Javascript

Most important basics, concepts, methods, functions, and more

JavaScript Basics Cheat Sheet

Getting Started: Introduction

JavaScript is a high-level, interpreted programming language primarily used to add interactivity to websites.

- **Key Features:**
  - Dynamically typed
  - Prototype-based object-oriented
  - First-class functions

To include JavaScript in an HTML file, use the `<script>` tag:

```
<script>
  // Your JavaScript code here
</script>
```

Alternatively, link an external JavaScript file:

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

Basic Syntax:

- Statements are terminated with semicolons (`;`), though they are often optional.
- JavaScript is case-sensitive.
- Comments are denoted by `//` for single-line and `/* ... */` for multi-line comments.

Console

The `console` object provides access to the browser's debugging console.

- `console.log(message)`: Prints a message to the console.

```
console.log("Hello, world!");
```

- `console.error(message)`: Prints an error message to the console.

```
console.error("An error occurred!");
```

- `console.warn(message)`: Prints a warning message to the console.

```
console.warn("This is a warning!");
```

- `console.table(data)`: Displays tabular data as a table.

```
const data = [{name: 'John', age: 30},
{name: 'Jane', age: 25}];
console.table(data);
```

- `console.time(label)` and `console.timeEnd(label)`: Starts and stops a timer to measure execution time.

```
console.time('myTimer');
// Some code to measure
console.timeEnd('myTimer');
```

Comments

Comments are used to explain code and are ignored by the JavaScript interpreter.

- Single-line comments start with `//`.
- Multi-line comments are enclosed between `/*` and `*/`.

```
// This is a single-line comment

/*
This is a multi-line comment.
It can span multiple lines.
*/
```

Good commenting practices:

- Explain the purpose of the code.
- Describe complex algorithms.
- Document parameters and return values of functions.

Numbers

JavaScript has a single number type which can represent both integers and floating-point numbers.

```
let integer = 10;
let decimal = 3.14;
```

`parseInt(string, radix)`

Converts a string to an integer. `radix` specifies the base (e.g., 10 for decimal).

```
parseInt("42", 10); // Returns 42
parseInt("0x1A", 16); // Returns 26
```

`parseFloat(string)`

Converts a string to a floating-point number.

```
parseFloat("3.14"); // Returns 3.14
```

Special Number Values

- `NaN`: Not-a-Number, represents an invalid number.
- `Infinity`: Represents infinity.
- `-Infinity`: Represents negative infinity.

Number Methods

- `toFixed(digits)`: Formats a number using fixed-point notation.
- `toPrecision(precision)`: Formats a number to a specified precision.

```
let num = 3.14159;
num.toFixed(2); // Returns "3.14"
num.toPrecision(3); // Returns "3.14"
```

Variables

Variables are used to store data values.

- **Declaration:** Use `var`, `let`, or `const` keywords.
- **Naming:** Must start with a letter, underscore, or dollar sign. Can contain letters, numbers, underscores, and dollar signs. Case-sensitive.

```
var x;
let y;
const z = 10;
```

`var`

Function-scoped or globally-scoped if declared outside a function. Can be re-declared and re-assigned.

```
var x = 5;
var x = 10; // Allowed
x = 12; //Allowed
```

`let`

Block-scoped. Cannot be re-declared within the same scope, but can be re-assigned.

```
let y = 5;
//let y = 10; // Not allowed
y = 12; //Allowed
```

`const`

Block-scoped. Cannot be re-declared or re-assigned. Must be initialized upon declaration.

```
const z = 5;
//const z = 10; // Not allowed
//z = 12; //Not allowed
```

Strings

Strings are sequences of characters enclosed in single quotes (`'`) or double quotes (`"`).

```
let str1 = 'Hello';
let str2 = "World";
```

String Length

Use `.length` property to determine the length of a string.

```
let str = "Hello";
console.log(str.length); // Outputs 5
```

String Methods

- `charAt(index)`: Returns the character at the specified index.
- `substring(startIndex, endIndex)`: Extracts a part of a string.
- `toUpperCase()`: Converts a string to uppercase.
- `toLowerCase()`: Converts a string to lowercase.
- `indexOf(searchValue)`: Returns the index of the first occurrence of a value in a string.
- `replace(searchValue, newValue)`: Replaces a specified value with another value.

String Concatenation

Use the `+` operator to concatenate strings.

```
let str1 = "Hello";
let str2 = "World";
let result = str1 + " " + str2; //
Returns "Hello World"
```

Arithmetic Operators

JavaScript supports standard arithmetic operators:

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `%`: Modulus (remainder)
- `**`: Exponentiation

```
let a = 10;
let b = 5;

console.log(a + b);   // Outputs 15
console.log(a - b);   // Outputs 5
console.log(a * b);   // Outputs 50
console.log(a / b);   // Outputs 2
console.log(a % b);   // Outputs 0
console.log(a ** b);  // Outputs 100000
```

Increment and Decrement Operators:

- `++`: Increment
- `--`: Decrement

```
let x = 5;
x++; // x is now 6
x--; // x is now 5
```

Assignment Operators

Assignment operators assign values to variables.

- `=` : Assigns the value on the right to the variable on the left.

```
let x = 10;
```

- `+=` : Adds the right operand to the left operand and assigns the result to the left operand.

```
let x = 5;
x += 3;  // x is now 8 (x = x + 3)
```

- `-=` : Subtracts the right operand from the left operand and assigns the result to the left operand.

```
let x = 5;
x -= 3;  // x is now 2 (x = x - 3)
```

- `*=` : Multiplies the left operand by the right operand and assigns the result to the left operand.

```
let x = 5;
x *= 3;  // x is now 15 (x = x * 3)
```

- `/=` : Divides the left operand by the right operand and assigns the result to the left operand.

```
let x = 6;
x /= 3;  // x is now 2 (x = x / 3)
```

- `%=` : Calculates the modulus of the left operand divided by the right operand and assigns the result to the left operand.

```
let x = 5;
x %= 3;  // x is now 2 (x = x % 3)
```

String Interpolation

String interpolation allows you to embed expressions directly within string literals, using template literals (backticks `` ` ``).

```
let name = "John";
let greeting = `Hello, ${name}!`; //
greeting is "Hello, John!"
```

Basic Usage

Use `${expression}` to insert the value of an expression into a string.

```
let age = 30;
let message = `You are ${age} years
old.`;
```

Expression Evaluation

Expressions inside `${}` are evaluated at runtime.

```
let a = 5;
let b = 10;
let result = `The sum of ${a} and ${b}
is ${a + b}.`;
```

Multiline Strings

Template literals can span multiple lines without needing concatenation or escape characters.

```
let multiline = `This is a
multiline string.`;
```

`let` Keyword

The `let` keyword declares a block-scoped local variable.

- `let` allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used.

```
function example() {
  let x = 10;
  if (true) {
    let x = 20;  // Different variable
than the outer x
    console.log(x); // Outputs 20
  }
  console.log(x);   // Outputs 10
}
```

- Cannot be re-declared within the same scope.
- Can be re-assigned.

```
let y = 5;
//let y = 10; // Not allowed
y = 12; // Allowed
```

Temporal Dead Zone (TDZ):
Variables declared with `let` are not accessible before they are declared in the code. This is known as the Temporal Dead Zone.

`const` Keyword

The `const` keyword declares a block-scoped constant variable.

- `const` is used to declare variables whose values should not be re-assigned after initialization.
- Constants must be initialized when they are declared.

```
const PI = 3.14159;
// PI = 3.14; //Not allowed, will throw
an error
```

- Block-scoped, similar to `let`.
- Cannot be re-declared or re-assigned.

```
const z = 5;
//const z = 10; // Not allowed
//z = 12; //Not allowed
```

Note: `const` does not mean the value is immutable. For objects and arrays, the variable cannot be re-assigned, but the properties or elements can be modified.

```
const obj = {name: 'John'};
obj.name = 'Jane'; // Allowed
console.log(obj.name); // Outputs 'Jane'
```

JavaScript Conditionals

if Statement

The `if` statement executes a block of code if a specified condition is true.

```
if (condition) {
  // Code to execute if the condition is true
}
```

Example:

```
let age = 20;
if (age >= 18) {
  console.log("You are an adult.");
}
```

The condition is a Boolean expression that evaluates to `true` or `false`.

If the condition is `true`, the code within the curly braces `{}` is executed.

If the condition is `false`, the code block is skipped.

You can also nest `if` statements inside other `if` statements.

```
if (condition1) {
  if (condition2) {
    // Code to execute if both conditions are true
  }
}
```

Ternary Operator

The ternary operator is a shorthand way of writing an `if...else` statement. It is also known as the conditional operator.

```
condition ? expressionIfTrue : expressionIfFalse;
```

Example:

```
let age = 20;
let status = (age >= 18) ? "Adult" : "Minor";
console.log(status); // Output: Adult
```

If `condition` is `true`, `expressionIfTrue` is executed; otherwise, `expressionIfFalse` is executed.

The ternary operator can be useful for assigning values based on a condition.

It is often used for simple, concise conditional logic.

Avoid using nested ternary operators, as they can become difficult to read and understand.

Operators

Comparison Operators:	Used to compare values and return a Boolean result.
`==`	Equal to (value only)
`===`	Strict equal to (value and type)
`!=`	Not equal to (value only)
`!==`	Strict not equal to (value and type)
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

else if Statement

The `else if` statement allows you to check multiple conditions in sequence.

```
if (condition1) {
  // Code to execute if condition1 is true
} else if (condition2) {
  // Code to execute if condition1 is false and condition2 is true
} else {
  // Code to execute if both conditions are false
}
```

Example:

```
let score = 75;
if (score >= 90) {
  console.log("A");
} else if (score >= 80) {
  console.log("B");
} else if (score >= 70) {
  console.log("C");
} else {
  console.log("D");
}
```

You can have multiple `else if` blocks to check various conditions.

The `else` block is optional and executes if none of the preceding conditions are true.

JavaScript executes the first `if` or `else if` block where the condition evaluates to `true` and skips the rest.

switch Statement

The `switch` statement executes different blocks of code based on the value of a single expression.

```
switch (expression) {
  case value1:
    // Code to execute if expression === value1
    break;
  case value2:
    // Code to execute if expression === value2
    break;
  default:
    // Code to execute if expression doesn't match any case
}
```

Example:

```
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("It's Monday!");
    break;
  case "Tuesday":
    console.log("It's Tuesday!");
    break;
  default:
    console.log("It's another day.");
}
```

The `expression` is evaluated once and its value is compared to each `case`.

If a `case` matches the `expression`, the code within that `case` is executed.

The `break` statement is crucial to prevent fall-through to the next `case`. If omitted, the code will continue to execute the next `case` even if it doesn't match.

The `default` case is executed if no other `case` matches the `expression`.

You can group multiple `case` statements together to execute the same code block.

== vs ===

`==` (Equality Operator)

Checks for equality of values after performing type coercion (if necessary).

Example:

```
5 == "5"  // true (string "5" is converted to number 5)
0 == false // true (false is converted to number 0)
null == undefined // true
```

`===` (Strict Equality Operator)

Checks for equality of both value and type without type coercion.

Example:

```
5 === "5"  // false (number and string are different types)
0 === false // false (number and boolean are different types)
null === undefined // false
```

Recommendation:

Always use `===` and `!==` to avoid unexpected behavior due to type coercion. This leads to more predictable and maintainable code.

JavaScript Functions

Function Declaration

A function declaration defines a named function.

```
function functionName(parameters) {
  // code to be executed
}
```

Example:

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

Function declarations are hoisted, meaning they can be called before they are defined in the code.

The `function` keyword is followed by the function name, a list of parameters in parentheses, and the function body in curly braces.

Function Expressions

A function expression defines a function inside an expression.

```
const functionName =
function(parameters) {
  // code to be executed
};
```

Example:

```
const multiply = function(a, b) {
  return a * b;
};
```

Function expressions are not hoisted, so they must be defined before they are called.

The function can be anonymous (as shown above) or have a name (named function expression).

Anonymous Functions

An anonymous function is a function without a name. They are often used in function expressions or as arguments to other functions.

```
function(parameters) {
  // code to be executed
}
```

Example:

```
setTimeout(function() {
  console.log('Delayed message');
}, 1000);
```

Anonymous functions are commonly used as callback functions.

They can be immediately invoked using the IIFE (Immediately Invoked Function Expression) pattern.

Arrow Functions (ES6)

Arrow functions provide a concise syntax for writing function expressions. Introduced in ES6.

```
(parameters) => expression

// or

(parameters) => {
  // code to be executed
  return expression;
}
```

Example:

```
const square = (x) => x * x;

const add = (a, b) => {
  return a + b;
};
```

If there is only one parameter, the parentheses can be omitted: `x => x * x`.

If the function body is a single expression, the curly braces and `return` keyword can be omitted.

Arrow functions do not have their own `this` context; they inherit it from the surrounding scope (lexical `this`).

Return Keyword

The `return` keyword specifies the value to be returned from a function.

```
function functionName(parameters) {
  // code to be executed
  return value;
}
```

Example:

```
function getFullName(firstName,
lastName) {
  return firstName + ' ' + lastName;
}
```

If the `return` keyword is not used, the function returns `undefined` by default.

The `return` keyword also terminates the execution of the function.

Calling Functions

Functions are called by using their name followed by parentheses, optionally including arguments.

```
functionName(arguments);
```

Example:

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

greet('Alice'); // Output: Hello, Alice!
```

Functions can be called directly or assigned to variables.

If a function expects arguments but none are provided, the missing arguments will have a value of `undefined` inside the function.

If more arguments are passed than expected, the extra arguments are ignored.

Function Parameters

Function parameters are the names listed in the function's definition.
Arguments are the actual values passed to the function when it is called.

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

functionName(argument1, argument2);
```

Example:

```
function divide(a, b) {
  return a / b;
}

let result = divide(10, 2); // a = 10, b
= 2
```

Parameters are local variables within the function's scope.

ES6 allows default parameter values:

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

greet(); // Output: Hello, Guest!
greet('Bob'); // Output: Hello, Bob!
```

Rest parameters allow a function to accept an indefinite number of arguments as an array:

```
function sum(...numbers) {
  return numbers.reduce((acc, num) =>
acc + num, 0);
}

console.log(sum(1, 2, 3, 4)); // Output:
10
```

JavaScript Scope

Scope Basics

Scope determines the accessibility of variables. In JavaScript, scope can be global or local.

- **Global Scope:** Variables declared outside of any function have global scope and are accessible from anywhere in your code.
- **Local Scope:** Variables declared within a function have local scope and are only accessible within that function.

JavaScript has function scope: Each function creates a new scope. Variables defined inside a function are not accessible from outside the function.

Example:

```
let globalVar = 'Global';

function myFunction() {
  let localVar = 'Local';
  console.log(globalVar); // Output: Global
  console.log(localVar);  // Output: Local
}

myFunction();
console.log(globalVar); // Output: Global
//console.log(localVar);  // Error: localVar is not defined
```

Block Scoped Variables (let and const)

`let` and `const` are block-scoped, meaning they are only accessible within the block they are defined in (a block is code within `{}`).

```
function exampleBlockScope() {
  if (true) {
    let x = 10;
    const y = 20;
    var z = 30; // var is function-scoped
  }
  // x and y are not accessible here
  console.log(z); // Output: 30 (because var is function-scoped)
}

exampleBlockScope();
```

Using `let` prevents variable hoisting within the block, reducing potential errors.

Global Variables

When you declare a variable outside any function, it becomes a global variable. Global variables can be accessed and modified from anywhere in your code.

```
var globalVar = 'I am global';

function anotherFunction() {
  console.log(globalVar); // Accessible here
}

anotherFunction();
```

Important Note: Avoid using global variables excessively, as they can lead to naming conflicts and make code harder to maintain and debug.

Accidental Global Variables: If you assign a value to a variable that has not been declared, JavaScript will automatically declare it as a global variable (in non-strict mode). Always declare variables using `var`, `let`, or `const`.

let vs var

`var`	Function-scoped or globally-scoped. Can be re-declared and updated. Hoisted to the top of its scope and initialized with `undefined`.
`let`	Block-scoped. Can be updated but not re-declared within its scope. Hoisted but not initialized, resulting in a `ReferenceError` if accessed before declaration.

Example:

```
function varLetExample() {
  var x = 1;
  let y = 2;
  if (true) {
    var x = 3;  // same x!
    let y = 4;  // different y
    console.log("Block x:", x); // Output: Block x: 3
    console.log("Block y:", y); // Output: Block y: 4
  }
  console.log("Function x:", x); // Output: Function x: 3
  console.log("Function y:", y); // Output: Function y: 2
}

varLetExample();
```

Loops with closures

A common pitfall involves using `var` in loops with closures, as `var`'s function scope can cause unexpected behavior. Use `let` to capture the variable's value for each iteration.

Incorrect (using `var`):

```
for (var i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i); // Output: 5, 5, 5, 5, 5
  }, 100);
}
```

In this case, by the time the `setTimeout` functions execute, the loop has already completed, and `i` is 5.

Correct (using `let`):

```
for (let i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i); // Output: 0, 1, 2, 3, 4
  }, 100);
}
```

`let` creates a new binding for `i` in each iteration, so the closure captures the correct value.

Alternatively, using an IIFE (Immediately Invoked Function Expression) with `var`:

```
for (var i = 0; i < 5; i++) {
  (function(j) {
    setTimeout(function() {
      console.log(j); // Output: 0, 1, 2, 3, 4
    }, 100);
  })(i);
}
```

Scope Chain

When a variable is used, JavaScript engine tries to find the variable in the current scope. If it's not found, it looks in the outer scope, and continues up the chain of scopes until it reaches the global scope. This is called the scope chain.

```
var globalVar = "Global";

function outerFunction() {
  var outerVar = "Outer";

  function innerFunction() {
    var innerVar = "Inner";
    console.log(innerVar);   // Output: Inner
    console.log(outerVar);   // Output: Outer
    console.log(globalVar);  // Output: Global
  }

  innerFunction();
}

outerFunction();
```

In the `innerFunction`, JavaScript looks for `innerVar` first in its own scope, then `outerVar` in the `outerFunction`'s scope, and finally `globalVar` in the global scope.

Lexical Scope

JavaScript uses lexical scoping (also known as static scoping). The scope of a variable is determined by its position in the source code.

```
var value = 1;

function foo() {
  console.log(value);
}

function bar() {
  var value = 2;
  foo(); // Output: 1 (lexical scope)
}

bar();
```

`foo` is defined in the global scope, so when it logs `value`, it looks for `value` in the global scope, not in the scope where it is called (`bar`).

JavaScript Arrays

Arrays Basics

Arrays in JavaScript are ordered lists of values. They can hold values of any data type, including numbers, strings, booleans, objects, and even other arrays.

Arrays are dynamic, meaning their size can grow or shrink as needed.

Arrays are zero-indexed, meaning the first element is at index 0, the second at index 1, and so on.

Creating Arrays:

Using array literal:

```
let myArray = [1, 2, 3, 'hello', true];
```

Using the `Array` constructor:

```
let myArray = new Array(1, 2, 3, 'hello', true);
// or
let myArray = new Array(5); // Creates an array with a length of 5
```

Note:
While using `new Array()` is possible, using array literals `[]` is generally preferred for readability and performance.

Property .length

The `.length` property of an array returns the number of elements in the array.

```
let myArray = [10, 20, 30];
console.log(myArray.length); // Output: 3
```

You can modify the length of an array by assigning a new value to the `.length` property. If you shorten the length, elements at the end of the array are removed. If you lengthen it, the new elements are `undefined`.

```
let myArray = [1, 2, 3, 4, 5];
myArray.length = 3;
console.log(myArray); // Output: [1, 2, 3]

myArray.length = 5;
console.log(myArray); // Output: [1, 2, 3, undefined, undefined]
```

Accessing Elements by Index

Array elements are accessed using their index, starting from 0.

```
let myArray = ['apple', 'banana', 'cherry'];
console.log(myArray[0]); // Output: 'apple'
console.log(myArray[2]); // Output: 'cherry'
```

If you try to access an index that doesn't exist, you'll get `undefined`.

```
let myArray = [1, 2, 3];
console.log(myArray[5]); // Output: undefined
```

Arrays are Mutable

Arrays in JavaScript are mutable, meaning you can change their contents after they are created.

You can modify an array by assigning new values to specific indices.

```
let myArray = [1, 2, 3];
myArray[0] = 10;
console.log(myArray); // Output: [10, 2, 3]
```

Method .push()

The `.push()` method adds one or more elements to the end of an array and returns the new length of the array.

```
let myArray = [1, 2, 3];
let newLength = myArray.push(4, 5);
console.log(myArray);     // Output: [1, 2, 3, 4, 5]
console.log(newLength); // Output: 5
```

Method .pop()

The `.pop()` method removes the last element from an array, reduces the array's length by 1, and returns the removed element.

```
let myArray = [1, 2, 3];
let removedElement = myArray.pop();
console.log(myArray);        // Output: [1, 2]
console.log(removedElement); // Output: 3
```

If the array is empty, `.pop()` returns `undefined`.

```
let myArray = [];
let removedElement = myArray.pop();
console.log(removedElement); // Output: undefined
```

Method .shift()

The `.shift()` method removes the first element from an array, shifts all other elements to a lower index, and returns the removed element.

```
let myArray = [1, 2, 3];
let removedElement = myArray.shift();
console.log(myArray);        // Output: [2, 3]
console.log(removedElement); // Output: 1
```

If the array is empty, `.shift()` returns `undefined`.

```
let myArray = [];
let removedElement = myArray.shift();
console.log(removedElement); // Output: undefined
```

Method .unshift()

The `.unshift()` method adds one or more elements to the beginning of an array and returns the new length of the array.

```
let myArray = [1, 2, 3];
let newLength = myArray.unshift(0);
console.log(myArray);     // Output: [0, 1, 2, 3]
console.log(newLength); // Output: 4
```

Method .concat()

The `.concat()` method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

```
let array1 = [1, 2, 3];
let array2 = [4, 5, 6];
let newArray = array1.concat(array2);

console.log(array1);   // Output: [1, 2, 3]
console.log(array2);   // Output: [4, 5, 6]
console.log(newArray); // Output: [1, 2, 3, 4, 5, 6]
```

You can concatenate multiple arrays or values:

```
let array1 = [1, 2];
let array2 = [3, 4];
let array3 = [5, 6];
let newArray = array1.concat(array2, array3, 7, 8);

console.log(newArray); // Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

JavaScript Loops

While Loop

The `while` loop executes a block of code as long as a specified condition is true.

```
while (condition) {
  // Code to be executed
}
```

Example:

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
// Output: 0 1 2 3 4
```

Important: Ensure the condition eventually becomes false to avoid infinite loops.

Reverse Loop

Looping backwards is useful for certain array manipulations or when needing to process elements in reverse order.

```
for (let i = array.length - 1; i >= 0; i--) {
  // Code to be executed
}
```

Example:

```
const arr = [1, 2, 3, 4, 5];
for (let i = arr.length - 1; i >= 0; i--) {
  console.log(arr[i]);
}
// Output: 5 4 3 2 1
```

Do...While Statement

The `do...while` statement executes a block of code once, and then repeats as long as a condition is true. The key difference from `while` is that the code block is executed at least once.

```
do {
  // Code to be executed
} while (condition);
```

Example:

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
// Output: 0 1 2 3 4
```

Even if the condition is initially false, the loop body will execute once:

```
let i = 5;
do {
  console.log(i);
  i++;
} while (i < 5);
// Output: 5
```

For Loop

The `for` loop is a versatile loop construct with three optional expressions:

```
for ([initialization]; [condition]; [final-expression]) {
  // Code to be executed
}
```

- **initialization:** Executed before the loop starts.
- **condition:** Evaluated before each loop iteration; if true, the loop continues.
- **final-expression:** Executed at the end of each loop iteration.

Example:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// Output: 0 1 2 3 4
```

Looping Through Arrays

Arrays are commonly iterated over using `for` loops or `forEach` method.

```
const arr = ['a', 'b', 'c'];

// Using for loop
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}

// Using forEach
arr.forEach(element => {
  console.log(element);
});
// Output: a b c (for both examples)
```

The `forEach` method provides a more concise way to iterate through arrays.

Break

The `break` statement terminates the current loop, switch, or labeled statement and transfers program control to the statement following the terminated statement.

```
for (let i = 0; i < 10; i++) {
  if (i === 3) {
    break;
  }
  console.log(i);
}
// Output: 0 1 2
```

Continue

The `continue` statement terminates execution of the statements in the current iteration of the current loop, and continues execution of the loop with the next iteration.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    continue;
  }
  console.log(i);
}
// Output: 0 1 2 4
```

Nested Loops

Loops can be nested inside other loops to handle multi-dimensional data or complex iterative tasks.

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 2; j++) {
    console.log(`i: ${i}, j: ${j}`);
  }
}
// Output:
// i: 0, j: 0
// i: 0, j: 1
// i: 1, j: 0
// i: 1, j: 1
// i: 2, j: 0
// i: 2, j: 1
```

for...in loop

The `for...in` loop iterates over all enumerable properties of an object. It is generally used for objects, not arrays.

```
const obj = { a: 1, b: 2, c: 3 };

for (let key in obj) {
  console.log(`Key: ${key}, Value: ${obj[key]}`);
}
// Output:
// Key: a, Value: 1
// Key: b, Value: 2
// Key: c, Value: 3
```

Note: Be cautious when using `for...in` with arrays, as it may not iterate in the order you expect and can include inherited properties.

for...of loop

The `for...of` loop iterates over iterable objects (including arrays, strings, maps, sets, etc.), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

```
const arr = ['a', 'b', 'c'];

for (let value of arr) {
  console.log(value);
}
// Output: a b c
```

It is a cleaner and more direct way to iterate over values in iterable objects compared to the traditional `for` loop.

JavaScript Iterators and Array Methods

Functions Assigned to Variables

In JavaScript, functions can be assigned to variables, allowing for flexible and reusable code.

Declaration:

```
const myFunction = function(param1, param2) {
  // Function body
  return param1 + param2;
};
```

Usage:

```
const result = myFunction(5, 3); // result will be 8
console.log(result);
```

Arrow Function Assignment:

```
const myArrowFunction = (param1, param2) => {
  // Function body
  return param1 * param2;
};
```

Usage:

```
const product = myArrowFunction(5, 3);
// product will be 15
console.log(product);
```

Benefits:

- Functions can be treated as first-class citizens.
- Enables functional programming paradigms.
- Allows passing functions as arguments to other functions.

Callback Functions

A callback function is a function passed as an argument to another function, which is then invoked inside the outer function to complete an action.

Example:

```
function greet(name, callback) {
  console.log('Hello, ' + name + '!');
  callback();
}

function sayGoodbye() {
  console.log('Goodbye!');
}

greet('John', sayGoodbye); // Output: Hello, John!
                           // Goodbye!
```

Asynchronous Callbacks:
Used in asynchronous operations such as `setTimeout` and event listeners.

Example:

```
setTimeout(function() {
  console.log('This will be executed after 2 seconds.');
}, 2000);
```

Benefits:

- Enables asynchronous programming.
- Allows for custom behavior within functions.
- Promotes modular and reusable code.

Array Method .reduce()

The `.reduce()` method executes a reducer function (provided by you) on each element of the array, resulting in a single output value.

Syntax:

```
array.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])
```

Parameters:

- `callback`: Function to execute on each element in the array.
- `accumulator`: Accumulates the callback's return values; it is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied.
- `currentValue`: The current element being processed in the array.
- `index` (optional): The index of the current element being processed in the array.
- `array` (optional): The array `reduce()` was called upon.
- `initialValue` (optional): Value to use as the first argument to the first call of the callback. If no initialValue is supplied, the first element in the array will be used as the initial accumulator value and skipped as currentValue.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(sum); // Output: 15
```

Use Cases:

- Summing values in an array.
- Flattening an array of arrays.
- Grouping objects by a property.

Array Method .map()

The `.map()` method creates a new array populated with the results of calling a provided function on every element in the calling array.

Syntax:

```
array.map(callback(currentValue[, index[, array]])[, thisArg])
```

Parameters:

- `callback` : Function that produces an element of the new array, taking three arguments:
  - `currentValue` : The current element being processed in the array.
  - `index` (optional): The index of the current element being processed in the array.
  - `array` (optional): The array `map()` was called upon.
- `thisArg` (optional): Value to use as `this` when executing callback.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers =
numbers.map(number => number * number);
console.log(squaredNumbers); // Output:
[1, 4, 9, 16, 25]
```

Use Cases:

- Transforming array elements.
- Formatting data for display.
- Creating a new array with modified values.

Array Method .forEach()

The `.forEach()` method executes a provided function once for each array element.

Syntax:

```
array.forEach(callback(currentValue[, index[, array]])[, thisArg])
```

Parameters:

- `callback` : Function to execute on each element, taking three arguments:
  - `currentValue` : The current element being processed in the array.
  - `index` (optional): The index of the current element being processed in the array.
  - `array` (optional): The array `forEach()` was called upon.
- `thisArg` (optional): Value to use as `this` when executing callback.

Example:

```
const numbers = [1, 2, 3];
numbers.forEach(number => {
  console.log(number * 2);
}); // Output: 2
   //         4
   //         6
```

Use Cases:

- Iterating over array elements to perform an action.
- Updating variables outside the array.
- Performing side effects for each element.

Array Method .filter()

The `.filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Syntax:

```
array.filter(callback(element[, index[, array]])[, thisArg])
```

Parameters:

- `callback` : Function to test each element in the array. Return `true` to keep the element, `false` otherwise, taking three arguments:
  - `element` : The current element being processed in the array.
  - `index` (optional): The index of the current element being processed in the array.
  - `array` (optional): The array `filter()` was called upon.
- `thisArg` (optional): Value to use as `this` when executing callback.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers =
numbers.filter(number => number % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
```

Use Cases:

- Selecting elements based on a condition.
- Removing unwanted elements from an array.
- Creating a new array with a subset of elements.

Chaining Array Methods

Array methods can be chained together to perform multiple operations in a concise way.

Example:

```
const numbers = [1, 2, 3, 4, 5, 6];
const sumOfEvenSquares = numbers
  .filter(number => number % 2 === 0)
  .map(number => number * number)
  .reduce((accumulator, currentValue) =>
accumulator + currentValue, 0);

console.log(sumOfEvenSquares); //
Output: 56 (2^2 + 4^2 + 6^2 = 4 + 16 +
36 = 56)
```

Explanation:

1. `.filter()` selects even numbers: `[2, 4, 6]`
2. `.map()` squares each number: `[4, 16, 36]`
3. `.reduce()` sums the squared numbers: `56`

Benefits:

- Improves code readability.
- Reduces the need for intermediate variables.
- Enables a more functional programming style.

Practical Examples

Example 1: Extracting Names from Objects

```
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];

const names = users.map(user =>
user.name);
console.log(names); // Output: ['Alice',
'Bob', 'Charlie']
```

Example 2: Calculating Average Score

```
const scores = [85, 90, 78, 92, 88];
const average = scores.reduce((acc,
score) => acc + score, 0) /
scores.length;
console.log(average); // Output: 86.6
```

Example 3: Filtering Active Users

```
const users = [
  { id: 1, name: 'Alice', isActive: true
},
  { id: 2, name: 'Bob', isActive: false
},
  { id: 3, name: 'Charlie', isActive:
true }
];

const activeUsers = users.filter(user =>
user.isActive);
console.log(activeUsers);
// Output:
// [
//   { id: 1, name: 'Alice', isActive:
true },
//   { id: 3, name: 'Charlie', isActive:
true }
// ]
```

JavaScript Objects

Accessing Properties

Dot notation

Access properties using a dot followed by the property name.

```
const obj = { name: 'John', age: 30 };
console.log(obj.name); // Output: John
```

Bracket notation

Access properties using square brackets with the property name as a string.

```
const obj = { name: 'John', age: 30 };
console.log(obj['age']); // Output: 30
```

Dynamic access

Bracket notation allows accessing properties with dynamically determined names.

```
const propertyName = 'name';
const obj = { name: 'John' };
console.log(obj[propertyName]); // Output: John
```

Naming Properties

Valid names

Property names can be strings or symbols. String names must follow JavaScript identifier rules or be quoted.

```
const obj = {
  firstName: 'John', // Valid
  'last-name': 'Doe', // Valid, but requires bracket notation
  123: 'Number' // Valid, but requires bracket notation
};
```

Reserved words

Avoid using reserved words as property names to prevent unexpected behavior.

```
const obj = { for: 'example' }; // Avoid this
```

Non-existent Properties

Accessing undefined properties

Accessing a property that doesn't exist returns `undefined`.

```
const obj = { name: 'John' };
console.log(obj.age); // Output: undefined
```

Checking property existence

Use `in` operator or `hasOwnProperty` method to check if a property exists.

```
const obj = { name: 'John' };
console.log('name' in obj); // Output: true
console.log(obj.hasOwnProperty('name')); // Output: true
console.log('age' in obj); // Output: false
```

Mutable

Object mutability

Objects are mutable, meaning their properties can be changed after creation.

```
const obj = { name: 'John' };
obj.name = 'Jane';
console.log(obj.name); // Output: Jane
```

Const objects

`const` prevents reassignment of the variable, but doesn't make the object immutable.

```
const obj = { name: 'John' };
obj.name = 'Jane'; // Allowed
// obj = { age: 30 }; // Not allowed
```

Assignment Shorthand Syntax

Property value shorthand

If a variable name matches the property name, you can use the shorthand.

```
const name = 'John';
const age = 30;
const obj = { name, age }; // Equivalent to { name: name, age: age }
console.log(obj); // Output: { name: 'John', age: 30 }
```

Computed property names

Use square brackets to define property names dynamically.

```
const keyName = 'age';
const obj = { [keyName]: 30 };
console.log(obj); // Output: { age: 30 }
```

Delete Operator

Deleting properties

The `delete` operator removes a property from an object.

```
const obj = { name: 'John', age: 30 };
delete obj.age;
console.log(obj); // Output: { name: 'John' }
```

Non-configurable properties

Properties defined as non-configurable cannot be deleted.

```
const obj = {};
Object.defineProperty(obj, 'name', { value: 'John', configurable: false });
delete obj.name; // Returns false in strict mode, otherwise no effect
console.log(obj.name); // Output: John
```

Objects as Arguments

Passing objects

Objects are passed by reference. Modifying the object inside a function affects the original object.

```
function modifyObject(obj) {
  obj.age = 31;
}

const person = { name: 'John', age: 30 };
modifyObject(person);
console.log(person.age); // Output: 31
```

Destructuring arguments

Use destructuring to extract properties from an object passed as an argument.

```
function greet({ name, age }) {
  console.log(`Hello, ${name}! You are ${age} years old.`);
}

const person = { name: 'John', age: 30 };
greet(person); // Output: Hello, John! You are 30 years old.
```

Shorthand object creation

Example

When a variable name is the same as the object key you can simplify the object declaration

```
const name = 'John';
const age = 30;

const person = {
 name,
 age
};

console.log(person); // { name: 'John', age: 30 }
```

this Keyword

Context

Refers to the context in which a function is executed.

```
const person = {
  name: 'John',
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

person.greet(); // Output: Hello, my name is John
```

Arrow functions

Arrow functions do not bind their own `this` value; they inherit it from the enclosing scope.

```
const person = {
  name: 'John',
  greet: () => {
    console.log(`Hello, my name is ${this.name}`); // this will refer to the window object
  }
};

person.greet();
```

Factory functions

Definition

A function that returns a new object.

```
function createPerson(name, age) {
  return {
    name: name,
    age: age,
    greet: function() {
      console.log(`Hello, my name is ${this.name}`);
    }
  };
}

const john = createPerson('John', 30);
john.greet(); // Hello, my name is John
```

Methods

Definition

Functions assigned as properties of objects.

```
const person = {
  name: 'John',
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

person.greet(); // Hello, my name is John
```

Getters and setters

Definition

Special methods that allow you to define custom behavior when getting or setting a property value.

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  set fullName(value) {
    const parts = value.split(' ');
    this.firstName = parts[0];
    this.lastName = parts[1];
  }
};

console.log(person.fullName); // John Doe
person.fullName = 'Jane Smith';
console.log(person.firstName); // Jane
console.log(person.lastName);  // Smith
```

JavaScript Classes

Class Declaration

Classes are a template for creating objects. They encapsulate data with code to work on that data. JavaScript classes are built on prototypes but also have some syntax and semantics that are unique to classes.

```
class MyClass {
  constructor(param1, param2) {
    // ...
  }

  method1() {
    // ...
  }

  method2() {
    // ...
  }
}
```

```
// Class expression
const MyClass = class {
  constructor(param1, param2) {
    // ...
  }
};

// Named class expression
const MyClass = class MyNamedClass {
  constructor(param1, param2) {
    // ...
  }
};
```

Classes are in fact “special functions”, and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

Class Constructor

The `constructor` method is a special method for creating and initializing an object created with a `class`. There can only be one special method with the name “constructor” in a class. A `SyntaxError` will be thrown if the class contains more than one occurrence of a `constructor` method.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

If you do not specify a constructor method, a default constructor is used. The arguments for the default constructor are the arguments passed to the class’s `new` operator.

```
class Parent {
  constructor() {
    this.name = 'Parent';
  }
}

class Child extends Parent {
  constructor() {
    super(); // Call the parent's constructor
    this.age = 10;
  }
}
```

Class Methods

Class methods are defined inside the class body. They can access the class’s properties and other methods using `this`.

```
class MyClass {
  myMethod() {
    console.log('Hello from myMethod!');
  }
}

const instance = new MyClass();
instance.myMethod(); // Output: Hello from myMethod!
```

Methods can return values or perform actions within the class.

```
class Calculator {
  add(a, b) {
    return a + b;
  }
}

const calc = new Calculator();
console.log(calc.add(5, 3)); // Output: 8
```

```
class MyClass {
  set myProperty(value) {
    this._myProperty = value;
  }

  get myProperty() {
    return this._myProperty;
  }
}

const instance = new MyClass();
instance.myProperty = 'New Value';
console.log(instance.myProperty); // Output: New Value
```

Static Methods

Static methods are called directly on the class (not on an instance) and are often used for utility functions.

```
class MyClass {
  static myStaticMethod() {
    console.log('Hello from static method!');
  }
}

MyClass.myStaticMethod(); // Output: Hello from static method!
```

Static methods are useful for creating utility functions that are related to the class but don't require an instance.

```
class MathUtils {
  static add(a, b) {
    return a + b;
  }
}

console.log(MathUtils.add(5, 3)); // Output: 8
```

```
class MyClass {
  static myStaticProperty = 'Static Value';

  static myStaticMethod() {

console.log(MyClass.myStaticProperty);
  }
}

MyClass.myStaticMethod(); // Output: Static Value
```

Class Inheritance (extends)

The `extends` keyword is used in class declarations or class expressions to create a child class. The child class inherits all the properties and methods from the parent class.

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog('Buddy');
dog.speak(); // Output: Buddy barks.
```

Use `super()` to call the constructor of the parent class and access its properties and methods.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }

  info() {
    console.log(`${this.name} is a ${this.breed}.`);
  }
}

const dog = new Dog('Buddy', 'Golden Retriever');
dog.info(); // Output: Buddy is a Golden Retriever.
```

You can override methods from the parent class in the child class to provide specialized behavior.

```
class Animal {
  speak() {
    console.log('Generic animal sound.');
  }
}

class Dog extends Animal {
  speak() {
    console.log('Woof!');
  }
}

const animal = new Animal();
const dog = new Dog();
animal.speak(); // Output: Generic animal sound.
dog.speak(); // Output: Woof!
```

JavaScript Modules

Introduction to Modules

Modules in JavaScript allow you to split your code into separate files. This improves code organization, reusability, and maintainability. Modules help avoid naming conflicts and provide a way to manage dependencies.

Key Concepts:

- **Encapsulation:** Modules encapsulate code, hiding implementation details and exposing only necessary parts.
- **Reusability:** Modules can be reused in different parts of an application or in different applications.
- **Dependency Management:** Modules explicitly declare their dependencies on other modules.

JavaScript modules primarily come in two flavors:

- **ES Modules (ESM):** The standard format for modules in modern JavaScript.
- **CommonJS (CJS):** Primarily used in Node.js environments.

ES Modules: Exporting

Named Exports:

Exporting specific variables, functions, or classes by their names.

```
// module.js
export const PI = 3.14159;
export function add(x, y) {
  return x + y;
}
export class MyClass {
  // ...
}
```

Default Export:

Exporting a single value as the default export.

```
// module.js
const myValue = 'Hello, world!';
export default myValue;
```

Exporting at the End:

You can also declare variables and functions first, then export them.

```
// module.js
const PI = 3.14159;
function add(x, y) {
  return x + y;
}

export { PI, add };
```

ES Modules: Importing

Named Imports:

Importing specific values using their names.

```
// main.js
import { PI, add } from './module.js';

console.log(PI);
console.log(add(2, 3));
```

Default Import:

Importing the default export. You can name the import anything you like.

```
// main.js
import myValue from './module.js';

console.log(myValue);
```

Importing Everything:

Importing all exports into a single namespace.

```
// main.js
import * as Module from './module.js';

console.log(Module.PI);
console.log(Module.add(2, 3));
```

Renaming Imports:

Renaming imports using the `as` keyword.

```
// main.js
import { add as sum } from
'./module.js';

console.log(sum(2, 3));
```

CommonJS: Exporting (Node.js)

Exporting Properties:

In CommonJS, you use the `module.exports` object to export values.

```
// module.js
module.exports.PI = 3.14159;
module.exports.add = function(x, y) {
  return x + y;
};

//Alternative way
exports.PI = 3.14159;
exports.add = function(x, y) {
  return x + y;
};
```

Exporting a Single Value:

You can also export a single value by assigning it directly to `module.exports`.

```
// module.js
module.exports = 'Hello, world!';
```

CommonJS: Importing (Node.js)

Importing Modules:

Use the `require()` function to import modules. `require()` returns the `module.exports` object of the imported module.

```
// main.js
const Module = require('./module.js');

console.log(Module.PI);
console.log(Module.add(2, 3));
```

Importing a Single Value:

If a module exports a single value, `require()` will return that value.

```
// main.js
const myValue = require('./module.js');

console.log(myValue);
```

Dynamic Imports (ES Modules)

Dynamic imports allow you to load modules asynchronously, typically used for code splitting or loading modules on demand. They use the `import()` function, which returns a promise.

```
async function loadModule() {
  const module = await import('./my-
module.js');
  module.myFunction();
}

loadModule();
```

Use Cases:

- Loading modules based on user interactions.
- Code splitting to reduce initial load time.
- Loading optional dependencies.

JavaScript Promises

Promise States

Pending: Initial state; neither fulfilled nor rejected.

Fulfilled: The operation completed successfully.

Rejected: The operation failed.

A promise can only resolve once. Subsequent calls to `resolve()` or `reject()` are ignored. Once a promise is fulfilled or rejected, it is considered **settled**.

Promises help manage asynchronous operations in JavaScript, providing a cleaner alternative to callbacks.

Executor Function

The executor function is passed to the `Promise` constructor.

It takes two arguments: `resolve` and `reject`, which are functions used to control the promise's state.

```
const myPromise = new Promise((resolve,
reject) => {
  // Asynchronous operation here
  if (/* operation succeeds */) {
    resolve('Operation successful!');
  } else {
    reject('Operation failed!');
  }
});
```

The executor function is executed immediately when the `Promise` is created.

setTimeout() with Promises

`setTimeout()` is commonly used within Promises to simulate asynchronous behavior.

It delays the execution of a function by a specified number of milliseconds.

```
const delayedPromise = new
Promise((resolve) => {
  setTimeout(() => {
    resolve('Resolved after 2 seconds');
  }, 2000);
});

delayedPromise.then((value) => {
  console.log(value); // Output:
Resolved after 2 seconds
});
```

This allows you to work with asynchronous code in a synchronous-like manner.

.then() Method

The `.then()` method is used to handle the fulfillment of a promise.

It takes one or two arguments: a callback function for fulfillment and an optional callback function for rejection.

```
myPromise.then(
  (value) => {
    console.log('Fulfilled:', value);
  },
  (reason) => {
    console.error('Rejected:', reason);
  }
);
```

It returns a new promise, allowing for chaining.

.catch() Method

The `.catch()` method is used to handle rejections of a promise.

It's a shorthand for `.then(null, rejectionCallback)`.

```
myPromise.catch((reason) => {
  console.error('Rejected:', reason);
});
```

Using `.catch()` makes error handling more readable and maintainable.

Promise.all()

`Promise.all()` takes an array of promises and returns a new promise that fulfills when all the input promises fulfill, or rejects when any of the input promises reject.

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve,
reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2,
promise3]).then((values) => {
  console.log(values); // Output: [3,
42, 'foo']
});
```

It's useful when you need to wait for multiple asynchronous operations to complete before proceeding.

Avoiding Nested Promises and .then()

Nesting `.then()` calls (also known as "callback hell" or "pyramid of doom") can make code difficult to read and maintain.

Use promise chaining to avoid excessive nesting.

```
// Avoid:
promise1.then((result1) => {
  promise2.then((result2) => {
    promise3.then((result3) => {
      // ...
    });
  });
});

// Prefer:
promise1
  .then((result1) => promise2(result1))
  .then((result2) => promise3(result2))
  .then((result3) => {
    // ...
  });
```

Chaining makes the code flow more linearly and easier to understand.

Creating Promises

To create a Promise, use the `new Promise()` constructor and pass in an executor function.

```
const myPromise = new Promise((resolve,
reject) => {
  // Perform asynchronous operation
  if (/* operation successful */) {
    resolve(value);
  } else {
    reject(error);
  }
});
```

`resolve()` is called when the operation is successful, and `reject()` is called when it fails.

Chaining Multiple .then()

You can chain multiple `.then()` calls to perform a sequence of asynchronous operations.

Each `.then()` receives the result of the previous `.then()`.

```
fetchData()
  .then((data) => processData(data))
  .then((processedData) =>
updateUI(processedData))
  .catch((error) => handleError(error));
```

This allows you to build complex asynchronous workflows in a readable manner.

Fake HTTP Request with Promise

Simulating an HTTP request with a Promise and `setTimeout()` demonstrates how Promises can be used with asynchronous operations.

```
function fakeHttpRequest(url) {
  return new Promise((resolve, reject)
=> {
    setTimeout(() => {
      const data = { id: 1, name:
'Example Data' };
      resolve(data);
      // If there was an error:
      // reject('Request failed');
    }, 1000);
  });
}

fakeHttpRequest('/data')
  .then((data) => console.log('Data
received:', data))
  .catch((error) =>
console.error('Error:', error));
```

This example simulates fetching data from a server and handles both success and failure scenarios.

JavaScript Async-Await

Asynchronous JavaScript

JavaScript is single-threaded, meaning it executes code sequentially. Asynchronous operations allow the program to continue running while waiting for long-running tasks (like network requests) to complete.

Key Concepts:

- **Callbacks:** Traditional way of handling async operations; can lead to 'callback hell'.
- **Promises:** Represent the eventual result of an asynchronous operation.
- **Async/Await:** Syntactic sugar built on top of promises, making asynchronous code look and behave a bit more like synchronous code.

Async/await simplifies asynchronous JavaScript, making it easier to read and write asynchronous code. It's built on promises.

Resolving Promises

Promises represent the eventual result of an asynchronous operation. They can be in one of three states:

- **Pending:** Initial state, neither fulfilled nor rejected.
- **Fulfilled:** Operation completed successfully.
- **Rejected:** Operation failed.

To handle the result of a promise, you use `.then()` for success and `.catch()` for errors.

```
const myPromise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve('Promise resolved!');
  }, 1000);
});

myPromise
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Async Functions

An `async` function is declared with the `async` keyword before the function declaration. It allows you to use the `await` keyword inside the function.

- Async functions always return a promise, whether you explicitly return a promise or not.
- If the async function returns a value, that value will be wrapped in a resolved promise.
- If the async function throws an error, it will return a rejected promise.

```
async function myFunction() {
  return 'Hello, Async!';
}

myFunction().then(result =>
console.log(result)); // Output: Hello,
Async!
```

Await Operator

The `await` operator is used inside an `async` function to pause the execution of the function until the promise is resolved. It only works inside async functions.

- `await` waits for the promise to resolve or reject.
- If the promise resolves, `await` returns the resolved value.
- If the promise rejects, `await` throws an error (which can be caught using `try...catch`).

```
async function fetchData() {
  const response = await
fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
}

fetchData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Async Await Promises

Async/await makes asynchronous code look and behave a bit more like synchronous code. This can drastically improve the readability and maintainability of your asynchronous JavaScript.

Benefits:

- **Improved Readability:** Reduces nesting and complexity compared to callbacks and promises.
- **Easier Debugging:** Easier to step through async code with debuggers.
- **Concise Code:** Less boilerplate code compared to traditional promise chains.

```
async function getUsers() {
  const response = await
fetch('/users');
  const users = await response.json();
  return users;
}

async function displayUsers() {
  const users = await getUsers();
  users.forEach(user =>
console.log(user.name));
}

displayUsers();
```

Error Handling

Error handling in async/await is done using the standard `try...catch` statement. Wrap the `await` call in a `try` block, and catch any errors in the `catch` block.

Best Practices:

- Wrap each `await` call that might throw an error in its own `try...catch` block.
- Consider using a global error handling mechanism to catch unhandled rejections.

```
async function fetchData() {
  try {
    const response = await
fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Fetch error:',
error);
    throw error; // Re-throw the error
to be handled further up the call stack
  }
}

fetchData()
  .then(data => console.log(data))
  .catch(error => console.error('Global
error handler:', error));
```

Aysnc await operator

The `await` operator is used to wait for a Promise to resolve or reject. It can only be used inside an `async` function. When encountered, it pauses the execution of the `async` function and waits for the Promise to resolve. Once resolved, it resumes the `async` function's execution, returning the resolved value.

If the Promise rejects, the `await` expression will throw an error, which you can catch using `try...catch` blocks.

Example:

```
async function processData() {
  try {
    const data = await fetchData();
    console.log('Data:', data);
  } catch (error) {
    console.error('Error fetching
data:', error);
  }
}
```

```
async function getJson() {
  try {
    let response = await fetch('your-
api-endpoint');
    let data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching
JSON:', error);
  }
}
```

Practical Examples and Use Cases

Async/await can be used in various scenarios, such as fetching data from an API, reading files, or interacting with databases.

Fetching Multiple APIs:

```
async function fetchMultiple() {
  const [users, posts] = await
Promise.all([
    fetch('/users').then(res =>
res.json()),
    fetch('/posts').then(res =>
res.json())
  ]);
  console.log('Users:', users);
  console.log('Posts:', posts);
}
```

Sequential API Calls:

```
async function sequentialCalls() {
  const user = await
fetch('/user/1').then(res =>
res.json());
  const posts = await
fetch(`/posts/${user.id}`).then(res =>
res.json());
  console.log('User:', user);
  console.log('Posts:', posts);
}
```

JavaScript Requests

JSON Basics

JSON (JavaScript Object Notation): A lightweight data-interchange format. Easy for humans to read and write. Easy for machines to parse and generate.

Key Features:

- Key-value pairs.
- Uses JavaScript syntax to describe data objects.
- Commonly used for transmitting data in web applications (e.g., sending data from the server to the client, so it can be displayed on a web page).

JSON Syntax Rules:

- Data is in name/value pairs.
- Data is separated by commas.
- Curly braces hold objects.
- Square brackets hold arrays.

Example:

```
{
  "name":"John Doe",
  "age":30,
  "city":"New York"
}
```

JSON Values:
JSON values can be:

- A string
- A number
- An object (JSON object)
- An array
- A boolean
- null

JSON.stringify(): Converts a JavaScript object to a JSON string.

Example:

```
const obj = {name: "John", age: 30};
const jsonString = JSON.stringify(obj);
console.log(jsonString); // Output:
{"name":"John","age":30}
```

JSON.parse(): Converts a JSON string to a JavaScript object.

Example:

```
const jsonString =
'{"name":"John","age":30}';
const obj = JSON.parse(jsonString);
console.log(obj.name); // Output: John
```

XMLHttpRequest (XHR)

XMLHttpRequest (XHR): A browser API to transfer data between a client and a server without a full page refresh.

Note: `XMLHttpRequest` is an older method, often superseded by the Fetch API, but understanding it can be valuable, especially when working with legacy code.

Creating an XHR Object:

```
const xhr = new XMLHttpRequest();
```

Opening a Request:

```
xhr.open(method, url, async);
```

- `method`: GET, POST, PUT, DELETE, etc.
- `url`: The URL to which the request is sent.
- `async`: true (asynchronous) or false (synchronous).

Example:

```
xhr.open('GET',
'https://example.com/data', true);
```

Sending the Request:

```
xhr.send(body);
```

- `body`: The data sent to the server (for POST, PUT, etc.). Can be null for GET.

Example:

```
xhr.send(); // For GET
xhr.send(JSON.stringify({key:
'value'})); // For POST
```

Handling the Response:

```
xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status <
300) {
    // Request succeeded
    console.log(xhr.responseText);
  } else {
    // Request failed
    console.log('Request failed with
status:', xhr.status);
  }
};

xhr.onerror = function() {
  console.log('Request failed');
};
```

Setting Headers:

```
xhr.setRequestHeader('Content-Type',
'application/json');
```

Common headers include `Content-Type`, `Authorization`, etc.

Example (GET request):

```
const xhr = new XMLHttpRequest();
xhr.open('GET',
'https://api.example.com/data', true);
xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status <
300) {
    console.log(xhr.responseText);
  }
};
xhr.onerror = function() {
  console.log('Request failed');
};
xhr.send();
```

GET Requests

GET Requests: Used to retrieve data from a server. Data is appended to the URL.

Characteristics:

- Data is visible in the URL.
- Limited amount of data can be sent.
- Should not be used for sensitive data.

Using XMLHttpRequest:

```
const xhr = new XMLHttpRequest();
xhr.open('GET',
'https://example.com/data?
param1=value1&param2=value2', true);
xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status <
300) {
    console.log(xhr.responseText);
  }
};
xhr.send();
```

Using Fetch API:

```
fetch('https://example.com/data?
param1=value1&param2=value2')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error =>
console.error('Error:', error));
```

Adding Parameters to URL:

```
const url = new
URL('https://example.com/data');
const params = {param1: 'value1',
param2: 'value2'};
Object.keys(params).forEach(key =>
url.searchParams.append(key,
params[key]));

console.log(url.toString()); // Output:
https://example.com/data?
param1=value1&param2=value2
```

POST Requests

POST Requests: Used to send data to a server to create/update a resource. Data is sent in the request body.

Characteristics:

- Data is not visible in the URL.
- Larger amounts of data can be sent.
- Suitable for sending sensitive data.

Using XMLHttpRequest:

```
const xhr = new XMLHttpRequest();
xhr.open('POST',
'https://example.com/data', true);
xhr.setRequestHeader('Content-Type',
'application/json');
xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status <
300) {
    console.log(xhr.responseText);
  }
};
xhr.send(JSON.stringify({key:
'value'}));
```

Using Fetch API:

```
fetch('https://example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({key: 'value'})
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:',
error));
```

Fetch API

Fetch API: A modern interface for making network requests. It provides a more powerful and flexible feature set than `XMLHttpRequest`.

Basic Syntax:

```
fetch(url, options)
  .then(response => { /* handle response */ })
  .catch(error => { /* handle error */ });
```

Options:

- `method`: HTTP method (GET, POST, PUT, DELETE, etc.).
- `headers`: An object containing HTTP headers.
- `body`: The request body (for POST, PUT, etc.).
- `mode`: CORS mode (cors, no-cors, same-origin).
- `credentials`: Request credentials (omit, same-origin, include).

Handling Responses:

- `response.json()`: Parses the response body as JSON.
- `response.text()`: Parses the response body as text.
- `response.blob()`: Parses the response body as a Blob.
- `response.status`: The HTTP status code (e.g., 200, 404, 500).
- `response.ok`: A boolean indicating whether the request was successful (status in the range 200-299).

Simple GET Request:

```
fetch('https://example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Simple POST Request:

```
fetch('https://example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({key: 'value'})
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

JSON Formatted Requests

JSON Formatted Requests: Sending and receiving data in JSON format is common when using APIs.

Key Considerations:

- Setting the `Content-Type` header to `application/json` for POST/PUT requests.
- Parsing the response body as JSON using `response.json()`.

Example (Fetch API POST with JSON):

```
fetch('https://example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({key: 'value'})
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Example (Fetch API GET and parse JSON):

```
fetch('https://example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Promise and URL Parameters with Fetch API

Promises: Fetch API uses promises to handle asynchronous operations. `.then()` is used for successful responses and `.catch()` for errors.

URL Parameters: You can add parameters to the URL when making a GET request.

Example (Fetch API with URL Parameters):

```
const url = new URL('https://example.com/data');
url.searchParams.append('param1', 'value1');
url.searchParams.append('param2', 'value2');

fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Chaining Promises:

```
fetch('https://example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log('Data received:', data);
    return data;
  })
  .then(processedData => {
    console.log('Processed data:', processedData);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Fetch API Function

Fetch API Function: Encapsulating fetch requests into reusable functions.

Benefits:

- Code reusability.
- Easier to manage complex requests.
- Improved readability.

Example (Reusable Fetch Function):

```
async function fetchData(url, options = {}) {
  try {
    const response = await fetch(url, options);
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
    throw error; // Re-throw the error to be caught by the caller
  }
}

// Usage:
fetchData('https://example.com/data')
  .then(data => console.log('Data:', data))
  .catch(error => console.error('Error:', error));

fetchData('https://example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({key: 'value'})
})
  .then(data => console.log('Data:', data))
  .catch(error => console.error('Error:', error));
```

Async/Await Syntax

Async/Await: Syntactic sugar built on top of promises, making asynchronous code easier to read and write.

Key Concepts:

- `async`: Marks a function as asynchronous, allowing the use of `await`.
- `await`: Pauses the execution of the async function until the promise is resolved.

Example (Async/Await with Fetch API):

```
async function getData() {
  try {
    const response = await fetch('https://example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}

getData();
```

Example (Async/Await POST Request):

```
async function postData() {
  try {
    const response = await fetch('https://example.com/data', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({key: 'value'})
    });

    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}

postData();
```