booleans can be created as objects by:
***const count = new Boolean(true);***
***console.log(count); // true***

**Note**: It is recommended to use primitive data types and create them in a normal way, such as const name = 'John';, const number = 57; and const count = true;

You should not declare strings, numbers, and boolean values as objects because they slow down the program.

**Note**: In JavaScript, the keyword class was introduced in ES6 (ES2015) that also allows us to create objects. Classes are similar to constructor functions in JavaScript. To learn more, visit JavaScript Classes.

## *Lecture 4: JS Getter and Setter*

In JavaScript, there are two kinds of object properties:

- Data properties
- Accessor properties

Data Property

Here's an example of data property that we have been using in the previous lessons.

```
const student = {

   // data property
   firstName: 'Monica';
};
```

**Accessor Property**

In JavaScript, accessor properties are methods that get or set the value of an object. For that, we use these two keywords:

- get - to define a getter method to get the property value
- set - to define a setter method to set the property value

**JavaScript Getter**

In JavaScript, getter methods are used to access the properties of an object. For example,

```
const student = {

   // data property
   firstName: 'Monica',

   // accessor property(getter)
   get getName() {
      return this.firstName;
   }
};

// accessing data property
console.log(student.firstName); // Monica

// accessing getter methods
console.log(student.getName); // Monica

// trying to access as a method
console.log(student.getName()); // error
```

In the above program, a getter method getName() is created to access the property of an object.

```
get getName() {
   return this.firstName;
}
```

**Note**: To create a getter method, the get keyword is used.

And also when accessing the value, we access the value as a property.

***student.getName;***

When you try to access the value as a method, an error occurs.

***console.log(student.getName()); // error***

**JavaScript Setter**

In JavaScript, setter methods are used to change the values of an object. For example,

```
const student = {
   firstName: 'Monica',

   //accessor property(setter)
   set changeName(newName) {
      this.firstName = newName;
   }
};

console.log(student.firstName); // Monica

// change(set) object property using a setter
student.changeName = 'Sarah';

console.log(student.firstName); // Sarah
```

In the above example, the setter method is used to change the value of an object.

```
set changeName(newName) {
   this.firstName = newName;
}
```

**Note**: To create a setter method, the set keyword is used.

As shown in the above program, the value of firstName is Monica.

Then the value is changed to Sarah.

***student.changeName = 'Sarah';***

**Note**: Setter must have exactly one formal parameter.

**JavaScript Object.defineProperty()**

In JavaScript, you can also use Object.defineProperty() method to add getters and setters. For example,

```
const student = {
   firstName: 'Monica'
```

```
}

// getting property
Object.defineProperty(student, "getName", {
   get : function () {
      return this.firstName;
   }
});

// setting property
Object.defineProperty(student, "changeName", {
   set : function (value) {
      this.firstName = value;
   }
});

console.log(student.firstName); // Monica

// changing the property value
student.changeName = 'Sarah';

console.log(student.firstName); // Sarah
```

In the above example, Object.defineProperty() is used to access and change the property of an object.

The syntax for using Object.defineProperty() is:

***Object.defineProperty(obj, prop, descriptor)***

The Object.defineProperty() method takes three arguments.

- The first argument is the objectName.
- The second argument is the name of the property.
- The third argument is an object that describes the property.

## *Lecture 5: JS Prototype*

As you know, you can create an object in JavaScript using an object constructor function. For example,

```
// constructor function
function Person () {
   this.name = 'John',
   this.age = 23
}
```

```
// creating objects
const person1 = new Person();
const person2 = new Person();
```

In the above example, function Person() is an object constructor function. We have created two objects person1 and person2 from it.

**JavaScript Prototype**

In JavaScript, every function and object has a property named prototype by default. For example,

```
function Person () {
   this.name = 'John',
   this.age = 23
}

const person = new Person();

// checking the prototype value
console.log(Person.prototype); // { ... }
```

In the above example, we are trying to access the prototype property of a Person constructor function.

Since the prototype property has no value at the moment, it shows an empty object { ... }.

**Prototype Inheritance**

In JavaScript, a prototype can be used to add properties and methods to a constructor function. And objects inherit properties and methods from a prototype. For example,

```
// constructor function
function Person () {
   this.name = 'John',
   this.age = 23
}

// creating objects
const person1 = new Person();
const person2 = new Person();

// adding property to constructor function
```

```
Person.prototype.gender = 'male';

// prototype value of Person
console.log(Person.prototype);

// inheriting the property from prototype
console.log(person1.gender);
console.log(person2.gender);
```

**Output**

```
{ gender: "male" }
male
male
```

In the above program, we have added a new property gender to the Person constructor function using:

```
Person.prototype.gender = 'male';
```

Then object person1 and person2 inherits the property gender from the prototype property of Person constructor function.

Hence, both objects person1 and person2 can access the gender property.

**Note**: The syntax to add the property to an object constructor function is:

```
objectConstructorName.prototype.key = 'value';
```

Prototype is used to provide additional property to all the objects created from a constructor function.

**Add Methods to a Constructor Function Using Prototype**

You can also add new methods to a constructor function using prototype. For example,

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// creating objects
const person1 = new Person();
const person2 = new Person();
```

```
// adding a method to the constructor function
Person.prototype.greet = function() {
   console.log('hello' + ' ' +  this.name);
}

person1.greet(); // hello John
person2.greet(); // hello John
```

In the above program, a new method greet is added to the Person constructor function using a prototype.

**Changing Prototype**

If a prototype value is changed, then all the new objects will have the changed property value. All the previously created objects will have the previous value. For example,

```
// constructor function
function Person() {
   this.name = 'John'
}

// add a property
Person.prototype.age = 20;

// creating an object
const person1 = new Person();

console.log(person1.age); // 20

// changing the property value of prototype
Person.prototype = { age: 50 }

// creating new object
const person3 = new Person();

console.log(person3.age); // 50
console.log(person1.age); // 20
```

**Note**: You should not modify the prototypes of standard JavaScript built-in objects like strings, arrays, etc. It is considered a bad practice.

**JavaScript Prototype Chaining**

If an object tries to access the same property that is in the constructor function and the prototype object, the object takes the property from the constructor function. For example,

```
function Person() {
    this.name = 'John'
}

// adding property
Person.prototype.name = 'Peter';
Person.prototype.age = 23

const person1 = new Person();

console.log(person1.name); // John
console.log(person1.age); // 23
```

In the above program, a property name is declared in the constructor function and also in the prototype property of the constructor function.

When the program executes, person1.name looks in the constructor function to see if there is a property named name. Since the constructor function has the name property with value 'John', the object takes value from that property.

When the program executes, person1.age looks in the constructor function to see if there is a property named age. Since the constructor function doesn't have age property, the program looks into the prototype object of the constructor function and the object inherits property from the prototype object (if available).

**Note**: You can also access the prototype property of a constructor function from an object.

```
function Person () {
    this.name = 'John'
}
// adding a prototype
Person.prototype.age = 24;
// creating object
const person = new Person();
// accessing prototype property
console.log(person.__proto__);   // { age: 24 }
```

In the above example, a person object is used to access the prototype property using __proto__. However, __proto__ has been deprecated and you should avoid using it.

# Chapter 5: Types

## *Lecture 1: JS Arrays*

An array is an object that can store multiple values at once. For example,

***const words = ['hello', 'world', 'welcome'];***

Here, words is an array. The array is storing 3 values.

**Create an Array**

You can create an array using two ways:

**1. Using an array literal**

The easiest way to create an array is by using an array literal []. For example,

***const array1 = ["eat", "sleep"];***

**2. Using the new keyword**

You can also create an array using JavaScript's new keyword.

**const array2 = new Array("eat", "sleep");**

In both of the above examples, we have created an array having two elements.

**Note**: It is recommended to use array literal to create an array.

Here are more examples of arrays:

```
// empty array
const myList = [ ];

// array of numbers
const numberArray = [ 2, 4, 6, 8];

// array of strings
const stringArray = [ 'eat', 'work', 'sleep'];

// array with mixed data types
const newData = ['work', 'exercise', 1, true];
```

You can also store arrays, functions and other objects inside an array. For example,

```
const newData = [
   {'task1': 'exercise'},
   [1, 2 ,3],
   function hello() { console.log('hello')}
];
```

### Access Elements of an Array

You can access elements of an array using indices (0, 1, 2 ...). For example,

```
const myArray = ['h', 'e', 'l', 'l', 'o'];

// first element
console.log(myArray[0]);  // "h"

// second element
console.log(myArray[1]); // "e"
```


Note: Array's index starts with 0, not 1.

### Add an Element to an Array

You can use the built-in method push() and unshift() to add elements to an array.

The push() method adds an element at the end of the array. For example,

```
let dailyActivities = ['eat', 'sleep'];

// add an element at the end
dailyActivities.push('exercise');
```

***console.log(dailyActivities); //  ['eat', 'sleep', 'exercise']***

The unshift() method adds an element at the beginning of the array. For example,
***let dailyActivities = ['eat', 'sleep'];***

***//add an element at the start***
***dailyActivities.unshift('work');***

***console.log(dailyActivities); // ['work', 'eat', 'sleep']***

**Change the Elements of an Array**

You can also add elements or change the elements by accessing the index value.
***let dailyActivities = ['eat', 'sleep'];***

***// this will add the new element 'exercise' at the 2 index***
***dailyActivities[2] = 'exercise';***

***console.log(dailyActivities); // ['eat', 'sleep', 'exercise']***

Suppose, an array has two elements. If you try to add an element at index 3 (fourth element), the third element will be undefined. For example,
***let dailyActivities = ['eat', 'sleep'];***

***// this will add the new element 'exercise' at the 3 index***
***dailyActivities[3] = 'exercise';***

***console.log(dailyActivities); // ["eat", "sleep", undefined, "exercise"]***

Basically, if you try to add elements to high indices, the indices in between will have undefined value.

**Remove an Element from an Array**

You can use the pop() method to remove the last element from an array. The pop() method also returns the returned value. For example,
***let dailyActivities = ['work', 'eat', 'sleep', 'exercise'];***

***// remove the last element***
***dailyActivities.pop();***

```
console.log(dailyActivities); // ['work', 'eat', 'sleep']

// remove the last element from ['work', 'eat', 'sleep']
const removedElement = dailyActivities.pop();

//get removed element
console.log(removedElement); // 'sleep'
console.log(dailyActivities);  // ['work', 'eat']
```

If you need to remove the first element, you can use the shift() method. The shift() method removes the first element and also returns the removed element. For example,

```
let dailyActivities = ['work', 'eat', 'sleep'];

// remove the first element
dailyActivities.shift();

console.log(dailyActivities); // ['eat', 'sleep']
```

**Array length**

You can find the length of an element (the number of elements in an array) using the length property. For example,

```
const dailyActivities = ['eat', 'sleep'];

// this gives the total number of elements in an array
console.log(dailyActivities.length); // 2
```

**Array Methods**

In JavaScript, there are various array methods available that makes it easier to perform useful calculations.

Some of the commonly used JavaScript array methods are:

| Method | Description |
| --- | --- |
| concat() | joins two or more arrays and returns a result |
| indexOf() | searches an element of an array and returns its position |
| find() | returns the first value of an array element that passes a test |
| findIndex() | returns the first index of an array element that passes a test |
| forEach() | calls a function for each element |
| includes() | checks if an array contains a specified element |
| push() | aads a new element to the end of an array and returns the new length of an array |
| unshift() | adds a new element to the beginning of an array and returns the new length of an array |
| pop() | removes the last element of an array and returns the removed element |
| shift() | removes the first element of an array and returns the removed element |
| sort() | sorts the elements alphabetically in strings and in ascending order |
| slice() | selects the part of an array and returns the new array |
| splice() | removes or replaces existing elements and/or adds new elements |

Example: JavaScript Array Methods

```
let dailyActivities = ['sleep', 'work', 'exercise']
let newRoutine = ['eat'];

// sorting elements in the alphabetical order
dailyActivities.sort();
console.log(dailyActivities); // ['exercise', 'sleep', 'work']

//finding the index position of string
const position = dailyActivities.indexOf('work');
console.log(position); // 2
```

```
// slicing the array elements
const newDailyActivities = dailyActivities.slice(1);
console.log(newDailyActivities); // [ 'sleep', 'work']

// concatenating two arrays
const routine = dailyActivities.concat(newRoutine);
console.log(routine); // ["exercise", "sleep", "work", "eat"]
```

**Note**: If the element is not in an array, indexOf() gives -1.

**Working of JavaScript Arrays**

In JavaScript, an array is an object. And, the indices of arrays are objects keys.

Since arrays are objects, the array elements are stored by reference. Hence, when an array value is copied, any change in the copied array will also reflect in the original array. For example,

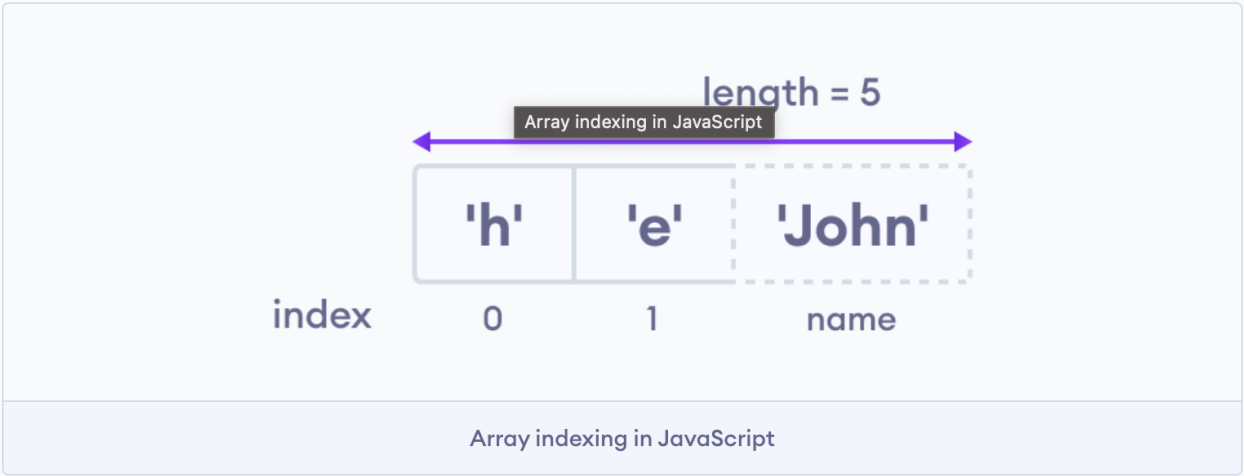
```
let arr = ['h', 'e'];
let arr1 = arr;
arr1.push('l');

console.log(arr); // ["h", "e", "l"]
console.log(arr1); // ["h", "e", "l"]
```

You can also store values by passing a named key in an array. For example,

```
let arr = ['h', 'e'];
arr.name = 'John';

console.log(arr); // ["h", "e"]
console.log(arr.name); // "John"
console.log(arr['name']); // "John"
```



Array indexing in JavaScript

However, it is not recommended to store values by passing arbitrary names in an array.

Hence in JavaScript, you should use an array if values are in ordered collection. Otherwise it's better to use object with { }.

## *Lecture 2: JS Multidimensional Array*

A multidimensional array is an array that contains another array. For example,

***// multidimensional array***

***const data = [[1, 2, 3], [1, 3, 4], [4, 5, 6]];***

**Create a Multidimensional Array**

Here is how you can create multidimensional arrays in JavaScript.

Example 1

***let studentsData = [['Jack', 24], ['Sara', 23], ['Peter', 24]];***

Example 2
***let student1 = ['Jack', 24];***
***let student2 = ['Sara', 23];***
***let student3 = ['Peter', 24];***

***// multidimensional array***

***let studentsData = [student1, student2, student3];***

Here, both example 1 and example 2 creates a multidimensional array with the same data.

### Access Elements of an Array

You can access the elements of a multidimensional array using indices (0, 1, 2 ...). For example,

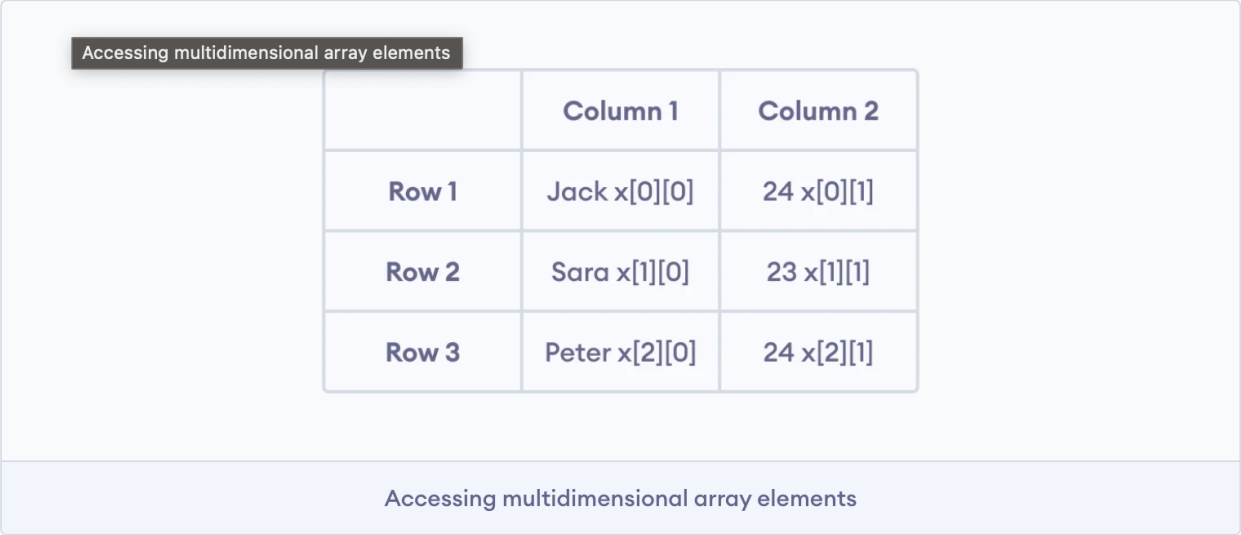
```
let x = [
['Jack', 24],
['Sara', 23],
['Peter', 24]
];

// access the first item
console.log(x[0]); // ["Jack", 24]

// access the first item of the first inner array
console.log(x[0][0]); // Jack

// access the second item of the third inner array
console.log(x[2][1]); // 24
```

You can think of a multidimensional array (in this case, x), as a table with 3 rows and 2 columns.




### Add an Element to a Multidimensional Array

You can use the Array's push() method or an indexing notation to add elements to a multidimensional array.

Adding Element to the Outer Array

```
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData.push(['Peter', 24]);

console.log(studentsData); //[["Jack", 24], ["Sara", 23], ["Peter", 24]
```

Adding Element to the Inner Array

```
// using index notation
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData[1][2] = 'hello';

console.log(studentsData); // [["Jack", 24], ["Sara", 23, "hello"]]


// using push()
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData[1].push('hello');

console.log(studentsData); // [["Jack", 24], ["Sara", 23, "hello"]]
```

You can also use the Array's splice() method to add an element at a specified index. For example,

```
let studentsData = [['Jack', 24], ['Sara', 23],];

// adding element at 1 index
studentsData.splice(1, 0, ['Peter', 24]);

console.log(studentsData); // [["Jack", 24], ["Peter", 24], ["Sara", 23]]
```

**Remove an Element from a Multidimensional Array**

You can use the Array's pop() method to remove the element from a multidimensional array. For example,

Remove Element from Outer Array

```
// remove the array element from outer array
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData.pop();
```

```
console.log(studentsData); // [["Jack", 24]]
```

Remove Element from Inner Array

```
// remove the element from the inner array
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData[1].pop();

console.log(studentsData); // [["Jack", 24], ["Sara"]]
```

You can also use the splice() method to remove an element at a specified index. For example,

```
let studentsData = [['Jack', 24], ['Sara', 23],];

// removing 1 index array item
studentsData.splice(1,1);
console.log(studentsData); // [["Jack", 24]]
```

**Iterating over Multidimensional Array**

You can iterate over a multidimensional array using the Array's forEach() method to iterate over the multidimensional array. For example,

```
let studentsData = [['Jack', 24], ['Sara', 23],];

// iterating over the studentsData
studentsData.forEach((student) => {
   student.forEach((data) => {
      console.log(data);
   });
});
```

Output

```
Jack
24
Sara
23
```

The first forEach() method is used to iterate over the outer array elements and the second forEach() is used to iterate over the inner array elements.

You can also use the for...of loop to iterate over the multidimensional array. For example,

```
let studentsData = [['Jack', 24], ['Sara', 23],];

for (let i of studentsData) {
  for (let j of i) {
    console.log(j);
  }
}
```

You can also use the for loop to iterate over a multidimensional array. For example,

```
let studentsData = [['Jack', 24], ['Sara', 23],];

// looping outer array elements
for(let i = 0; i < studentsData.length; i++){

    // get the length of the inner array elements
    let innerArrayLength = studentsData[i].length;

    // looping inner array elements
    for(let j = 0; j < innerArrayLength; j++) {
        console.log(studentsData[i][j]);
    }
}
```

## *Lecture 3: JS String*

JavaScript string is a primitive data type that is used to work with texts. For example,

```
const name = 'John';
```

**Create JavaScript Strings**

In JavaScript, strings are created by surrounding them with quotes. There are three ways you can use quotes.

```
Single quotes: 'Hello'
Double quotes: "Hello"
Backticks: `Hello`
```

For example,
```
//strings example
const name = 'Peter';
const name1 = "Jack";
const result = `The names are ${name} and ${name1}`;
```

Single quotes and double quotes are practically the same and you can use either of them.

Backticks are generally used when you need to include variables or expressions into a string. This is done by wrapping variables or expressions with ${variable or expression} as shown above.

You can also write a quote inside another quote. For example,
```
const name = 'My name is "Peter".';
```

However, the quote should not match the surrounding quotes. For example,
```
const name = 'My name is 'Peter'.'; // error
```

**Access String Characters**

You can access the characters in a string in two ways.

One way is to treat strings as an array. For example,
```
const a = 'hello';
console.log(a[1]); // "e"
```

Another way is to use the method charAt(). For example,
```
const a = 'hello';
console.log(a.charAt(1)); // "e"
```

**JavaScript Strings are immutable**

In JavaScript, strings are immutable. That means the characters of a string cannot be changed. For example,
```
let a = 'hello';
a[0] = 'H';
console.log(a); // "hello"
```

However, you can assign the variable name to a new string. For example,
```
let a = 'hello';
a = 'Hello';
console.log(a); // "Hello"
```

**JavaScript is Case-Sensitive**

JavaScript is case-sensitive. That means in JavaScript, the lowercase and uppercase letters are treated as different values. For example,

```
const a = 'a';
const b = 'A'
console.log(a === b); // false
```

In JavaScript, a and A are treated as different values.

**JavaScript Multiline Strings**

To use a multiline string, you can either use the + operator or the \ operator. For example,

```
// using the + operator
const message1 = 'This is a long message ' +
    'that spans across multiple lines' +
    'in the code.'

// using the \ operator
const message2 = 'This is a long message \
that spans across multiple lines \
in the code.'
```

**JavaScript String Length**

To find the length of a string, you can use built-in length property. For example,

```
const a = 'hello';
console.log(a.length); // 5
```

**JavaScript String Objects**

You can also create strings using the new keyword. For example,

```
const a = 'hello';
const b = new String('hello');

console.log(a); // "hello"
console.log(b); // "hello"

console.log(typeof a); // "string"
console.log(typeof b); // "object"
```

**Note**: It is recommended to avoid using string objects. Using string objects slows down the program.

### JavaScript String Methods

Here are the commonly used JavaScript String methods:

| Method | Description |
|---|---|
| charAt(index) | returns the character at the specified index |
| concat() | joins two or more strings |
| replace() | replaces a string with another string |
| split() | converts the string to an array of strings |
| substr(start, length) | returns a part of a string |
| substring(start,end) | returns a part of a string |
| slice(start, end) | returns a part of a string |
| toLowerCase() | returns the passed string in lower case |
| toUpperCase() | returns the passed string in upper case |
| trim() | removes whitespace from the strings |
| includes() | searches for a string and returns a boolean value |
| search() | searches for a string and returns a position of a match |

Example: JavaScript String Methods

```
const text1 = 'hello';
const text2 = 'world';
const text3 = '     JavaScript    ';

// concatenating two strings
const result1 = text1.concat(' ', text2);
```

```
console.log(result1); // "hello world"

// converting the text to uppercase
const result2 = text1.toUpperCase();
console.log(result2); // HELLO

// removing whitespace from the string
const result3 = text3.trim();
console.log(result3); // JavaScript

// converting the string to an array
const result4 = text1.split();
console.log(result4); // ["hello"]

// slicing the string
const result5= text1.slice(1, 3);
console.log(result5); // "el"
```

**JavaScript String() Function**

The String() function is used to convert various data types to strings. For example,

```
const a = 225; // number
const b = true; // boolean

//converting to string
const result1 = String(a);
const result2 = String(b);

console.log(result1); // "225"
console.log(result2); // "true"
```

**Escape Character**

You can use the backslash escape character \ to include special characters in a string. For example,

```
const name = 'My name is \'Peter\'.';
console.log(name);
```

Output

**My name is 'Peter'.**

In the above program, the same quote is included using \.

Here are other ways that you can use \:

| Code | Output |
| --- | --- |
| \" | include double quote |
| \\ | include backslash |
| \n | new line |
| \r | carriage return |
| \v | vertical tab |
| \t | horizontal tab |
| \b | backspace |
| \f | form feed |

## *Lecture 4: JS for...in loop*

There are also other types of loops. The for..in loop in JavaScript allows you to iterate over all property keys of an object.

**JavaScript for...in loop**

The syntax of the for...in loop is:

```
for (key in object) {
    // body of for...in
}
```

In each iteration of the loop, a key is assigned to the key variable. The loop continues for all object properties.

**Note**: Once you get keys, you can easily find their corresponding values.

Example 1: Iterate Through an Object

```
const student = {
   name: 'Monica',
   class: 7,
   age: 12
}

// using for...in
for ( let key in student ) {

   // display the properties
   console.log(`${key} => ${student[key]}`);
}
```

Output

```
name => Monica
class => 7
age => 12
```

In the above program, the for...in loop is used to iterate over the student object and print all its properties.

The object key is assigned to the variable key.

student[key] is used to access the value of key.

Example 2: Update Values of Properties

```
const salaries = {
   Jack : 24000,
   Paul : 34000,
   Monica : 55000
}

// using for...in
for ( let i in salaries) {

   // add a currency symbol
   let salary = "$" + salaries[i];
```

```
    // display the values
    console.log(`${i} : ${salary}`);
}
```

Output

```
Jack : $24000,
Paul : $34000,
Monica : $55000
```

In the above example, the for...in loop is used to iterate over the properties of the salaries object. Then, the string $ is added to each value of the object.

**for...in with Strings**

You can also use for...in loop to iterate over string values. For example,

```
const string = 'code';

// using for...in loop
for (let i in string) {
    console.log(string[i]);
}
```

Output

```
c
o
d
e
```

**for...in with Arrays**

You can also use for...in with arrays. For example,

```
// define array
const arr = [ 'hello', 1, 'JavaScript' ];

// using for...in loop
for (let x in arr) {
    console.log(arr[x]);
}
```

*Output*

```
hello
```

**Note**: You should not use for…in to iterate over an array where the index order is important.

One of the better ways to iterate over an array is using the for…of loop.

## *Lecture 5: JS Number*

In JavaScript, numbers are primitive data types. For example,
***const a = 3;***
***const b = 3.13;***

Unlike in some other programming languages, you don't have to specifically declare for integer or floating values using int, float, etc.

You can use exponential notation e to include too large or too small numbers. For example,
***const a1 = 5e9;***
***console.log(a1); //5000000000***

***const a2 = 5e-5;***
***console.log(a2); // 0.00005***

Numbers can also be denoted in hexadecimal notation. For example,
***const a = 0xff;***
***console.log(a); // 255***

***const b = 0x00 ;***
***console.log(b); // 0***

**+ Operator with Numbers**

When + is used with numbers, it is used to add the numbers. For example,
***const a = 4 + 9;***
***console.log(a); // 13***

When + is used with numbers and strings, it is used to concatenate them. For example,
***const a = '4' + 9;***

***console.log(a); // 49***

When a numeric string is used with other numeric operations, the numeric string is converted to a number. For example,
***const a = '4' - 2;***
***console.log(a); // 2***

***const a = '4' / 2;***
***console.log(a); // 2***

***const a = '4' * 2;***
***console.log(a); // 8***

**JavaScript NaN**

In JavaScript, NaN(Not a Number) is a keyword that indicates that the value is not a number.

Performing arithmetic operations (except + ) to numeric value with string results in NaN. For example,
***const a = 4 - 'hello';***
***console.log(a); // NaN***

The built-in function isNaN() can be used to find if a value is a number. For example,
***const a = isNaN(9);***
***console.log(a); // false***

***const a = isNaN(4 - 'hello');***
***console.log(a); // true***

When the typeof operator is used for NaN value, it gives a number output. For example,
***const a = 4 - 'hello';***
***console.log(a); // NaN***
***console.log(typeof a); // "number"***

**JavaScript Infinity**

In JavaScript, when calculation is done that exceeds the largest (or smallest) possible number, Infinity (or -Infinity) is returned. For example,
***const a = 2 / 0;***

***console.log(a); // Infinity***

***const a = -2 / 0;***
***console.log(a); // -Infinity***

**JavaScript BigInt**

In JavaScript, Number type can only represent numbers less than (253 - 1) and more than -(253 - 1). However, if you need to use a larger number than that, you can use the BigInt data type.

A BigInt number is created by appending n to the end of an integer. For example,
***// BigInt value***
***const value = 900719925124740998n;***

***// Adding two big integers***
***const value1 = value + 1n;***
***console.log(value1); // returns "900719925124740999n"***

**Note**: BigInt was introduced in the newer version of JavaScript and is not supported by many browsers.

**JavaScript Numbers Are Stored in 64-bit**

In JavaScript, numbers are stored in 64-bit format IEEE-754, also known as "double precision floating point numbers".

The numbers are stored in 64 bits (the number is stored in 0 to 51 bit positions, the exponent in 52 to 62 bit positions and the sign in 63 bit position).

| Numbers | Exponent | Sign |
| --- | --- | --- |
| 52 bits(0 - 51) | 11 bits(52- 62) | 1 bit(63) |

**Precision Problems**

Operations on floating-point numbers results in some unexpected results. For example,
***const a = 0.1 + 0.2;***

***console.log(a); // 0.30000000000000004***

The result should be 0.3 instead of 0.30000000000000004. This error occurs because in JavaScript, numbers are stored in binary form to represent decimal digits internally. And decimal numbers can't be represented in binary form exactly.

To solve the above problem, you can do something like this:
***const a = (0.1 * 10 + 0.2 * 10) / 10;***
***console.log(a); // 0.3***

You can also use the toFixed() method.
***const a = 0.1 + 0.2;***
***console.log(a.toFixed(2)); // 0.30***

toFixed(2) rounds up the decimal number to two decimal values.
***const a = 9999999999999999***
***console.log(a); // 10000000000000000***

**Note**: Integers are accurate up to 15 digits.

**Number Objects**

You can also create numbers using the new keyword. For example,
***const a = 45;***

***// creating a number object***
***const b = new Number(45);***

***console.log(a); // 45***
***console.log(b); // 45***

***console.log(typeof a); // "number"***
***console.log(typeof b); // "object"***

**Note**: It is recommended to avoid using number objects. Using number objects slows down the program.

**JavaScript Number Methods**

Here is a list of built-in number methods in JavaScript.

| Method | Description |
| --- | --- |
| isNaN() | determines whether the passed value is NaN |
| isFinite() | determines whether the passed value is a finite number |
| isInteger() | determines whether the passed value is an integer |
| isSafeInteger() | determines whether the passed value is a safe integer |
| parseFloat(string) | converts the numeric floating string to floating-point number |
| parseInt(string, [radix]) | converts the numeric string to integer |
| toExponential(fractionDigits) | returns a string value for a number in exponential notation |
| toFixed(digits) | returns a string value for a number in fixed-point notation |
| toPrecision() | returns a string value for a number to a specified precision |
| toString([radix]) | returns a string value in a specified radix(base) |
| valueof() | returns the numbers value |
| toLocaleString() | returns a string with a language sensitive representation of a number |

For example,

```
// check if a is integer
const a = 12;
console.log(Number.isInteger(a)); // true

// check if b is NaN
const b = NaN;
console.log(Number.isNaN(b)); // true

// display upto two decimal point
const d = 5.1234;
console.log(d.toFixed(2)); // 5.12
```

### JavaScript Number Properties

Here is a list of Number properties in JavaScript.

| Property | Description |
|---|---|
| EPSILON | returns the smallest interval between two representable numbers |
| MAX_SAFE_INTEGER | returns the maximum safe integer |
| MAX_VALUE | returns the largest possible value |
| MIN_SAFE_INTEGER | returns the minimum safe integer |
| MIN_VALUE | returns the smallest possible value |
| NaN | represents 'Not-a-Number' value |
| NEGATIVE_INFINITY | represents negative infinity |
| POSITIVE_INFINITY | represents positive infinity |
| prototype | allows the addition of properties to Number objects |

For example,

```
// largest possible value
const a = Number.MAX_VALUE;
console.log(a); // 1.7976931348623157e+308

// maximum safe integer
const a = Number.MAX_SAFE_INTEGER;
console.log(a); // 9007199254740991
```

### JavaScript Number() Function

The Number() function is used to convert various data types to numbers. For example,

```
const a = '23'; // string
const b = true; // boolean
```

```
//converting to number
const result1 = Number(a);
const result2 = Number(b);

console.log(result1); // 23
console.log(result2); // 1
```

## *Lecture 6: JS Symbol*

**JavaScript Symbol**

The JavaScript ES6 introduced a new primitive data type called Symbol. Symbols are immutable (cannot be changed) and are unique. For example,

```
// two symbols with the same description

const value1 = Symbol('hello');
const value2 = Symbol('hello');

console.log(value1 === value2); // false
```

Though value1 and value2 both contain the same description, they are different.

**Creating Symbol**

You use the Symbol() function to create a Symbol. For example,

```
// creating symbol
const x = Symbol()

typeof x; // symbol
```

You can pass an optional string as its description. For example,

```
const x = Symbol('hey');
console.log(x); // Symbol(hey)
```

**Access Symbol Description**

To access the description of a symbol, we use the . operator. For example,

```
const x = Symbol('hey');
console.log(x.description); // hey
```

**Add Symbol as an Object Key**

You can add symbols as a key in an object using square brackets []. For example,

```
let id = Symbol("id");

let person = {
   name: "Jack",

   // adding symbol as a key
   [id]: 123 // not "id": 123,
   greet: function(){},
};

console.log(person); // {name: "Jack", Symbol(id): 123}
```

Symbols are not included in for...in Loop

The for...in loop does not iterate over Symbolic properties. For example,

```
let id = Symbol("id");

let person = {
   name: "Jack",
   age: 25,
   [id]: 12
};

// using for...in
for (let key in person) {
   console.log(key);
}
```

Output

```
name
age
```

**Benefit of Using Symbols in Object**

If the same code snippet is used in various programs, then it is better to use Symbols in the object key. It's because you can use the same key name in different codes and avoid duplication issues. For example,

```
let person = {
    name: "Jack"
};

// creating Symbol
let id = Symbol("id");

// adding symbol as a key
person[id] = 12;
```

In the above program, if the person object is also used by another program, then you wouldn't want to add a property that can be accessed or changed by another program. Hence by using Symbol, you create a unique property that you can use.

Now, if the other program also needs to use a property named id, just add a Symbol named id and there won't be duplication issues. For example,

```
let person = {
    name: "Jack"
};

let id = Symbol("id");

person[id] = "Another value";
```

In the above program, even if the same name is used to store values, the Symbol data type will have a unique value.

In the above program, if the string key was used, then the later program would have changed the value of the property. For example,

```
let person = {
    name: "Jack"
};

// using string as key
```

```
person.id = 12;
console.log(person.id); // 12

// Another program overwrites value
person.id = 'Another value';
console.log(person.id); // Another value
```

In the above program, the second user.id overwrites the previous value.

### Symbol Methods

There are various methods available with Symbol.

| Method | Description |
|---|---|
| `for()` | Searches for existing symbols |
| `keyFor()` | Returns a shared symbol key from the global symbol registry. |
| `toSource()` | Returns a string containing the source of the Symbol object |
| `toString()` | Returns a string containing the description of the Symbol |
| `valueOf()` | Returns the primitive value of the Symbol object. |

Example: Symbol Methods

```
// get symbol by name
let sym = Symbol.for('hello');
let sym1 = Symbol.for('id');

// get name by symbol
console.log( Symbol.keyFor(sym) ); // hello
console.log( Symbol.keyFor(sym1) ); // id
```

### Symbol Properties

| Properties | Description |
| --- | --- |
| `asyncIterator` | Returns the default AsyncIterator for an object |
| `hasInstance` | Determines if a constructor object recognizes an object as its instance |
| `isConcatSpreadable` | Indicates if an object should be flattened to its array elements |
| `iterator` | Returns the default iterator for an object |
| `match` | Matches against a string |
| `matchAll` | Returns an iterator that yields matches of the regular expression against a string |
| `replace` | Replaces matched substrings of a string |
| `search` | Returns the index within a string that matches the regular expression |
| `split` | Splits a string at the indices that match a regular expression |
| `species` | Creates derived objects |
| `toPrimitive` | Converts an object to a primitive value |
| `toStringTag` | Gives the default description of an object |
| `description` | Returns a string containing the description of the symbol |

Example: Symbol Properties Example

```
const x = Symbol('hey');

// description property
console.log(x.description); // hey

const stringArray = ['a', 'b', 'c'];
const numberArray = [1, 2, 3];

// isConcatSpreadable property
```

```
numberArray[Symbol.isConcatSpreadable] = false;

let result = stringArray.concat(numberArray);
console.log(result); // ["a", "b", "c", [1, 2, 3]]
```

# Chapter 6: Exceptions & Modules

## *Lecture 1: JS try...catch...finally Statement*

The try, catch and finally blocks are used to handle exceptions (a type of an error). Before you learn about them, you need to know about the types of errors in programming.

**Types of Errors**

In programming, there can be two types of errors in the code:

**Syntax Error**: Error in the syntax. For example, if you write consol.log('your result');, the above program throws a syntax error. The spelling of console is a mistake in the above code.

**Runtime Error**: This type of error occurs during the execution of the program. For example,

calling an invalid function or a variable.

These errors that occur during runtime are called exceptions. Now, let's see how you can handle these exceptions.

**JavaScript try...catch Statement**

The try...catch statement is used to handle the exceptions. Its syntax is:

```
try {
   // body of try
}
catch(error) {
   // body of catch
}
```

The main code is inside the try block. While executing the try block, if any error occurs, it goes to the catch block. The catch block handles the errors as per the catch statements.

If no error occurs, the code inside the try block is executed and the catch block is skipped.

Example 1: Display Undeclared Variable

```
// program to show try...catch in a program

const numerator= 100, denominator = 'a';
```

```
try {
    console.log(numerator/denominator);

   // forgot to define variable a
   console.log(a);
}
catch(error) {
   console.log('An error caught');
   console.log('Error message: ' + error);
}
```

Output

```
NaN
An error caught
Error message: ReferenceError: a is not defined
```

In the above program, a variable is not defined. When you try to print the a variable, the program throws an error. That error is caught in the catch block.

**JavaScript try…catch…finally Statement**

You can also use the try…catch…finally statement to handle exceptions. The finally block executes both when the code runs successfully or if an error occurs.

The syntax of try…catch…finally block is:

```
try {
   // try_statements
}
catch(error) {
   // catch_statements
}
finally() {
   // codes that gets executed anyway
}
```

Example 2: try…catch…finally Example

```
const numerator= 100, denominator = 'a';

try {
    console.log(numerator/denominator);
```

```
    console.log(a);
}
catch(error) {
   console.log('An error caught');
   console.log('Error message: ' + error);
}
finally {
    console.log('Finally will execute every time');
}
```

Output

```
NaN
An error caught
Error message: ReferenceError: a is not defined
Finally will execute every time
```

In the above program, an error occurs and that error is caught by the catch block. The finally block will execute in any situation ( if the program runs successfully or if an error occurs).

**Note**: You need to use catch or finally statement after try statement. Otherwise, the program will throw an error Uncaught SyntaxError: Missing catch or finally after try.

**JavaScript try...catch in setTimeout**

The try...catch won't catch the exception if it happened in "timed" code, like in setTimeout(). For example,

```
try {
   setTimeout(function() {
      // error in the code
   }, 3000);
} catch (e) {
  console.log( "won't work" );
}
```

The above try...catch won't work because the engine has already left the try..catch construct and the function is executed later.

The try..catch block must be inside that function to catch an exception inside a timed function. For example,

```
setTimeout(function() {
```

```
    try {
        // error in the code
    } catch {
        console.log( "error is caught" );
    }
}, 3000);
```

You can also use the throw statement with the try...catch statement to use user-defined exceptions. For example, a certain number is divided by 0. If you want to consider Infinity as an error in the program, then you can throw a user-defined exception using the throw statement to handle that condition.

## *Lecture 2: JS throw Statement*

In the previous tutorial, you learned to handle exceptions using JavaScript try..catch statement. The try and catch statements handle exceptions in a standard way which is provided by JavaScript. However, you can use the throw statement to pass user-defined exceptions.

In JavaScript, the throw statement handles user-defined exceptions. For example, if a certain number is divided by 0, and if you need to consider Infinity as an exception, you can use the throw statement to handle that exception.

**JavaScript throw statement**

The syntax of throw statement is:
***throw expression;***

Here, expression specifies the value of the exception.

For example,
```
const number = 5;
throw number/0; // generate an exception when divided by 0
```

**Note**: The expression can be string, boolean, number, or object value.

**JavaScript throw with try...catch**

The syntax of try...catch...throw is:
```
try {
```

```
    // body of try
    throw exception;
}
catch(error) {
    // body of catch
}
```

**Note**: When the throw statement is executed, it exits out of the block and goes to the catch block. And the code below the throw statement is not executed.

Example 1: try...catch...throw Example

```
const number = 40;
try {
    if(number > 50) {
        console.log('Success');
    }
    else {
        // user-defined throw statement
        throw new Error('The number is low');
    }

    // if throw executes, the below code does not execute
    console.log('hello');
}
catch(error) {
    console.log('An error caught');
    console.log('Error message: ' + error);
}
```

Output

```
An error caught
Error message: Error: The number is low
```

In the above program, a condition is checked. If the number is less than 51, an error is thrown. And that error is thrown using the throw statement.

The throw statement specifies the string The number is low as an expression.

**Note**: You can also use other built-in error constructors for standard errors: TypeError, SyntaxError, ReferenceError, EvalError, InternalError, and RangeError.

For example,

```
throw new ReferenceError('this is reference error');
```

### Rethrow an Exception

You can also use throw statement inside the catch block to rethrow an exception. For example,

```
const number = 5;
try {
    // user-defined throw statement
    throw new Error('This is the throw');
}
catch(error) {
   console.log('An error caught');
   if( number + 8 > 10) {

      // statements to handle exceptions
       console.log('Error message: ' + error);
      console.log('Error resolved');
   }
   else {
      // cannot handle the exception
      // rethrow the exception
      throw new Error('The value is low');
   }
}
```

Output

```
An error caught
Error message: Error: This is the throw
Error resolved
```

In the above program, the throw statement is used within the try block to catch an exception. And the throw statement is rethrown in the catch block which gets executed if the catch block cannot handle the exception.

Here, the catch block handles the exception and no error occurs. Hence, the throw statement is not rethrown.

If the error was not handled by the catch block, the throw statement would be rethrown with error message ***Uncaught Error: The value is low***

## *Lecture 3: JS Modules*

As our program grows bigger, it may contain many lines of code. Instead of putting everything in a single file, you can use modules to separate codes in separate files as per their functionality. This makes our code organized and easier to maintain.

Module is a file that contains code to perform a specific task. A module may contain variables, functions, classes etc. Let's see an example,

Suppose, a file named greet.js contains the following code:

```
// exporting a function
export function greetPerson(name) {
    return `Hello ${name}`;
}
```

Now, to use the code of greet.js in another file, you can use the following code:

```
// importing greetPerson from greet.js file
import { greetPerson } from './greet.js';

// using greetPerson() defined in greet.js
let displayName = greetPerson('Jack');

console.log(displayName); // Hello Jack
```

Here,

The greetPerson() function in the greet.js is exported using the export keyword

```
export function greetPerson(name) {
    ...
}
```

Then, we imported greetPerson() in another file using the import keyword. To import functions, objects, etc., you need to wrap them around { }.

import { greet } from '/.greet.js';

**Note**: You can only access exported functions, objects, etc. from the module. You need to use the export keyword for the particular function, objects, etc. to import them and use them in other files.

**Export Multiple Objects**

It is also possible to export multiple objects from a module. For example,

In the file module.js

```
// exporting the variable
export const name = 'JavaScript Program';

// exporting the function
export function sum(x, y) {
    return x + y;
}
```

In main file,

```
import { name, sum } from './module.js';

console.log(name);
let add = sum(4, 9);
console.log(add); // 13
```

Here,

```
import { name, sum } from './module.js';
```

This imports both the name variable and the sum() function from the module.js file.

**Renaming imports and exports**

If the objects (variables, functions etc.) that you want to import are already present in your main file, the program may not behave as you want. In this case, the program takes value from the main file instead of the imported file.

To avoid naming conflicts, you can rename these functions, objects, etc. during the export or during the import .

1. Rename in the module (export file)

```
// renaming import inside module.js
export {
    function1 as newName1,
    function2 as newName2
};

// when you want to use the module
```

```
// import in the main file
import { newName1, newName2 } from './module.js';
```

Here, while exporting the function from module.js file, new names (here, newName1 & newName2 ) are given to the function. Hence, when importing that function, the new name is used to reference that function.

2. Rename in the import file

```
// inside module.js
export {
   function1,
   function2
};

// when you want to use the module
// import in the required file with different name

import { function1 as newName1, function2 as newName2 } from './module.js';
```

Here, while importing the function, the new names (here, newName1 & newName2 ) are used for the function name. Now you use the new names to reference these functions.

**Default Export**

You can also perform default export of the module. For example,

In the file greet.js:

```
// default export
export default function greet(name) {
   return `Hello ${name}`;
}

export const age = 23;
```

Then when importing, you can use:

```
import random_name from './greet.js';
```

While performing default export,

random_name is imported from greet.js. Since, random_name is not in greet.js, the default export (greet() in this case) is exported as random_name.

You can directly use the default export without enclosing curly brackets {}.

**Note**: A file can contain multiple exports. However, you can only have one default export in a file.

**Modules Always use Strict Mode**

By default, modules are in strict mode. For example,

```
// in greet.js
function greet() {
    // strict by default
}

export greet();
```

**Benefits of Using Module**

- The code base is easier to maintain because different code having different functionalities are in different files.
- Makes code reusable. You can define a module and use it numerous times as per your needs.

# Chapter 7: ES6

## *Lecture 1: JS ES6 Features*

JavaScript ES6 (also known as ECMAScript 2015 or ECMAScript 6) is the newer version of JavaScript that was introduced in 2015.

ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

### JavaScript let

JavaScript let is used to declare variables. Previously, variables were declared using the ***var*** keyword.

The variables declared using let are ***block-scoped***. This means they are only accessible within a particular block. For example,

```
// variable declared using let
let name = 'Sara';
{
   // can be accessed only inside
   let name = 'Peter';

   console.log(name); // Peter
}
console.log(name); // Sara
```

### JavaScript const

The const statement is used to declare constants in JavaScript. For example,

```
// name declared with const cannot be changed
const name = 'Sara';
```

Once declared, you cannot change the value of a const variable.

### JavaScript Arrow Function

In the ES6 version, you can use arrow functions to create function expressions. For example,

This function

```
// function expression
let x = function(x, y) {
   return x * y;
}
```

can be written as

```
// function expression using arrow function
let x = (x, y) => x * y;
```

**JavaScript Classes**

JavaScript class is used to create an object. Class is similar to a constructor function. For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

Keyword class is used to create a class. The properties are assigned in a constructor function.

Now you can create an object. For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const person1 = new Person('John');

console.log(person1.name); // John
```

**Default Parameter Values**

In the ES6 version, you can pass default values in the function parameters. For example,

```
function sum(x, y = 5) {
```

```
    // take sum
    // the value of y is 5 if not passed
    console.log(x + y);
}

sum(5); // 10
sum(5, 15); // 20
```

In the above example, if you don't pass the parameter for y, it will take 5 by default.

### JavaScript Template Literals

The template literal has made it easier to include variables inside a string. For example, before you had to do:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log('Hello ' + first_name + ' ' + last_name);
```

This can be achieved using template literal by:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log(`Hello ${first_name} ${last_name}`);
```

### JavaScript Destructuring

The destructuring syntax makes it easier to assign values to a new variable. For example,

```
// before you would do something like this
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let name = person.name;
let age = person.age;
let gender = person.gender;
```

```
console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

Using ES6 Destructuring syntax, the above code can be written as:

```
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}

let { name, age, gender } = person;
let { name, age } = person;


console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

**JavaScript import and export**

You could export a function or a program and use it in another program by importing it. This helps to make reusable components. For example, if you have two JavaScript files named contact.js and home.js.

In contact.js file, you can export the contact() function:

```
// export
export default function contact(name, age) {
   console.log(`The name is ${name}. And age is ${age}.`);
}
```

Then when you want to use the contact() function in another file, you can simply import the function. For example, in home.js file:

```
import contact from './contact.js';

contact('Sara', 25);
// The name is Sara. And age is 25
```

**JavaScript Promises**

Promises are used to handle asynchronous tasks. For example,

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
  reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
   function successValue(result) {
      console.log(result); // Promise resolved
   },
 )
```

**JavaScript Rest Parameter and Spread Operator**

You can use the rest parameter to represent an indefinite number of arguments as an array. For example,

```
function show(a, b, ...args) {
  console.log(a); // one
  console.log(b); // two
  console.log(args); // ["three", "four", "five", "six"]
}

show('one', 'two', 'three', 'four', 'five', 'six')
```

You pass the remaining arguments using ... syntax. Hence, the name rest parameter.

You use the spread syntax ... to copy the items into a single array. For example,

```
let arr1 = ['one', 'two'];
let arr2 = [...arr1, 'three', 'four', 'five'];
console.log(arr2); // ["one", "two", "three", "four", "five"]
```

Both the rest parameter and the spread operator use the same syntax. However, the spread operator is used with arrays (iterable values).

## *Lecture 2: JS Arrow Function*

Arrow function is one of the features introduced in the ES6 version of JavaScript. It allows you to create functions in a cleaner way compared to regular functions. For example,

This function
```
// function expression
let x = function(x, y) {
  return x * y;
}
```

can be written as
```
// using arrow functions
let x = (x, y) => x * y;
```

using an arrow function.

**Arrow Function Syntax**

The syntax of the arrow function is:
```
let myFunction = (arg1, arg2, ...argN) => {
   statement(s)
}
```

Here,

- myFunction is the name of the function
- arg1, arg2, ...argN are the function arguments
- statement(s) is the function body

If the body has single statement or expression, you can write arrow function as:
```
let myFunction = (arg1, arg2, ...argN) => expression
```

Example 1: Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,
```
let greet = () => console.log('Hello');
greet(); // Hello
```

Example 2: Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

Example 3: Arrow Function as an Expression

You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;

let welcome = (age < 18) ?
  () => console.log('Baby') :
  () => console.log('Adult');

welcome(); // Baby
```

Example 4: Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets {}. For example,

```
let sum = (a, b) => {
    let result = a + b;
    return result;
}

let result1 = sum(5,7);
console.log(result1); // 12
```

**this with Arrow Function**

Inside a regular function, this keyword refers to the function where it is called.

However, this is not associated with arrow functions. Arrow function does not have its own this. So whenever you call this, it refers to its parent scope. For example,

Inside a regular function

```
function Person() {
    this.name = 'Jack',
    this.age = 25,
    this.sayName = function () {
        // this is accessible
        console.log(this.age);
        function innerFunc() {
```

```
        // this refers to the global object
        console.log(this.age);
        console.log(this);
      }
      innerFunc();
   }
}

let x = new Person();
x.sayName();
```

Output

```
25
undefined
Window {}
```

Here, this.age inside this.sayName() is accessible because this.sayName() is the method of an object.

However, innerFunc() is a normal function and this.age is not accessible because this refers to the global object (Window object in the browser). Hence, this.age inside the innerFunc() function gives undefined.

Inside an arrow function

```
function Person() {
   this.name = 'Jack',
   this.age = 25,
   this.sayName = function () {
      console.log(this.age);
      let innerFunc = () => {
         console.log(this.age);
      }
      innerFunc();
   }
}

const x = new Person();
x.sayName();
```

Output
***25***
***25***

Here, the innerFunc() function is defined using the arrow function. And inside the arrow function, this refers to the parent's scope. Hence, this.age gives 25.

**Arguments Binding**

Regular functions have arguments binding. That's why when you pass arguments to a regular function, you can access them using the arguments keyword. For example,

```
let x = function () {
    console.log(arguments);
}
x(4,6,7); // Arguments [4, 6, 7]
```

Arrow functions do not have arguments binding.

When you try to access an argument using the arrow function, it will give an error. For example,

```
let x = () => {
    console.log(arguments);
}

x(4,6,7);
// ReferenceError: Can't find variable: arguments
```

To solve this issue, you can use the spread syntax. For example,

```
let x = (...n) => {
    console.log(n);
}

x(4,6,7); // [4, 6, 7]
```

**Arrow Function with Promises and Callbacks**

Arrow functions provide better syntax to write promises and callbacks. For example,

```
// ES5
asyncFunction().then(function() {
    return asyncFunction1();
```

```
}).then(function() {
    return asyncFunction2();
}).then(function() {
    finish;
});
```

can be written as

```
// ES6
asyncFunction()
.then(() => asyncFunction1())
.then(() => asyncFunction2())
.then(() => finish);
```

**Things You Should Avoid With Arrow Functions**

1. You should not use arrow functions to create methods inside objects.

```
let person = {
    name: 'Jack',
    age: 25,
    sayName: () => {

        // this refers to the global .....
        //
        console.log(this.age);
    }
}

person.sayName(); // undefined
```

2. You cannot use an arrow function as a constructor. For example,

```
let Foo = () => {};
let foo = new Foo(); // TypeError: Foo is not a constructor
```

## *Lecture 3: JS Default Parameters*

The concept of default parameters is a new feature introduced in the ES6 version of JavaScript. This allows us to give default values to function parameters. Let's take an example,

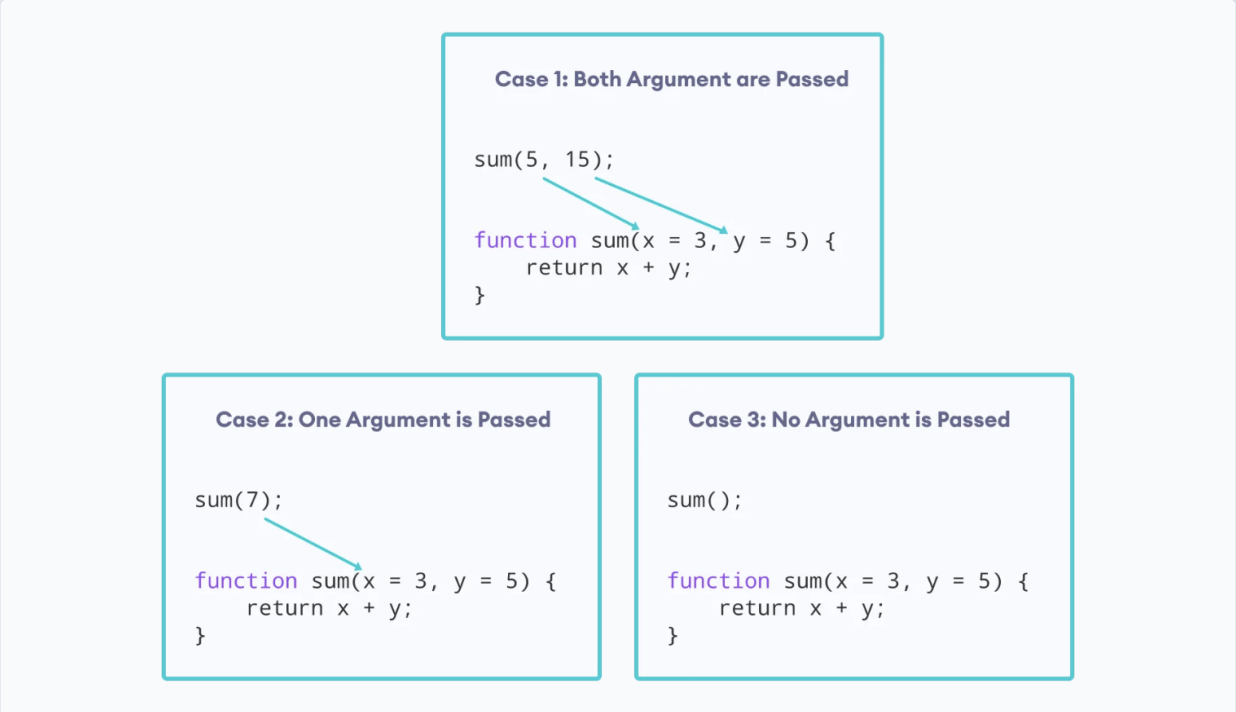
```
function sum(x = 3, y = 5) {
    // return sum
```

```
    return x + y;
}

console.log(sum(5, 15));  // 20
console.log(sum(7));      // 12
console.log(sum());       // 8
```

In the above example, the default value of x is 3 and the default value of y is 5.

- sum(5, 15) - When both arguments are passed, x takes 5 and y takes 15.
- sum(7) - When 7 is passed to the sum() function, x takes 7 and y takes default value 5.
- sum() - When no argument is passed to the sum() function, x takes default value 3 and y takes default value 5.




How default arguments work in JavaScript

**Using Expressions as Default Values**

It is also possible to provide expressions as default values.

Example 1: Passing Parameter as Default Values

```
function sum(x = 1, y = x,  z = x + y) {
    console.log( x + y + z );
}
```

***sum(); // 4***

In the above program,

- The default value of x is 1
- The default value of y is set to x parameter
- The default value of z is the sum of x and y

If you reference the parameter that has not been initialized yet, you will get an error. For example,

```
function sum( x = y, y = 1 ) {
   console.log( x + y);
}
sum();
```

Output

***ReferenceError: Cannot access 'y' before initialization***

Example 2: Passing Function Value as Default Value

```
// using a function in default value expression

const sum = () => 15;

const calculate = function( x, y = x * sum() ) {
   return x + y;
}

const result = calculate(10);
console.log(result);          // 160
```

In the above program,

- 10 is passed to the calculate() function.
- x becomes 10, and y becomes 150 (the sum function returns 15).
- The result will be 160.

**Passing undefined Value**

In JavaScript, when you pass undefined to a default parameter function, the function takes the default value. For example,

```
function test(x = 1) {
```

```
  console.log(x);
}

// passing undefined
// takes default value 1
test(undefined); // 1
```

## *Lecture 4: JS Template Literals*

Template literals (template strings) allow you to use strings or embedded expressions in the form of a string. They are enclosed in backticks ``. For example,

```
const name = 'Jack';
console.log(`Hello ${name}!`); // Hello Jack!
```

**Note**: Template literal was introduced in 2015 (also known as ECMAScript 6 or ES6 or ECMAScript 2015). Some browsers may not support the use of template literals.

### Template Literals for Strings

In the earlier versions of JavaScript, you would use a single quote '' or a double quote "" for strings. For example,

```
const str1 = 'This is a string';

// cannot use the same quotes
const str2 = 'A "quote" inside a string';  // valid code
const str3 = 'A 'quote' inside a string';  // Error

const str4 = "Another 'quote' inside a string"; // valid code
const str5 = "Another "quote" inside a string"; // Error
```

To use the same quotations inside the string, you can use the escape character \.

```
// escape characters using \
const str3 = 'A \'quote\' inside a string';  // valid code
const str5 = "Another \"quote\" inside a string"; // valid code
```

Instead of using escape characters, you can use template literals. For example,

```
const str1 = `This is a string`;
const str2 = `This is a string with a 'quote' in it`;
```

***const str3 = `This is a string with a "double quote" in it`;***

As you can see, the template literals not only make it easy to include quotations but also make our code look cleaner.

**Multiline Strings Using Template Literals**

Template literals also make it easy to write multiline strings. For example,

Using template literals, you can replace
***// using the + operator***
***const message1 = 'This is a long message\n' +***
***'that spans across multiple lines\n' +***
***'in the code.'***

***console.log(message1)***

with
***const message1 = `This is a long message***
***that spans across multiple lines***
***in the code.`***

***console.log(message1)***

The output of both these programs will be the same.
***This is a long message***
***that spans across multiple lines***
***in the code.***

**Expression Interpolation**

Before JavaScript ES6, you would use the + operator to concatenate variables and expressions in a string. For example,
***const name = 'Jack';***
***console.log('Hello ' + name); // Hello Jack***

With template literals, it's a bit easier to include variables and expressions inside a string. For that, we use the ${...} syntax.
***const name = 'Jack';***
***console.log(`Hello ${name}`);***

```
const result = 4 + 5;

// template literals used with expressions
console.log(`The sum of 4 + 5 is ${result}`);

console.log(`${result < 10 ? 'Too low': 'Very high'}`)
```

Output

```
Hello Jack
The sum of  4 + 5 is 9
Too low
```

The process of assigning variables and expressions inside the template literal is known as interpolation.

**Tagged Templates**

Normally, you would use a function to pass arguments. For example,

```
function tagExample(strings) {
    return strings;
}

// passing argument
const result = tagExample('Hello Jack');

console.log(result);
```

However, you can create tagged templates (that behave like a function) using template literals. You use tags that allow you to parse template literals with a function.

Tagged template is written like a function definition. However, you do not pass parentheses () when calling the literal. For example,

```
function tagExample(strings) {
    return strings;
}

// creating tagged template
const result = tagExample`Hello Jack`;
```

```
console.log(result);
```

Output

```
["Hello Jack"]
```

An array of string values are passed as the first argument of a tag function. You could also pass the values and expressions as the remaining arguments. For example,

```
const name = 'Jack';
const greet = true;
function tagExample(strings, nameValue) {
   let str0 = strings[0]; // Hello
   let str1 = strings[1]; // , How are you?
   if(greet) {
      return `${str0}${nameValue}${str1}`;
   }
}
// creating tagged literal
// passing argument name
const result = tagExample`Hello ${name}, How are you?`;

console.log(result);
```

Output

```
Hello Jack, How are you?
```

In this way, you can also pass multiple arguments in the tagged temaplate.

## *Lecture 5: JS Spread Operator*

The spread operator is a new addition to the features available in the JavaScript ES6 version.

### Spread Operator

The spread operator ... is used to expand or spread an iterable or an array. For example,

```
const arrValue = ['My', 'name', 'is', 'Jack'];
console.log(arrValue);   // ["My", "name", "is", "Jack"]
console.log(...arrValue); // My name is Jack
```

In this case, the code:

***console.log(...arrValue)***

is equivalent to:
***console.log('My', 'name', 'is', 'Jack');***

**Copy Array Using Spread Operator**

You can also use the spread syntax ... to copy the items into a single array. For example,

```
const arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];

console.log(arr2);
//  Output:
//  ["one", "two", "three", "four", "five"]
```

**Clone Array Using Spread Operator**

In JavaScript, objects are assigned by reference and not by values. For example,

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Here, both variables arr1 and arr2 are referring to the same array. Hence the change in one variable results in the change in both variables.

However, if you want to copy arrays so that they do not refer to the same array, you can use the spread operator. This way, the change in one array is not reflected in the other. For example,

```
let arr1 = [ 1, 2, 3];

// copy using spread syntax
let arr2 = [...arr1];
```

```
console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

**Spread Operator with Object**

You can also use the spread operator with object literals. For example,

```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };

// add members obj1 and obj2  to obj3
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Here, both obj1 and obj2 properties are added to obj3 using the spread operator.

**Rest Parameter**

When the spread operator is used as a parameter, it is known as the rest parameter.

You can also accept multiple arguments in a function call using the rest parameter. For example,

```
let func = function(...args) {
    console.log(args);
}

func(3); // [3]
func(4, 5, 6); // [4, 5, 6]
```

Here,

- When a single argument is passed to the func() function, the rest parameter takes only one parameter.
- When three arguments are passed, the rest parameter takes all three parameters.

**Note**: Using the rest parameter will pass the arguments as array elements.

You can also pass multiple arguments to a function using the spread operator. For example,

```
function sum(x, y ,z) {
   console.log(x + y + z);
}

const num1 = [1, 3, 4, 5];

sum(...num1); // 8
```

If you pass multiple arguments using the spread operator, the function takes the required arguments and ignores the rest.

## *Lecture 6: JS Map*

The JavaScript ES6 has introduced two new data structures, i.e Map and WeakMap.

Map is similar to objects in JavaScript that allows us to store elements in a key/value pair.

The elements in a Map are inserted in an insertion order. However, unlike an object, a map can contain objects, functions and other data types as key.

### Create JavaScript Map

To create a Map, we use the new Map() constructor. For example,

```
// create a Map
const map1 = new Map(); // an empty map
console.log(map1); // Map {}
```

### Insert Item to Map

After you create a map, you can use the set() method to insert elements to it. For example,

```
// create a set
```

```
let map1 = new Map();

// insert key-value pair
map1.set('info', {name: 'Jack', age: 26});
console.log(map1); // Map {"info" => {name: "Jack", age: 26}}
```

You can also use objects or functions as keys. For example,

```
// Map with object key
let map2 = new Map();

let obj = {};
map2.set(obj, {name: 'Jack', age: "26"});

console.log(map2); // Map {{} => {name: "Jack", age: "26"}}
```

**Access Map Elements**

You can access Map elements using the get() method. For example,

```
let map1 = new Map();
map1.set('info', {name: 'Jack', age: "26"});

// access the elements of a Map
console.log(map1.get('info')); // {name: "Jack", age: "26"}
```

**Check Map Elements**

You can use the has() method to check if the element is in a Map. For example,

```
const set1 = new Set([1, 2, 3]);

let map1 = new Map();
map1.set('info', {name: 'Jack', age: "26"});

// check if an element is in Set
console.log(map1.has('info')); // true
```

**Removing Elements**

You can use the clear() and the delete() method to remove elements from a Map.

The delete() method returns true if a specified key/value pair exists and has been removed or else returns false. For example,

```
let map1 = new Map();
map1.set('info', {name: 'Jack', age: "26"});

// removing a particular element
map1.delete('address'); // false
console.log(map1); // Map {"info" => {name: "Jack", age: "26"}}

map1.delete('info'); // true
console.log(map1); // Map {}
```

The clear() method removes all key/value pairs from a Map object. For example,

```
let map1 = new Map();
map1.set('info', {name: 'Jack', age: "26"});

// removing all element
map1.clear();
console.log(map1); // Map {}
```

### JavaScript Map Size

You can get the number of elements in a Map using the size property. For example,

```
let map1 = new Map();
map1.set('info', {name: 'Jack', age: "26"});

console.log(map1.size); // 1
```

### Iterate Through a Map

You can iterate through the Map elements using the for...of loop or forEach() method. The elements are accessed in the insertion order. For example,

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');
```

```
// looping through Map
for (let [key, value] of map1) {
    console.log(key + '- ' + value);
}
```

Output

```
name- Jack
age- 27
```

You could also get the same results as the above program using the forEach() method. For example,

```
// using forEach method()
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through Map
map1.forEach(function(value, key) {
  console.log(key + '- ' + value)
})
```

**Iterate Over Map Keys**

You can iterate over the Map and get the key using the keys() method. For example,

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through the Map
for (let key of map1.keys()) {
  console.log(key)
}
```

Output

```
name
age
```

**Iterate Over Map Values**

You can iterate over the Map and get the values using the values() method. For example,

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through the Map
for (let value of map1.values()) {
    console.log(value);
}
```

Output

```
Jack
27
```

**Get Key/Values of Map**

You can iterate over the Map and get the key/value of a Map using the entries() method. For example,

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');

// looping through the Map
for (let elem of map1.entries()) {
    console.log(`${elem[0]}: ${elem[1]}`);
}
```

Output

```
name: Jack
age: 27
```

**JavaScript Map vs Object**

| Map | Object |
| --- | --- |
| Maps can contain objects and other data types as keys. | Objects can only contain strings and symbols as keys. |
| Maps can be directly iterated and their value can be accessed. | Objects can be iterated by accessing its keys. |
| The number of elements of a Map can be determined by `size` property. | The number of elements of an object needs to be determined manually. |
| Map performs better for programs that require the addition or removal of elements frequently. | Object does not perform well if the program requires the addition or removal of elements frequently. |

**JavaScript WeakMap**

The WeakMap is similar to a Map. However, WeakMap can only contain objects as keys. For example,

```
const weakMap = new WeakMap();
console.log(weakMap); // WeakMap {}

let obj = {};

// adding object (element) to WeakMap
weakMap.set(obj, 'hello');

console.log(weakMap); // WeakMap {{} => "hello"}
```

When you try to add other data types besides objects, WeakMap throws an error. For example,

```
const weakMap = new WeakMap();

// adding string as a key to WeakMap
weakMap.set('obj', 'hello');
Run Code
// throws error
// TypeError: Attempted to set a non-object key in a WeakMap
```

**WeakMap Methods**

WeakMaps have methods get(), set(), delete(), and has(). For example,

```
const weakMap = new WeakMap();
console.log(weakMap); // WeakMap {}

let obj = {};

// adding object (element) to WeakMap
weakMap.set(obj, 'hello');

console.log(weakMap); // WeakMap {{} => "hello"}

// get the element of a WeakMap
console.log(weakMap.get(obj)); // hello

// check if an element is present in WeakMap
console.log(weakMap.has(obj)); // true

// delete the element of WeakMap
console.log(weakMap.delete(obj)); // true

console.log(weakMap); // WeakMap {}
```

**WeakMaps Are Not iterable**

Unlike Maps, WeakMaps are not iterable. For example,

```
const weakMap = new WeakMap();
console.log(weakMap); // WeakMap {}

let obj = {};

// adding object (element) to WeakMap
weakMap.set(obj, 'hello');


// looping through WeakMap
for (let i of weakMap) {
```

```
    console.log(i);  // TypeError
}
```

## *Lecture 7: JS Set*

The JavaScript ES6 has introduced two new data structures, i.e Set and WeakSet.

Set is similar to an array that allows us to store multiple items like numbers, strings, objects, etc. However, unlike an array, a set cannot contain duplicate values.

**Create JavaScript Set**

To create a Set, you need to use the new Set() constructor. For example,

```
// create Set
const set1 = new Set(); // an empty set
console.log(set1); // Set {}

// Set with multiple types of value
const set2 = new Set([1, 'hello', {count : true}]);
console.log(set2); // Set {1, "hello", {count: true}}
```

When duplicate values are passed to a Set object, the duplicate values are excluded.

```
// Set with duplicate values
const set3 = new Set([1, 1, 2, 2]);
console.log(set3); // Set {1, 2}
```

**Access Set Elements**

You can access Set elements using the values() method and check if there is an element inside Set using has() method. For example,

```
const set1 = new Set([1, 2, 3]);

// access the elements of a Set
console.log(set1.values()); // Set Iterator [1, 2, 3]
```

You can use the has() method to check if the element is in a Set. For example,

```
const set1 = new Set([1, 2, 3]);

// check if an element is in Set
console.log(set1.has(1));
```

**Adding New Elements**

You can add elements to a Set using the add() method. For example,

```
const set = new Set([1, 2]);
console.log(set.values());

// adding new elements
set.add(3);
console.log(set.values());

// adding duplicate elements
// does not add to Set
set.add(1);
console.log(set.values());
```

Output

```
Set Iterator [1, 2]
Set Iterator [1, 2, 3]
Set Iterator [1, 2, 3]
```

**Removing Elements**

You can use the clear() and the delete() method to remove elements from a Set.

The delete() method removes a specific element from a Set. For example,

```
const set = new Set([1, 2, 3]);
console.log(set.values()); // Set Iterator [1, 2, 3]

// removing a particular element
set.delete(2);
console.log(set.values()); // Set Iterator [1, 3]
```

The clear() method removes all elements from a Set. For example,

```
const set = new Set([1, 2, 3]);
```

```
console.log(set.values()); // Set Iterator [1, 2, 3]

// remove all elements of Set
set.clear();
console.log(set.values()); // Set Iterator []
```

### Iterate Sets

You can iterate through the Set elements using the for...of loop or forEach() method. The elements are accessed in the insertion order. For example,

```
const set = new Set([1, 2, 3]);

// looping through Set
for (let i of set) {
    console.log(i);
}
```

Output

```
1
2
3
```

### JavaScript WeakSet

The WeakSet is similar to a Set. However, WeakSet can only contain objects whereas a Set can contain any data types such as strings, numbers, objects, etc. For example,

```
const weakSet = new WeakSet();
console.log(weakSet); // WeakSet {}

let obj = {
    message: 'Hi',
    sendMessage: true
}

// adding object (element) to WeakSet
weakSet.add(obj);

console.log(weakSet); // WeakSet {{message: "Hi", sendMessage: true}}
```

When you try to add other data types besides objects, WeakSet throws an error. For example,

```
// trying to add string to WeakSet
weakSet.add('hello');

// throws error
// TypeError: Attempted to add a non-object key to a WeakSet
console.log(weakSet);
```

**WeakSet Methods**

WeakSets have methods add(), delete(), and has(). For example,

```
const weakSet = new WeakSet();
console.log(weakSet); // WeakSet {}

const obj = {a:1};

// add to a weakSet
weakSet.add(obj);
console.log(weakSet); // WeakSet {{a: 1}}

// check if an element is in Set
console.log(weakSet.has(obj)); // true

// delete elements
weakSet.delete(obj);
console.log(weakSet); // WeakSet {}
```

**WeakSets Are Not iterable**

Unlike Sets, WeakSets are not iterable. For example,

```
const weakSet = new WeakSet({a:1});

// looping through WeakSet
for (let i of weakSet) {

    // TypeError
    console.log(i);
```

```
}
```

**Mathematical Set Operations**

In JavaScript, Set does not provide built-in methods for performing mathematical operations such as union, intersection, difference, etc. However, we can create programs to perform those operations.

Example: Set Union Operation

```
// perform union operation
// contain elements of both sets
function union(a, b) {
    let unionSet = new Set(a);
    for (let i of b) {
        unionSet.add(i);
    }
    return unionSet
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['grapes', 'apple', 'banana']);

let result = union(setA, setB);

console.log(result);
```

Output

```
Set {"apple", "mango", "orange", "grapes", "banana"}
```

Example: Set Intersection Operation

```
// perform intersection operation
// elements of set a that are also in set b
function intersection(setA, setB) {
    let intersectionSet = new Set();

    for (let i of setB) {
        if (setA.has(i)) {
            intersectionSet.add(i);
```

```
        }
    }
    return intersectionSet;
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['grapes', 'apple', 'banana']);

let result = intersection(setA, setB);

console.log(result);
```

Output

```
Set {"apple"}
```

Example: Set Difference Operation

```
// perform difference operation
// elements of set a that are not in set b
function difference(setA, setB) {
    let differenceSet = new Set(setA)
    for (let i of setB) {
        if(!differenceSet.delete(i)) {
            differenceSet.add(i)
        }
    }
    return differenceSet
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['grapes', 'apple', 'banana']);

let result = difference(setA, setB);

console.log(result);
```

Output

```
Set {"mango", "orange"}
```

Example: Set Subset Operation

```
// perform subset operation
// true if all elements of set b is in set a
function subset(setA, setB) {
    for (let i of setB) {
        if (!setA.has(i)) {
            return false
        }
    }
    return true
}

// two sets of fruits
let setA = new Set(['apple', 'mango', 'orange']);
let setB = new Set(['apple', 'orange']);

let result = subset(setA, setB);

console.log(result);
```

Output

```
true
```

## *Lecture 8: JS Destructuring Assignment*

**JavaScript Destructuring**

The destructuring assignment introduced in ES6 makes it easy to assign array values and object properties to distinct variables. For example,

Before ES6:

```
// assigning object attributes to variables
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let name = person.name;
```

```
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

From ES6:

```
// assigning object attributes to variables
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}

// destructuring assignment
let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

**Note**: The order of the name does not matter in object destructuring.

For example, you could write the above program as:

```
let { age, gender, name } = person;
console.log(name); // Sara
```

**Note**: When destructuring objects, you should use the same name for the variable as the corresponding object key.

For example,

```
let {name1, age, gender} = person;
console.log(name1); // undefined
```

If you want to assign different variable names for the object key, you can use:

```
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}
```

```
// destructuring assignment
// using different variable names
let { name: name1, age: age1, gender:gender1 } = person;

console.log(name1); // Sara
console.log(age1); // 25
console.log(gender1); // female
```

**Array Destructuring**

You can also perform array destructuring in a similar way. For example,

```
const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, y, z] = arrValue;

console.log(x); // one
console.log(y); // two
console.log(z); // three
```

**Assign Default Values**

You can assign the default values for variables while using destructuring. For example,

```
let arrValue = [10];

// assigning default value 5 and 7
let [x = 5,  y = 7] = arrValue;

console.log(x); // 10
console.log(y); // 7
```

In the above program, arrValue has only one element. Hence,

- the x variable will be 10
- the y variable takes the default value 7

In object destructuring, you can pass default values in a similar way. For example,

```
const person = {
```

```
    name: 'Jack',
}

// assign default value 26 to age if undefined
const { name, age = 26} = person;

console.log(name); // Jack
console.log(age); // 26
```

**Swapping Variables**

In this example, two variables are swapped using the destructuring assignment syntax.

```
// program to swap variables
let x = 4;
let y = 7;

// swapping variables vbmn,./
L
:P{
[x, y] = [y, x];

console.log(x); // 7
console.log(y); // 4
```

**Skip Items**

You can skip unwanted items in an array without assigning them to local variables. For example,

```
const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, , z] = arrValue;

console.log(x); // one
console.log(z); // three
```

In the above program, the second element is omitted by using the comma separator ,.

**Assign Remaining Elements to a Single Variable**

You can assign the remaining elements of an array to a variable using the spread syntax .... For example,

```
const arrValue = ['one', 'two', 'three', 'four'];

// destructuring assignment in arrays
// assigning remaining elements to y
const [x, ...y] = arrValue;

console.log(x); // one
console.log(y); // ["two", "three", "four"]
```

Here, one is assigned to the x variable. And the rest of the array elements are assigned to y variable.

You can also assign the rest of the object properties to a single variable. For example,

```
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}

// destructuring assignment
// assigning remaining properties to rest
let { name, ...rest } = person;

console.log(name); // Sara
console.log(rest); // {age: 25, gender: "female"}
```

**Note**: The variable with the spread syntax cannot have a trailing comma ,. You should use this rest element (variable with spread syntax) as the last variable.

For example,

```
const arrValue = ['one', 'two', 'three', 'four'];

// throws an error
const [ ...x, y] = arrValue;
```

```
console.log(x); // eror
```

**Nested Destructuring Assignment**

You can perform nested destructuring for array elements. For example,

```
// nested array elements
const arrValue = ['one', ['two', 'three']];

// nested destructuring assignment in arrays
const [x, [y, z]] = arrValue;

console.log(x); // one
console.log(y); // two
console.log(z); // three
```

Here, the variable y and z are assigned nested elements two and three.

In order to execute the nested destructuring assignment, you have to enclose the variables in an array structure (by enclosing inside []).

You can also perform nested destructuring for object properties. For example,

```
const person = {
   name: 'Jack',
   age: 26,
   hobbies: {
      read: true,
      playGame: true
   }
}
// nested destructuring
const {name, hobbies: {read, playGame}} = person;

console.log(name); // Jack
console.log(read); // true
console.log(playGame); // true
```

In order to execute the nested destructuring assignment for objects, you have to enclose the variables in an object structure (by enclosing inside {}).

## *Lecture 9: JS Classes*

Classes are one of the features introduced in the ES6 version of JavaScript.

A class is a blueprint for the object. You can create an object from the class.

You can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, you build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

**Creating JavaScript Class**

JavaScript class is similar to the Javascript constructor function, and it is merely a syntactic sugar.

The constructor function is defined as:

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// create an object
const person1 = new Person();
```

Instead of using the function keyword, you use the class keyword for creating JS classes. For example,

```
// creating a class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

The class keyword is used to create a class. The properties are assigned in a constructor function.

Now you can create an object. For example,

```
// creating a class
class Person {
  constructor(name) {
    this.name = name;
  }
}

// creating an object
const person1 = new Person('John');
const person2 = new Person('Jack');

console.log(person1.name); // John
console.log(person2.name); // Jack
```

Here, person1 and person2 are objects of Person class.

**Note**: The constructor() method inside a class gets called automatically each time an object is created.

**Javascript Class Methods**

While using constructor function, you define methods as:

```
// constructor function
function Person (name) {

   // assigning  parameter values to the calling object
    this.name = name;

    // defining method
    this.greet = function () {
        return ('Hello' + ' ' + this.name);
    }
}
```

It is easy to define methods in the JavaScript class. You simply give the name of the method followed by (). For example,

```
class Person {
    constructor(name) {
    this.name = name;
```

```
  }

    // defining method
    greet() {
        console.log(`Hello ${this.name}`);
    }
}

let person1 = new Person('John');

// accessing property
console.log(person1.name); // John

// accessing method
person1.greet(); // Hello John
```

**Note**: To access the method of an object, you need to call the method using its name followed by ().

### Getters and Setters

In JavaScript, getter methods get the value of an object and setter methods set the value of an object.

JavaScript classes may include getters and setters. You use the get keyword for getter methods and set for setter methods. For example,

```
class Person {
    constructor(name) {
        this.name = name;
    }

    // getter
    get personName() {
        return this.name;
    }

    // setter
    set personName(x) {
        this.name = x;
```

```
    }
}

let person1 = new Person('Jack');
console.log(person1.name); // Jack

// changing the value of name property
person1.personName = 'Sarah';
console.log(person1.name); // Sarah
console.log(person1.personName); // Sarah
```

**Hoisting**

A class should be defined before using it. Unlike functions and other JavaScript declarations, the class is not hoisted. For example,

```
// accessing class
const p = new Person(); // ReferenceError

// defining class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

As you can see, accessing a class before defining it throws an error.

**'use strict'**

Classes always follow 'use-strict'. All the code inside the class is automatically in strict mode. For example,

```
class Person {
  constructor() {
    a = 0;
    this.name = a;
```

```
  }
}
```

***let p = new Person(); // ReferenceError: Can't find variable: a***

**Note**: JavaScript class is a special type of function. And the typeof operator returns function for a class.

For example,

```
class Person {}
console.log(typeof Person); // function
```

## *Lecture 10: JS Inheritance*

**Class Inheritance**

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.

Using class inheritance, a class can inherit all the methods and properties of another class.

Inheritance is a useful feature that allows code reusability.

To use class inheritance, you use the extends keyword. For example,

```
// parent class
class Person {
   constructor(name) {
      this.name = name;
   }

   greet() {
      console.log(`Hello ${this.name}`);
   }
}

// inheriting parent class
class Student extends Person {

}
```

```
let student1 = new Student('Jack');
student1.greet();
```

Output

```
Hello Jack
```

In the above example, the Student class inherits all the methods and properties of the Person class. Hence, the Student class will now have the name property and the greet() method.

Then, we accessed the greet() method of Student class by creating a student1 object.

**JavaScript super() keyword**

The super keyword used inside a child class denotes its parent class. For example,

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        console.log(`Hello ${this.name}`);
    }
}

// inheriting parent class
class Student extends Person {

    constructor(name) {

        console.log("Creating student class");

        // call the super class constructor and pass in the name parameter
        super(name);
    }

}

let student1 = new Student('Jack');
```

```
student1.greet();
```

Here, super inside Student class refers to the Person class. Hence, when the constructor of Student class is called, it also calls the constructor of the Person class which assigns a name property to it.

**Overriding Method or Property**

If a child class has the same method or property name as that of the parent class, it will use the method and property of the child class. This concept is called method overriding. For example,

```
// parent class
class Person {
  constructor(name) {
    this.name = name;
    this.occupation = "unemployed";
  }

  greet() {
    console.log(`Hello ${this.name}.`);
  }

}

// inheriting parent class
class Student extends Person {

  constructor(name) {

    // call the super class constructor and pass in the name parameter
    super(name);

    // Overriding an occupation property
    this.occupation = 'Student';
  }

  // overriding Person's method
  greet() {
    console.log(`Hello student ${this.name}.`);
```

```
        console.log('occupation: ' + this.occupation);
    }
}

let p = new Student('Jack');
p.greet();
```

Output
***Hello student Jack.***
***occupation: Student***

Here, the occupation property and the greet() method are present in parent Person class and the child Student class. Hence, the Student class overrides the occupation property and the greet() method.

**Uses of Inheritance**

- Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
- Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
- Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

## *Lecture 11: JS for...of*

In JavaScript, there are three ways we can use a for loop.

- JavaScript for loop
- JavaScript for...in loop
- JavaScript for...of loop

The for...of loop was introduced in the later versions of JavaScript ES6.

The for..of loop in JavaScript allows you to iterate over iterable objects (arrays, sets, maps, strings etc).

**JavaScript for...of loop**

The syntax of the for...of loop is:

```
for (element of iterable) {
   // body of for...of
}
```

Here,

- iterable - an iterable object (array, set, strings, etc).
- element - items in the iterable

In plain English, you can read the above code as: for every element in the iterable, run the body of the loop.

### for...of with Arrays

The for..of loop can be used to iterate over an array. For example,

```
// array
const students = ['John', 'Sara', 'Jack'];

// using for...of
for ( let element of students ) {

   // display the values
   console.log(element);
}
```

Output

```
John
Sara
Jack
```

In the above program, the for...of loop is used to iterate over the students array object and display all its values.

### for...of with Strings

You can use for...of loop to iterate over string values. For example,

```
// string
const string = 'code';
```

```
// using for...of loop
for (let i of string) {
    console.log(i);
}
```

Output

```
c
o
d
e
```

**for...of with Sets**

You can iterate through Set elements using the for...of loop. For example,

```
// define Set
const set = new Set([1, 2, 3]);

// looping through Set
for (let i of set) {
    console.log(i);
}
```

Output

```
1
2
3
```

**for...of with Maps**

You can iterate through Map elements using the for...of loop. For example,

```
// define Map
let map = new Map();

// inserting elements
map.set('name', 'Jack');
map.set('age', '27');
```

```
// looping through Map
for (let [key, value] of map) {
    console.log(key + '- ' + value);
}
```

Output

```
name- Jack
age- 27
```

**User Defined Iterators**

You can create an iterator manually and use the for...of loop to iterate through the iterators. For example,

```
// creating iterable object
const iterableObj = {

    // iterator method
    [Symbol.iterator]() {
        let step = 0;
        return {
            next() {
                step++;
                if (step === 1) {
                    return { value: '1', done: false};
                 }
                else if (step === 2) {
                    return { value: '2', done: false};
                }
                else if (step === 3) {
                    return { value: '3', done: false};
                }
                return { value: '', done: true };
            }
        }
    }
}

// iterating using for...of
```

```
for (const i of iterableObj) {
 console.log(i);
}
```

Output

```
1
2
3
```

**for...of with Generators**

Since generators are iterables, you can implement an iterator in an easier way. Then you can iterate through the generators using the for...of loop. For example,

```
// generator function
function* generatorFunc() {
   yield 10;
   yield 20;
   yield 30;
}

const obj = generatorFunc();

// iteration through generator
for (let value of obj) {
   console.log(value);
}
Run Code
Output
10
20
30
```

**for...of Vs for...in**

| for...of | for...in |
| --- | --- |
| The `for...of` loop is used to iterate through the values of an iterable. | The `for...in` loop is used to iterate through the keys of an object. |
| The `for...of` loop cannot be used to iterate over an object. | You can use `for...in` to iterate over an iterable such arrays and strings but you should avoid using `for...in` for iterables. |

## *Lecture 12: JS Proxies*

In JavaScript, proxies (proxy objects) are used to wrap an object and redefine various operations into the object such as reading, insertion, validation, etc. Proxy allows you to add custom behavior to an object or a function.

**Creating a Proxy Object**

The syntax of proxy is:
***new Proxy(target, handler);***

Here,

- new Proxy() - the constructor.
- target - the object/function which you want to proxy
- handler - can redefine the custom behavior of the object

For example,

```
let student1 = {
   age: 24,
   name: "Felix"
}

const handler = {
   get: function(obj, prop) {
      return obj[prop] ? obj[prop] : 'property does not exist';
```

```
    }
}

const proxy = new Proxy(student1, handler);
console.log(proxy.name); // Felix
console.log(proxy.age); // 24
console.log(proxy.class); // property does not exist
```

Here, the get() method is used to access the object's property value. And if the property is not available in the object, it returns property does not exist.

As you can see, you can use a proxy to create new operations for the object. A case may arise when you want to check if an object has a particular key and perform an action based on that key. In such cases, proxies can be used.

You can also pass an empty handler. When an empty handler is passed, the proxy behaves as an original object. For example,

```
let student = {
    name: 'Jack',
    age: 24
}

const handler = { };

// passing empty handler
const proxy1 = new Proxy(student, {});

console.log(proxy1); // Proxy {name: "Jack", age: 24}
console.log(proxy1.name); // Jack
```

### Proxy handlers

Proxy provides two handler methods get() and set().

### get() handler

The get() method is used to access the properties of a target object. For example,

```
let student = {
    name: 'Jack',
```

```
    age: 24
}

const handler = {

    // get the object key and value
    get(obj, prop) {

        return obj[prop];
  }
}

const proxy = new Proxy(student, handler);
console.log(proxy.name); // Jack
```

Here, the get() method takes the object and the property as its parameters.

**set() handler**

The set() method is used to set the value of an object. For example,

```
let student = {
    name: 'John'
}

let setNewValue = {
  set: function(obj, prop, value) {

    obj[prop] = value;
    return;
  }
};

// setting new proxy
let person = new Proxy(student, setNewValue);

// setting new key/value
person.age = 25;
console.log(person); // Proxy {name: "John", age: 25}
```

Here, a new property age is added to the student object.

**Uses of Proxy**

**1. For Validation**

You can use a proxy for validation. You can check the value of a key and perform an action based on that value.

For example,

```
let student = {
   name: 'Jack',
   age: 24
}

const handler = {

   // get the object key and value
   get(obj, prop) {

   // check condition
   if (prop == 'name') {
     return obj[prop];
   } else {
     return 'Not allowed';
   }
  }
}

const proxy = new Proxy(student, handler);
console.log(proxy.name); // Jack
console.log(proxy.age); // Not allowed
```

Here, only the name property of the student object is accessible. Else, it returns Not allowed.

**2. Read Only View of an Object**

There may be times when you do not want to let others make changes in an object. In such cases, you can use a proxy to make an object readable only. For example,

```
let student = {
   name: 'Jack',
   age: 23
}

const handler = {
   set: function (obj, prop, value) {
      if (obj[prop]) {

         // cannot change the student value
         console.log('Read only')
      }
   }
};

const proxy = new Proxy(student, handler);

proxy.name = 'John'; // Read only
proxy.age = 33; // Read only
```

In the above program, one cannot mutate the object in any way.

If one tries to mutate the object in any way, you'll only receive a string saying Read Only.

**3. Side Effects**

You can use a proxy to call another function when a condition is met. For example,

```
const myFunction = () => {
   console.log("execute this function")
};

const handler = {
   set: function (target, prop, value) {
      if (prop === 'name' && value === 'Jack') {
         // calling another function
         myFunction();
```

```
        }
        else {
            console.log('Can only access name property');
        }
    }
};

const proxy = new Proxy({}, handler);

proxy.name = 'Jack'; // execute this function
proxy.age = 33; // Can only access name property
```

# Chapter 8: Asynchronous

## *Lecture 1: JS setTimeout()*

The setTimeout() method executes a block of code after the specified time. The method executes the code only once.

The commonly used syntax of JavaScript setTimeout is:
***setTimeout(function, milliseconds);***

Its parameters are:

- function - a function containing a block of code
- milliseconds - the time after which the function is executed

The setTimeout() method returns an intervalID, which is a positive integer.

Example 1: Display a Text Once After 3 Second
```
// program to display a text using setTimeout method
function greet() {
   console.log('Hello world');
}

setTimeout(greet, 3000);
console.log('This message is shown first');
```

Output
***This message is shown first***
***Hello world***

In the above program, the setTimeout() method calls the greet() function after 3000 milliseconds (3 second).

Hence, the program displays the text Hello world only once after 3 seconds.

**Note**: The setTimeout() method is useful when you want to execute a block of once after some time. For example, showing a message to a user after the specified time.

The setTimeout() method returns the interval id. For example,
```
// program to display a text using setTimeout method
function greet() {
   console.log('Hello world');
}
```

```
let intervalId = setTimeout(greet, 3000);
console.log(`Id: ' + intervalId);
```

Output

```
Id: 3
Hello world
```

Example 2: Display Time Every 3 Second

```
// program to display time every 3 seconds
function showTime() {

    // return new date and time
    let dateTime= new Date();

    // returns the current local time
    let time = dateTime.toLocaleTimeString();

    console.log(time)

    // display the time after 3 seconds
     setTimeout(showTime, 3000);
}

// calling the function
showTime();
```

Output

```
5:45:39 PM
5:45:43 PM
5:45:47 PM
5:45:50 PM
```

.....

The above program displays the time every 3 seconds.

The setTimeout() method calls the function only once after the time interval (here 3 seconds).

However, in the above program, since the function is calling itself, the program displays the time every 3 seconds.

This program runs indefinitely (until the memory runs out).

**Note**: If you need to execute a function multiple times, it's better to use the setInterval() method.

**JavaScript clearTimeout()**

As you have seen in the above example, the program executes a block of code after the specified time interval. If you want to stop this function call, you can use the clearTimeout() method.

The syntax of clearTimeout() method is:
***clearTimeout(intervalID);***

Here, the intervalID is the return value of the setTimeout() method.

Example 3: Use clearTimeout() Method

```
// program to stop the setTimeout() method

let count = 0;

// function creation
function increaseCount(){

    // increasing the count by 1
    count += 1;
    console.log(count)
}

let id = setTimeout(increaseCount, 3000);

// clearTimeout
clearTimeout(id);
console.log('setTimeout is stopped.');
```

Output

```
setTimeout is stopped.
```

In the above program, the setTimeout() method is used to increase the value of count after 3 seconds. However, the clearTimeout() method stops the function call of the setTimeout() method. Hence, the count value is not increased.

**Note**: You generally use the clearTimeout() method when you need to cancel the setTimeout() method call before it happens.

You can also pass additional arguments to the setTimeout() method. The syntax is:
***setTimeout(function, milliseconds, parameter1, ....paramenterN);***

When you pass additional parameters to the setTimeout() method, these parameters (parameter1, parameter2, etc.) will be passed to the specified function.

For example,

```
// program to display a name
function greet(name, lastName) {
    console.log('Hello' + ' ' + name + ' ' + lastName);
}

// passing argument to setTimeout
setTimeout(greet, 1000, 'John', 'Doe');
```

Output
***Hello John Doe***

In the above program, two parameters John and Doe are passed to the setTimeout() method. These two parameters are the arguments that will be passed to the function (here, greet() function) that is defined inside the setTimeout() method.

## *Lecture 2: JS CallBack Function*

...

## *Lecture 3: JS Promise*

...

## *Lecture 4: JS async/await*

...

## *Lecture 5: JS setInterval()*

...

# Chapter 9: Miscellaneous

### *Lecture 1: JS JSON*

...

### *Lecture 2: JS Date and Time*

...

### *Lecture 3: Closure*

...

### *Lecture 4: this*

...

### *Lecture 5: 'use strict'*

...

### *Lecture 6: Iterators and Iterables*

...

### *Lecture 7: Generators*

...

### *Lecture 8: Regular Expressions*

...

### *Lecture 9: Browser Debugging*

...

### *Lecture 10: Uses of JavaScript*

...