Javascript

By LAKSHMIKANT DESHPANDE

Introduction to Javascript

- What is Javascript?
- Use of javascript?
- Trend of Javascript?
- Types of Implementing Javascript?
    - Internal
    - External

Javascript Console

- Console
  - log
  - error
  - warn
  - info
  - clear
  - assert

- Document
  - getElementById
  - querySelector
  - querySelectorAll
  - getElementsByClassName
  - getElementsByTagName
  - title
  - style

JavaScript REPL - Read-Eval-Print Loop

- 2 + 3
- 10 - 4 * 2
- Math.sqrt(16)

Basics of Javascript

- Variables
  - Variables are declared using the var, let, or const keyword.
  - Variables declared with var and let can be reassigned to new values.
  - The var keyword has function scope or global scope, while let has block scope (limited to the nearest enclosing block).
  - Variables declared with var or let can be declared without an initial value and assigned later.
- Constants
  - Constants are declared using the const keyword.
  - Constants cannot be reassigned to a new value once they are declared. They are immutable.
  - Constants must be assigned a value at the time of declaration and cannot be declared without an initial value.
  - Constants also have block scope like let.

Data Types

- Primitive data types.
  - Number
  - String
  - Boolean
  - Null
  - Undefined
  - Symbol

- Non-primitive data types.
  - Object
  - Function
  - Array
  - Date
  - RegExp

Alert, Prompt and Confirm

Alert, Prompt, and Confirm are built-in functions that allow interaction with the user through dialog boxes in the browser.

- Alert

  alert('Hello Excelrians!');

- Prompt

  const name = prompt('Please enter your name:');

  console.log('Hello, ' + name); // Logs a message with the entered name

- Confirm

  const result = confirm('Are you sure you want to learn full stack development?');

Operators

JavaScript provides a wide range of operators that allow you to perform various operations on values.

Arithmetic Operators

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulo)
- ++ (increment)
- -- (decrement)

Assignment Operators

- = (simple assignment)
- += (addition assignment)
- -= (subtraction assignment)
- *= (multiplication assignment)
- /= (division assignment)
- %= (modulo assignment)

Comparison Operators

- == (equality)
- === (strict equality)
- != (inequality)
- !== (strict inequality)
- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)

Logical Operators

- && (logical AND)
- || (logical OR)
- ! (logical NOT)
    -

Bitwise Operators

- & (bitwise AND)
- | (bitwise OR)
- ^ (bitwise XOR)
- ~ (bitwise NOT)
- << (left shift)
- >> (right shift)
- >>> (unsigned right shift)

- **Conditional (Ternary) Operator:**
  - condition ? expr1 : expr2
  - It's a shorthand way of writing an if-else statement.
- **Type Operators:**
  - typeof (returns the type of a value)
  - instanceof (checks if an object is an instance of a particular class)
- **String Operators:**
  - + (concatenation operator, used to concatenate strings)

Unary Operators

- + (unary plus)
- - (unary minus)
- ! (logical NOT)
- ++ (prefix/postfix increment)
- -- (prefix/postfix decrement)

Conditions / Decision making

- **Boolean Conditions**
- **Comparison Conditions**
- **Logical Conditions**
- **Ternary Operator**
- **Nested Conditions**
- **Switch Statement**

Switch statements

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]
  case 'value2':  // if (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
```

Loops / Iterations

for loop: It executes a block of code for a specified number of times, based on a defined initialization, condition, and iteration expression.

while loop: It repeatedly executes a block of code as long as a specified condition is true.

do-while loop: It is similar to the while loop, but it executes the block of code at least once before checking the condition.

for...in loop: It iterates over the properties of an object, providing access to each key.

for...of loop: It iterates over iterable objects (arrays, strings, etc.), providing access to each element.

Entry controlled vs exit controlled loops

for loop

```
for (begin; condition; step) {
    // ... loop body ...
}
```

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

while loop

```
while (condition) {
  //  ..  "loop body"
}

let i = 0;
while (i < 5) {
continue;
    console.log(i);
    i++;
}
```

do-while loop

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

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 5);
```

for...in loop

```
for (variable in object) {
  // Code block to be executed for each property
}
const person = {
  name: ‘LAKSHMIKANT’,
  age: 22,
  city: ‘Bengaluru’
};
for (let key in person) {
  console.log(key + ': ' + person[key]);
}
```

for...of loop

```
for (variable of iterable) {
  // Code block to be executed for each iteration
}
const fruits = ['apple', 'banana', 'cherry'];
for (const fruit of fruits) {
  console.log(fruit);
}
```

Data Storage

Data Storage Types

- Booleans
- Numbers
- Strings
- Arrays
- Objects

Data Structures

- Arrays
- Objects
- Maps
- Sets
- Linked Lists
- Stacks
- Queues
- Tree
- Graph

Strings

Strings are immutable

- let gameTitle = "Gamers Zone";
- let welcomeMessage = "Welcome to Gamers Zone!";
- let str = "Hello";

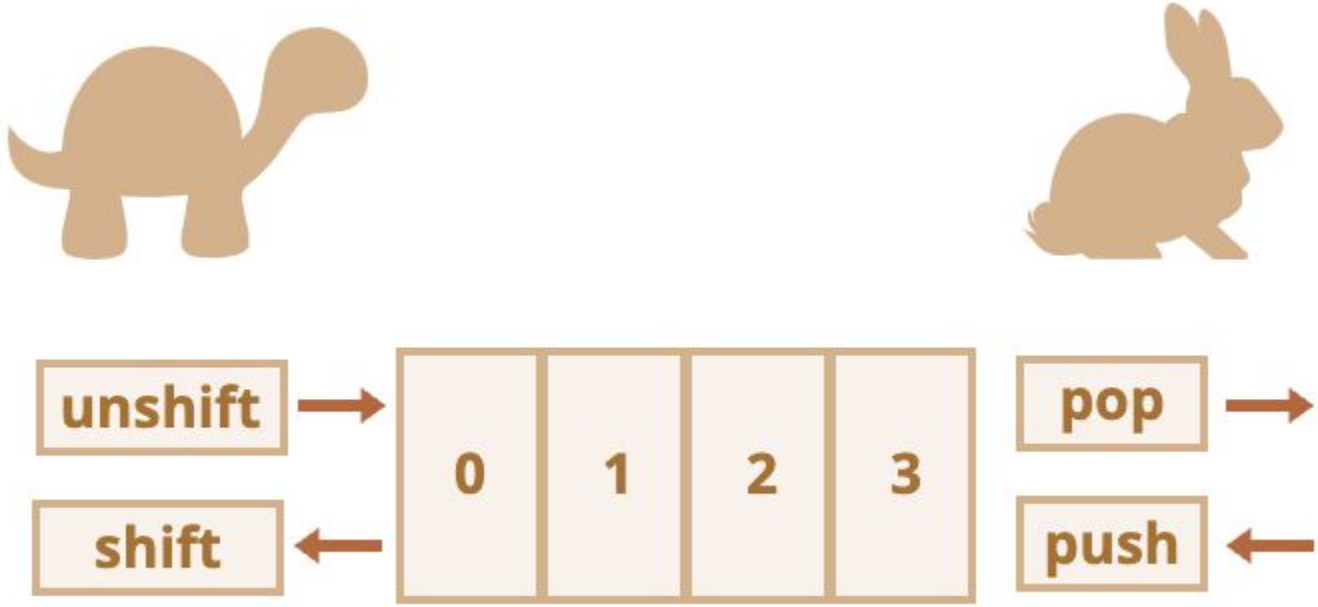
  str += " World"; // Creates a new string, "Hello World", but does not modify the string

JavaScript String Methods

1. charAt()
2. charCodeAt()
3. concat(str1, str2, ...)
4. includes()
5. endsWith()
6. indexOf()
7. lastIndexOf()
8. match()
9. matchAll()
10. repeat()
11. replace()
12. replaceAll()
13. search()
14. slice()
15. split()
16. startsWith()
17. substr()
18. substring()
19. toLowerCase()
20. toUpperCase()
21. toString()
22. trim()
23. valueOf()

Arrays

```
let gameGenres = ["Action", "Adventure", "RPG"];

let topPlayers = ["Sumanth", "Vikash", "Deepak", "Alayja", "Rakshit", "Manoj"];
```


unshift
shift
0
1
2
3
pop
push

A LIST OF

JAVASCRIPT

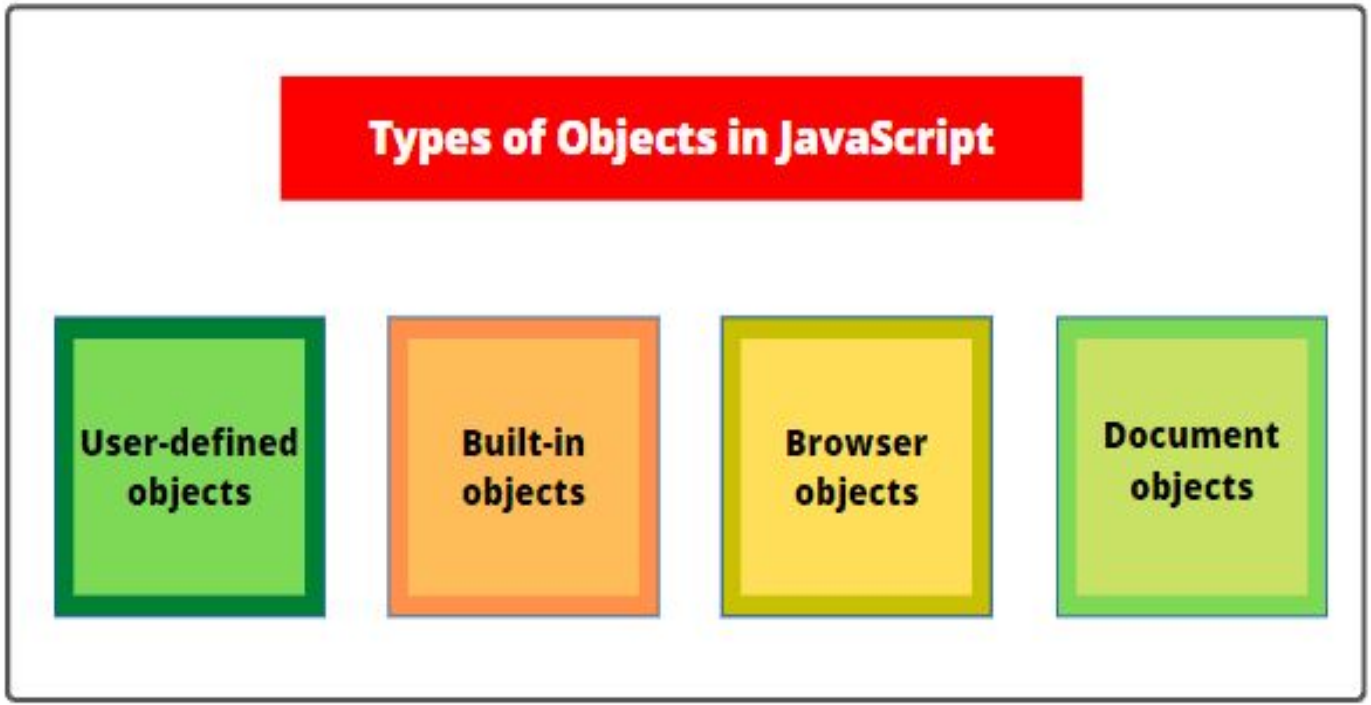
ARRAY METHODS

.map()
.filter()
.sort()
.forEach()
.concat()
.every()
.some()
.includes()
.join()
.reduce()

.find()
.findIndex()
.indexOf()
.fill()
.slice()
.reverse()
.push()
.pop()
.shift()
.unshift()

Objects

```
const player = {
    username: "Gamer123",
    level: 15,
    score: 2500,
    achievements: ["First Blood", "Sharpshooter"],
    inventory: {
        weapon: "Blaster",
        armor: "Power Suit",
    },
};
```


Types of Objects in JavaScript
User-defined objects
Built-in objects
Browser objects
Document objects

JAVASCRIPT OBJECT CHEATSHEET

Object Declaration

```
const twit = {
  name: "Proful",
  follower: 7815,
  1 : "hi"
}
```

any type

converted to string

- storing key-value pairs.
- data unordered
- keys are unique

Dot Notation

```
twit.name          // "Proful"
twit.follower      // 7815
twit.follower.count
```

Accessing nested props

Square Notation

```
twit['name']       // "Proful"
```

Can be dynamic/variable

```
const {name, followers} = twit
name               // Proful
followers          // 7815

const linkedin = { name }

                   { name: 'Proful' }
```

Empty Object creations

```
const person = {}
const person = new Object()
```

```
const twit = {
  name: "Proful"
}

function change(insta){
  insta.name = "Steve"
}

change(twit)
twit.name // "Steve"
```

- Pass by reference

```
delete twit.name // both key & value

twit.randomKey    // undefined

twit.follower = 5000
```

declared as const but mutable

```
for(const key in twit) {
  console.log(key)       // name
                         // followers
}
```

```
const twit = {
  name: "Proful",
  get profile() {
    return `Hi ${this.name.toLowerCase()}`
  },
  set profile(prof) {
    this.name = "Mr " + prof
  }
}
twit.profile                  // 'Hi proful'
twit.profile = 'Steve'
twit.name                     // 'Mr Steve'
```

getter

seter

```
const twit = {
  name: "Proful",
  hi() {
    console.log(`Hi ${this.name}`)
  },
  hello: () => {
    console.log(`Hello ${twit.name}`)
  },
}
```

this refer to twit object

You cannot use this here

Map

```
let myMap = new Map();

myMap.set("name", "Subhash");

myMap.set("age", 28);
```

Map Properties and Methods

Methods	Descriptions
map.set(key, value)	sets the value for the key in the map object, and returns the map object itself.
map.get(key)	returns the value associated with the key, if no key exist returns undefined.
map.has(key)	returns true if a value associated with the key exists, otherwise false.
map.delete(key)	removes an element specified by the key. It returns true if the element is in the map, false if it does not.
map.clear()	removes all elements from the map object.
map.size	returns a count of all the elements in the map object.
map.entries	returns a new Iterator object that contains an array of [key, value] for each element in the map object.
map.keys()	returns a new Iterator that contains the keys for elements in insertion order.
map.values()	returns a new Iterator object that contains values for each element in insertion order.
map.forEach(callback[, thisArg])	invokes a callback for each key-value pair in the map in the insertion order. thisArg (optional parameter) sets the this value for each callback.

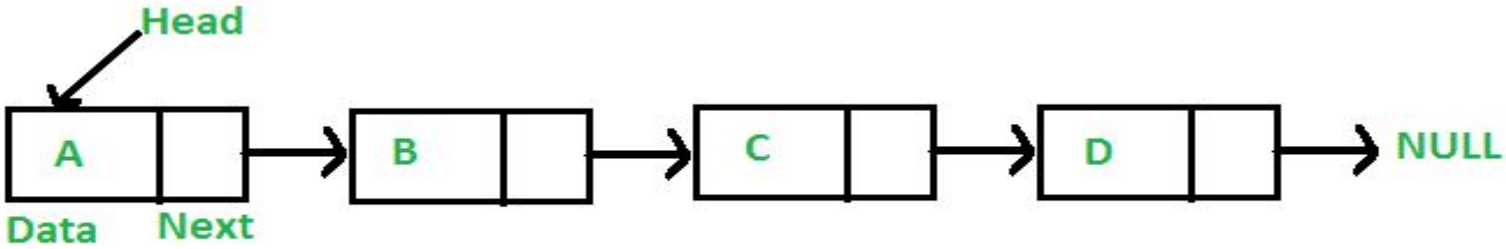
Set

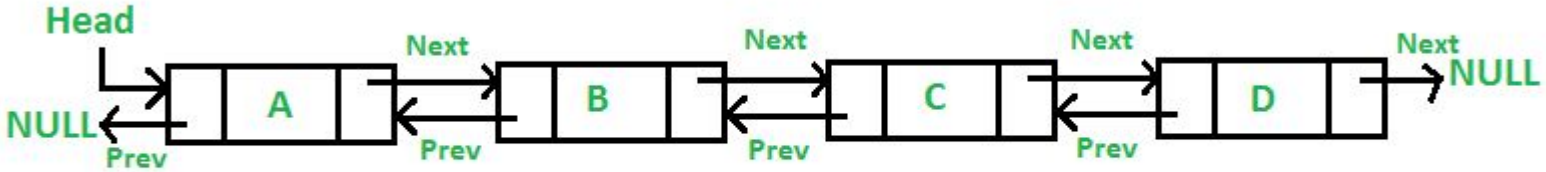
```
let mySet = new Set();

mySet.add(1);

mySet.add(2);

mySet.add(2); // Duplicate value, not added

mySet.add(1); // The duplicate value 1 will not be added
```

Linked List

A linked list is a data structure composed of a collection of nodes, where each node contains a value and a reference (or link) to the next node in the list. It provides an ordered way to store and manipulate data.

- A linked list is a linear data structure.
- It consists of a sequence of elements called nodes.
- Each node contains data and a reference (link) to the next node in the sequence.
- The first node in the list is called the head.
- The last node in the list points to null, indicating the end of the list.


Head
A
B
C
D
NULL
Data
Next


Head
NULL
A
B
C
D
NULL
Next
Next
Next
Next
Prev
Prev
Prev


Start
Data
Data
Data
Next
Next
Prev
Prev

Stack

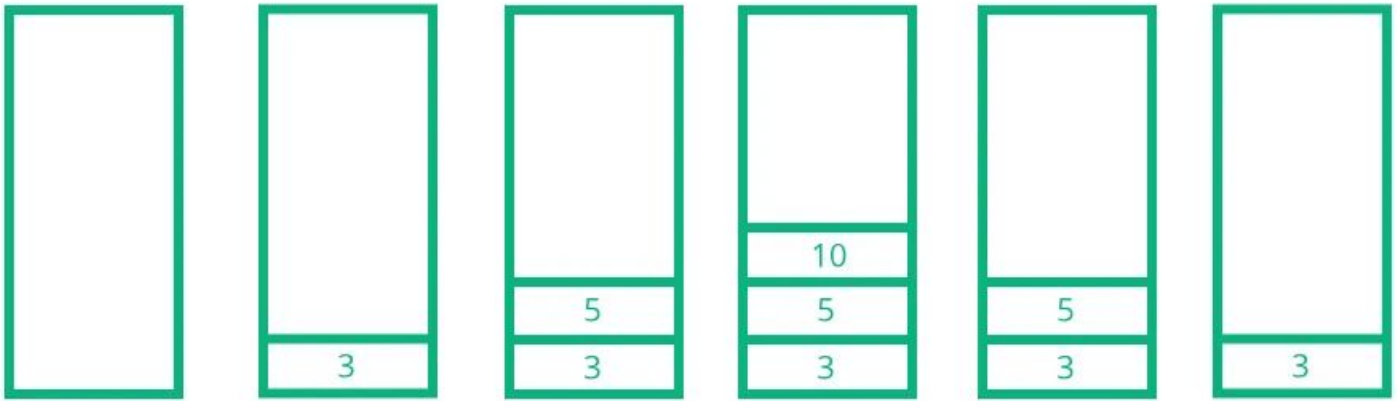
A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It behaves like a stack of objects where the last item added is the first one to be removed.

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle.
- It represents a collection of elements with two main operations: push and pop.
- Elements are added and removed from only one end of the stack, called the top of the stack.
- The top of the stack represents the most recently added item, while the bottom represents the least recently added item.
- Adding an element to the stack is called pushing. It places the new element at the top of the stack.
- Removing an element from the stack is called popping. It removes the top element from the stack.
- Accessing the top element without removing it is also a common operation and is called peeking.

STACK

PUSH : Insert Element at Top of Stack

POP : Deletes the Element at Top of Stack

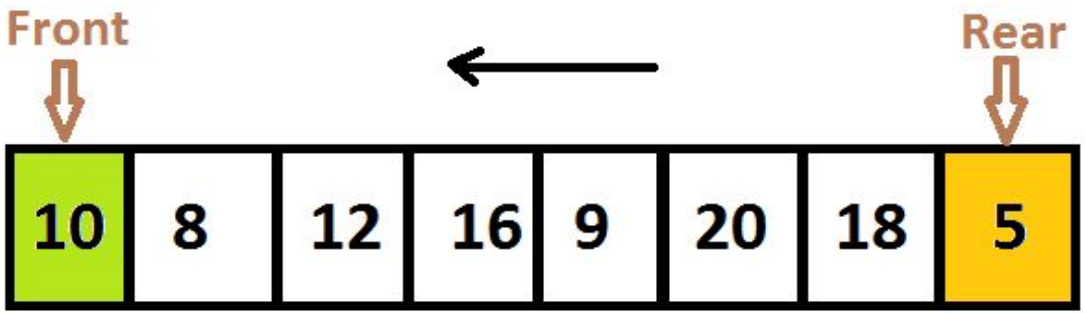


Queue

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It represents a collection of elements with two primary operations: enqueue and dequeue.

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle.
- It represents a collection of elements with two main operations: enqueue and dequeue.
- Elements are added to the back of the queue and removed from the front.
- Adding an element to the queue is called enqueueing or offer.
- Removing an element from the queue is called dequeueing or poll.
- The element at the front of the queue is the oldest element, while the element at the back is the newest.

Queue Data Structure (First In First Out)



Enqueue(10)
Enqueue(8)
Enqueue(12)
Enqueue(16)
Enqueue(9)
Enqueue(20)
Enqueue(18)
Dequeue() -->10
Dequeue() -->8
Dequeue() -->12
Dequeue() -->16
Dequeue() -->9
Dequeue() -->20
Dequeue() -->18

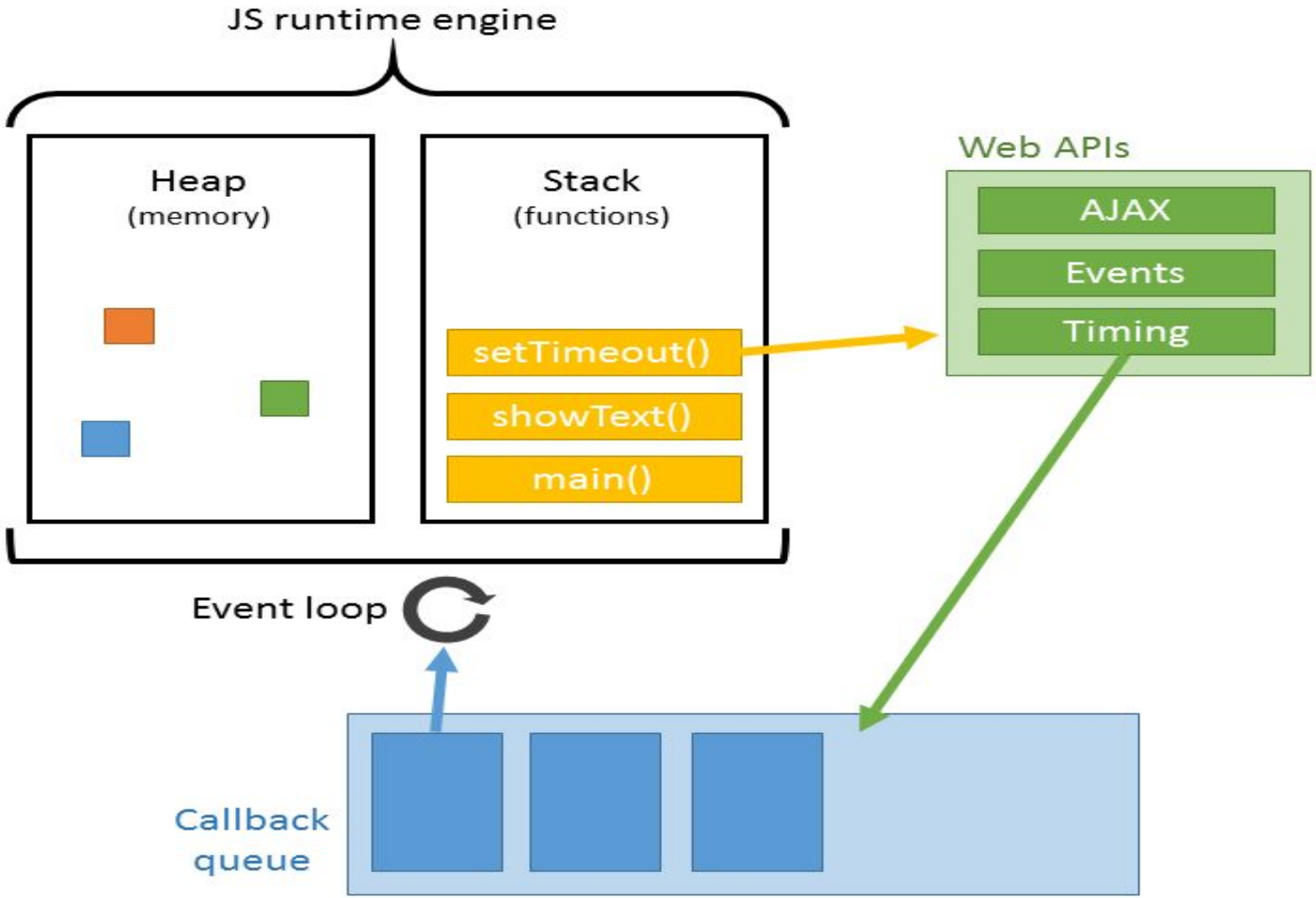
Web APIs

DOM
AJAX (XMLHttpRequest)
setImmediate
setTimeout
setInterval

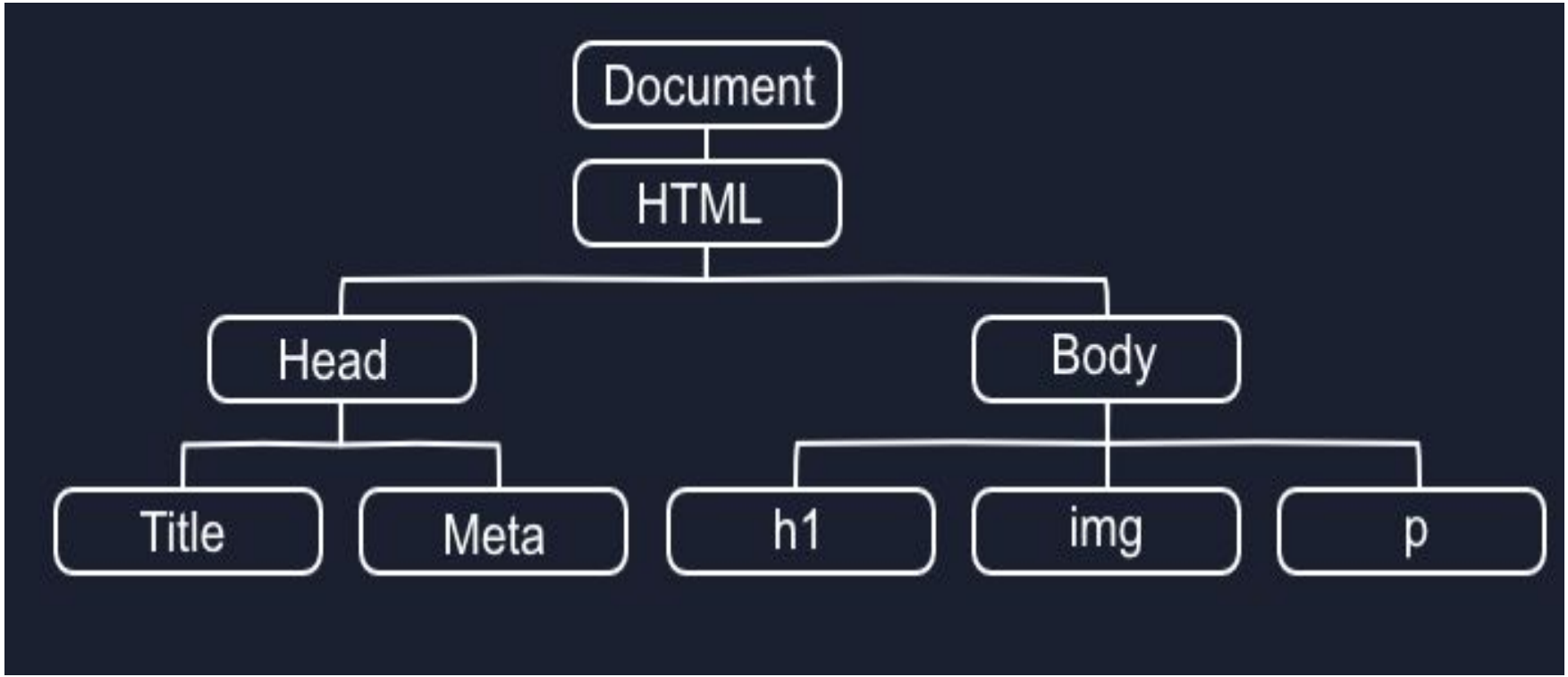
Event Loop

Callback Queue

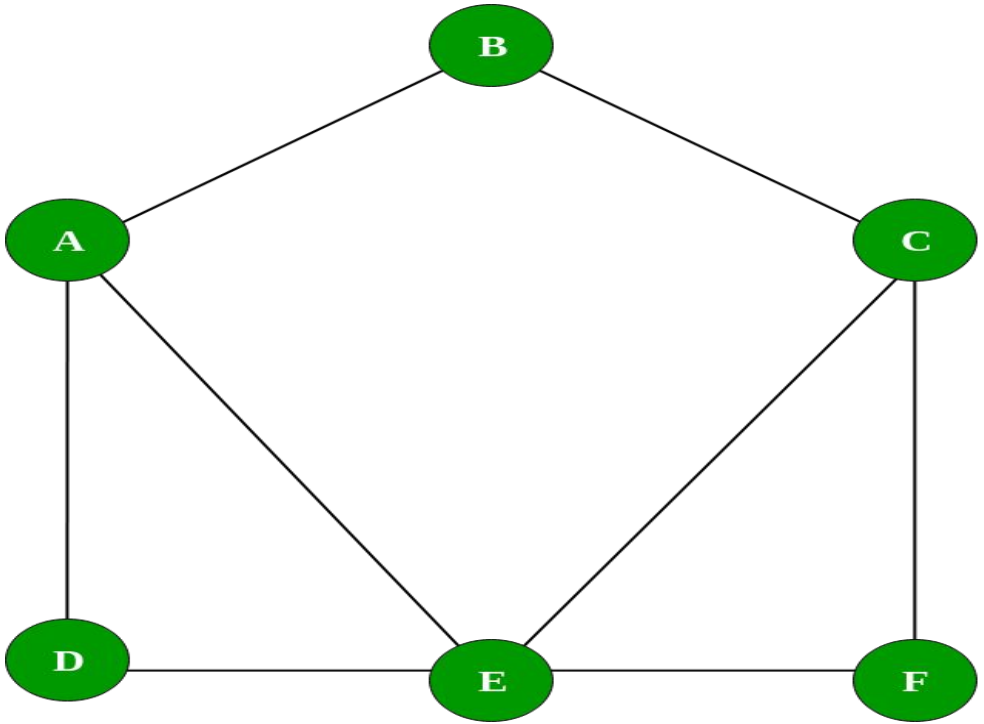
onClick
onLoad
onDone


JS runtime engine
Heap
(memory)
Stack
(functions)
setTimeout()
showText()
main()
Web APIs
AJAX
Events
Timing
Event loop
Callback
queue

Trees



Graph



Functions

Function Declaration

```
function greet(name) {
  console.log("Hello, " + name + "!");
}
function playGame(gameName) {
  console.log("Welcome to the Gamers Zone!");
  console.log("Starting " + gameName + "...");
}
playGame("Super Mario Bros"); // Calling the playGame function with the argument "Super Mario Bros"
```

Function Expressions (Anonymous)

```
const greet = function(name) {
  console.log("Hello, " + name + "!");
};
const playGame = function(gameName) {
  console.log("Welcome to the Gamers Zone!");
  console.log("Starting " + gameName + "...");
};
playGame("Super Mario Bros"); // Calling the playGame function with the argument "Super Mario Bros"
```

Arrow Functions

```
const greet = (name) => {
  console.log("Hello, " + name + "!");
};
const playGame = (gameName) => {
  console.log("Welcome to the Gamers Zone!");
  console.log("Starting " + gameName + "...");
  // Game logic goes here
};
playGame("Super Mario Bros"); // Calling the playGame function with the argument "Super Mario Bros"
```

Function Expression (Named)

```
var gamersZone = function playGame() {
  console.log("Welcome to the Gamers Zone!");
  // function body
};

gamersZone(); // Calling the named function expression
```

Method Definition (Within an Object)

```
var object = {
  functionName: function() {
    // function body
  }
};
var gamersZone = {
  name: "Gamers Zone",
  playGame: function() {
    console.log("Welcome to " + this.name + "!");

  }
};
gamersZone.playGame(); // Calling the playGame method
```

Function as a Constructor

```
// Syntax

function FunctionName() {
  // function body
}
var instance = new FunctionName();
```

```
function GamersZone() {
  this.name = "Gamers Zone";
  this.playGame = function() {
    console.log("Welcome to " + this.name + "!");
  };
  this.title = “Call of Duty”;
this.getTitleAndName = function(){
          return this.name+” “ + this.title;
}

}

var gameInst = new GamersZone(); // Creating an instance using the constructor
gameInst.playGame(); // Calling the playGame method of the instance
```

Immediately Invoked Function Expression (IIFE)

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

Or

```
(function functionName() {
  // function body
})();
```

```
(function() {
  var gamersZone = {
    name: "Gamers Zone",
    playGame: function() {
      console.log("Welcome to " + this.name + "!");
    }
  };
  gamersZone.playGame(); // Calling the playGame method
})();
```

Generator Function

```
function* gamersZone() {
  yield "Gamers Zone";
  yield "Welcome to the gaming zone!";

}

var zoneGenerator = gamersZone(); // Creating a generator object
console.log(zoneGenerator.next().value); // "Gamers Zone"
console.log(zoneGenerator.next().value); // "Welcome to the gaming zone!"
```

Object Prototype Method

```
function ConstructorName() {
  // constructor body
}

ConstructorName.prototype.functionName = function() {
  // function body
};
```

Advantages of Regular Functions

- Function Names: Regular functions can have explicit names, making them easier to identify and debug in stack traces and error messages.
- Function Declarations: Function declarations are hoisted, meaning they can be called before they are defined in the code.
- Dynamic this Binding: Regular functions have dynamic this binding, allowing the this context to be determined at runtime based on how the function is called.
- Prototype Property: Regular functions can be used as constructors to create objects, and they have a prototype property that can be used for inheritance and prototype-based object-oriented programming.
- Access to arguments: Regular functions have access to the arguments object, which contains all the arguments passed to the function, providing flexibility in handling variable numbers of arguments.

Disadvantages of Regular Functions

- Syntax Length: Regular functions typically have a longer syntax compared to arrow functions, especially when dealing with simple, short functions.
- this Binding Complexity: The dynamic binding of this in regular functions can sometimes be confusing, as it depends on how the function is called and can lead to unexpected behavior if not handled correctly.
- Lack of Lexical Scoping: Regular functions create their own scope, which can lead to variable shadowing and potential naming conflicts when dealing with nested functions or closures.
- No Implicit Return: Regular functions require an explicit return statement to return a value, even for single expressions, which can result in more verbose code compared to arrow functions.

Advantages of Arrow Functions

- Concise Syntax: Arrow functions have a shorter and more concise syntax compared to regular function expressions, making the code more readable and reducing the need for excessive typing.
- Lexical this Binding: Arrow functions do not have their own this context. Instead, they inherit the this value from the surrounding scope. This behavior can help avoid confusion and inconsistencies with this binding in regular functions.
- Implicit Return: Arrow functions with a single expression automatically return the result of that expression without requiring an explicit return statement. This feature makes the code more concise for simple functions.
- No Implicit Arguments Binding: Arrow functions do not have an arguments object. Instead, they inherit the arguments object from the enclosing scope. This behavior prevents the potential confusion caused by the implicit binding of arguments in regular functions.

Disadvantages of Arrow Functions

- Lack of Function Name: Arrow functions are always anonymous. They cannot be named, which can make debugging and stack traces more challenging as the function name will not be available.
- Cannot Be Used as Constructors: Arrow functions do not have a prototype property and cannot be used as constructors to create objects. They are primarily intended for use as concise function expressions.
- No arguments Object: As mentioned earlier, arrow functions do not have their own arguments object. If you need access to the arguments passed to the function, you need to rely on the arguments object of the enclosing scope.
- Lexical this Binding: While the lexical this binding can be an advantage, it can also be a disadvantage in certain scenarios where you need the dynamic this context provided by regular functions.

Strings

- length: Returns the number of characters in a string.
- toUpperCase(): Converts a string to uppercase.
- toLowerCase(): Converts a string to lowercase.
- substring(startIndex, endIndex): Extracts a portion of a string based on start and end indices.
- split(delimiter): Splits a string into an array of substrings based on a delimiter.
- indexOf(substring): Returns the index of the first occurrence of a substring within a string.
- replace(searchValue, replaceValue): Replaces a specified substring with another string.
- charAt(index): Returns the character at a specified index in a string.
- slice(startIndex, endIndex): Extracts a portion of a string based on start and end indices.
- trim(): Removes leading and trailing whitespace from a string.

Special Characters

Character	Description
`\n`	New line
`\r`	In Windows text files a combination of two characters `\r\n` represents a new break, while on non-Windows OS it's just `\n`. That's for historical reasons, most Windows software also understands `\n`.
`\'`, `\"`, `\``	Quotes
`\\`	Backslash
`\t`	Tab
`\b`, `\f`, `\v`	Backspace, Form Feed, Vertical Tab – mentioned for completeness, coming from old times, not used nowadays (you can forget them right now).

Array

- push(): Adds one or more elements to the end of an array and returns the new length of the array.
- pop(): Removes the last element from an array and returns that element.
- shift(): Removes the first element from an array and shifts all other elements down by one index. It returns the removed element.
- unshift(): Adds one or more elements to the beginning of an array and shifts existing elements up by one index. It returns the new length of the array.
- splice(): Adds or removes elements from an array. It modifies the original array and returns an array containing the removed elements.
- slice(): Returns a shallow copy of a portion of an array into a new array. It does not modify the original array.
- concat(): Combines two or more arrays and returns a new array without modifying the original arrays.
- join(): Joins all elements of an array into a string, using a specified separator.
- reverse(): Reverses the order of elements in an array.

- sort(): Sorts the elements of an array in place. By default, it converts elements to strings and sorts them based on their Unicode values.
- filter(): Creates a new array with all elements that pass a certain condition based on a callback function.
- map(): Creates a new array by applying a callback function to each element of the original array.
- reduce(): Applies a callback function on each element of an array, resulting in a single value. It reduces the array to a single value.
- forEach(): Calls a provided function once for each element in an array, in order.
- indexOf(): Returns the first index at which a given element is found in an array, or -1 if it is not present.
- lastIndexOf(): Returns the last index at which a given element is found in an array, or -1 if it is not present.
- includes(): Determines whether an array includes a certain element, returning true or false.
- some(): Checks if at least one element in the array passes a certain condition based on a callback function. It returns true if the condition is satisfied, otherwise false.
- every(): Checks if all elements in the array pass a certain condition based on a callback function. It returns true if all elements satisfy the condition, otherwise false.

Iterating over an array

```
let numbers = [1, 2, 3, 4, 5];
// Using a for loop
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
// Using a for...of loop
for (let number of numbers) {
  console.log(number);
}
// Using forEach method
numbers.forEach(function (number) {
  console.log(number);
});
```

Some Array Methods Examples

```
let myArray = [1, 2, 3, 4, 5];
console.log(myArray);
let fruits = ['apple', 'banana', 'orange'];
fruits.push('grape');
console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']
fruits.pop();
console.log(fruits); // Output: ['apple', 'banana', 'orange']
fruits.shift();
console.log(fruits); // Output: ['banana', 'orange']
fruits.unshift('kiwi');
console.log(fruits); // Output: ['kiwi', 'banana', 'orange']
fruits.splice(1, 1, 'cherry', 'mango');
console.log(fruits); // Output: ['kiwi', 'cherry', 'mango', 'orange']
```

Array Higher-Order Functions

```
let numbers = [1, 2, 3, 4, 5];
// Using map to double each number
let doubledNumbers = numbers.map(function (number) {
  return number * 2;
});
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
// Using filter to get even numbers
let evenNumbers = numbers.filter(function (number) {
  return number % 2 === 0;
});
console.log(evenNumbers); // Output: [2, 4]
```

Array Destructuring

```
let numbers = [1, 2, 3];
let [a, b, c] = numbers;
console.log(a, b, c); // Output: 1 2 3
```

Spread Operator

```
let array1 = [1, 2, 3];
let array2 = [4, 5, 6];

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

let clonedArray = [...array1];
console.log(clonedArray); // Output: [1, 2, 3]
```

Multidimensional Arrays

```
let grid = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log(grid[0][0]); // Output: 1
console.log(grid[1][2]); // Output: 6
console.log(grid[2][1]); // Output: 8
grid[1][1] = 10;
console.log(grid);

// Output: [[1, 2, 3], [4, 10, 6], [7, 8, 9]]
```

Object

- Object.keys(): Returns an array of the object's own enumerable property names.
- Object.values(): Returns an array of the object's own enumerable property values.
- Object.entries(): Returns an array of the object's own enumerable property key-value pairs as arrays.
- Object.hasOwnProperty(): Returns a boolean indicating whether an object has a specific property as its own property (not inherited).
- Object.freeze(): Freezes an object, making it immutable by preventing adding, modifying, or removing properties.
- Object.seal(): Seals an object, preventing the addition or deletion of properties while allowing the modification of existing properties.
- Object.create(): Creates a new object with the specified prototype object and properties.

Object Destructuring

```
const person = {
  name: "Lakshmikant",
  age: 30,
  city: "BLR"
};
const { name, age } = person;
console.log(name);  // Lakshmikant
console.log(age);   // 30
```

- JavaScript objects are mutable, which means you can change their properties even after they are created
- Array of Objects
- Object configuration and properties
- Object properties have attributes that define their behavior and characteristics. Two important attributes are the "writable" and "enumerable" attributes.
- Object Clone

Modules

module.exports = { value1, function1 }

Or

exports.value1 = value1

exports.function1 = function1

Each file in a Node.js project is treated as a module that can export values to be used by other modules.

module.exports is an object in a Node.js file that holds the exported values and functions from that module.

Declaring a module.exports object in a file specifies the values to be exported from that file. When exported, another module can import this values with the require global method.

JS : Class (Basics - OOJS)

- Objects: JavaScript uses objects to represent and store data. Objects can have properties (variables) and methods (functions) associated with them.
- Classes: Classes serve as blueprints or templates for creating objects. They define the structure and behavior of objects that will be instantiated from them.
- Constructors: Constructors are special methods within a class that are used to initialize objects. They are called automatically when an object is created from a class.
- Inheritance: Inheritance allows classes to inherit properties and methods from other classes. It promotes code reuse and the creation of hierarchical relationships between classes.
- Encapsulation: Encapsulation refers to the bundling of data and methods within an object. It allows for data hiding and the organization of related functionality.
- Polymorphism: Polymorphism enables objects of different classes to be treated as instances of a common superclass. It allows for flexibility and code extensibility.

- Method Overriding: Method overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass. The overridden method in the subclass takes precedence over the superclass's implementation.
- Getters and Setters: Getters and setters are special methods used to retrieve and set the values of object properties. They allow for controlled access to object data.
- Prototype-based Inheritance: JavaScript uses a prototype-based inheritance model, where objects can inherit properties and methods directly from other objects.
- Object Composition: Object composition refers to the practice of building complex objects by combining simpler objects. It promotes code reuse and modularity.
- This Keyword: The this keyword refers to the current instance of an object. It is commonly used within methods to access object properties and methods.
- Polishing: Polishing involves adding or modifying properties or methods to an existing object or class dynamically.

OOPS with Gamers Zone Example

- Objects: In a Gamers Zone, objects could represent individual gamers, gaming consoles, game characters, or gaming accessories. Each object would have its own properties and methods.
- Classes: Classes in a Gamers Zone context could include classes like Gamer, Console, Game, or Accessory. These classes would define the structure and behavior of the corresponding objects.
- Constructors: Constructors within the Gamers Zone classes would be used to initialize objects. For example, the Gamer class constructor could set properties like the gamer's name, age, preferred game genre, etc.

- Inheritance: Inheritance could be used to create subclasses like CasualGamer and ProGamer that inherit properties and methods from a base Gamer class. This would allow for distinguishing between different types of gamers with specialized behaviors.
- Encapsulation: Encapsulation in a Gamers Zone would involve bundling data and methods within objects. For instance, a Console object could encapsulate properties like the console type, storage capacity, and methods like powering on/off or launching games.
- Polymorphism: Polymorphism can be applied in a Gamers Zone to treat different types of gaming objects, such as Game or Accessory, as instances of a common superclass. This allows for flexibility in handling various gaming-related objects uniformly.
- Method Overriding: Method overriding could be used in the Game class, where different types of games could override a common method like play() with their own implementation, providing unique gameplay experiences.

- Getters and Setters: Getters and setters can be utilized in a Gamers Zone to retrieve and set properties of objects. For example, a Gamer object may have a getter method to retrieve the gamer's high score and a setter method to update the score.
- Prototype-based Inheritance: Prototype-based inheritance allows objects to inherit properties and methods directly from other objects. In a Gamers Zone, game characters could inherit common properties and behaviors from a shared prototype object.
- Object Composition: Object composition can be used in a Gamers Zone to build complex gaming objects by combining simpler ones. For instance, a gaming setup object could be composed of a Console object, multiple Accessory objects, and a Gamer object.

ES 6

- Arrow Functions: Shorter syntax for writing function expressions.
- Template Literals: Enhanced string literals that support interpolation and multiline strings.
- Destructuring Assignment: Extracts values from arrays or objects into individual variables.
- Default Parameters: Allows default values to be specified for function parameters.
- Rest and Spread Operators: Collects elements into arrays or objects (rest) or spreads them out (spread).
- Object Literal Extensions: Shorthand syntax for defining methods and computed property names.
- Class Syntax: Simplified syntax for creating and working with classes and objects.
- Modules: Support for creating reusable modules with export and import statements.
- Promises: Provides a cleaner and more flexible approach to asynchronous programming.
- let and const: Block-scoped variables that offer better control over variable declarations.
- Enhanced for...of Loop: Iterates over iterable objects (arrays, strings, etc.) with a simpler syntax.
- Symbol: A new primitive type that allows the creation of unique and non-enumerable property keys.

- Block-scoped variables and constants (let and const keywords):
    - Gamers Zone wants to track the number of online players and the maximum number of players allowed per game.
    - Using let, they can declare a variable onlinePlayers to store the current number of online players and update it as players join or leave.
    - Using const, they can declare a constant maxPlayers to store the maximum number of players allowed per game, which remains the same throughout.
- Arrow functions (() => {} syntax):
    - Gamers Zone wants to display the player's nickname whenever a new player joins.
    - They can use an arrow function to define an event handler: const handlePlayerJoin = (nickname) => { console.log(nickname + " joined the game!"); }.
    - The arrow function syntax provides a concise way to define the function without using the function keyword.

- Default function parameters:
  - Gamers Zone wants to allow players to customize their nickname when joining the game but provide a default nickname if they don't choose one.
  - They can define the joinGame function with a default parameter: const joinGame = (nickname = "Guest") => { console.log(nickname + " joined the game!"); }.
  - If a player doesn't provide a nickname when joining, the default value "Guest" will be used.
- Rest and spread operators (...):
  - Gamers Zone wants to implement a function to calculate the total score of a team.
  - They can define a function calculateTotalScore with the rest parameter to accept an arbitrary number of scores: const calculateTotalScore = (...scores) => { /* calculate total score */ }.
  - The rest operator ... gathers all the individual scores into an array, which can then be used for calculations.

- Template literals:
  - Gamers Zone wants to display a personalized welcome message for each player.
  - They can use template literals to concatenate strings and variables: console.log(Welcome, ${nickname}! Enjoy your game.);.
  - Template literals allow embedding expressions and variables within a string using ${} syntax.
- Destructuring assignment:
  - Gamers Zone wants to retrieve the player's name and score from an object and use them in different parts of their application.
  - They can destructure the player object to extract the name and score: const { name, score } = player;.
  - This creates two variables (name and score) and assigns their values from the corresponding properties in the player object.

- Classes and inheritance (class keyword):
  - Gamers Zone wants to create a class for game characters, with specific properties and methods.
  - They can define a Character class using the class keyword, with properties like name and health, and methods like attack and heal.
- Modules (import and export statements):
  - Gamers Zone wants to organize their code into reusable modules for different game functionalities.
  - They can create a separate module for character-related functions and export the necessary functions using the export statement.
  - In another module, they can import those functions using the import statement, allowing access to the exported functionality.

- Promises:
  - Gamers Zone wants to handle asynchronous operations, such as fetching player data from a server.
  - They can use promises to handle these asynchronous operations more efficiently and avoid callback hell.
  - For example, they can create a promise to fetch player data and handle success and failure using then and catch methods.
- Iterators and generators:
  - Gamers Zone wants to create an iterator to loop through an array of game levels.
  - They can define a generator function that returns an iterator object using the function* syntax and yield keyword.
  - The iterator can then be used to iterate through the levels one by one.

- Symbols:
  - Gamers Zone wants to create unique identifiers for each game level.
  - They can use symbols to generate unique and immutable identifiers for each level, ensuring they are distinct from other identifiers in the system.
- Enhanced object literals:
  - Gamers Zone wants to define game levels with specific properties and methods.
  - They can use enhanced object literals to define levels concisely, including shorthand property syntax and computed property keys.
- Map and Set data structures:
  - Gamers Zone wants to keep track of unique players who have joined the game.
  - They can use a Set data structure to store unique player objects, eliminating duplicates automatically.
  - Additionally, they can use a Map data structure to map player IDs to their corresponding player objects for efficient lookup.

- Array methods:
  - Gamers Zone wants to find a player with a specific ID from an array of players.
  - They can use the find method provided by the Array object to find the player matching the given criteria.
  - The findIndex method can also be used to find the index of the player in the array.
- String methods:
  - Gamers Zone wants to check if a player's nickname starts with a certain prefix.
  - They can use the startsWith method provided by the String object to perform the desired check conveniently.
- Binary and octal literals:
  - Gamers Zone wants to display the number of players in binary and octal formats.
  - They can use binary and octal literals by prefixing the numbers with 0b and 0o, respectively.

- Enhanced error handling:
  - Gamers Zone wants to handle different types of errors separately when validating player input.
  - They can use multiple catch clauses in a try-catch block to catch specific types of exceptions and handle them accordingly.
- Proxy and Reflect objects:
  - Gamers Zone wants to intercept and customize operations on game objects, such as logging property access or preventing certain modifications.
  - They can use the Proxy object to intercept and handle operations performed on a target object.
  - The Reflect object provides methods that mirror common object operations, enabling more flexible and controlled object manipulation.

- Array and object destructuring in function parameters:
  - Gamers Zone wants to define a function that takes an object as a parameter and extracts specific properties within the function body.
  - They can use destructuring assignment directly in the function parameter definition to extract the desired properties in a concise manner.
- Tail-call optimization:
  - Gamers Zone wants to implement a recursive function to calculate the total score of a team without causing stack overflow errors.
  - They can use tail-call optimization to optimize the recursive function, allowing it to reuse the same stack frame for consecutive recursive calls.

Special Functions

- setInterval(): Executes a function repeatedly at a specified interval.
- clearInterval(): Stops the execution of a function that is repeatedly executed using setInterval().
- setTimeout(): Executes a function once after a specified delay.
- clearTimeout(): Cancels the execution of a function scheduled to run after a specified delay using setTimeout().