Author: Lokang Jackson

Book Name: JavaScript and MySQL

Variable

In JavaScript, a variable can be thought of as a storage name for data. You can use variables to store data, such as numbers, strings, arrays, objects, and more, in the computer's memory. A variable must be declared before it can be used, and you can define a variable using **var**, **let**, or **const** keywords. Here's what each of these declarations means:

var: Declares a variable, optionally initializing it to a value. **var** is function-scoped, which means the variable is available within the function it's declared in (or globally if declared outside of a function), regardless of block scope.

```
var message = "Hello, World!";
```

let: Declares a block-scoped, local variable, optionally initializing it to a value. Unlike **var**, a variable declared with **let** is only accessible in the block it's enclosed in.

```
let number = 10;
```

const: Declares a block-scoped, read-only named constant. Variables declared with **const** cannot be re-declared or re-assigned, but they are not immutable. For example, the properties of an object declared with **const** can be altered.

```
const PI = 3.14159;
```

Here are some additional rules and conventions for JavaScript variables:

- **Names**: A variable name must begin with a letter, dollar sign ($), or underscore (_). It can't start with a number.
- **Case Sensitivity**: Variable names are case-sensitive. For instance, **myVar**, **myvar**, and **MYVAR** are three different variables.
- **Camel Case**: JavaScript follows the camel case convention. Start with a lowercase letter and then subsequent words are written with their first letter capitalized (e.g., **myVariableName**).
- **Re-declaring**: Using **var**, a variable can be re-declared with no problem. If you re-declare a variable using **let**, you'll get an error.
- **Hoisting**: Variables declared with **var** are hoisted to the top of the function or global scope. However, **let** and **const** are not hoisted in the same way.

Remember, good variable naming and the thoughtful use of **var**, **let**, and **const** can make your code much more readable and understandable to both you and others who might read your code.

Arrays

In JavaScript, an array is a special variable that can hold more than one value at a time. Arrays are used to store multiple values in a single variable. These values can be of various types, including other arrays, and they're maintained in an ordered collection. Here's how you work with arrays in JavaScript:

Creating an Array

You can create an array by using the array literal syntax, or the Array constructor.

Array Literal Syntax (preferred way because it's more concise):

```
let fruits = ["Apple", "Banana", "Mango"];
```

Array Constructor:

```
let vegetables = new Array("Carrot", "Pea");
```

Accessing Array Elements

You can access an element in an array by referring to the index number. This reference is zero-based, meaning that the first element has the index 0, the second has index 1, and so on.

```
let fruits = ["Apple", "Banana", "Mango"];
let firstFruit = fruits[0]; // "Apple"
```

Changing an Array Element

Similarly, you can change the value of a specific element in an array by using the index number.

```
fruits[1] = "Strawberry"; // Now the array is ["Apple", "Strawberry",
"Mango"]
```

Accessing the Full Array

If you want to access the whole array, you can just use the array's name:

```
console.log(fruits); // ["Apple", "Strawberry", "Mango"]
```

Array Properties and Methods

Arrays have properties and methods that you can use to manipulate them. Here are a few common ones:

length: This property returns the number of elements in the array.

```
let count = fruits.length; // 3
```

push(): This method adds a new element to the end of an array.

```
fruits.push("Orange"); // ["Apple", "Strawberry", "Mango", "Orange"]
```

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

```
fruits.pop(); // Removes "Orange" from the fruits array
```

shift(): This method removes the first item of an array.

```
fruits.shift(); // Removes "Apple" from the fruits array
```

unshift(): This method adds a new item at the beginning of an array.

```
fruits.unshift("Pineapple"); // Adds "Pineapple" at the beginning of the
fruits array
```

indexOf(): This method searches the array for an element and returns its first index. If it's not found, it returns -1.

```
let pos = fruits.indexOf("Mango"); // 1
```

splice(): This method can be used to add new items to an array or remove existing items.

```
fruits.splice(1, 1, "Peach"); // Removes "Strawberry" at index 1 and adds
"Peach" at that position
```

Iterating Over an Array

To loop through each item in an array, you can use a simple **for** loop, **for...of** loop, or the **forEach()** method, among others.

```
// for loop
for (let i = 0; i < fruits.length; i++) {
 console.log(fruits[i]);
}
// for...of loop
for (let fruit of fruits) {
 console.log(fruit);
}
// forEach() method
fruits.forEach(function(item, index, array) {
 console.log(item, index);
});
```

Arrays in JavaScript are versatile and powerful, and they include many more methods than what's listed here. They are a fundamental part of the language and are used to store and manipulate collections of data efficiently.

operator

Sure, here's a more comprehensive list including all the symbols for each type of operator in

JavaScript and one example for each:

1. Arithmetic Operators

- **Symbols**: **+**, **-**, *, /, **%**, **++**, **--**, **

- **Examples**:

```
let add = 5 + 3;       // 8
let sub = 5 - 3;       // 2
let mul = 5 * 3;       // 15
let div = 5 / 3;       // 1.6666...
let mod = 5 % 3;       // 2
let num = 5;
num++;                 // 6
num--;                 // 5 again
let exp = 5 ** 3;      // 125
```

2. Assignment Operators

- **Symbols**: **=**, **+=**, **-=**, ***=**, **/=**, **%=**, ****=**
- **Examples**:

```
let x = 5;      // 5
x += 3;         // 8
x -= 3;         // 5
x *= 3;         // 15
x /= 3;         // 5
x %= 3;         // 2
x **= 3;        // 8
```

3. String Operators

- **Symbol**: **+** (there's only one symbol for string operation which is used for concatenation)
- **Example**:

```
let text = "Hello" + " " + "World"; // "Hello World"
```

4. Comparison Operators

- **Symbols**: **==**, **===**, **!=**, **!==**, **>**, **<**, **>=**, **<=**
- **Examples**:

```
let isEqual = (5 == '5');          // true
let isStrictEqual = (5 === '5');  // false
let isNotEqual = (5 != '5');      // false
let isStrictNotEqual = (5 !== '5'); // true
let isGreater = (5 > 3);          // true
let isLesser = (5 < 3);           // false
let isGreaterOrEqual = (5 >= 5);  // true
let isLesserOrEqual = (5 <= 5);   // true
```

5. Logical Operators

- **Symbols**: **&&**, **||**, **!**
- **Examples**:

```
let and = (true && false);   // false
let or = (true || false);    // true
let not = !(true);           // false
```

6. Bitwise Operators

- **Symbols**: **&**, **|**, **^**, **~**, **<<**, **>>**, **>>>**
- **Examples**:

```
let bitwiseAND = (5 & 3);          // 1
let bitwiseOR = (5 | 3);           // 7
let bitwiseXOR = (5 ^ 3);          // 6
let bitwiseNOT = (~5);             // -6
let bitwiseLeftShift = (5 << 1);   // 10
let bitwiseRightShift = (5 >> 1);  // 2
let bitwiseZeroFillRightShift = (5 >>> 1);  // 2
```

7. Conditional (Ternary) Operator

- **Symbol**: **? :**
- **Example**:

```
let status = (age >= 18) ? "adult" : "minor"; // "adult" if age is 18 or
over, otherwise "minor"
```

8. Nullish Coalescing Operator

- **Symbol**: **??**
- **Example**:

```
let username = userInput ?? "defaultUser"; // "defaultUser" if userInput
is null or undefined
```

This set of examples should give you a comprehensive view of the various operators available in JavaScript, represented by their respective symbols, along with practical usage examples.

condition and loop

In JavaScript, conditions and loops are fundamental constructs that control the flow of execution in scripts. Conditions allow the execution of code segments based on whether a certain criterion is met, while loops repeat a code segment based on a condition or until a certain criterion is met. Let's explore both, along with examples:

Conditions:

if...else statement - Executes a block of code if a specified condition is true. If the condition is false, another block of code can be executed.

```
let temperature = 20;
if (temperature < 25) {
   console.log("It's cool outside.");
} else {
   console.log("It's warm outside.");
}
// Output: It's cool outside.
```

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

```
let score = 70;
if (score >= 90) {
   console.log("Grade A");
} else if (score >= 70) {
   console.log("Grade B");
} else {
   console.log("Grade C");
}
// Output: Grade B
```

switch statement - Evaluates an expression, matches the expression's value to a case clause, and executes statements associated with that case.

```
let day = 3;
switch (day) {
   case 1:
       console.log("Monday");
       break;
   case 2:
       console.log("Tuesday");
       break;
   case 3:
       console.log("Wednesday");
       break;
   case 4:
       console.log("Thursday");
       break;
   case 5:
       console.log("Friday");
       break;
   case 6:
       console.log("Saturday");
       break;
   case 7:
       console.log("Sunday");
```

```
        break;
    default:
        console.log("Invalid Day");
}
// Output: Wednesday
```

Loops:

for loop - Repeats a block of code a specified number of times.

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

while loop - Repeats a block of code while a specified condition is true.

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

do...while loop - Also repeats a block of code while a specified condition is true but guarantees the code to be executed at least once.

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

for...in loop - Iterates over the enumerable properties of an object.

```
const car = {make: "Ford", model: "Mustang", year: 1969};
for (let prop in car) {
   console.log(`${prop}: ${car[prop]}`);
}
// Output: make: Ford, model: Mustang, year: 1969
```

for...of loop - Iterates over iterable objects such as arrays and strings.

```
const colors = ["red", "green", "blue"];
for (let color of colors) {
   console.log(color);
}
```

```
// Output: red, green, blue
```

Combining conditions and loops is a powerful technique, allowing for complex logic to be expressed cleanly and efficiently. Depending on the scenario, you might use conditions to determine when loops should start or end or what logic should be performed within a loop.

Built in

Sure, here are examples of each of the JavaScript built-in functions listed:

1. String Functions

charAt(index)

```
let str = "Hello, World!";
console.log(str.charAt(0)); // "H"
```

concat(string2, string3, ..., stringX)

```
let str1 = "Hello";
let str2 = "World";
console.log(str1.concat(", ", str2, "!")); // "Hello, World!"
```

includes(searchString, position)

```
let str = "Hello, World!";
console.log(str.includes("World")); // true
```

indexOf(searchValue, fromIndex)

```
let str = "Hello, World!";
console.log(str.indexOf("World")); // 7
```

replace(searchValue, newValue)

```
let str = "Hello, World!";
console.log(str.replace("World", "Everyone")); // "Hello, Everyone!"
```

slice(beginIndex, endIndex)

```
let str = "Hello, World!";
console.log(str.slice(7, 12)); // "World"
```

split(separator, limit)

```
let str = "Hello, World!";
console.log(str.split(", ")); // ["Hello", "World!"]
```

toLowerCase()

```
let str = "Hello, World!";
console.log(str.toLowerCase()); // "hello, world!"
```

toUpperCase()

```
let str = "Hello, World!";
console.log(str.toUpperCase()); // "HELLO, WORLD!"
```

trim()

```
let str = "   Hello, World!   ";
console.log(str.trim()); // "Hello, World!"
```

2. Array Functions

concat(array2, array3, ..., arrayX)

```
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
console.log(arr1.concat(arr2)); // [1, 2, 3, 4, 5, 6]
```

every(callback)

```
let arr = [1, 2, 3, 4, 5];
console.log(arr.every(num => num < 6)); // true
```

filter(callback)

```
let arr = [1, 2, 3, 4, 5];
console.log(arr.filter(num => num % 2 === 0)); // [2, 4]
```

forEach(callback)

```
let arr = [1, 2, 3, 4, 5];
arr.forEach(num => console.log(num * 2)); // 2, 4, 6, 8, 10
```

indexOf(searchElement, fromIndex)

```
let arr = [1, 2, 3, 4, 5];
console.log(arr.indexOf(3)); // 2
```

join(separator)

```
let arr = ["Hello", "World"];
console.log(arr.join(", ")); // "Hello, World"
```

3. **Math Functions**

abs(x)

```
console.log(Math.abs(-10)); // 10
```

ceil(x)

```
console.log(Math.ceil(4.2)); // 5
```

floor(x)

```
console.log(Math.floor(4.9)); // 4
```

4. **Date Functions**

getDate()

```
let date = new Date();
console.log(date.getDate()); // Current day of the month
```

getDay()

```
let date = new Date();
console.log(date.getDay()); // Current day of the week (0-6, 0=Sunday)
```

getFullYear()

```
let date = new Date();
console.log(date.getFullYear()); // Current year
```

getMonth()

```
let date = new Date();
console.log(date.getMonth()); // Current month (0-11, 0=January)
```

5. **Number Functions**

isFinite(value)

```
console.log(Number.isFinite(10)); // true
console.log(Number.isFinite(Infinity)); // false
```

isInteger(value)

```
console.log(Number.isInteger(10)); // true
console.log(Number.isInteger(10.5)); // false
```

isNaN(value)

```
console.log(Number.isNaN(NaN)); // true
console.log(Number.isNaN(10)); // false
```

parseFloat(string)

```
console.log(Number.parseFloat("10.5")); // 10.5
```

parseInt(string, radix)

```
console.log(Number.parseInt("10", 10)); // 10
console.log(Number.parseInt("10", 2)); // 2 (binary)
```

These examples cover a wide range of JavaScript's built-in functions, showcasing how they can be used in various contexts.

class

In JavaScript, classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JavaScript are built on prototypes but also have some syntax and semantics that are not shared with ES5 classalike semantics.

Here's the syntax for classes in JavaScript:

```
class ClassName {
 constructor() {
   // constructor is a special method used to create and initialize an
object
 }
 method1() {
   // method1 is a function within a class
 }
 method2() {
   // method2 is another function within a class
 }
}
```

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.

Here's an example of how to define a class and create an object using that class:

```
// Define a class
class Rectangle {
 // The constructor initializes the properties
 constructor(height, width) {
   this.height = height;
   this.width = width;
 }
 // Method to return the area
 area() {
   return this.height * this.width;
 }
 // Method to return the perimeter
 perimeter() {
   return 2 * (this.height + this.width);
 }
}
// Create an instance of the class
const myRectangle = new Rectangle(10, 20);
console.log(myRectangle.area());        // Outputs: 200
console.log(myRectangle.perimeter());    // Outputs: 60
```

In this example, **Rectangle** is a class and **myRectangle** is an object (an instance of **Rectangle**). The **new** keyword is used to create an instance of the class, which means a new object is created from the class, and the constructor of the class is called automatically.

Classes support inheritance, meaning you can create a new class that inherits properties and methods from another class. This is done using the **extends** keyword.

```
class Square extends Rectangle {
 constructor(sideLength) {
   super(sideLength, sideLength);
 }
}
const mySquare = new Square(10);
console.log(mySquare.area());        // Outputs: 100
console.log(mySquare.perimeter());    // Outputs: 40
```

In this code, **Square** is a new class that inherits from **Rectangle**. The **super** keyword is used to call the constructor of the superclass. The **mySquare** object is an instance of the **Square** class, and it can use the **area** and **perimeter** methods defined in **Rectangle**.

dom

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as a tree of nodes. Each node represents part of the document (such as an element, attribute, or text).

Here are some fundamental concepts and examples of working with the DOM in JavaScript:

Accessing DOM Elements

getElementById()

```
<!DOCTYPE html>
<html>
<body>
 <p id="myParagraph">Hello, World!</p>
 <script>
   let paragraph = document.getElementById("myParagraph");
   console.log(paragraph.innerText); // "Hello, World!"
 </script>
</body>
</html>
```

getElementsByClassName()

```
<!DOCTYPE html>
<html>
<body>
 <p class="myClass">Hello</p>
 <p class="myClass">World</p>
 <script>
   let elements = document.getElementsByClassName("myClass");
   for (let i = 0; i < elements.length; i++) {
     console.log(elements[i].innerText);
   }
   // "Hello"
   // "World"
 </script>
</body>
</html>
```

getElementsByTagName()

```
<!DOCTYPE html>
<html>
<body>
 <p>Paragraph 1</p>
 <p>Paragraph 2</p>
 <script>
   let paragraphs = document.getElementsByTagName("p");
   for (let i = 0; i < paragraphs.length; i++) {
     console.log(paragraphs[i].innerText);
   }
   // "Paragraph 1"
   // "Paragraph 2"
 </script>
```

```
</body>
</html>
```

querySelector()

```
<!DOCTYPE html>
<html>
<body>
 <p id="myParagraph">Hello, World!</p>
 <script>
   let paragraph = document.querySelector("#myParagraph");
   console.log(paragraph.innerText); // "Hello, World!"
 </script>
</body>
</html>
```

querySelectorAll()

```
<!DOCTYPE html>
<html>
<body>
 <p class="myClass">Hello</p>
 <p class="myClass">World</p>
 <script>
   let elements = document.querySelectorAll(".myClass");
   elements.forEach(element => {
     console.log(element.innerText);
   });
   // "Hello"
   // "World"
 </script>
</body>
</html>
```

Manipulating DOM Elements

Changing Content

```
<!DOCTYPE html>
<html>
<body>
 <p id="myParagraph">Hello, World!</p>
 <button onclick="changeContent()">Change Content</button>
 <script>
   function changeContent() {
     document.getElementById("myParagraph").innerText = "Content
Changed!";
   }
 </script>
</body>
</html>
```

Changing Styles

```
<!DOCTYPE html>
<html>
<body>
 <p id="myParagraph">Hello, World!</p>
 <button onclick="changeStyle()">Change Style</button>
 <script>
   function changeStyle() {
     document.getElementById("myParagraph").style.color = "red";
     document.getElementById("myParagraph").style.fontSize = "20px";
   }
 </script>
</body>
</html>
```

Adding and Removing Elements

```
<!DOCTYPE html>
<html>
<body>
 <ul id="myList">
   <li>Item 1</li>
   <li>Item 2</li>
 </ul>
 <button onclick="addItem()">Add Item</button>
 <button onclick="removeItem()">Remove Item</button>
 <script>
   function addItem() {
     let li = document.createElement("li");
     li.innerText = "New Item";
     document.getElementById("myList").appendChild(li);
   }
   function removeItem() {
     let list = document.getElementById("myList");
     if (list.hasChildNodes()) {
       list.removeChild(list.lastChild);
     }
   }
 </script>
</body>
</html>
```

Event Handling

Adding Event Listeners

```
<!DOCTYPE html>
<html>
<body>
 <p id="myParagraph">Click me!</p>
 <script>
   document.getElementById("myParagraph").addEventListener("click",
function() {
     alert("Paragraph clicked!");
   });
 </script>
</body>
</html>
```

Removing Event Listeners

```
<!DOCTYPE html>
<html>
<body>
 <p id="myParagraph">Click me!</p>
 <button id="removeListener">Remove Event Listener</button>
 <script>
   function clickHandler() {
     alert("Paragraph clicked!");
   }
   document.getElementById("myParagraph").addEventListener("click",
clickHandler);
   document.getElementById("removeListener").addEventListener("click",
function() {
     document.getElementById("myParagraph").removeEventListener("click",
clickHandler);
   });
 </script>
</body>
</html>
```

These examples provide a foundational understanding of interacting with and manipulating the DOM using JavaScript. There are many more aspects and methods within the DOM API that can be explored as needed for more complex interactions and manipulations.

custom

Let's create custom JavaScript functions that mimic the behavior of some of these built-in functions. This can be a useful exercise to understand how these functions work under the hood.

1. Custom String Functions

charAt(index)

```
function customCharAt(str, index) {
 return str[index] || '';
}
console.log(customCharAt("Hello, World!", 0)); // "H"
```

concat(string2, string3, ..., stringX)

```
function customConcat(...strings) {
 return strings.join('');
}
console.log(customConcat("Hello", ", ", "World", "!")); // "Hello,
World!"
```

includes(searchString, position)

```
function customIncludes(str, searchString, position = 0) {
 return str.indexOf(searchString, position) !== -1;
}
console.log(customIncludes("Hello, World!", "World")); // true
```

indexOf(searchValue, fromIndex)

```
function customIndexOf(str, searchValue, fromIndex = 0) {
 for (let i = fromIndex; i < str.length; i++) {
   if (str.substring(i, i + searchValue.length) === searchValue) {
     return i;
   }
 }
 return -1;
}
console.log(customIndexOf("Hello, World!", "World")); // 7
```

replace(searchValue, newValue)

```
function customReplace(str, searchValue, newValue) {
 return str.split(searchValue).join(newValue);
}
console.log(customReplace("Hello, World!", "World", "Everyone")); //
"Hello, Everyone!"
```

slice(beginIndex, endIndex)

```
function customSlice(str, beginIndex, endIndex = str.length) {
 let result = '';
```

```
  for (let i = beginIndex; i < endIndex; i++) {
    result += str[i];
  }
  return result;
}
console.log(customSlice("Hello, World!", 7, 12)); // "World"
```

split(separator, limit)

```
function customSplit(str, separator, limit) {
  let result = [];
  let temp = '';
  for (let i = 0; i < str.length; i++) {
    if (str[i] === separator) {
      result.push(temp);
      temp = '';
      if (result.length === limit) {
        return result;
      }
    } else {
      temp += str[i];
    }
  }
  result.push(temp);
  return result;
}
console.log(customSplit("Hello, World!", ", ")); // ["Hello", "World!"]
```

toLowerCase()

```
function customToLowerCase(str) {
  let result = '';
  for (let i = 0; i < str.length; i++) {
    let charCode = str.charCodeAt(i);
    if (charCode >= 65 && charCode <= 90) {
      result += String.fromCharCode(charCode + 32);
    } else {
      result += str[i];
    }
  }
  return result;
}
console.log(customToLowerCase("Hello, World!")); // "hello, world!"
```

toUpperCase()

```
function customToUpperCase(str) {
  let result = '';
  for (let i = 0; i < str.length; i++) {
    let charCode = str.charCodeAt(i);
    if (charCode >= 97 && charCode <= 122) {
      result += String.fromCharCode(charCode - 32);
    } else {
      result += str[i];
    }
  }
  return result;
}
console.log(customToUpperCase("Hello, World!")); // "HELLO, WORLD!"
```

2. **Custom Array Functions**

concat(array2, array3, ..., arrayX)

```
function customArrayConcat(...arrays) {
  let result = [];
  arrays.forEach(array => {
    array.forEach(item => result.push(item));
  });
  return result;
}
console.log(customArrayConcat([1, 2, 3], [4, 5, 6])); // [1, 2, 3, 4, 5,
6]
```

every(callback)

```
function customEvery(array, callback) {
  for (let i = 0; i < array.length; i++) {
    if (!callback(array[i], i, array)) {
      return false;
    }
  }
  return true;
}
console.log(customEvery([1, 2, 3, 4, 5], num => num < 6)); // true
```

3. **Custom Math Functions**

random()

```
function customRandom() {
  return Math.random();
}
```

```
console.log(customRandom()); // Random number between 0 and 1
```

round(x)

```
function customRound(x) {
 return (x % 1 >= 0.5) ? Math.ceil(x) : Math.floor(x);
}
console.log(customRound(4.5)); // 5
```

sqrt(x)

```
function customSqrt(x) {
 if (x < 0) return NaN;
 let approx = x / 2;
 let betterApprox;
 while (true) {
   betterApprox = (approx + x / approx) / 2;
   if (Math.abs(betterApprox - approx) < 0.000001) break;
   approx = betterApprox;
 }
 return betterApprox;
}
console.log(customSqrt(16)); // 4
```

min(x, y, ..., z)

```
function customMin(...numbers) {
 let minNum = numbers[0];
 for (let i = 1; i < numbers.length; i++) {
   if (numbers[i] < minNum) {
     minNum = numbers[i];
   }
 }
 return minNum;
}
console.log(customMin(1, 2, 3)); // 1
```

4. **Custom Date Functions**

getDate()

```
function customGetDate(date) {
 return date.getDate();
}
let date1 = new Date();
console.log(customGetDate(date1)); // Current day of the month
```

setDate(day)

```
function customSetDate(date, day) {
 date.setDate(day);
 return date;
}
let date9 = new Date();
console.log(customSetDate(date9, 15)); // Date with day set to 15
```

setFullYear(year, month, day)

```
function customSetFullYear(date, year, month, day) {
 date.setFullYear(year, month, day);
 return date;
}
let date10 = new Date();
console.log(customSetFullYear(date10, 2024, 5, 15)); // Date set to June
15, 2024
```

setHours(hour, min, sec, millisec)

```
function customSetHours(date, hour, min, sec, millisec) {
 date.setHours(hour, min, sec, millisec);
 return date;
}
let date11 = new Date();
console.log(customSetHours(date11, 15, 30, 0, 0)); // Date with time set
to 15:30:00.000
```

5. **Custom Number Functions**

isInteger(value)

```
function customIsInteger(value) {
 return typeof value === 'number' && isFinite(value) && Math.floor(value)
=== value;
}
console.log(customIsInteger(10)); // true
console.log(customIsInteger(10.5)); // false
```

isNaN(value)

```
function customIsNaN(value) {
 return value !== value;
}
```

```
console.log(customIsNaN(NaN)); // true
console.log(customIsNaN(10)); // false
```

These custom functions are simplified versions of the built-in JavaScript functions, demonstrating the core logic behind them. They may not cover all edge cases and nuances of the built-in versions but should provide a good understanding of how they work.

Oop

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code: data in the form of fields (often known as properties or attributes), and code, in the form of procedures (often known as methods).

In JavaScript, OOP can be implemented using classes and objects, as I've described in the previous messages. However, it's important to note that JavaScript uses prototypal inheritance, which is different from the classical inheritance pattern found in many other programming languages.

Here are the main principles of OOP:

Encapsulation: Encapsulation is the bundling of data and the methods that operate on that data into a single unit, commonly referred to as an object. This restricts direct access to some of an object's components and can prevent the accidental modification of data. To achieve encapsulation in JavaScript, you use classes and the **constructor** keyword to bind data and methods.

```
class Car {
   constructor(brand) {
       this.carname = brand; // property
   }
   present() { // method
       return 'I have a ' + this.carname;
   }
}
```

Inheritance: This is a mechanism where you can to derive a class from another class for a hierarchy of classes that share a set of attributes and methods. You can use it to re-use, extend, or modify the attributes and behaviors that are defined in other classes. In JavaScript, you use the **extends** keyword to implement inheritance between classes.

```
class Model extends Car {
   constructor(brand, mod) {
       super(brand);
       this.model = mod;
   }
   show() {
       return this.present() + ', it is a ' + this.model;
   }
}
```

Polymorphism: Polymorphism is a core concept in OOP that allows objects of different classes to be treated as objects of a common super class. It's the ability of different objects to respond in a unique way to the same method call. In JavaScript, polymorphism can be achieved using inheritance and method overriding. Here's an example that demonstrates polymorphism:

```
// A base class
class Animal {
 speak() {
   return 'Generic animal sound!';
 }
}
// A derived class
class Dog extends Animal {
 speak() {
   return 'Woof!'; // This is a Dog's unique way to handle the 'speak'
method
 }
}
// Another derived class
class Cat extends Animal {
 speak() {
   return 'Meow!'; // This is a Cat's unique way to handle the 'speak'
method
 }
}
// A function that can work with any kind of animal
function triggerSound(animal) {
 console.log(animal.speak()); // Doesn't need to know the type of animal
to call the 'speak' method
}
// Creating instances
const myDog = new Dog();
const myCat = new Cat();
// The 'triggerSound' function doesn't need to know the type of animal to
call the 'speak' method
triggerSound(myDog); // Outputs: Woof!
triggerSound(myCat); // Outputs: Meow!
// Here, 'triggerSound' is able to process objects of different types
(Dog and Cat),
// and each type is able to respond to the 'speak' method call in its own
way.
```

In this example, **Dog** and **Cat** both override the **speak** method provided by **Animal**, their superclass. The **triggerSound** function is an example of polymorphism where it can accept any object that has a **speak** method. It doesn't need to know whether the object is a **Dog** or a **Cat**, just that it can **speak**. So, when **triggerSound** calls **speak** on a **Dog** or **Cat** object, it makes the corresponding animal sound. This is polymorphism in action, where different classes have a different implementation of the same method.

Abstraction: Abstraction means hiding the complex reality while exposing only the necessary parts. It can be thought of as a form of data protection or secure data. JavaScript doesn't have traditional abstract classes, but you can achieve abstraction by using class constructors or by returning an object from a function.

Here's an example of how you might use these principles in a JavaScript program:

```
// Inheritance and Encapsulation
class Animal {
   constructor(name) {
       this.name = name;  // Encapsulating properties
   }
   speak() {   // Encapsulating method
       console.log(`${this.name} makes a noise.`);
   }
}
class Dog extends Animal {  // Inheritance
   speak() {
       console.log(`${this.name} barks.`);
   }
}
let dog = new Dog('Rex');
dog.speak(); // Output: Rex barks.
// Abstraction (not fully supported, but we can simulate)
function CreateAbstractAnimal() {
   return {
       abstractMethod() {
           throw new Error("This is an abstract method!");
       }
   }
}
// Polymorphism (JavaScript's dynamic typing supports it inherently)
function doSomething(animal) {
   animal.speak();
}
doSomething(new Animal('Cat'));  // Output: Cat makes a noise.
doSomething(new Dog('Bulldog')); // Output: Bulldog barks.
```

In the example above, the **Animal** and **Dog** classes illustrate encapsulation and inheritance. The **CreateAbstractAnimal** function shows how you might simulate an abstract class in JavaScript, which is a part of the abstraction principle. The **doSomething** function is an example of polymorphism; it's a single function that can operate on any type of animal. This is possible because of JavaScript's dynamic typing system.

2d

To draw 2D shapes in JavaScript, you can use the **Canvas** element and its associated **CanvasRenderingContext2D** object. The **Canvas** element is an HTML element that you can use to draw graphics using JavaScript. Here's an example of how you can use it to draw a rectangle:

```
<canvas id="canvas" width="200" height="100"></canvas>
<script>
 // get the canvas element and its context
 let canvas = document.getElementById('canvas');
 let ctx = canvas.getContext('2d');
 // draw a rectangle
 ctx.fillStyle = 'red';
 ctx.fillRect(10, 10, 180, 80);
</script>
```

This code will create a canvas element with the ID "canvas", and it will use the **fillRect()** method of the **CanvasRenderingContext2D** object to draw a filled rectangle with the color red. The rectangle will be positioned at (10, 10) and will have a width of 180 and a height of 80.

You can use the **CanvasRenderingContext2D** object to draw other shapes as well, such as circles and lines. Here's an example of how you can use it to draw a circle:

```
<canvas id="canvas" width="200" height="100"></canvas>
<script>
 // get the canvas element and its context
 let canvas = document.getElementById('canvas');
 let ctx = canvas.getContext('2d');
 // draw a circle
 ctx.beginPath();
 ctx.arc(100, 50, 40, 0, 2 * Math.PI);
 ctx.fillStyle = 'blue';
 ctx.fill();
</script>
```

This code will create a canvas element with the ID "canvas", and it will use the **arc()** method of the **CanvasRenderingContext2D** object to draw a filled circle with the color blue. The circle will be centered at (100, 50) and will have a radius of 40.

3d

This example demonstrates how to create a 3D rotating cube using Three.js, a popular JavaScript library for 3D graphics. Here's the complete HTML and JavaScript code:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>3D Cube with Three.js</title>
   <script
src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"><
/script>
</head>
<body>
<script>
   // Set up the scene, camera, and renderer
   var scene = new THREE.Scene();
   var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
   var renderer = new THREE.WebGLRenderer();
   renderer.setSize(window.innerWidth, window.innerHeight);
```

```
    document.body.appendChild(renderer.domElement);
    // Create a cube and add it to the scene
    var geometry = new THREE.BoxGeometry(1, 1, 1);
    var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
    var cube = new THREE.Mesh(geometry, material);
    scene.add(cube);
    // Position the camera
    camera.position.z = 5;
    // Create an animation loop
    var animate = function () {
        requestAnimationFrame(animate);
        // Rotate the cube
        cube.rotation.x += 0.01;
        cube.rotation.y += 0.01;
        // Render the scene from the perspective of the camera
        renderer.render(scene, camera);
    };
    // Start the animation loop
    animate();
</script>
</body>
</html>
```

Explanation:

Setting up the Scene:

- var scene = new THREE.Scene();: Creates a new scene.
- var camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);: Sets up a perspective camera with a 75-degree field of view, an aspect ratio matching the window's aspect ratio, and clipping planes at 0.1 and 1000 units from the camera.

Creating the Renderer:

- var renderer = new THREE.WebGLRenderer();: Creates a WebGL renderer.
- renderer.setSize(window.innerWidth, window.innerHeight);: Sets the renderer to fill the window.
- document.body.appendChild(renderer.domElement);: Adds the renderer's output (a canvas element) to the HTML document.

Creating a Cube:

- var geometry = new THREE.BoxGeometry(1, 1, 1);: Defines the geometry for a cube.
- var material = new THREE.MeshBasicMaterial({color: 0x00ff00});: Defines a basic material with a green color.
- var cube = new THREE.Mesh(geometry, material);: Creates a mesh from the geometry and material.
- scene.add(cube);: Adds the cube to the scene.

Positioning the Camera:

- camera.position.z = 5;: Moves the camera back 5 units so that the cube is in view.

Animating the Cube:

- var animate = function () { ... };: Defines the animation loop.
- requestAnimationFrame(animate);: Schedules the animate function to be called on the next frame.
- cube.rotation.x += 0.01;: Rotates the cube slightly around the x-axis.
- cube.rotation.y += 0.01;: Rotates the cube slightly around the y-axis.
- renderer.render(scene, camera);: Renders the scene from the perspective of the camera.

Starting the Animation:

- animate();: Starts the animation loop.

This code sets up a basic 3D scene with a rotating cube, which is a good starting point for exploring more complex 3D graphics using Three.js.

comments

Comments in JavaScript are used to explain code and make it more readable. They can also be used to prevent execution when testing alternative code. There are two types of comments in JavaScript:

Single-line comments: These comments start with //. Any text between // and the end of the line is ignored by JavaScript (will not be executed).

```
// This is a single-line comment
let x = 5; // This is another single-line comment
```

Multi-line comments: These comments start with /* and end with */. Any text between /* and */ will be ignored by JavaScript.

```
/*
 This is a multi-line comment
 It can span multiple lines
*/
let y = 10;
```

Best Practices for Using Comments

- **Clarity**: Use comments to explain the "why" behind your code, not the "what". The code itself should be self-explanatory when it comes to the "what".
- **Conciseness**: Keep comments concise and to the point.
- **Update**: Ensure comments are updated when the code changes to avoid discrepancies between code and comments.
- **Avoid Redundancy**: Do not state the obvious in comments. For instance, avoid comments like // increment x by 1 for code like x++;.

Here's an example demonstrating these practices:

```
// Calculate the factorial of a number
```

```
function factorial(n) {
   // If the number is less than 0, reject it
   if (n < 0) return -1;
   // If the number is 0, its factorial is 1
   if (n === 0) return 1;
   // Otherwise, recursively calculate the factorial
   return n * factorial(n - 1);
}
/*
 Example usage:
 factorial(5); // returns 120
*/
```

Using comments effectively can greatly enhance the maintainability and readability of your code.

data type

In programming, a data type is a classification that specifies which type of value a variable or expression can hold. JavaScript has several built-in data types, including:

- String: a sequence of characters, enclosed in single or double quotes. For example: "hello" or 'world'.

```
// String
let name = "John";
console.log(name); // Output: "John"
```

- Number: numeric values, including integers and floating-point values. For example: 42 or 3.14.

```
// Number
let age = 30;
console.log(age); // Output: 30
```

- Boolean: a value that can be either true or false.

```
// Boolean
let isStudent = false;
console.log(isStudent); // Output: false
```

- Undefined: a value that indicates a variable has been declared but has not been assigned a value.

```
// Undefined
let address;
console.log(address); // Output: undefined
```

- Symbol: a unique and immutable data type that can be used as an identifier for object properties.

```
// Symbol
```

```
let sym = Symbol("uniqueId");
console.log(sym); // Output: Symbol(uniqueId)
```

- Object: a collection of properties and methods.

```
// Object
let person = {
 name: "John",
 age: 30
};
console.log(person); // Output: { name: "John", age: 30 }
```

- Array: a special type of object that holds a list of values.

```
// Array
let colors = ["red", "green", "blue"];
console.log(colors); // Output: ["red", "green", "blue"]
```

- Function: a block of code that can be executed when called.

```
// Function
function add(a, b) {
 return a + b;
}
console.log(add(2, 3)); // Output: 5
```

JavaScript also supports a special data type called **null**, which represents the absence of a value or object.

It's also important to note that JavaScript is a loosely typed language, which means that variables do not have to be declared with a specific data type, and their data type can change at runtime.

Event handling

Event handling in JavaScript enables interactive web pages and is essential in client-side web development. JavaScript can be used to detect when an event occurs, such as when a user clicks a button, moves the mouse, or submits a form, and then run a specified function or block of code in response to that event. Below is a general overview of event handling in JavaScript:

Basic Steps in Event Handling:

1. **Select an Element**: Choose the HTML element to which the event will be attached.
2. **Define a Function**: Create a function that will run when the event occurs.
3. **Attach an Event**: Use JavaScript to attach the event to the selected HTML element.

Basic Syntax:

```
element.eventType = functionName;
```

Example 1: Inline Event Handling

HTML:

```
<button onclick="alert('Button Clicked!')">Click Me!</button>
```

In the above example, an inline **onclick** event is used to show an alert when the button is clicked.

Example 2: Traditional DOM Event Handling

JavaScript:

```
let button = document.getElementById('myButton');
button.onclick = function() {
   alert('Button Clicked!');
};
```

HTML:

```
<button id="myButton">Click Me!</button>
```

In this example, the **onclick** event is attached to a button element using JavaScript.

Event Listeners:

A more modern approach to handle events in JavaScript is using event listeners.

Syntax:

```
element.addEventListener('eventType', function/eventHandler);
```

Example: Using Event Listeners

JavaScript:

```
let button = document.getElementById('myButton');
button.addEventListener('click', function() {
   alert('Button Clicked!');
});
```

HTML:

```
<button id="myButton">Click Me!</button>
```

Event Object:

The event object is a special object created when an event occurs. It contains properties and methods related to the event.

Example using the Event Object:

```
button.addEventListener('click', function(event) {
   alert('Button Clicked! Event type is: ' + event.type);
});
```

Event Propagation:

- **Event Bubbling**: The event starts from the target element that triggered the event and bubbles up to the root of the DOM tree.
- **Event Capturing** (or **Event Capturing**): The event starts from the root and trickles down to the target element.

You can choose whether to handle an event in the capturing or bubbling phase by using the third parameter of **addEventListener**:

```
element.addEventListener('click', functionHandler, useCapture);
```

- **useCapture** is optional and is **false** by default (bubbling phase). Set it to **true** to handle the event in the capturing phase.

Prevent Default Behavior:

To prevent the default behavior of an event, use the **preventDefault()** method of the event object.

Example:

```
let form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
   event.preventDefault();
   alert('Form Submitted!');
});
```

Event Delegation:

Event delegation refers to the practice of using a single event listener to manage all of a particular type of event for child elements.

Example:

```
let ul = document.getElementById('myList');
ul.addEventListener('click', function(event) {
   alert('Item Clicked: ' + event.target.textContent);
});
```

HTML:

```
<ul id="myList">
   <li>Item 1</li>
   <li>Item 2</li>
   <li>Item 3</li>
</ul>
```

Note:

Ensure to add your script tag just before the closing body tag **</body>** or use **DOMContentLoaded** event to make sure your JavaScript runs only after the DOM has fully loaded, especially when your script tags are in the **<head>**.

```
document.addEventListener('DOMContentLoaded', function() {
   // Your code here
});
```

This is a broad overview of event handling in JavaScript. Specific use cases may require additional handling and more advanced techniques.

Arrays

Arrays in JavaScript are a data structure that can hold a collection of values of any data type, such as numbers, strings, objects, and even other arrays. Each element in an array is assigned a unique index starting from 0, which can be used to access or modify the element.

Here is an example of creating an array in JavaScript:

```
let myArray = [1, 2, 3, 4, 5];
```

This creates an array with 5 elements, each containing a number.

We can access individual elements in an array using square bracket notation and the index of the element we want to access. For example:

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

We can also modify the elements in an array by assigning new values using the same square bracket notation:

```
myArray[3] = 10;
console.log(myArray); // Output: [1, 2, 3, 10, 5]
```

We can add new elements to an array using the **push()** method:

```
myArray.push(6);
console.log(myArray); // Output: [1, 2, 3, 10, 5, 6]
```

We can also remove elements from an array using the **pop()** method, which removes the last element:

```
myArray.pop();
console.log(myArray); // Output: [1, 2, 3, 10, 5]
```

Arrays in JavaScript also have many other built-in methods, such as **slice()**, **splice()**, **concat()**, and **reverse()**, which can be used to manipulate arrays in a variety of ways.

Here is an example of using some of these methods:

```
let fruits = ['apple', 'banana', 'cherry', 'date'];
// Add an element to the beginning of the array
fruits.unshift('apricot');
console.log(fruits); // Output: ['apricot', 'apple', 'banana', 'cherry',
'date']
// Remove the second and third elements from the array
fruits.splice(1, 2);
console.log(fruits); // Output: ['apricot', 'cherry', 'date']
// Create a new array with the last two elements of the original array
let newFruits = fruits.slice(-2);
console.log(newFruits); // Output: ['cherry', 'date']
```

Arrays are a fundamental data structure in JavaScript and are used extensively in programming.

Linked lists

A linked list is a data structure that consists of a sequence of nodes, each of which contains a value and a reference to the next node in the sequence. Linked lists are a useful data structure when we need to store and manipulate data in a dynamic way, meaning that the size of the data may change over time. In a linked list, each node is not required to be contiguous in memory.

The first node of a linked list is called the head, and the last node is called the tail. The tail node has a reference to null, indicating the end of the list. Linked lists can be used to implement stacks, queues, and other data structures.

Here is an example of creating a linked list in JavaScript:

```
class Node {
 constructor(value) {
   this.value = value;
   this.next = null;
 }
}
class LinkedList {
 constructor() {
   this.head = null;
   this.tail = null;
   this.length = 0;
 }
 append(value) {
   const newNode = new Node(value);
   if (!this.head) {
     this.head = newNode;
     this.tail = newNode;
   } else {
     this.tail.next = newNode;
     this.tail = newNode;
   }
   this.length++;
 }
 prepend(value) {
   const newNode = new Node(value);
   if (!this.head) {
     this.head = newNode;
     this.tail = newNode;
   } else {
     newNode.next = this.head;
     this.head = newNode;
   }
   this.length++;
 }
 insert(index, value) {
   if (index === 0) {
     this.prepend(value);
     return;
   }
   if (index >= this.length) {
     this.append(value);
     return;
   }
   const newNode = new Node(value);
   const leader = this.traverseToIndex(index - 1);
   const temp = leader.next;
   leader.next = newNode;
   newNode.next = temp;
   this.length++;
 }
 remove(index) {
   if (index === 0) {
     this.head = this.head.next;
     this.length--;
     return;
   }
   const leader = this.traverseToIndex(index - 1);
   leader.next = leader.next.next;
   if (index === this.length - 1) {
     this.tail = leader;
   }
   this.length--;
 }
 traverseToIndex(index) {
   let currentNode = this.head;
   let counter = 0;
   while (counter !== index) {
     currentNode = currentNode.next;
     counter++;
   }
   return currentNode;
 }
}
```

In this example, the **LinkedList** class is defined with several methods, including **append()**, **prepend()**, **insert()**, and **remove()**. The **append()** method adds a new node to the end of the list, while the **prepend()** method adds a new node to the beginning of the list. The **insert()** method adds a new node at a specified index in the list, while the **remove()** method removes a node at a specified index.

The **Node** class represents a single node in the list and has a **value** property to store the value of the node and a **next** property to reference the next node in the list. The **LinkedList** class has a **head** property to reference the first node in the list and a **tail** property to reference the last node in the list. The **length** property keeps track of the number of nodes in the list.

The **traverseToIndex()** method is used internally to find the node at a specified index in the list. This method starts at the **head** of the list and iterates through the nodes until the specified index is reached Here are some examples of how to use the **LinkedList** class:

```
const list = new LinkedList();
list.append(1);
list.append(2);
list.append(3);
console.log(list); // Output: { head: Node { value: 1, next: Node {
value: 2, next: Node { value: 3, next: null } } }, tail: Node { value: 3,
next: null }, length: 3 }
list.prepend(0);
console.log(list); // Output: { head: Node { value: 0, next: Node {
value: 1, next: Node { value: 2, next: Node { value: 3, next: null } } }
}, tail: Node { value: 3, next: null }, length: 4 }
list.insert(2, 1.5);
console.log(list); // Output: { head: Node { value: 0, next: Node {
value: 1, next: Node { value: 1.5, next: Node { value: 2, next: Node {
value: 3, next: null } } } } }, tail: Node { value: 3, next: null },
length: 5 }
list.remove(3);
console.log(list); // Output: { head: Node { value: 0, next: Node {
value: 1, next: Node { value: 1.5, next: Node { value: 3, next: null } }
} }, tail: Node { value: 3, next: null }, length: 4 }
```

In this example, we create a new **LinkedList** object and add several nodes to it using the **append()** method. We then prepend a new node to the list using the **prepend()** method, insert a new node at index 2 using the **insert()** method, and remove a node at index 3 using the **remove()** method. The output shows the resulting **LinkedList** object after each operation.

Linked lists can be used in a variety of applications, such as implementing caches, file systems, and web browsers.

Stacks

A stack is a data structure that follows the LIFO (last-in, first-out) principle, which means that the last element added to the stack will be the first one to be removed. A stack can be implemented using an array or a linked list. Stacks are used in many algorithms and applications, such as expression evaluation, backtracking, and parsing.

Here is an example of creating a stack using an array in JavaScript:

```
class Stack {
 constructor() {
   this.items = [];
 }
 push(item) {
   this.items.push(item);
 }
 pop() {
   if (this.isEmpty()) {
     return null;
   }
   return this.items.pop();
 }
 peek() {
   if (this.isEmpty()) {
     return null;
   }
   return this.items[this.items.length - 1];
 }
 isEmpty() {
   return this.items.length === 0;
 }
 size() {
   return this.items.length;
 }
 clear() {
   this.items = [];
 }
}
```

In this example, the **Stack** class is defined with several methods, including **push()**, **pop()**, **peek()**, **isEmpty()**, **size()**, and **clear()**. The **push()** method adds a new item to the top of the stack, while the **pop()** method removes and returns the top item from the stack. The **peek()** method returns the top item from the stack without removing it. The **isEmpty()** method returns true if the stack is empty, and false otherwise. The **size()** method returns the number of items in the stack. The **clear()** method removes all items from the stack.

Here are some examples of how to use the **Stack** class:

```
const stack = new Stack();
stack.push(1);
stack.push(2);
stack.push(3);
console.log(stack.size()); // Output: 3
console.log(stack.peek()); // Output: 3
console.log(stack.pop()); // Output: 3
console.log(stack.size()); // Output: 2
console.log(stack.isEmpty()); // Output: false
stack.clear();
console.log(stack.size()); // Output: 0
```

In this example, we create a new **Stack** object and add several items to it using the **push()** method. We then use the **size()**, **peek()**, **pop()**, and **isEmpty()** methods to access and manipulate the items in the stack, and the **clear()** method to remove all items from the stack.

Stacks can be used in a variety of applications, such as function call stacks, undo/redo functionality, and browser history.

Queues

A queue is a data structure that follows the FIFO (first-in, first-out) principle, which means that the first element added to the queue will be the first one to be removed. A queue can be implemented using an array or a linked list. Queues are used in many algorithms and applications, such as job scheduling, task processing, and network packet handling.

Here is an example of creating a queue using an array in JavaScript:

```
class Queue {
 constructor() {
   this.items = [];
 }
 enqueue(item) {
   this.items.push(item);
 }
 dequeue() {
   if (this.isEmpty()) {
     return null;
   }
   return this.items.shift();
 }
 peek() {
   if (this.isEmpty()) {
     return null;
   }
   return this.items[0];
 }
 isEmpty() {
   return this.items.length === 0;
 }
 size() {
   return this.items.length;
 }
 clear() {
   this.items = [];
 }
}
```

In this example, the **Queue** class is defined with several methods, including **enqueue()**, **dequeue()**, **peek()**, **isEmpty()**, **size()**, and **clear()**. The **enqueue()** method adds a new item to the back of the queue, while the **dequeue()** method removes and returns the front item from the queue. The **peek()** method returns the front item from the queue without removing it. The **isEmpty()** method returns true if the queue is empty, and false otherwise. The **size()** method returns the number of items in the queue. The **clear()** method removes all items from the queue.

Here are some examples of how to use the **Queue** class:

```
const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
console.log(queue.size()); // Output: 3
console.log(queue.peek()); // Output: 1
console.log(queue.dequeue()); // Output: 1
console.log(queue.size()); // Output: 2
console.log(queue.isEmpty()); // Output: false
queue.clear();
console.log(queue.size()); // Output: 0
```

In this example, we create a new **Queue** object and add several items to it using the **enqueue()** method. We then use the **size()**, **peek()**, **dequeue()**, and **isEmpty()** methods to access and manipulate the items in the queue, and the **clear()** method to remove all items from the queue.

Queues can be used in a variety of applications, such as message queues, printer queues, and web server request handling.

Trees

A tree is a hierarchical data structure that consists of nodes connected by edges. The first node in a tree is called the root, and the nodes below the root are called the children. Nodes with no children are called leaf nodes. Trees can be used to model hierarchical relationships, such as file systems, organizational charts, and HTML document object models.

Here is an example of creating a binary tree in JavaScript:

```
class Node {
 constructor(value) {
   this.value = value;
   this.left = null;
   this.right = null;
 }
}
class BinaryTree {
 constructor() {
   this.root = null;
 }
 insert(value) {
   const newNode = new Node(value);
   if (!this.root) {
     this.root = newNode;
     return;
   }
   let currentNode = this.root;
   while (true) {
     if (value < currentNode.value) {
       if (!currentNode.left) {
         currentNode.left = newNode;
         return;
       }
       currentNode = currentNode.left;
     } else {
       if (!currentNode.right) {
         currentNode.right = newNode;
         return;
       }
       currentNode = currentNode.right;
     }
   }
 }
 search(value) {
   let currentNode = this.root;
   while (currentNode) {
     if (value === currentNode.value) {
       return currentNode;
     }
     if (value < currentNode.value) {
       currentNode = currentNode.left;
     } else {
       currentNode = currentNode.right;
     }
   }
   return null;
 }
}
```

In this example, the **BinaryTree** class is defined with several methods, including **insert()** and **search()**. The **insert()** method adds a new node with the given value to the tree in the appropriate position according to the value. The **search()** method searches the tree for a node with the given value and returns the node if found, or null otherwise.

The **Node** class represents a single node in the tree and has a **value** property to store the value of the node and **left**and **right** properties to reference the left and right child nodes.

Here are some examples of how to use the **BinaryTree** class:

```
const tree = new BinaryTree();
tree.insert(5);
tree.insert(3);
tree.insert(7);
tree.insert(2);
tree.insert(4);
tree.insert(6);
tree.insert(8);
console.log(tree.search(4)); // Output: Node { value: 4, left: Node {
value: 2, left: null, right: null }, right: Node { value: 5, left:
[Node], right: [Node] } }
console.log(tree.search(9)); // Output: null
```

In this example, we create a new **BinaryTree** object and add several nodes to it using the **insert()** method. We then use the **search()** method to find a node with a specific value in the tree.

Trees can be used in a variety of applications, such as representing family trees, decision trees, and search trees.

Graphs

A graph is a data structure that consists of a set of vertices (also called nodes or points) and a set of edges that connect these vertices. Graphs can be used to model many real-world situations, such as social networks, road networks, and computer networks. There are many types of graphs, including directed and undirected graphs, weighted and unweighted graphs, and cyclic and acyclic graphs.

Here is an example of creating an undirected graph using an adjacency list representation in JavaScript:

```
class Graph {
 constructor() {
   this.vertices = {};
 }
 addVertex(vertex) {
   this.vertices[vertex] = [];
 }
 addEdge(vertex1, vertex2) {
   this.vertices[vertex1].push(vertex2);
   this.vertices[vertex2].push(vertex1);
 }
 removeVertex(vertex) {
   while (this.vertices[vertex].length) {
     const adjacentVertex = this.vertices[vertex].pop();
     this.removeEdge(vertex, adjacentVertex);
   }
   delete this.vertices[vertex];
 }
 removeEdge(vertex1, vertex2) {
   this.vertices[vertex1] = this.vertices[vertex1].filter(v => v !==
vertex2);
```

```
    this.vertices[vertex2] = this.vertices[vertex2].filter(v => v !==
vertex1);
  }
  depthFirstSearch(startVertex, visited = new Set()) {
    visited.add(startVertex);
    console.log(startVertex);
    for (const neighbor of this.vertices[startVertex]) {
      if (!visited.has(neighbor)) {
        this.depthFirstSearch(neighbor, visited);
      }
    }
  }
  breadthFirstSearch(startVertex) {
    const queue = [startVertex];
    const visited = new Set([startVertex]);
    while (queue.length > 0) {
      const vertex = queue.shift();
      console.log(vertex);
      for (const neighbor of this.vertices[vertex]) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }
  }
}
```

In this example, the **Graph** class is defined with several methods, including **addVertex()**, **addEdge()**, **removeVertex()**, **removeEdge()**, **depthFirstSearch()**, and **breadthFirstSearch()**. The **addVertex()** method adds a new vertex to the graph. The **addEdge()** method adds an undirected edge between two vertices. The **removeVertex()** method removes a vertex and all its incident edges from the graph. The **removeEdge()** method removes an edge between two vertices. The **depthFirstSearch()** method performs a depth-first search starting from a given vertex. The **breadthFirstSearch()** method performs a breadth-first search starting from a given vertex.

Here are some examples of how to use the **Graph** class:

```
const graph = new Graph();
graph.addVertex('A');
graph.addVertex('B');
graph.addVertex('C');
graph.addVertex('D');
graph.addVertex('E');
graph.addEdge('A', 'B');
graph.addEdge('B', 'C');
graph.addEdge('C', 'D');
graph.addEdge('D', 'E');
graph.addEdge('E', 'A');
graph.depthFirstSearch('A');
// Output: A, B, C, D, E
```

```
graph.breadthFirstSearch('A');
// Output: A, B, E, C, D
```

In this example, we create a new **Graph** object and add several vertices and edges to it using the **addVertex()** and **addEdge()** methods. We then use the **depthFirstSearch()** and **breadthFirstSearch()** methods to traverse the graph starting from a given vertex.

Graphs can be used in a variety of applications, such as route planning, recommendation systems, and social network analysis. Graph algorithms are also used in many fields, including computer science, operations research, and engineering.

Some common graph algorithms include:

- Dijkstra's algorithm: finds the shortest path between two vertices in a weighted graph
- Kruskal's algorithm: finds the minimum spanning tree of a weighted graph
- Breadth-first search: finds the shortest path between two vertices in an unweighted graph
- Depth-first search: traverses a graph in a depth-first order
- Topological sort: orders the vertices of a directed acyclic graph based on their dependencies
- Bellman-Ford algorithm: finds the shortest path between two vertices in a weighted graph with negative edges

Graphs are a powerful data structure and algorithmic tool that can be used to solve a wide range of problems.

Hash tables

A hash table is a data structure that uses a hash function to map keys to values. Hash tables provide fast insertion, deletion, and lookup of key-value pairs, with an average time complexity of O(1) for each operation. In JavaScript, objects can be used as hash tables, where the keys are strings and the values can be any type.

Here is an example of creating a hash table in JavaScript using an object:

```
const hashTable = {};
function hashFunction(key) {
 let hash = 0;
 for (let i = 0; i < key.length; i++) {
   const char = key.charCodeAt(i);
   hash = ((hash << 5) - hash) + char;
   hash |= 0;
 }
 return hash;
}
function insert(key, value) {
 const hashedKey = hashFunction(key);
 hashTable[hashedKey] = value;
}
function get(key) {
 const hashedKey = hashFunction(key);
 return hashTable[hashedKey];
}
```

```
function remove(key) {
 const hashedKey = hashFunction(key);
 delete hashTable[hashedKey];
}
```

In this example, the **hashTable** object is used to store the key-value pairs. The **hashFunction()** function is used to convert a string key into a numerical index that can be used to access the corresponding value in the hash table. The **insert()**, **get()**, and **remove()** functions are used to insert, retrieve, and remove key-value pairs from the hash table.

Here are some examples of how to use the hash table:

```
insert('John', 42);
insert('Jane', 'hello');
insert('Bob', { foo: 'bar' });
console.log(get('John')); // Output: 42
console.log(get('Jane')); // Output: 'hello'
console.log(get('Bob')); // Output: { foo: 'bar' }
remove('Jane');
console.log(get('Jane')); // Output: undefined
```

In this example, we insert several key-value pairs into the hash table using the **insert()** function. We then use the **get()** function to retrieve the values associated with the keys, and the **remove()** function to remove a key-value pair from the hash table.

Hash tables are used in many applications, such as caching, symbol tables, and database indexing. The efficiency of hash tables makes them a popular choice for many software systems.

Sets

In JavaScript, a Set is a built-in data structure that represents an unordered collection of unique values. Sets can be used to efficiently remove duplicates from an array, perform set operations such as union, intersection, and difference, and check if a value is a member of a set.

Here is an example of using a Set in JavaScript:

```
const mySet = new Set();
mySet.add(1);
mySet.add(2);
mySet.add(3);
console.log(mySet); // Output: Set { 1, 2, 3 }
mySet.add(2);
console.log(mySet); // Output: Set { 1, 2, 3 }
mySet.delete(2);
console.log(mySet); // Output: Set { 1, 3 }
console.log(mySet.has(1)); // Output: true
console.log(mySet.size); // Output: 2
mySet.clear();
console.log(mySet); // Output: Set {}
```

In this example, we create a new Set object called **mySet** using the **new Set()** constructor. We add several values to the set using the **add()** method, which only adds a value if it doesn't already exist in the set. We then use the **delete()**method to remove a value from the set and the **has()** method to check if a value is a member of the set. The **size**property returns the number of elements in the set. Finally, we use the **clear()** method to remove all elements from the set.

Sets can also be used to perform set operations, such as union, intersection, and difference. Here is an example of using the **Set** object to perform a union operation:

```
const set1 = new Set([1, 2, 3]);
const set2 = new Set([2, 3, 4]);
const unionSet = new Set([...set1, ...set2]);
console.log(unionSet); // Output: Set { 1, 2, 3, 4 }
```

In this example, we create two **Set** objects, **set1** and **set2**, and use the spread operator to create a new set that contains all the elements of both sets. The resulting set, **unionSet**, contains the elements 1, 2, 3, and 4.

Sets are a useful data structure for efficiently storing and manipulating collections of unique values in JavaScript.

Types of errors

There are several types of errors that can occur in JavaScript. Here are some of the most common ones:

Syntax errors: These occur when you have a mistake in your code that violates the rules of the language. For example, forgetting to close a quote, missing a semicolon, or using a reserved keyword.

```
// Example of a syntax error
let name = "John"; // Missing semicolon
console.log(name);
```

Runtime errors: These occur during the execution of your code. For example, when you try to access a variable that is not defined or when you call a method that does not exist.

```
// Example of a runtime error
let x;
console.log(x.toUpperCase()); // "toUpperCase" method does not exist for
"undefined"
```

Logical errors: These occur when your code does not produce the expected result, but it runs without generating an error. These types of errors can be difficult to detect and fix.

```
// Example of a logical error
function calculateTotal(price, quantity) {
 return price * quantity; // Forgot to add tax
}
let total = calculateTotal(100, 2);
```

```
console.log(total); // Output: 200 (but it should be 220 with 10% tax)
```

It's important to handle these errors properly in your code, so that you can provide a good user experience and prevent your application from crashing. One way to do this is to use a try-catch statement, which allows you to catch and handle errors gracefully.

```
// Example of a try-catch statement
try {
 let x;
 console.log(x.toUpperCase());
} catch (error) {
 console.log("An error occurred: " + error.message);
}
```

In this example, the code inside the **try** block generates a runtime error, but the catch block catches the error and logs a message to the console, preventing the application from crashing.

try-catch statements and their syntax

A try-catch statement is a JavaScript feature that allows you to handle errors gracefully in your code. The basic syntax of a try-catch statement is as follows:

```
try {
 // Code that might throw an error
} catch (error) {
 // Code that handles the error
}
```

Here's how it works:

1. The code inside the **try** block is executed.
2. If an error occurs while executing the **try** block, JavaScript immediately jumps to the **catch** block.
3. The **catch** block takes an **error** parameter, which contains information about the error that was thrown. You can use this information to handle the error in an appropriate way (e.g., logging an error message, displaying an error to the user, etc.).
4. After the **catch** block is executed, the program continues with the code that comes after the **try-catch** statement.

Here's an example of a try-catch statement:

```
try {
 let x;
 console.log(x.toUpperCase()); // This line will throw a runtime error
} catch (error) {
 console.log("An error occurred: " + error.message);
}
```

In this example, the **try** block contains code that tries to call the **toUpperCase()** method on an undefined variable. This will throw a runtime error, but the **catch** block will catch the error and log a message to the console, preventing the program from crashing.

Throwing and handling custom errors

In addition to the built-in errors in JavaScript, you can also create and throw your own custom errors. This can be useful when you need to handle specific cases in your application that are not covered by the standard errors.

Here's an example of how to create and throw a custom error:

```
class MyError extends Error {
 constructor(message) {
   super(message);
   this.name = "MyError";
 }
}
throw new MyError("Something went wrong");
```

In this example, we create a custom error class called **MyError** that extends the built-in **Error** class. The **constructor** method takes a **message** parameter, which is passed to the parent **Error** constructor using the **super** keyword. We also set the **name** property to "MyError" to make it easier to identify the error later.

To throw the custom error, we simply use the **throw** keyword followed by a new instance of the **MyError** class. This will generate an error with the message "Something went wrong" and the name "MyError".

To handle a custom error, you can use a try-catch statement just like with the built-in errors:

```
try {
 // Code that might throw a custom error
} catch (error) {
 if (error instanceof MyError) {
   console.log("A MyError occurred: " + error.message);
 } else {
   console.log("An error occurred: " + error.message);
 }
}
```

In this example, we check if the error that was caught is an instance of the **MyError** class using the **instanceof** operator. If it is, we log a custom message to the console. If it's not, we log a generic error message.

Custom errors can be useful for creating more specific error messages and handling errors in a more structured way in your code.

Error objects

Error objects in JavaScript are objects that contain information about an error that has occurred during the execution of your code. They are typically generated when an exception is thrown and can provide useful information for debugging and error handling.

In JavaScript, there are several types of built-in error objects that inherit from the **Error** object:

Error: The base error object from which all other errors inherit. It contains a message property that describes the error.

SyntaxError: An error that occurs when there is a syntax error in the code (e.g., a missing semicolon).

ReferenceError: An error that occurs when an undefined variable or function is referenced.

TypeError: An error that occurs when a value is not of the expected type (e.g., calling a method on a null or undefined value).

RangeError: An error that occurs when a value is outside the expected range (e.g., creating an array with a negative length).

URIError: An error that occurs when a URI-related function is used incorrectly.

Here's an example of how to create and throw a built-in error object:

```
try {
 let x;
 console.log(x.toUpperCase());
} catch (error) {
 if (error instanceof TypeError) {
   console.log("A type error occurred: " + error.message);
 } else {
   console.log("An error occurred: " + error.message);
 }
}
```

In this example, we catch the error that is thrown when we try to call the **toUpperCase()** method on an undefined variable **x**. We check if the error is an instance of the **TypeError** object, and if it is, we log a custom message to the console. Otherwise, we log a generic error message with the error message property.

You can also create your own custom error objects by extending the built-in **Error** object or by creating a new object with a **message** property. Custom error objects can provide more specific information about the error and can be useful for handling errors in a more structured way in your code.

Error handling with Promises and async/await

In JavaScript, Promises and async/await provide a way to handle asynchronous operations and can be useful for error handling.

Promises are objects that represent a value that may not be available yet, but will be resolved at some point in the future. They have a **then()** method that allows you to chain callbacks that will be executed when the Promise is resolved or rejected.

Here's an example of how to use a Promise for error handling:

```
function fetchData() {
 return new Promise((resolve, reject) => {
   // Make an asynchronous API call
   fetch("https://example.com/data")
     .then(response => response.json())
     .then(data => resolve(data))
     .catch(error => reject(error));
 });
}
fetchData()
 .then(data => console.log(data))
 .catch(error => console.log("An error occurred: " + error.message));
```

In this example, the **fetchData()** function returns a Promise that makes an asynchronous API call using the **fetch()** method. If the call is successful, the Promise is resolved with the data returned by the API. If there's an error, the Promise is rejected with an error object.

We can handle the result of the Promise using the **then()** and **catch()** methods. If the Promise is resolved, the **then()** callback is executed and the data is logged to the console. If the Promise is rejected, the **catch()** callback is executed and an error message is logged to the console.

Async/await is a syntax that provides a more concise way to handle Promises. It allows you to write asynchronous code that looks like synchronous code.

Here's an example of how to use async/await for error handling:

```
async function fetchData() {
 try {
   const response = await fetch("https://example.com/data");
   const data = await response.json();
   return data;
 } catch (error) {
   throw error;
 }
}
try {
 const data = await fetchData();
 console.log(data);
} catch (error) {
 console.log("An error occurred: " + error.message);
}
```

In this example, the **fetchData()** function is marked as **async**, which allows us to use the **await** keyword to wait for the Promise to resolve or reject. If the Promise is resolved, the **await** keyword returns the data, and if it's rejected, it throws an error.

We can handle the result of the function using a try-catch block. If the function returns data, the try block is executed, and the data is logged to the console. If the function throws an error, the catch block is executed, and an error message is logged to the console.

Async/await can be useful for error handling because it allows you to write more readable and structured code. However, it's important to note that async/await is still based on Promises, so you should still be familiar with Promise syntax and error handling.

AI

Creating an AI system is a broad task and can range from a simple chatbot to a complex machine learning model. Below is an example of a simple chatbot AI in JavaScript using Node.js and the "readline" module to interact with the user via the command line.

First, make sure you have Node.js installed on your system. You can download it from nodejs.org.

Create a new directory for your project and navigate to it in your terminal or command prompt.

Initialize a new Node.js project by running **npm init** and following the prompts to create a **package.json** file.

Create a new file in your project directory named **chatbot.js**.

Paste the following code into **chatbot.js**:

```
const readline = require('readline');
const rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout
});
const responses = {
 "hi": "Hello! How can I help you today?",
 "how are you?": "I'm just a program, so I don't have feelings, but I'm
running well, thank you!",
 "bye": "Goodbye! Have a great day!",
 "default": "I'm not sure how to respond to that."
};
const getResponse = (input) => {
 return responses[input.toLowerCase()] || responses['default'];
};
const chat = () => {
 rl.question('You: ', (input) => {
   if (input.toLowerCase() === 'bye') {
     console.log('Bot: Goodbye! Have a great day!');
     return rl.close();
   }
   const response = getResponse(input);
   console.log(`Bot: ${response}`);
   chat(); // repeat the process
 });
};
console.log('Bot: Hi! Type "bye" to exit.');
chat();
```

Run your chatbot by executing **node chatbot.js** in your terminal or command prompt.

Start typing messages to interact with your chatbot. Type "bye" to exit the chat.

This is a very basic example, and real AI systems usually require more advanced techniques and libraries, possibly integrating with machine learning models trained on large datasets. For more advanced AI implementations in JavaScript, you might want to look into libraries such as TensorFlow.js, which brings machine learning capabilities to JavaScript.

MySQL

To create a JavaScript CRUD (Create, Read, Update, Delete) application with a MySQL database, you can use Node.js along with Express for the server-side, and any frontend framework or plain JavaScript for the client-side. Here’s a basic outline of the steps involved:

Backend: Node.js with Express and MySQL

Set up the project:

```
mkdir crud-app
cd crud-app
npm init -y
npm install express mysql body-parser cors
```

Create the server (server.js):

```
const express = require('express');
const mysql = require('mysql');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();
app.use(cors());
app.use(bodyParser.json());
const db = mysql.createConnection({
 host: 'localhost',
 user: 'root',
 password: 'password',
 database: 'crud_db'
});
db.connect(err => {
 if (err) {
   console.error('Database connection failed:', err);
 } else {
   console.log('Database connected.');
 }
});
// Create
app.post('/create', (req, res) => {
 const { name, email } = req.body;
 const sql = 'INSERT INTO users (name, email) VALUES (?, ?)';
 db.query(sql, [name, email], (err, result) => {
   if (err) {
     res.status(500).send(err);
   } else {
     res.send('User added.');
   }
 });
});
// Read
app.get('/users', (req, res) => {
 const sql = 'SELECT * FROM users';
 db.query(sql, (err, results) => {
   if (err) {
     res.status(500).send(err);
   } else {
     res.json(results);
   }
 });
});
// Update
app.put('/update/:id', (req, res) => {
 const { id } = req.params;
 const { name, email } = req.body;
 const sql = 'UPDATE users SET name = ?, email = ? WHERE id = ?';
 db.query(sql, [name, email, id], (err, result) => {
   if (err) {
     res.status(500).send(err);
   } else {
     res.send('User updated.');
   }
 });
});
// Delete
app.delete('/delete/:id', (req, res) => {
 const { id } = req.params;
 const sql = 'DELETE FROM users WHERE id = ?';
 db.query(sql, [id], (err, result) => {
   if (err) {
     res.status(500).send(err);
   } else {
     res.send('User deleted.');
   }
 });
});
app.listen(3000, () => {
 console.log('Server is running on port 3000');
});
```

Frontend: HTML, CSS, and JavaScript

Create an index.html file to serve as the frontend:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>CRUD App</title>
 <style>
   /* Add some basic styling */
 </style>
</head>
<body>
 <h1>CRUD Application</h1>

 <form id="userForm">
   <input type="text" id="name" placeholder="Name" required>
   <input type="email" id="email" placeholder="Email" required>
   <button type="submit">Add User</button>
 </form>
 <ul id="userList"></ul>
 <script>
   const userForm = document.getElementById('userForm');
   const userList = document.getElementById('userList');
   userForm.addEventListener('submit', async (e) => {
     e.preventDefault();
     const name = document.getElementById('name').value;
     const email = document.getElementById('email').value;
     const response = await fetch('http://localhost:3000/create', {
       method: 'POST',
       headers: {
         'Content-Type': 'application/json'
       },
       body: JSON.stringify({ name, email })
     });
     if (response.ok) {
       alert('User added.');
       fetchUsers();
     }
   });
   async function fetchUsers() {
     const response = await fetch('http://localhost:3000/users');
     const users = await response.json();
     userList.innerHTML = '';
     users.forEach(user => {
       const li = document.createElement('li');
       li.textContent = `${user.name} (${user.email})`;
       li.innerHTML += ` <button
onclick="deleteUser(${user.id})">Delete</button>`;
       li.innerHTML += ` <button onclick="editUser(${user.id},
'${user.name}', '${user.email}')">Edit</button>`;
       userList.appendChild(li);
     });
   }
```

```
    async function deleteUser(id) {
      const response = await fetch(`http://localhost:3000/delete/${id}`, {
method: 'DELETE' });
      if (response.ok) {
        alert('User deleted.');
        fetchUsers();
      }
    }
    async function editUser(id, name, email) {
      const newName = prompt('Enter new name:', name);
      const newEmail = prompt('Enter new email:', email);
      const response = await fetch(`http://localhost:3000/update/${id}`, {
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ name: newName, email: newEmail })
      });
      if (response.ok) {
        alert('User updated.');
        fetchUsers();
      }
    }
    fetchUsers();
  </script>
</body>
</html>
```

Setup MySQL Database

Open your MySQL client and create a database and a table:

```
CREATE DATABASE crud_db;
USE crud_db;
CREATE TABLE users (
 id INT AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(255),
 email VARCHAR(255)
);
```

Running the Application

Start your Node.js server:

```
node server.js
```

Open index.html in your browser.

You now have a basic CRUD application using JavaScript, Node.js, Express, and MySQL. This can be expanded with more features, error handling, and frontend frameworks like React or Vue.js for a more sophisticated application.

static

A static class in JavaScript, in the sense of the static keyword in other programming languages like C# or Java, does not exist natively. However, you can mimic the behavior of a static class using a module pattern or by creating a class with static methods. Here's an example of how to create a class with static methods in JavaScript:

Using a Class with Static Methods

```
class MyStaticClass {
 static myStaticMethod() {
   console.log('This is a static method');
 }
 static anotherStaticMethod() {
   console.log('This is another static method');
 }
}
// Calling static methods
MyStaticClass.myStaticMethod();
MyStaticClass.anotherStaticMethod();
```

In this example, MyStaticClass has two static methods: myStaticMethod and anotherStaticMethod. You can call these methods directly on the class itself, without needing to create an instance of the class.

Using a Module Pattern

Alternatively, you can use a module pattern to create a static-like class:

```
const MyStaticModule = (function() {
 function myStaticMethod() {
   console.log('This is a static method');
 }
 function anotherStaticMethod() {
   console.log('This is another static method');
 }
 return {
   myStaticMethod: myStaticMethod,
   anotherStaticMethod: anotherStaticMethod
 };
})();
// Calling methods
MyStaticModule.myStaticMethod();
MyStaticModule.anotherStaticMethod();
```

In this example, MyStaticModule is an object with methods that behave like static methods. You can call these methods directly on the module object.

Both approaches allow you to create methods that can be called without creating an instance of a class, mimicking the behavior of static classes in other programming languages.

Inheritance

Inheritance in JavaScript allows a class to inherit properties and methods from another class. This is done using the extends keyword. Here’s an example to illustrate how inheritance works in JavaScript:

Basic Example of Inheritance

```
class Animal {
 constructor(name) {
   this.name = name;
 }
 speak() {
   console.log(`${this.name} makes a sound.`);
 }
}
class Dog extends Animal {
 constructor(name, breed) {
   super(name); // Calls the constructor of the parent class (Animal)
   this.breed = breed;
 }
 speak() {
   super.speak(); // Calls the speak method of the parent class
   console.log(`${this.name} barks.`);
 }
 getBreed() {
   console.log(`${this.name} is a ${this.breed}.`);
 }
}
// Creating an instance of the Dog class
const myDog = new Dog('Buddy', 'Golden Retriever');
// Calling methods
myDog.speak();     // Outputs: "Buddy makes a sound." followed by "Buddy
barks."
myDog.getBreed();  // Outputs: "Buddy is a Golden Retriever."
```

Explanation

Animal Class: This is the parent class with a constructor that initializes the name property and a method speak that logs a message.

Dog Class: This class inherits from the Animal class using the extends keyword. It has its own constructor that calls the parent class constructor using super(name), and it also initializes an additional property breed.

Method Overriding: The speak method in the Dog class calls the speak method of the parent class using super.speak() and then adds additional behavior.

Creating an Instance: An instance of the Dog class is created with the name 'Buddy' and breed 'Golden Retriever'. Methods speak and getBreed are called on this instance.

Points to Note

- The super keyword is used to call the constructor and methods of the parent class.
- You can override methods in the child class and still call the parent class methods using super.methodName().
- Inheritance allows the creation of a hierarchy of classes, making code reusable and easier to manage.

This example demonstrates the basic principles of inheritance in JavaScript, showing how a class can extend another class, inherit its properties and methods, and override or add new functionality.