```
console.log('Compconste JavaScript Course')

// https://youtu.be/lGmRnu--iU8?t=2852

//=-=--=-=-==-=-=-=-==--=-=-=-=-=-==-=-=

// constiable,constant and Comment in js

const score = 34;

const score = 54; // redeclear and this is not use

console.log(score)

//===========END===============

// const and const

//==========================

//03 Data type in Values

//===========END===============

//04 String in js

console.log("Hello Coders")

const firstName = "Owais"

const lastName = "Programmer"

console.log(firstName, lastName)

//===========END===============

//05 String Contination in js

const fullName = firstName + " " + lastName

console.log(fullName)

//===========END===============

//06 Method  2 Using Templates Literals

console.log(`hello ${firstName} is ${lastName}`)
```

```
//===========END===============

//07 Getting String Character in js

console.log(firstName[2])

//===========END===============

//08 String Methods in JavaScript

console.log(firstName.toUpperCase())

console.log(lastName.indexOf('r'))

//===========END===============

//09 Common String Methods in JavaScript

const hobies = '   coding reading running  '

//trim Methods in JavaScript

const result = hobies.trim() //remove Extra spaces

console.log(hobies)

console.log(result)

//===========END===============

//10 Include Methods in JavaScript (check string exist or Not exist) Case Sensitive

console.log(result.includes('reading'))

//===========END===============

//11 Slice Methods in JavaScript

const FullName = " Muhammad Owais Rao"

// console.log(FullName.slice(0, 9)); //Excluding the Last One

const result = FullName.slice(0, 15)

//important Note Does Not Mutate original String

console.log("Original String = ", FullName);
console.log("Extracted String = ", result);

//===========END===============
```

//12 String "Split" Method in JavaScript

```
const favoriteColors = "Brown Blue Black Green"

const arrColors = favoriteColors.split(' ') //String convert to Array

console.log(arrColors);

//===========END===============
```

// 13 JavaScript String are Immutable String (means not Modify)

```
const str = 'Hello'

str[0] = 'p'

str[1] = 'q'


//there will be no change in 'str' constiable

console.log(str);

//===========END===============
```

// 14 Numbers

```
const score = 50

console.log(score, typeof score);

//===========END===============
```

//15 Mathematical Expression

```
const result = score * 2 + (4 * 3) - 8 / 2 % 4;

//Using Parioty and Precedence

//1 () Brackets

//2 ** Power Operator
```

```
//3 * / % (From Left to Right)

//4 + - (From Left to Right)


console.log(result);

//===========END===============
```

//16 Concatination of Nmbers

```
const resultline = 'My Total Score is = ' + result;

console.log(resultline);

//===========END===============
```

// 17 Loose Equality (==) Vs Strict Equality Operator (===)

//Loose Equality (==)

```
const age = 22; //Number Type Value

console.log(age == 22); //Focus Only On Value But Not Type
```

//Strict Equality Operator (===)

```
const age = 22;

console.log(age === "22"); //Focus on Both Value

//===========END===============
```

//18 Type Conversion

```
const stringType = "54"

console.log(stringType, typeof stringType);
```

//Number Method

```
const numberType = Number(stringType)

console.log(numberType, typeof numberType);

//https://youtu.be/lGmRnu--iU8?t=2852 47: 32

//===========END===============
```

//19 Array IN JavaScript

//Creating an Array

```
const fruits = ['applw', 'Banana', 'orange'];
```

//Accesing Elements in an array

```
console.log(fruits[0]); // Output: 'apple'

console.log(fruits[1]); // Output: 'banana'

console.log(fruits[2]); // Output: 'orange'
```

//Adding elements in an array

```
fruits.push('grape');

console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']
```

//Removing elements From an array

```
fruits.pop();

console.log(fruits); // Output: ['apple', 'banana', 'orange']
```

//Finding the lenght of an array

```
console.log(fruits.length); // Output: 3
```

```
//Looping through the array
for (const i = 0; i < fruits.length; i++) {
   console.log(fruits[i]);
}


//Sorting an Array
fruits.sort();
console.log(fruits); // Output: ['apple', 'banana', 'orange']


//Reversing an array
fruits.reverse();
console.log(fruits); // Output: ['orange', 'banana', 'apple']
//===========END===============
//20 Boolean Value and Comparision Operator


//Boolean Value
const isTrue = true;
const isFalse = false;


//Comparison Operator


const num1 = 10;
const num2 = 5;
```

```
console.log(num1 > num2); // Output: true
console.log(num1 < num2); // Output: false
console.log(num1 >= num2); // Output: true
console.log(num1 <= num2); // Output: false
console.log(num1 == num2); // Output: false
console.log(num1 != num2); // Output: true

//Logical Operator

const isSunny = true;
const isWarm = false;

console.log(isSunny && isWarm); // Output: false
console.log(isSunny || isWarm); // Output: true
console.log(!isSunny); // Output: false

//Conditional operator

const age = 18;

if (age >= 18) {
  console.log('You are an adult.');
} else {
  console.log('You are not an adult yet.');
}


//Ternary operator


const age = 18;

const message = (age >= 18) ? 'You are an adult.' : 'You are not an adult yet.';

console.log(message); // Output: 'You are an adult.'

//===========END===============

//21 Loop in javascript


//For Loops

for (const i = 0; i < 5; i++) {

   console.log(i);

}


//While Loops

const i = 0;

while (i < 5) {

   console.log(i);

   i++;

}


//Do-While Loops

const i = 0;
```

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

//For-in-loop(For iteratinng over object properties)
const person = {
   name: 'John',
   age: 30,
   gender: 'male'
};

for (const prop in person) {
   console.log(prop + ': ' + person[prop]);
}

//For-of-loop(For iterating over arrays)
const fruits = ['apple', 'banana', 'orange'];

for (const fruit of fruits) {
   console.log(fruit);
}

//Neated Loops
```

```
for (const i = 0; i < 3; i++) {
   for (const j = 0; j < 3; j++) {
      console.log(i + ',' + j);
   }
}
//===========END===============
```

//22 Condition in JavaScript

//If Statement

```
const age = 18;

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

// If-else statements

```
const age = 16;

if (age >= 18) {
   console.log('You are an adult.');
} else {
   console.log('You are not an adult yet.');
}
```

```
//If-else if-else statements
const grade = 80;

if (grade >= 90) {
    console.log('A');
} else if (grade >= 80) {
    console.log('B');
} else if (grade >= 70) {
    console.log('C');
} else {
    console.log('F');
}

//Switch statements
const day = 'Monday';

switch (day) {
    case 'Monday':
        console.log('Today is Monday.');
        break;
    case 'Tuesday':
        console.log('Today is Tuesday.');
        break;
    case 'Wednesday':
        console.log('Today is Wednesday.');
        break;
    default:
        console.log('Today is not Monday, Tuesday, or Wednesday.');
}
//===========END===============
```

//23 Logical Operator in Javascript

// AND Operator(&&)

```
const isSunny = true;
const isWarm = false;

if (isSunny && isWarm) {
    console.log('It is sunny and warm.');
} else {
    console.log('It is not sunny and warm.');
}
```

//OR Operator(||)

```
const isSunny = true;
const isWarm = false;

if (isSunny || isWarm) {
    console.log('It is either sunny or warm.');
} else {
  console.log('It is neither sunny nor warm.');
}

//NOT Operator(!)
const isSunny = true;

if (!isSunny) {
  console.log('It is not sunny.');
} else {
  console.log('It is sunny.');
}

//Combinig Operator
const isSunny = true;
const isWarm = false;
const isHumid = true;

if ((isSunny && isWarm) || isHumid) {
  console.log('It is either sunny and warm or humid.');
} else {
  console.log('It is neither sunny and warm nor humid.');
}
//==========END==============
```

//24 constiable and block scope in Javascript

//Global

```
const name = 'John';

function sayName() {
    console.log(name);
}

sayName(); // Output: 'John'
```

//Local Scope

```
function sayName() {
    const name = 'John';
    console.log(name);
}

sayName(); // Output: 'John'
console.log(name); // Output: Uncaught ReferenceError: name is not defined

//Block scope(using const and const)
if (true) {
    const name = 'John';
    const age = 30;
    console.log(name); // Output: 'John'
    console.log(age); // Output: 30
}

console.log(name); // Output: Uncaught ReferenceError: name is not defined
console.log(age); // Output: Uncaught ReferenceError: age is not defined

//Function scope(using const)
function sayName() {
    const name = 'John';
    console.log(name);
}

sayName(); // Output: 'John'
console.log(name); // Output: Uncaught ReferenceError: name is not defined

//Nested Scope
function sayName() {
    const name = 'John';

    function sayAge() {
        const age = 30;
        console.log(name + ' is ' + age + ' years old.');
    }
    sayAge();
}

sayName(); // Output: 'John is 30 years old.'
//===========END===============
```

//25 Function and Function Expression in JavaScript

```
//decleration
function sayHello() {
    console.log('Hello!');
}

sayHello(); // Output: 'Hello!'

//Function Expression
const sayHello = function () {
    console.log('Hello!');
};

sayHello(); // Output: 'Hello!'
//===========END===============
```

//26 Arrow Function

```
/*
```

```
Specail Form of Function expression

It allow us to write Function more fast because

no need to use function keyword

No need to use paranthesis() in case of single parameter

No need to use curely{} if single line code in function

No need to use return statement if single line code in function

*/


console.log('Hello, World!');


const sayHello = () => {

   console.log('Hello!');

};


sayHello(); // Output: 'Hello!'


//Function with Parmeter

function sayHello(name) {

   console.log('Hello, ' + name + '!');

}


sayHello('John'); // Output: 'Hello, John!'


//Function with return Value
```

```
function addNumbers(num1, num2) {
  return num1 + num2;
}

const result = addNumbers(5, 10);
console.log(result); // Output: 15

//Function eith default Parameter
function sayHello(name = 'World') {
  console.log('Hello, ' + name + '!');
}

sayHello(); // Output: 'Hello, World!'
sayHello('John'); // Output: 'Hello, John!'

//Function with rest Parameter
function addNumbers(...numbers) {
  const sum = 0;
  for (const number of numbers) {
    sum += number;
  }
  return sum;
}
```

```
const result = addNumbers(5, 10, 15);
console.log(result); // Output: 30
//===========END===============
```

//27 Function Parameter and Argument in JavaScript

```
//function with one
function sayHello(name) {
   console.log('Hello, ' + name + '!');
}

sayHello('John'); // Output: 'Hello, John!'

//Function with multiple Parmeter
function addNumbers(num1, num2) {
   console.log(num1 + num2);
}

addNumbers(5, 10); // Output: 15

//Function with default parameter
function sayHello(name = 'World') {
   console.log('Hello, ' + name + '!');
}
```

```
sayHello(); // Output: 'Hello, World!'
sayHello('John'); // Output: 'Hello, John!'

//Function with rest Parameter
function addNumbers(...numbers) {
  const sum = 0;
  for (const number of numbers) {
    sum += number;
  }
  console.log(sum);
}

addNumbers(5, 10, 15); // Output: 30

//function with callback function as parameter
function sayHello(name, callback) {
  console.log('Hello, ' + name + '!');
  callback();
}

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

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


//===========END===============
```

//28 HighOrder function-CallBack

//Higher-order function

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

function greet(callback) {
   const name = 'John';
   callback(name);
}

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

//CallBack Function

```
function addNumbers(num1, num2) {
   return num1 + num2;
}

function multiplyNumbers(num1, num2, callback) {
   const result = callback(num1, num2);
```

```
  console.log(result);
}

multiplyNumbers(5, 10, addNumbers); // Output: 15

//Array method with CallBack

const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(number => number * 2);

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

//Higher-Order function with Callback and closure

function createCounter() {
  const count = 0;

  function increment() {
    count++;
    console.log(count);
  }

  return increment;
```

```
}


const counter = createCounter();


counter(); // Output: 1

counter(); // Output: 2

counter(); // Output: 3

//===========END===============
```

//29 HighOrder Function Returning Function

// Basic Example

```
function createGreeting(greeting) {

   return function (name) {

      console.log(greeting + ', ' + name + '!');

   };

}


const sayHello = createGreeting('Hello');

const sayGoodbye = createGreeting('Goodbye');


sayHello('John'); // Output: 'Hello, John!'

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

//Example with Arrow Function

```
const createGreeting = greeting => name => console.log(greeting + ', ' + name + '!');

const sayHello = createGreeting('Hello');
const sayGoodbye = createGreeting('Goodbye');

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

//Example with clouser
function createCounter() {
  const count = 0;

  return function () {
    count++;
    console.log(count);
  };
}

const counter1 = createCounter();
const counter2 = createCounter();

counter1(); // Output: 1
counter1(); // Output: 2
counter2(); // Output: 1
```

```
counter2(); // Output: 2
//==========END==============
//30 IIFE (Immediately Invoked Function Expression)
//Exceuated Only Once (Never Agian run)

//Basic Example
(function () {
   console.log('Hello, World!');
})();

//Example with Parameter
(function (name) {
   console.log('Hello, ' + name + '!');
})('John');

//Example with returning Value
const result = (function (num1, num2) {
   return num1 + num2;
})(5, 10);

console.log(result); // Output: 15

//Example with Arrow Function
((name) => {
    console.log('Hello, ' + name + '!');
})('John');
//===========END===============
//31 SetTimeOut and setInterval in JavaScript
/*
setTimeOut -> Run Function "once" after "interval" of time
setInterval -> Run function repeatedly,Starting after the interval
of time , then repeating
*/

//SetTimeOut example
function sayHello() {
    console.log('Hello, World!');
}

setTimeout(sayHello, 3000); // Output: 'Hello, World!' (after 3 seconds)

//setInterval javaSscript
const count = 0;

function incrementCount() {
    count++;
    console.log(count);
}
```

```
setInterval(incrementCount, 1000); // Output: 1, 2, 3, 4, 5... (every second)


//setTimeout with anonymous Function:
setTimeout(function () {
  console.log('Hello, World!');
}, 3000); // Output: 'Hello, World!' (after 3 seconds)


//setInterval with arrow Function
const count = 0;

setInterval(() => {
  count++;
  console.log(count);
}, 1000); // Output: 1, 2, 3, 4, 5... (every second)
//===========END===============


//32 Hoisting
/*
constiable "decleration" are "hoisting" toward "top" of their scope
*/


//Function Decleration in JavaScript
sayHello();
```

```
function sayHello() {
  console.log('Hello, World!');
}

//constiable Decleration in JavaScript
console.log(naame); // Output: undefined
const naame = 'John';

//Function Expression in JavaScript
sayHello();

const sayHello = function () {
  console.log('Hello, World!');
};

//Block Scope
if (true) {
  const namme = 'John';
}

console.log(namme); // Output: 'John'
//===========END===============
```

//33 Object Introduction in javaScript

```
//Object Literals
const person = {
    name: 'John',
    age: 30,
    sayHello: function () {
        console.log('Hello, my name is ' + this.name + '!');
    }
};

console.log(person.name); // Output: 'John'
person.sayHello(); // Output: 'Hello, my name is John!'

//Object construction
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello = function () {
        console.log('Hello, my name is ' + this.name + '!');
    };
}

const person = new Person('John', 30);
```

```
console.log(person.name); // Output: 'John'

person.sayHello(); // Output: 'Hello, my name is John!'
```

//Object Methods

```
const calculator = {

  num1: 0,

  num2: 0,

  setNumbers: function (num1, num2) {

    this.num1 = num1;

    this.num2 = num2;

  },

  add: function () {

    return this.num1 + this.num2;

  },

  subtract: function () {

    return this.num1 - this.num2;

  }

};


calculator.setNumbers(5, 10);

console.log(calculator.add()); // Output: 15

console.log(calculator.subtract()); // Output: -5
```

//Object Inheritance

```
function Animal(name) {

   this.name = name;

}


Animal.prototype.sayHello = function () {

   console.log('Hello, my name is ' + this.name + '!');

};


function Dog(name, breed) {

   Animal.call(this, name);

   this.breed = breed;

}


Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.constructor = Dog;


Dog.prototype.bark = function () {

   console.log('Woof!');

};


const dog = new Dog('Fido', 'Labrador');


console.log(dog.name); // Output: 'Fido'

console.log(dog.breed); // Output: 'Labrador'
```

```
dog.sayHello(); // Output: 'Hello, my name is Fido!'
dog.bark(); // Output: 'Woof!'
//===========END===============
```

//34 Function VS Method in JavaScript

```

//Function:
function sayHello(name) {
   console.log('Hello, ' + name + '!');
}

sayHello('John'); // Output: 'Hello, John!'

//Method:
const person = {
   name: 'John',
   sayHello: function () {
      console.log('Hello, my name is ' + this.name + '!');
   }
};

person.sayHello(); // Output: 'Hello, my name is John!'

//Function with return value
function addNumbers(num1, num2) {
    return num1 + num2;
}

const result = addNumbers(5, 10);

console.log(result); // Output: 15


//Method with Object Property
const person = {
    name: 'John',
    age: 30,
    getAge: function () {
        return this.age;
    }
};

const age = person.getAge();

console.log(age); // Output: 30

//deconste the property
// deconste person.name
//console.log(person.name); // Output: undefined
```

```
//===========END===============
```

//35 This Keyword in JavaScript

//Global Context

```
console.log(this); // Output: Window
```

//Function Context

```
function sayHello() {
  console.log(this);
}

sayHello(); // Output: Window
```

//Object context

```
const person = {
  name: 'John',
  sayHello: function () {
    console.log(this);
  }
};

person.sayHello(); // Output: {name: 'John', sayHello: ƒ}
```

//Contruction Context

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.sayHello = function () {
    console.log('Hello, my name is ' + this.name + '!');
  };
}

const person = new Person('John', 30);

person.sayHello(); // Output: 'Hello, my name is John!'

//Event Context
const button = document.querySelector('button');

button.addEventListener('click', function () {
  console.log(this);
}); // Output: <button>Click me</button>
//===========END===============
//36 ForEach Method in JavaScript
//Basic example
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function (number) {
  console.log(number);
}); // Output: 1, 2, 3, 4, 5

//Example with arrow Function
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(number => console.log(number)); // Output: 1, 2, 3, 4, 5

//Example with index
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function (number, index) {
  console.log('Index ' + index + ': ' + number);
}); // Output: Index 0: 1, Index 1: 2, Index 2: 3, Index 3: 4, Index 4: 5

//Example with object
const people = [
  { name: 'John', age: 30 },
  { name: 'Mary', age: 25 },
  { name: 'Bob', age: 40 }
];

people.forEach(function (person) {
  console.log(person.name + ' is ' + person.age + ' years old.');
```

```
}); // Output: 'John is 30 years old.', 'Mary is 25 years old.', 'Bob is 40 years old.'
//===========END===============
```

//37 Object Inside Array in JavaScript

//Basic Example

```
const myArray [
  { name: "John", age: 25 },
  { name: "Jane", age: 30 },
  { name: "Bob", age: 40 }
];

console.log(myArray[0].name); // Output: John
console.log(myArray[1].age); // Output: 30
console.log(myArray[2].name); // Output: Bob
//===========END===============
```

//38 MATH Object in JavaScrript

//Generating a random number between 0 and 1

```
const randomNum = Math.random();
console.log(randomNum); // Output: a random number between 0 and 1
```

//Rounding a number to the nearst integer

```
const num = 3.7;
const roundedNum = Math.round(num);
console.log(roundedNum); // Output: 4
```

```
//Finding the maximum or minimum value in a array
const numbers = [5, 2, 8, 1, 9];
const maxNum = Math.max(...numbers);
const minNum = Math.min(...numbers);
console.log(maxNum); // Output: 9
console.log(minNum); // Output: 1

//Generating a random integer between two values
const min = 1;
const max = 10;
const randomInt = Math.floor(Math.random() * (max - min + 1)) + min;
console.log(randomInt); // Output: a random integer between 1 and 10

//Calculating the square root of a number
const num = 16;
const squareRoot = Math.sqrt(num);
console.log(squareRoot); // Output: 4
//===========END===============
```

//39 Call.Apply and Bind in Javascritp

```
//Call` Example:
const person = {
   name: "John",
   age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
};

const anotherPerson = {
  name: "Jane",
  age: 25
};

person.greet.call(anotherPerson); // Output: Hello, my name is Jane and I am 25 years old.

//apply example
const numbers = [1, 2, 3, 4, 5];

const sum = function (a, b, c, d, e) {
  return a + b + c + d + e;
};

const total = sum.apply(null, numbers);
console.log(total); // Output: 15

//bind example
const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
};

const anotherPerson = {
  name: "Jane",
  age: 25
};

const greetAnotherPerson = person.greet.bind(anotherPerson);
greetAnotherPerson(); // Output: Hello, my name is Jane and I am 25 years old.

//===========END===============
```

//40 Pass by Value and Pass by reference in JavaScript

//Pass By Value:

```
const num1 = 10;
const num2 = num1;

num2 = 20;

console.log(num1); // Output: 10
console.log(num2); // Output: 20

//Pass By Reference:
const arr1 = [1, 2, 3];
const arr2 = arr1;

arr2.push(4);

console.log(arr1); // Output: [1, 2, 3, 4]
console.log(arr2); // Output: [1, 2, 3, 4]

//Pass by reference example with object:
const obj1 = { name: "John", age: 30 };
const obj2 = obj1;

obj2.age = 25;

console.log(obj1); // Output: { name: "John", age: 25 }
console.log(obj2); // Output: { name: "John", age: 25 }
//==========END==============
//41 For in Loop in Javascript
//object's Properties
const person = {
  name: "John",
  age: 30,

  city: "New York"

};


for (const prop in person) {

  console.log(`${prop}: ${person[prop]}`);

}


// Output:

// name: John

// age: 30

// city: New York


//Looping through an array's indicates:

const numbers = [1, 2, 3, 4, 5];


for (const index in numbers) {

  console.log(index);

}


// Output:

// 0

// 1

// 2
// 3
// 4
```

//Looping through an arrray's elements:

```
const numbers = [1, 2, 3, 4, 5];

for (const index in numbers) {
   console.log(numbers[index]);
}

// Output:
// 1
// 2
// 3
// 4
// 5
//===========END===============
```

//42 DOM Introduction in Javascript(Document Object Model)

//Accesing an element By ID:

```
const element = document.getElementById("myElement");
```

//Accesing element by ClassName

```
const elements = document.getElementsByClassName("myClass");
```

```
//Accessing element by TagName

const elements = document.getElementsByTagName("p");


//Changing the text content of an element:

const element = document.getElementById("myElement");

element.textContent = "Hello, world!";


//Adding an event listener to an element

const button = document.getElementById("myButton");

button.addEventListener("click", function () {

   console.log("Button clicked!");

});


//===========END===============

//43 Query Selector in Javascript

//To select all example with a specific class name in javascript,

//you can use the queryselectorAll() method,

const elements = document.querySelectorAll('.example');


/*

This will select all elements with the class name "example" and store them in the elements

 constiable as a NodeList. You can then loop through this NodeList to access each individual
element:e

 */
```

```
elements.forEach(element => {
  // Do something with each element
});

/*
You can also use other CSS selectors with querySelectorAll()
 to select elements based on their tag name, ID, or other attributes. For example:
 */

// Select all <p> elements
const paragraphs = document.querySelectorAll('p');

// Select the element with ID "my-element"
const myElement = document.querySelector('#my-element');

// Select all elements with the "active" attribute
const activeElements = document.querySelectorAll('[active]');

//==========END==============
//44 OtherWay to access element in javascript
const elements = document.getElementsByClassName('example');

/**
```

* This will select all elements with the class name "example" and store them in the elements

constiable as an HTMLCollection. You can then loop through this HTMLCollection to access each individual element:

*/

```
for (const i = 0; i < elements.length; i++) {
   const element = elements[i];
   // Do something with each element
}
```

//===========END===============

//45 InnerText Vs InnerHTML in JavaScript

///InnerText return the text content of an element without any Html tags.For Example

```
<div id="my-element">This is <strong>bold</strong> text.</div>


const element = document.getElementById('my-element');
const text = element.innerText; // "This is bold text."
```

//innerHTML returns the HTML content of an element,including any HTML tags

```
<div id="my-element">This is <strong>bold</strong> text.</div>


const element = document.getElementById('my-element');
const html = element.innerHTML; // "This is <strong>bold</strong> text."
```

/*

* Note the innerHTML will also return the HTML content of any child

```
 * elements,uncluding their tags and attributes

 */

/**

 * When setting the content of an element,you can use either innerText or inner

 * or innerHTML to set the content,depending on whether you want to include HTML tags or not

 */


const element = document.getElementById('my-element');

element.innerText = 'New text'; // Sets the text content to "New text"

element.innerHTML = '<em>New text</em>'; // Sets the HTML content to "<em>New text</em>"

//===========END===============
```

//46 Getting an Setting in JavaScript

```
/**

 * In JavaScript, you can get and set attributes of HTML elements using the

getAttribute() and setAttribute() methods.


 To get the value of an attribute, you can use the getAttribute()

 method. For example, to get the value of the src attribute of an image element with the ID

myImage, you can use the following code:

 */

const imageSrc = document.getElementById("myImage").getAttribute("src");
```

```
/**

 * To set the value of an attribute, you can use the setAttribute()

 method. For example, to set the value of the src attribute of an image element with the ID

myImage, you can use the following code:

 */


document.getElementById("myImage").setAttribute("src", "newImage.jpg");



/**

 * You can also use the dot notation to get and set attributes. For example, to get the value of the

src attribute of an image element with the ID myImage

, you can use the following code:

 */

const imageSrc = document.getElementById("myImage").src;



/**

 * To set the value of the src

 attribute using dot notation, you can use the following code:

 */


document.getElementById("myImage").src = "newImage.jpg";

//==========END==============
```

//47 Adding the Style in JavaScript

```
/**
 * In JavaScript, you can add styles to HTML elements using the style
 property. The style property is an object that represents the inline style of an element.
 */

document.getElementById("myElement").style.backgroundColor = "red";

document.getElementById("myElement").style.cssText = "background-color: red; font-size: 16px;";

/**
 * Alternatively, you can create a CSS rule dynamically and add it to the document's stylesheet using the
insertRule() method. For example, to create a CSS rule that sets the background color of all elements with the class
myClass to red, you can use the following code:
 */

const sheet = document.createElement('style');
sheet.innerHTML = ".myClass { background-color: red; }";
document.head.appendChild(sheet);
//===========END===============
```

//48 Add,Remove and Replace in javacscript

```
/**
 * In JavaScript, you can add, remove, and replace classes of HTML elements using the
classList property. The classList property is an object that represents the class attribute of an
```

```
element.
 */

/**
 * To add a class to an element, you can use the add()
 method of the classList object. For example, to add the class
active to an element with the ID myElement, you can use the following code:
 */
document.getElementById("myElement").classList.add("active");

/**
 * To remove a class from an element, you can use the remove()
 method of the classList object. For example, to remove the class
active from an element with the ID myElement, you can use the following code:
 */
document.getElementById("myElement").classList.remove("active");

/**
 * To replace a class of an element with another class, you can use the
replace() method of the classList object. For example, to replace the class
active of an element with the ID myElement with the classinactive
, you can use the following code:
 */
document.getElementById("myElement").classList.replace("active", "inactive");
```

```
/**
 * You can also check if an element has a specific class using the
contains() method of the classList object. For example, to check if an element with the ID
myElement has the class active, you can use the following code:
 */
const hasActiveClass = document.getElementById("myElement").classList.contains("active");


/**
 * Note that the classList property is not supported in older versions of Internet Explorer. In
those cases, you can use the
className property to add, remove, and replace classes. For example, to add the class
active to an element with the ID myElement using the className
 property, you can use the following code:
 */
document.getElementById("myElement").className += " active";
/**
 * To remove the class active from an element with the ID
myElement using the className property, you can use the following code:
 */
document.getElementById("myElement").className =
document.getElementById("myElement").className.replace(" active", "");


/**
  * To replace the class active of an element with the class
```

inactive using the className property, you can use the following code:

*/

```
document.getElementById("myElement").className =
document.getElementById("myElement").className.replace(" active", " inactive");
```

//===========END===============

//49 Parent Children and Sibling in javascript

/**

* In JavaScript, you can access the parent, children, and siblings of an HTML element using properties and methods.

*/

/**

* To access the parent element of an element, you can use the

parentNode property. For example, to get the parent element of an element with the ID

myElement, you can use the following code

*/

```
const parentElement = document.getElementById("myElement").parentNode;
```

/**

* To access the children elements of an element, you can use the

children property. For example, to get the children elements of an element with the ID

myElement, you can use the following code:

*/

```
const childElements = document.getElementById("myElement").children;
```

/**

* To access the first child element of an element, you can use the

```
firstElementChild property. For example, to get the first child element of an element with the ID

myElement, you can use the following code:

 */

const firstChildElement = document.getElementById("myElement").firstElementChild;

/**

 * To access the last child element of an element, you can use the

lastElementChild property. For example, to get the last child element of an element with the ID

myElement, you can use the following code:

 */

const lastChildElement = document.getElementById("myElement").lastElementChild;

/**

 * To access the previous sibling element of an element, you can use the

previousElementSibling property. For example, to get the previous sibling element of an
element with the ID

myElement, you can use the following code:

 */

const previousSiblingElement =
document.getElementById("myElement").previousElementSibling;

/**

 * To access the next sibling element of an element, you can use the

nextElementSibling property. For example, to get the next sibling element of an element with
the ID

myElement, you can use the following code:

 */

const nextSiblingElement = document.getElementById("myElement").nextElementSibling;
```

```
/**

 * Note that thechildren, firstElementChild, lastElementChild, previousElementSibling

, and nextElementSibling properties are not supported in older versions of Internet Explorer. In those cases, you can use the

childNodes property to access the children elements and the

previousSibling and nextSibling properties to access the sibling elements.

 */

//===========END===============
```

//50 Events Basic in javascript

```
/**

 * In JavaScript, you can handle events that occur in the browser using event listeners. Event listeners are functions that are executed when a specific event occurs, such as a user clicking a button or a page finishing loading.


Here are some basic examples of how to handle events in JavaScript:

 */
```

//Click Event

```
document.getElementById("myButton").addEventListener("click", function () {

   // Code to execute when the button is clicked

});
```

//Load Event

```
window.addEventListener("load", function () {

   // Code to execute when the page finishes loading

});
```

```
//Mouseoever Event
document.getElementById("myElement").addEventListener("mouseover", function () {
   // Code to execute when the mouse is over the element
});
//Keydown Event:
document.addEventListener("keydown", function (event) {
   // Code to execute when a key is pressed
});
/**
 * In this example, the event
 parameter contains information about the key that was pressed, such as the key code and
whether the shift key was pressed.
 */
//Submit Event
document.getElementById("myForm").addEventListener("submit", function (event) {
   // Code to execute when the form is submitted
   event.preventDefault(); // Prevents the form from submitting normally
});
/**
 * In this example, the event parameter contains information about the form submission, such
as the form data.


Note that there are many other types of events that can be handled in JavaScript, such as
mousemove, scroll, and resize. You can find a full list of events in the JavaScript documentation.
 */
```

//===========END===============

//51 Creating and Removing Element all example in javascript

/**

 * In JavaScript, you can create and remove HTML elements using the

createElement(), appendChild(), and removeChild() methods.

 */

//Creating Elements:

/**

 * To create a new HTML element, you can use the

createElement() method.

For example, to create a new div element, you can use the following code:

 */

```
const newDiv = document.createElement("div");
```

/**

 * You can then set attributes and content for the new element using properties such as

id, className, and innerHTML.

 For example, to set the id and className attributes and the inner HTML content of the new

div element, you can use the following code:

 */

```
newDiv.id = "myNewDiv";

newDiv.className = "newDivClass";

newDiv.innerHTML = "This is a new div element.";
```

/**

 * Finally, you can add the new element to the HTML document using the

appendChild() method. For example, to add the new div element to the body of the document, you can use the following code:

*/

```
document.body.appendChild(newDiv);
```

//Removing Elements:

/**

* To remove an HTML element from the document, you can use the

removeChild() method.

For example, to remove an element with the ID myElement

from the document, you can use the following code:

*/

```
const elementToRemove = document.getElementById("myElement");

elementToRemove.parentNode.removeChild(elementToRemove);
```

/**

* In this example, the parentNode property is used to get the parent element of the element to be removed, and the

removeChild() method is used to remove the element from the parent element.

Note that you can also use the remove() method to remove an element directly, without needing to access its parent element. However, this method is not supported in older versions of Internet Explorer.

*/

//===========END==============

//52 Bubbling and Delegation in Javascript

/**

* In JavaScript, event bubbling and event delegation are techniques used to handle events

efficiently.

*/

//Event Bubbling

/**

* Event bubbling is a mechanism where an event that is triggered on a child element is propagated up the DOM tree to its parent elements. This means that if you have a nested set of elements, and an event is triggered on a child element, the event will also be triggered on all of its parent elements.

*

To handle an event using event bubbling, you can add an event listener to a parent element and use the

event.target property to determine which child element triggered the event. For example, to handle a click event on a list of items where each item is a child element of an unordered list element, you can use the following code:

*/

```
document.querySelector("ul").addEventListener("click", function (event) {
  if (event.target.tagName === "LI") {
    // Code to execute when an item is clicked
  }
});
```

/**

* In this example, the event listener is added to the unordered list element, and the

event.target property is used to determine if the clicked element is an li element. If it is, the code to execute is run.

*/

//Event Delegation

/**

* Event delegation is a technique where you add an event listener to a parent element and use the

event.target property to determine which child element triggered the event. This is similar to event bubbling, but instead of relying on the event to bubble up the DOM tree, you use the parent element to handle the event.

To handle an event using event delegation, you can add an event listener to a parent element and use the

event.target property to determine which child element triggered the event. For example, to handle a click event on a list of items where each item is a child element of an unordered list element, you can use the following code:

*/

```
document.querySelector("ul").addEventListener("click", function (event) {

   if (event.target.tagName === "BUTTON") {

      // Code to execute when a button is clicked

   }

});
```

/**

* In this example, the event listener is added to the unordered list element, and the

event.target property is used to determine if the clicked element is a

button element. If it is, the code to execute is run.

Event delegation is useful when you have a large number of child elements that need to handle the same event, as it allows you to handle the event on a single parent element instead of adding an event listener to each child element.

*/

//===========END===============

//53 Sublit Event all Example in Javascript

/**

* In JavaScript, you can handle the submit event of a form using the

submit event and the preventDefault() method.

To handle the submit event of a form, you can add an event listener to the form element and use the

preventDefault() method to prevent the form from submitting normally. For example, to handle the submit event of a form with the ID

myForm, you can use the following code:

*/

```
document.getElementById("myForm").addEventListener("submit", function (event) {

   event.preventDefault();

   // Code to execute when the form is submitted

});
```

/**

* In this example, the event.preventDefault()

method is used to prevent the form from submitting normally, which would cause the page to reload. Instead, the code to execute when the form is submitted is run.

*/

```
/**

 * You can then access the form data using the FormData

 object or by accessing the form elements directly. For example, to access the value of an input
element with the name

myInput, you can use the following code:

 */

const inputValue = document.getElementById("myInput").value;


/**

 * You can also use the submit() method to submit a form programmatically. For example, to
submit a form with the ID

myForm programmatically, you can use the following code:

 */

document.getElementById("myForm").submit();

/**

 * Note that the submit() method will trigger the submit

 event, so any event listeners attached to the form will be executed.

 */

//===========END===============
```

//54 Regular Expression all Example in JAvascript

```
/**

 * In JavaScript, regular expressions are used to match patterns in strings. Regular expressions
are defined using a syntax that consists of characters and special characters that represent
patterns.

Here are some basic examples of how to use regular expressions in JavaScript:

 */
```

```
//Matching a Pattern
const str = "Hello, world!";
const pattern = /hello/i;
const result = str.match(pattern);

//Replacing a Pattern
const str = "Hello, world!";
const pattern = /hello/i;
const replacement = "hi";
const result = str.replace(pattern, replacement);

//Testing a Pattern
const str = "Hello, world!";
const pattern = /hello/i;
const result = pattern.test(str);

//===========END===============
//55 Array Method all example in JavaScript
//Push():
const fruits = ["apple", "banana"];
fruits.push("orange", "kiwi");
console.log(fruits); // Output: ["apple", "banana", "orange", "kiwi"]

//pop():
```

```
const fruits = ["apple", "banana", "orange"];
const lastFruit = fruits.pop();
console.log(lastFruit); // Output: "orange"
console.log(fruits); // Output: ["apple", "banana"]

//shift():
const fruits = ["apple", "banana", "orange"];
const firstFruit = fruits.shift();
console.log(firstFruit); // Output: "apple"
console.log(fruits); // Output: ["banana", "orange"]

//Unshift():
const fruits = ["apple", "banana"];
fruits.unshift("orange", "kiwi");
console.log(fruits); // Output: ["orange", "kiwi", "apple", "banana"]

//Slice():
const fruits = ["apple", "banana", "orange", "kiwi"];
const citrusFruits = fruits.slice(1, 3);
console.log(citrusFruits); // Output: ["banana", "orange"]

//Splice():
const fruits = ["apple", "banana", "orange", "kiwi"];
fruits.splice(1, 2, "grape", "pear");
```

```
console.log(fruits); // Output: ["apple", "grape", "pear", "kiwi"]


/**
 * Note that there are many other array methods in JavaScript, such as
concat(), join(), reverse(), sort(), and forEach()
. You can find a full list of array methods in the JavaScript documentation.
 */
//==========END==============
//56 MAP in JAvascript
//Mapping an Array of Numbers
const numbers = [1, 2, 3, 4, 5];
const squares = numbers.map(function (num) {
   return num * num;
});
console.log(squares); // Output: [1, 4, 9, 16, 25]


//Mapping an Array of Objects
const people = [
   { name: "Alice", age: 25 },
   { name: "Bob", age: 30 },
   { name: "Charlie", age: 35 }
];
const names = people.map(function (person) {
   return person.name;
});

console.log(names); // Output: ["Alice", "Bob", "Charlie"]


//Mapping an Array of Strings

const words = ["hello", "world", "javascript"];

const capitalizedWords = words.map(function (word) {

   return word.charAt(0).toUpperCase() + word.slice(1);

});

console.log(capitalizedWords); // Output: ["Hello", "World", "Javascript"]


//Note the map() method is can also be used with arrow Function this ES6:For Example

const numbers = [1, 2, 3, 4, 5];

const squares = numbers.map(num => num * num);

console.log(squares); // Output: [1, 4, 9, 16, 25]

//==========END==============
```

//57 Filter all Example in Javascript

```
/**

 * In JavaScript, the filter()

 method is used to create a new array all elements that pass a certain condition. The filter()

 method does not modify the original array, but instead returns a new array with the filtered
elements.

 */


//Filtering an Array of Number
```

```
const numbers = [1, 2, 3, 4, 5];

const evenNumbers = numbers.filter(function (num) {

  return num % 2 === 0;

});

console.log(evenNumbers); // Output: [2, 4]


//Filtering an Array of Objects

const people = [

  { name: "Alice", age: 25 },

  { name: "Bob", age: 30 },

  { name: "Charlie", age: 35 }

];

const youngPeople = people.filter(function (person) {

  return person.age < 30;

});

console.log(youngPeople); // Output: [{ name: "Alice", age: 25 }]


//Filtering an Array of Strings

const words = ["hello", "world", "javascript"];

const longWords = words.filter(function (word) {

  return word.length > 5;

});

console.log(longWords); // Output: ["javascript"]
```

```
/**
 * Note that the filter() method can also be used with arrow function in ES6.For Example:
 */
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
//===========END===============
```

//58 Reduce all Example in Javascript

//Reducing an Array of Numbers:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function (total, num) {
   return total + num;
}, 0);
console.log(sum); // Output: 15
```

//Reducing an Array of Objects:

```
const people = [
   { name: "Alice", age: 25 },
   { name: "Bob", age: 30 },
   { name: "Charlie", age: 35 }
];
const totalAge = people.reduce(function (total, person) {
   return total + person.age;
}, 0);
console.log(totalAge); // Output: 90

//Reducing an Array of Strings:
const words = ["hello", "world", "javascript"];
const sentence = words.reduce(function (total, word) {
   return total + " " + word;
}, "");
console.log(sentence); // Output: "hello world javascript"

/**
 * Note that the reduce() method can also be used with arrow function in ES6.for Example:
 */
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // Output: 15
//===========END===============
//59 Find Method in Javascript

//Finding an Element in an Array of Numbers
const numbers = [1, 2, 3, 4, 5];
const evenNumber = numbers.find(function (num) {
   return num % 2 === 0;
});
```

```
console.log(evenNumber); // Output: 2


//Finding an Element in an Array of Objects

const people = [

   { name: "Alice", age: 25 },

   { name: "Bob", age: 30 },

   { name: "Charlie", age: 35 }

];

const youngPerson = people.find(function (person) {

   return person.age < 30;

});

console.log(youngPerson); // Output: { name: "Alice", age: 25 }


//Finding an Element in an Array of Strings

const words = ["hello", "world", "javascript"];

const wordStartingWithJ = words.find(function (word) {

   return word.charAt(0) === "j";

});

console.log(wordStartingWithJ); // Output: "javascript"




/**

 * Note that the find() method can also be used with arrow functions in ES6.For Example

 */
```

```
const numbers = [1, 2, 3, 4, 5];

const evenNumber = numbers.find(num => num % 2 === 0);

console.log(evenNumber); // Output: 2

//===========END===============
```

//60 Find Index all Example in JavaScript

//Finding the Index of an Element in an Array of Numbers

```
const numbers = [1, 2, 3, 4, 5];

const evenNumberIndex = numbers.findIndex(function (num) {

   return num % 2 === 0;

});

console.log(evenNumberIndex); // Output: 1
```

//Finding the Index of an Element in an Array of Objects

```
const people = [

   { name: "Alice", age: 25 },

   { name: "Bob", age: 30 },

   { name: "Charlie", age: 35 }

];

const youngPersonIndex = people.findIndex(function (person) {

   return person.age < 30;

});

console.log(youngPersonIndex); // Output: 0
```

//Finding the Index of an Element in an Array of String

```
const words = ["hello", "world", "javascript"];

const wordStartingWithJIndex = words.findIndex(function (word) {

   return word.charAt(0) === "j";

});

console.log(wordStartingWithJIndex); // Output: 2




/**

 * Note the findIndex() method can also be used with arrrow function ES6

 */

const numbers = [1, 2, 3, 4, 5];

const evenNumberIndex = numbers.findIndex(num => num % 2 === 0);

console.log(evenNumberIndex); // Output: 1

//===========END===============
```

//61 Some and Every all Example in javascript

```
/**

 * In JavaScript, the some() and every()

 methods are used to if at least one or all elements in an array satisfy a certain condition,
respectively. Both methods return a boolean value.

 */
```

//Checking if at Least One Element in an Array of Numbers Satisfies a Condition:

```
const numbers = [1, 2, 3, 4, 5];

const hasEvenNumber = numbers.some(function (num) {
   return num % 2 === 0;

});

console.log(hasEvenNumber); // Output: true


//Checking if All Elements in an Array of Objects Satisfy a Condition:


const people = [

   { name: "Alice", age: 25 },

   { name: "Bob", age: 30 },

   { name: "Charlie", age: 35 }

];

const allOver18 = people.every(function (person) {

   return person.age > 18;

});

console.log(allOver18); // Output: true


//Checking if at Least One Element in an Array of Strings Satisfies a Condition:

const words = ["hello", "world", "javascript"];

const hasWordStartingWithJ = words.some(function (word) {

   return word.charAt(0) === "j";

});

console.log(hasWordStartingWithJ); // Output: true


//Note that the some()andevery() methods can also be used with arrow functions in ES6.For
```

example:

```
const numbers = [1, 2, 3, 4, 5];

const hasEvenNumber = numbers.some(num => num % 2 === 0);

console.log(hasEvenNumber); // Output: true


const allOver18 = people.every(person => person.age > 18);

console.log(allOver18); // Output: true
```

//62 FlatMap in JAvascript

```
const arr = [1, 2, 3];


const result = arr.flatMap(num => [num, num * 2]);


console.log(result); // Output: [1, 2, 2, 4, 3, 6]
```

//===========END===============

//62 Sorting Array in JavaScript

//Sorting array of numbers in ascending

```
const arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];


arr.sort((a, b) => a - b);


console.log(arr); // Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

//Sorting an array of string in alphabetical order

```
const arr = ['banana', 'apple', 'cherry', 'date', 'elderberry'];

arr.sort();

console.log(arr); // Output: ['apple', 'banana', 'cherry', 'date', 'elderberry']
```

//Sorting an array of objects by a specific property

```
const arr = [
  { name: 'John', age: 25 },
  { name: 'Jane', age: 30 },
  { name: 'Bob', age: 20 },
  { name: 'Alice', age: 27 }
];

arr.sort((a, b) => a.age - b.age);

console.log(arr); // Output: [{ name: 'Bob', age: 20 }, { name: 'John', age: 25 }, { name: 'Alice', age: 27 }, { name: 'Jane', age: 30 }]
```

//===========END===============

//63 Chaining of method in javascript

//Chaining map and filter methods:

```
const arr = [1, 2, 3, 4, 5];

const result = arr
    .map(num => num * 2)
    .filter(num => num > 5);

console.log(result); // Output: [6, 8, 10]

//Chaining reduce and filter methods:
const arr = [1, 2, 3, 4, 5];

const result = arr
    .filter(num => num % 2 === 0)
    .reduce((acc, num) => acc + num, 0);

console.log(result); // Output: 6

//Chaining sort and map methods:
const arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];

const result = arr
    .sort((a, b) => a - b)
    .map(num => num * 2);
```

```
console.log(result); // Output: [2, 2, 6, 6, 8, 10, 10, 12, 18, 20, 18]
```

//===========END===============

//64 Date and time in javascript

//creating a new Date Objects

```
const now = new Date();


console.log(now); // Output: current date and time
```

//Getting the current date and time:

```
const now = new Date();


const year = now.getFullYear();

const month = now.getMonth() + 1;

const day = now.getDate();

const hours = now.getHours();

const minutes = now.getMinutes();

const seconds = now.getSeconds();


console.log(`${year}-${month}-${day} ${hours}:${minutes}:${seconds}`); // Output: current date and time in YYYY-MM-DD HH:MM:SS format
```

//Parsing a date string

```
const dateString = '2022-01-01T00:00:00.000Z';
const date = new Date(dateString);


console.log(date); // Output: Sat Jan 01 2022 01:00:00 GMT+0100 (Central European Standard Time)


//Formatting a date string

const now = new Date();


const formattedDate = now.toLocaleDateString('en-US', { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' });

const formattedTime = now.toLocaconstimeString('en-US', { hour: 'numeric', minute: 'numeric', second: 'numeric', hour12: true });


console.log(`${formattedDate} ${formattedTime}`); // Output: current date and time in long format

//===========END===============
```

//65 Digital Clock in javascript

```
//HtML

// < !DOCTYPE html >

  <html>

    <head>

      <title>Digital Clock</title>

    </head>

    <body>

      <h1 id="clock"></h1>
```

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


//Javascript
function updateTime() {
  const now = new Date();
  const hours = now.getHours().toString().padStart(2, '0');
  const minutes = now.getMinutes().toString().padStart(2, '0');
  const seconds = now.getSeconds().toString().padStart(2, '0');
  const timeString = `${hours}:${minutes}:${seconds}`;
  document.getElementById('clock').textContent = timeString;
}


setInterval(updateTime, 1000);
//===========END===============
```

//66 Local Storage intro in javascript

```
//---------Example 1
// Set a value in local storage
localStorage.setItem('name', 'John');


// Get a value from local storage
const name = localStorage.getItem('name');
console.log(name); // Output: John
```

```
// Remove a value from local storage

localStorage.removeItem('name');


//--------Example 2:

// Set an object in local storage

const person = { name: 'John', age: 30 };

localStorage.setItem('person', JSON.stringify(person));


// Get an object from local storage

const personString = localStorage.getItem('person');

const person = JSON.parse(personString);

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

//===========END===============
```

//67 Set and Get Items in JavaScript

```
//Setting and Getting a String Value

// Set a string value

localStorage.setItem('name', 'John');


// Get the string value

const name = localStorage.getItem('name');

console.log(name); // Output: John
```

```
//Setting and Getting an Object Value
// Set an object value
const person = { name: 'John', age: 30 };
localStorage.setItem('person', JSON.stringify(person));

// Get the object value
const personString = localStorage.getItem('person');
const person = JSON.parse(personString);
console.log(person); // Output: { name: 'John', age: 30 }
//===========END===============
//68 Deconsting Example in JavaScript
//Deconsting a Single Item:
// Set a value
localStorage.setItem('name', 'John');

// Deconste the value
localStorage.removeItem('name');

//Deconsting all items
// Set some values
localStorage.setItem('name', 'John');
localStorage.setItem('age', 30);

// Deconste all values
```

```
localStorage.clear();
//===========END===============
```

//69 Store Complex Data all Example in JavaScript

```
// Define an array of objects
const people = [
  { name: 'John', age: 30 },
  { name: 'Jane', age: 25 },
  { name: 'Bob', age: 40 }
];

// Store the array in local storage
localStorage.setItem('people', JSON.stringify(people));

// Retrieve the array from local storage
const peopleString = localStorage.getItem('people');
const people = JSON.parse(peopleString);

// Log the array to the console
console.log(people); // Output: [{ name: 'John', age: 30 }, { name: 'Jane', age: 25 }, { name: 'Bob', age: 40 }]
//===========END===============
```

//70 Contructor and New Operator in JavaScript

```
// Define a constructor function
```

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Create a new object using the constructor
const john = new Person('John', 30);

// Log the object to the console
console.log(john); // Output: Person { name: 'John', age: 30 }
//===========END===============
```

//71 Protype in Javascript

```
// Define a constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Add a method to the prototype
Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
}
```

```
// Create a new object using the constructor

const john = new Person('John', 30);


// Call the method on the object

john.greet(); // Output: Hello, my name is John and I'm 30 years old.


/**

 * Note that using prototypes allows you to add methods and properties to all objects created
using a particular constructor function, without having to add them to each individual object.
This can save memory and make your code more efficient.

 */

//===========END===============
```

//72 Prototypical interfaces in JavaScript

```
// Define a constructor function

function Person(name, age) {

   this.name = name;

   this.age = age;

}


// Define a prototypical interface

Person.prototype = {

   greet: function () {

      console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);

   },
  sayGoodbye: function () {
    console.log(`Goodbye from ${this.name}!`);
  }
}

// Create a new object using the constructor
const john = new Person('John', 30);

// Call the methods on the object
john.greet(); // Output: Hello, my name is John and I'm 30 years old.
john.sayGoodbye(); // Output: Goodbye from John!

/**
 * Note that using prototypical interfaces allows you to define a set of methods and properties that can be shared by all objects created using a particular constructor function. This can make your code more modular and easier to maintain.
 */
//===========END===============
```

//73 ES6 Classes in JavaScript

```
// Define a class
class Person {
  constructor (name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }

  sayGoodbye() {
    console.log(`Goodbye from ${this.name}!`);
  }
}

// Create a new object using the class
const john = new Person('John', 30);

// Call the methods on the object
john.greet(); // Output: Hello, my name is John and I'm 30 years old.
john.sayGoodbye(); // Output: Goodbye from John!
//===========END===============
//74 Getter and Setter in JavaScript
// Define a class
class Person {
  constructor (name, age) {
    this._name = name;
    this._age = age;
  }
  get name() {

    return this._name;

  }


  set name(value) {

    this._name = value;

  }


  get age() {

    return this._age;

  }


  set age(value) {

    if (value < 0) {

      throw new Error('Age cannot be negative');

    }

    this._age = value;

  }


  greet() {

    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);

  }

}
```

```
// Create a new object using the class
const john = new Person('John', 30);


// Call the methods on the object
john.greet(); // Output: Hello, my name is John and I'm 30 years old.


// Use the setters to update the object
john.name = 'Jane';
john.age = 25;


// Call the methods on the updated object
john.greet(); // Output: Hello, my name is Jane and I'm 25 years old.
//===========END===============
```

//75 Static Method in JavaScript

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


  static subtract(a, b) {
    return a - b;
  }
}

// Call the static methods on the class
const sum = MathUtils.add(5, 3);
console.log(sum); // Output: 8

const difference = MathUtils.subtract(5, 3);
console.log(difference); // Output: 2
//===========END===============
```

//76 Class Inheritance in JavaScript

```
// Define a base class
class Person {
  constructor (name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }
}

// Define a derived class
```

```
class Student extends Person {
  constructor (name, age, grade) {
    super(name, age);
    this.grade = grade;
  }

  study() {
    console.log(`${this.name} is studying hard for their ${this.grade} grade.`);
  }
}

// Create a new object using the derived class
const john = new Student('John', 15, '9th');

// Call the methods on the object
john.greet(); // Output: Hello, my name is John and I'm 15 years old.
john.study(); // Output: John is studying hard for their 9th grade.

/**
 * Note that using class inheritance allows us to create new classes that inherit properties and methods from existing classes, and add new properties and methods as needed. This can help us write more modular and reusable code.
 */
//===========END===============
```

//77 Chaining of Method in JavaScript

```
// Define a class

class Calculator {

  constructor (value = 0) {

    this.value = value;

  }



  add(num) {

    this.value += num;

    return this;

  }



  subtract(num) {

    this.value -= num;

    return this;

  }



  multiply(num) {

    this.value *= num;

    return this;

  }



  divide(num) {

    this.value /= num;

    return this;
  }
}

// Create a new object using the class and chain methods
const result = new Calculator(10)
  .add(5)
  .multiply(2)
  .subtract(3)
  .divide(4)
  .value;

console.log(result); // Output: 4
//==========END==============
```

//78 Async Code in JavaScript

```
// Define a function that returns a promise
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Data fetched successfully!');
        }, 2000);
    });
}
```

```
// Call the function and handle the promise
async function getData() {
    try {
        const data = await fetchData();
        console.log(data); // Output: Data fetched successfully!
    } catch (error) {
        console.error(error);
    }
}

// Call the async function
getData();
//===========END===============
//79 XML HTTP Request in JavaScript

// Define a function that returns a promise
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Data fetched successfully!');
        }, 2000);
    });
}
```

```
// Call the function and handle the promise
async function getData() {
  try {
    const data = await fetchData();
    console.log(data); // Output: Data fetched successfully!
  } catch (error) {
    console.error(error);
  }
}

// Call the async function
getData();
//===========END===============
//80 Status Code in JavaScript

// Create a new XHR object
const xhr = new XMLHttpRequest();

// Define a callback function to handle the response
xhr.onload = function () {
  if (xhr.status === 200) {
    console.log('Request succeeded. Status code:', xhr.status);
  } else {
    console.error('Request failed. Status code:', xhr.status);
  }
};


// Open a new request and send it
xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1');
xhr.send();
//===========END===============
```

//81 CallBack Function in JavaScript

```
// Define a function that takes a callback function as a parameter
function fetchData(callback) {
  setTimeout(() => {
    const data = 'Data fetched successfully!';
    callback(data);
  }, 2000);
}


// Call the function and pass in a callback function
fetchData((data) => {
  console.log(data); // Output: Data fetched successfully!
});


/**
 * Note that using callback functions allows us to pass functions as parameters to other
functions, and execute them at a later time. This can be useful for performing asynchronous
```

```
operations or for creating more modular and reusable code.

 */

//===========END===============
```

//82 Extraction JSON in JavaScript

```
// Define a JSON string

const jsonString = '{"name": "John", "age": 30, "city": "New York"}';


// Parse the JSON string into an object

const data = JSON.parse(jsonString);


// Access the properties of the object

console.log(data.name); // Output: John

console.log(data.age); // Output: 30

console.log(data.city); // Output: New York


/**

 * Note that JSON is a lightweight data interchange format that is commonly used for sending data between a client and a server. The

JSON.parse

 method is used to parse a JSON string into a JavaScript object, while the

JSON.stringify

 method is used to convert a JavaScript object into a JSON string.

 */
```

//===========END===============

//83 Callback Hell in JavaScript

/**

 * Callback hell is a situation that arises when we have multiple nested callbacks in our code, making it difficult to read and maintain. Here's an example of callback hell in JavaScript:

 */

```
function getUser(userId, callback) {
    setTimeout(() => {
        const user = {
            id: userId,
            name: 'John',
            age: 30,
            city: 'New York'
        };
        callback(user);
    }, 2000);
}

function getPosts(userId, callback) {
    setTimeout(() => {
        const posts = [
            { id: 1, title: 'Post 1' },
            { id: 2, title: 'Post 2' },
      { id: 3, title: 'Post 3' }
    ];
    callback(posts);
  }, 2000);
}

function getComments(postId, callback) {
  setTimeout(() => {
    const comments = [
      { id: 1, text: 'Comment 1' },
      { id: 2, text: 'Comment 2' },
      { id: 3, text: 'Comment 3' }
    ];
    callback(comments);
  }, 2000);
}

getUser(1, (user) => {
  console.log(user);
  getPosts(user.id, (posts) => {
    console.log(posts);
    getComments(posts[0].id, (comments) => {
      console.log(comments);
    });
  });
});


/**
 * To avoid callback hell, we can use techniques such as promises, async/await, or libraries such
as
async.js to write more readable and maintainable code.
 */
//===========END===============
```

//84 Basics of Promises in JavaScript

```
// a function that returns a promise
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Data fetched successfully!';
      resolve(data);
    }, 2000);
  });
}


// Call the function and handle the promise
fetchData()
  .then((data) => {
    console.log(data); // Output: Data fetched successfully!
  })
  .catch((error) => {
    console.error(error);
  });
//===========END===============
```

//85 Chaining of Promises in JavaScript

```
// a function that returns a promise
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Data fetched successfully!';
      resolve(data);
    }, 2000);
  });
}

// Call the function and chain promises
fetchData()
  .then((data) => {
    console.log(data); // Output: Data fetched successfully!
    return data.toUpperCase();
  })
  .then((data) => {

    console.log(data); // Output: DATA FETCHED SUCCESSFULLY!

  })

  .catch((error) => {

    console.error(error);

  });


//==========END==============
```

//86 Fetch API in JavaScript

```
// Fetch data from a URL

fetch('https://jsonplaceholder.typicode.com/todos/1')

  .then((response) => {

    if (!response.ok) {

      throw new Error('Network response was not ok');

    }

    return response.json();

  })

  .then((data) => {

    console.log(data); // Output: { userId: 1, id: 1, title: 'delectus aut autem', compconsted: false }

  })

  .catch((error) => {

    console.error('There was a problem with the fetch operation:', error);
  });
//==========END==============
//87 Async/Await im JavaScript
// a function that returns a promise
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Data fetched successfully!';
      resolve(data);
    }, 2000);
  });
}

// Call the function using async/await
async function getData() {
  try {
    const data = await fetchData();
    console.log(data); // Output: Data fetched successfully!
  } catch (error) {
    console.error(error);
  }
}

// Call the async function
```

```
getData();
//===========END===============
//88 Array De-structuring in JavaScript
// Define an array
const numbers = [1, 2, 3, 4, 5];

// Destructure the array
const [first, second, ...rest] = numbers;

// Log the values
console.log(first); // Output: 1
console.log(second); // Output: 2
console.log(rest); // Output: [3, 4, 5]
//===========END===============
//89 Object De-structuring in JavaScript
// Define an object
const person = {
   name: 'John',
   age: 30,
   city: 'New York'
};

// Destructure the object
const { name, age, city } = person;
```

```
// Log the values

console.log(name); // Output: John

console.log(age); // Output: 30

console.log(city); // Output: New York

//===========END===============
```

//90 Spread OPerattor in Javascript

//Combining arrays

```
const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];


const combinedArr = [...arr1, ...arr2];


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

//Copying arrays

```
const originalArr = [1, 2, 3];

const copyArr = [...originalArr];


console.log(copyArr); // Output: [1, 2, 3]
```

//Merging objects

```
const obj1 = { name: 'John', age: 30 };
const obj2 = { city: 'New York', country: 'USA' };

const mergedObj = { ...obj1, ...obj2 };

console.log(mergedObj); // Output: { name: 'John', age: 30, city: 'New York', country: 'USA' }
//===========END===============
```

//91 Rest Operator in javascript

//Function arguments

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

console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

//Destructing Array

```
const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log(first); // Output: 1
console.log(second); // Output: 2
console.log(rest); // Output: [3, 4, 5]
```

```
//Destructing Objects:

const person = {

   name: 'John',

   age: 30,

   city: 'New York',

   country: 'USA'

};


const { name, age, ...address } = person;


console.log(name); // Output: John

console.log(age); // Output: 30

console.log(address); // Output: { city: 'New York', country: 'USA' }

//===========END===============
```

//92 Short Circuiting in JavaScript

```
//Using the && Operator

const name = 'John';

const greeting = name && `Hello, ${name}!`;


console.log(greeting); // Output: Hello, John!


//Using the || Operator
```

```
const name = '';
const defaultName = 'Guest';
const displayName = name || defaultName;

console.log(displayName); // Output: Guest

// Using short-circuiting to avoid errors
const person = {
  name: 'John',
  age: 30
};

const city = person.address && person.address.city;

console.log(city); // Output: undefined

//===========END===============
```

//93 Nullish Coalescing operator in Javascript

```
/**
 * Sure, here's an example of using the nullish coalescing operator (
??) in JavaScript:
 */
```

```
const name = null;
const defaultName = 'Guest';
const displayName = name ?? defaultName;

console.log(displayName); // Output: Guest
//===========END===============
```

//94 for-of loop in javascript

```
const numbers = [1, 2, 3, 4, 5];

for (const number of numbers) {
   console.log(number);
}

// Output:
// 1
// 2
// 3
// 4
// 5
//===========END===============
```

//95 Enchanced Object Literal-shortcut in JavaScript

//Shorthand Property names

```
const name = 'John';
const age = 30;
const person = { name, age };

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

//Shorthand method names
const calculator = {
  add(a, b) {
    return a + b;
  },
  subtract(a, b) {
    return a - b;
  }
};

console.log(calculator.add(1, 2)); // Output: 3
console.log(calculator.subtract(5, 3)); // Output: 2

//Computed Property names:
const key = 'name';

const person = {
  [key]: 'John',
  age: 30
};

console.log(person.name); // Output: John

/**
 * Note that enhanced object literals and shortcuts can make our code more concise and readable, especially when working with objects that have many properties or methods. They can also make our code more dynamic and flexible, allowing us to create objects with properties and methods that are determined at runtime.
 */
//===========END===============
```

//96 Optaining Chaining in JavaScript

//Accesing nested Properties

```
const person = {
  name: 'John',
  age: 30,
  address: {
    city: 'New York',
    country: 'USA'
  }
};

const city = person.address?.city;
```

```
console.log(city); // Output: New York


//Calling Methods

const person = {
  name: 'John',
  age: 30,
  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
};


person.sayHello?.(); // Output: Hello, my name is John


//Chaining Multiple Properties

const person = {
  name: 'John',
  age: 30,
  address: {
    city: 'New York',
    country: 'USA'
  }
};


const country = person.address?.country?.toUpperCase();
```

```
console.log(country); // Output: USA


/**
 * Note that the optional chaining operator can be used to write concise and readable code that handles null or undefined values in a safe and efficient way. However, it should be used with caution and only when appropriate, as it can sometimes make code harder to read and understand.
 */


//===========END===============
```

//97 Looping Objects in JavaScript

//Using a for...in loop:

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};


for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}


// Output:
```

```
// name: John
// age: 30
// city: New York
```

//Using Object.key():

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};

Object.keys(person).forEach(key => {
  console.log(`${key}: ${person[key]}`);
});

// Output:
// name: John
// age: 30
// city: New York
```

//Using Obbject.entries():

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};

Object.entries(person).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});

// Output:
// name: John
// age: 30
// city: New York

/**
 * Note that the Object.entries() method is a more concise and readable way to iterate over the key-value pairs of an object, and it only iterates over the own properties of the object.
 */
//===========END===============
```

//98 Sets all Example in JavaScript

//creating a Set:

```
const set = new Set([1, 2, 3, 4, 5]);

console.log(set); // Output: Set { 1, 2, 3, 4, 5 }
```

```
//Adding and deconsting values
const set = new Set();

set.add(1);
set.add(2);
set.add(3);

console.log(set); // Output: Set { 1, 2, 3 }

set.deconste(2);

console.log(set); // Output: Set { 1, 3 }

//Checking for Values
const set = new Set([1, 2, 3]);

console.log(set.has(2)); // Output: true
console.log(set.has(4)); // Output: false

//Iterating over values:
const set = new Set([1, 2, 3]);

for (const value of set) {
  console.log(value);
}


// Output:

// 1

// 2

// 3

//===========END===============
```

//99 Map intro And Map Iteration in JavaScript

//Creating a Map:

```
const map = new Map();


map.set('name', 'John');

map.set('age', 30);

map.set('city', 'New York');


console.log(map); // Output: Map { 'name' => 'John', 'age' => 30, 'city' => 'New York' }
```

//Getting and Deconsting values:

```
const map = new Map();


map.set('name', 'John');

map.set('age', 30);

map.set('city', 'New York');
```

```
console.log(map.get('name')); // Output: John

map.deconste('age');

console.log(map); // Output: Map { 'name' => 'John', 'city' => 'New York' }

//Iterating over key-value pairs
const map = new Map();

map.set('name', 'John');
map.set('age', 30);
map.set('city', 'New York');

for (const [key, value] of map) {
  console.log(`${key}: ${value}`);
}

// Output:
// name: John
// age: 30
// city: New York

/**
```

```
 * Note that maps are a useful data structure for storing key-value pairs and performing map operations such as mapping, filtering, and reducing. They can also be used to remove duplicates from arrays and to check if an array contains a specific value.

 */

//==========END==============
```

```
// 100 Module Scpoe all Example in Javascript


// Module.js

module {

  // Private variable

  let privateVariable = 'I am private';


  // Private function

  function privateFunction() {

   console.log('This is a private function');

  }


  // Public function

  function publicFunction() {

   console.log('This is a public function');

  }


  // Expose public function

  return {
      publicFunction: publicFunction
    };
  }

  // Usage in another file
  const myModule = require('./Module');
  myModule.publicFunction(); // Output: "This is a public function"
  myModule.privateVariable; // undefined (privateVariable is not accessible)
  myModule.privateFunction(); // TypeError: myModule.privateFunction is not a function
//===========END===============
```

// 101 Global Object all Example in Javascript

```
// Accessing global variables:
console.log(window.innerWidth); // Accessing the inner width of the browser window
console.log(global.process.version); // Accessing the version of Node.js

//Defining global variables:
window.myVariable = 'Hello'; // Creating a global variable in the browser
global.myVariable = 'Hello'; // Creating a global variable in Node.js

// Using global functions:
setTimeout(() => {
    console.log('Delayed message'); // Using the setTimeout function from the global object
  }, 1000);
```

```
//  Accessing global constructors:

const myArray = new Array(); // Creating an array using the global Array constructor

const myDate = new Date(); // Creating a date object using the global Date constructor


// Using global constants:

console.log(Math.PI); // Accessing the PI constant from the global Math object

console.log(JSON.stringify({ name: 'John' })); // Using the JSON object from the global object


//===========END===============
```

// 102 Function scope and Lexical environment all Example in Javascript

```
// Basic function scope:

function myFunction() {

  var x = 10; // 'x' is function-scoped and accessible within myFunction

  console.log(x);

 }


 myFunction(); // Output: 10

 console.log(x); // Error: x is not defined (not accessible outside the function)


//  Nested functions and lexical environment:

function outerFunction() {

  var outerVariable = 'I am outside';
```

```
   function innerFunction() {
     var innerVariable = 'I am inside';
     console.log(outerVariable); // Accessing outerVariable from the lexical environment
     console.log(innerVariable);
   }

   innerFunction();
 }

 outerFunction();
 // Output:
 // "I am outside"
 // "I am inside"
 console.log(outerVariable); // Error: outerVariable is not defined (not accessible outside the function)
 console.log(innerVariable); // Error: innerVariable is not defined (not accessible outside the function)
```

//   Lexical scoping and closures:

```
function outerFunction() {
   var outerVariable = 'I am outside';

   function innerFunction() {
     var innerVariable = 'I am inside';
   function nestedFunction() {
    console.log(outerVariable); // Accessing outerVariable from the lexical environment
    console.log(innerVariable); // Accessing innerVariable from the lexical environment
   }

   return nestedFunction;
  }

  var closure = innerFunction();
  closure();
 }

 outerFunction();
 // Output:
 // "I am outside"
 // "I am inside"
//===========END===============
 // 102 Closures - real examples all Example in Javascript
//   Private variables and encapsulation:
function createCounter() {
  var count = 0;

  return {
    increment: function() {
      count++;
    },
    decrement: function() {
      count--;
    },
    getCount: function() {
      return count;
    }
  };
}

var counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.getCount()); // Output: 2
```

// Event handling:

```
function attachEventListener(element, eventType) {
    element.addEventListener(eventType, function() {
      console.log('Event type:', eventType);
    });
  }
```

```
 var button = document.querySelector('#myButton');
 attachEventListener(button, 'click');


// Iteration with setTimeout:
for (var i = 1; i <= 5; i++) {
   (function(index) {
     setTimeout(function() {
       console.log(index);
     }, i * 1000);
   })(i);
 }
//===========END===============
```

// 103 Currying- real examples all Example in Javascript

```
// Mathematical operations:
function add(x) {
   return function(y) {
     return x + y;
   };
 }


 var addFive = add(5);
 console.log(addFive(3)); // Output: 8
 console.log(addFive(7)); // Output: 12
```

```
//  Logging utility:

function createLogger(prefix) {

  return function(message) {

   console.log(`[${prefix}] ${message}`);

  };

 }


 var infoLogger = createLogger('INFO');

 var errorLogger = createLogger('ERROR');


 infoLogger('Application started');

 errorLogger('Something went wrong');


// Function composition:

function compose(...fns) {

  return function(x) {

   return fns.reduceRight(function(acc, fn) {

    return fn(acc);

   }, x);

  };

 }


 function double(x) {
   return x * 2;

 }


 function square(x) {

   return x * x;

 }


 var doubleThenSquare = compose(square, double);

 console.log(doubleThenSquare(5)); // Output: 100

//===========END===============
```

//   104 Object references- real examples all Example in Javascript

// Passing objects to functions:

```
function updateName(person) {

   person.name = 'John Doe';

 }


 var user = { name: 'Jane Smith' };

 updateName(user);

 console.log(user.name); // Output: "John Doe"
```

// Sharing data across components:

```
var data = { count: 0 };


function incrementCount() {
  data.count++;
}

function renderUI() {
  console.log('Count:', data.count);
}

incrementCount();
renderUI(); // Output: "Count: 1"

// Arrays and references:
var array1 = [1, 2, 3];
var array2 = array1;

array2.push(4);
console.log(array1); // Output: [1, 2, 3, 4]

// Object destructuring and references:
var person = { name: 'John', age: 30 };

var { name, age } = person;
name = 'Jane';
console.log(person.name); // Output: "John"
//===========END===============
```

// 105 Shallow copy and deep copy- real examples all Example in Javascript

// Shallow copy example:

```
var originalArray = [1, 2, 3];

var shallowCopyArray = originalArray.slice();


shallowCopyArray.push(4);

console.log(originalArray); // Output: [1, 2, 3]

console.log(shallowCopyArray); // Output: [1, 2, 3, 4]
```

// Shallow copy with objects:

```
var originalObj = { name: 'John', age: 30 };

var shallowCopyObj = Object.assign({}, originalObj);


shallowCopyObj.age = 35;

console.log(originalObj.age); // Output: 30

console.log(shallowCopyObj.age); // Output: 35
```

// Deep copy example:

```
var originalArray = [1, 2, [3, 4]];

var deepCopyArray = JSON.parse(JSON.stringify(originalArray));


deepCopyArray[2].push(5);

console.log(originalArray); // Output: [1, 2, [3, 4]]

console.log(deepCopyArray); // Output: [1, 2, [3, 4, 5]]
```

```
// Deep copy with objects:

var originalObj = { name: 'John', address: { city: 'New York', country: 'USA' } };

var deepCopyObj = JSON.parse(JSON.stringify(originalObj));


deepCopyObj.address.city = 'London';

console.log(originalObj.address.city); // Output: "New York"

console.log(deepCopyObj.address.city); // Output: "London"

//==========END==============
```

//   106 this keyword & Methods- real examples all Example in Javascript

```
// Object methods:

var car = {

  brand: 'Toyota',

  model: 'Camry',

  start: function() {

   console.log(`Starting ${this.brand} ${this.model}`);

  }

 };


 car.start(); // Output: "Starting Toyota Camry"


//   Event handlers:

var button = document.querySelector('#myButton');

button.addEventListener('click', function() {
 console.log(`Button ${this.id} clicked`);
});

// Constructor functions:
function Person(name, age) {
  this.name = name;
  this.age = age;

  this.introduce = function() {
   console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  };
 }

 var john = new Person('John', 25);
 john.introduce(); // Output: "Hi, I'm John and I'm 25 years old."

//   Method chaining:
var calculator = {
  value: 0,
  add: function(num) {
   this.value += num;
   return this;
  },
  multiply: function(num) {
   this.value *= num;
   return this;
  },
  getValue: function() {
   return this.value;
  }
 };

 var result = calculator.add(5).multiply(2).getValue();
 console.log(result); // Output: 10
//===========END===============
```

//   107 Symbol data type- real examples all Example in Javascript

// Creating unique property keys:

```
var nameSymbol = Symbol('name');
var person = {
 [nameSymbol]: 'John Doe',
 age: 30
};

console.log(person[nameSymbol]); // Output: "John Doe"
console.log(Object.getOwnPropertySymbols(person)); // Output: [Symbol(name)]
```

// Implementing custom iterators:

```
var myIterable = {
  data: [1, 2, 3],
  [Symbol.iterator]: function() {
    var index = 0;
    var data = this.data;
    return {
      next: function() {
        return index < data.length
          ? { value: data[index++], done: false }
          : { done: true };
      }
    };
  }
};

for (var item of myIterable) {
  console.log(item);
}
// Output:
// 1
// 2
// 3
```

//   Defining well-known symbols:

```
class MyCustomClass {
  [Symbol.toStringTag]() {
   return 'MyCustomClass';
  }
 }

 var instance = new MyCustomClass();
 console.log(instance.toString()); // Output: "[object MyCustomClass]"
//===========END===============
```

//   108 Functions as objects- real examples all Example in Javascript

```
// Assigning functions to variables:
var greet = function(name) {
  console.log('Hello, ' + name + '!');
 };

 greet('John'); // Output: "Hello, John!"
//   Functions as arguments:
function applyOperation(a, b, operation) {
  return operation(a, b);
 }

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

 var result1 = applyOperation(2, 3, add); // 2 + 3 = 5
 var result2 = applyOperation(4, 5, multiply); // 4 * 5 = 20

 console.log(result1); // Output: 5
 console.log(result2); // Output: 20

//  Functions as return values:
function createMultiplier(multiplier) {
 return function(number) {
  return number * multiplier;
 };
}

var double = createMultiplier(2);
var triple = createMultiplier(3);

console.log(double(4)); // Output: 8
console.log(triple(4)); // Output: 12
//==========END==============
```

//   109 Name function expressions (NFE)- real examples all Example in Javascript

// Named Function Expressions (NFE) are function expressions that have a name assigned to them. The name is only accessible within the function's scope and is useful for self-reference or improving debuggability.

// Recursive functions:

```
var factorial = function calculateFactorial(n) {
    if (n <= 1) {
      return 1;
    }
    return n * calculateFactorial(n - 1);
  };

  console.log(factorial(5)); // Output: 120
```

// Event listeners:

```
var button = document.querySelector('#myButton');
button.addEventListener('click', function handleClick() {
  console.log('Button clicked');
});

button.click(); // Output: "Button clicked"
```

// Debugging and stack traces:

```
var divide = function divideNumbers(a, b) {
    if (b === 0) {
      throw new Error('Division by zero');
```

```
  }
  return a / b;
 };

 try {
  console.log(divide(10, 0));
 } catch (error) {
  console.error(error.stack);
 }
//===========END===============
```

//   110 Decorator Pattern - Memoization- real examples all Example in Javascript

// The Decorator pattern and memoization are two separate concepts in JavaScript, but I can provide examples for both individually:

// Decorator Pattern:

```
// Base object
class Pizza {
  getDescription() {
   return 'Plain pizza';
  }
  cost() {
   return 10;
  }
 }
```

```
// Decorator objects
class Cheese extends Pizza {
  constructor(pizza) {
    super();
    this.pizza = pizza;
  }
  getDescription() {
    return this.pizza.getDescription() + ', with extra cheese';
  }
  cost() {
    return this.pizza.cost() + 2;
  }
}

class Mushroom extends Pizza {
  constructor(pizza) {
    super();
    this.pizza = pizza;
  }
  getDescription() {
    return this.pizza.getDescription() + ', with mushrooms';
  }
  cost() {
    return this.pizza.cost() + 3;
    }
  }

  // Usage
  let myPizza = new Pizza();
  console.log(myPizza.getDescription()); // Output: "Plain pizza"
  console.log(myPizza.cost()); // Output: 10

  myPizza = new Cheese(myPizza);
  console.log(myPizza.getDescription()); // Output: "Plain pizza, with extra cheese"
  console.log(myPizza.cost()); // Output: 12

  myPizza = new Mushroom(myPizza);
  console.log(myPizza.getDescription()); // Output: "Plain pizza, with extra cheese, with mushrooms"
  console.log(myPizza.cost()); // Output: 15

//   Memoization:
function expensiveOperation(n) {
  console.log('Computing for', n);
  // Simulating time-consuming computation
  let result = 0;
  for (let i = 0; i < n; i++) {
    result += i;
  }
  return result;
}

function memoize(fn) {
  const cache = {};
  return function(n) {
    if (n in cache) {
      console.log('Returning from cache for', n);
      return cache[n];
    } else {
      const result = fn(n);
      console.log('Caching result for', n);
      cache[n] = result;
      return result;
    }
  };
}

const memoizedOperation = memoize(expensiveOperation);
console.log(memoizedOperation(5)); // Output: Computing for 5, Caching result for 5, 10
console.log(memoizedOperation(5)); // Output: Returning from cache for 5, 10
console.log(memoizedOperation(7)); // Output: Computing for 7, Caching result for 7, 21
console.log(memoizedOperation(7)); // Output: Returning from cache for 7, 21
```

//===========END===============

//   111 This - call, apply, bind- real examples all Example in Javascript

// Certainly! The call, apply, and bind methods are used to manipulate the this value within functions in JavaScript. Here are some real-world examples that demonstrate the use of call, apply, and bind in JavaScript:

// Using call:

```
function greet(name) {
    console.log(`Hello, ${name}! I'm ${this.role}.`);
  }


  var person = {
    role: 'developer'
  };


  greet.call(person, 'John'); // Output: "Hello, John! I'm developer."
```

//   Using apply:

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


  var numbers = [1, 2, 3];


  var result = sum.apply(null, numbers);
  console.log(result); // Output: 6
```

// Using bind:

```
var module = {
  name: 'My Module',
  getDescription: function() {
    console.log(`Module name: ${this.name}`);
  }
 };


 var logDescription = module.getDescription.bind(module);
 logDescription(); // Output: "Module name: My Module"
//===========END===============
```

//   112 Debouncing- real examples all Example in Javascript

// Debouncing is a technique used to limit the frequency of executing a function by delaying its invocation until a certain period of inactivity has passed. It is often used to optimize performance and improve user experience in scenarios such as input events or scroll events. Here are some real-world examples that demonstrate the use of debouncing in JavaScript:

// Debouncing an input event:

```
function debounce(func, delay) {
  let timerId;
  return function(...args) {
    clearTimeout(timerId);
    timerId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
 }

 function handleInput(event) {
  console.log(event.target.value);
 }

 const debouncedInputHandler = debounce(handleInput, 300);

 document.querySelector('#myInput').addEventListener('input', debouncedInputHandler);
```

//   Debouncing a scroll event:

```
function debounce(func, delay) {
  let timerId;
  return function(...args) {
   clearTimeout(timerId);
   timerId = setTimeout(() => {
    func.apply(this, args);
   }, delay);
  };
 }

 function handleScroll() {
  console.log('Scrolled');
 }


 const debouncedScrollHandler = debounce(handleScroll, 200);


 window.addEventListener('scroll', debouncedScrollHandler);
```

//===========END===============

//   113 Throttling- real examples all Example in Javascript

// Throttling is a technique used to limit the frequency of executing a function by enforcing a maximum execution rate. It ensures that the function is invoked at a specified interval, preventing it from being called more often than desired. Throttling is commonly used for scenarios such as resizing events or mouse movement events

// Throttling a resize event:

```
function throttle(func, delay) {
   let isThrottled = false;
   return function(...args) {
     if (!isThrottled) {
       func.apply(this, args);
       isThrottled = true;
       setTimeout(() => {
         isThrottled = false;
       }, delay);
     }
   };
 }
 function handleResize() {
  console.log('Resized');
 }

 const throttledResizeHandler = throttle(handleResize, 200);

 window.addEventListener('resize', throttledResizeHandler);

//   Throttling a scroll event:
function throttle(func, delay) {
  let isThrottled = false;
  return function(...args) {
   if (!isThrottled) {
    func.apply(this, args);
    isThrottled = true;
    setTimeout(() => {
     isThrottled = false;
    }, delay);
   }
  };
 }

 function handleScroll() {
  console.log('Scrolled');
 }


 const throttledScrollHandler = throttle(handleScroll, 300);


 window.addEventListener('scroll', throttledScrollHandler);
```

//===========END===============

//   114 Iterable and iterator protocols- real examples all Example in Javascript

// The Iterable and Iterator protocols in JavaScript provide a standardized way to make objects iterable, allowing them to be looped over using the for...of loop or used with other iterable functions.

// Creating a custom iterable object:

```
const myIterable = {
  data: ['apple', 'banana', 'cherry'],
  [Symbol.iterator]() {
    let index = 0;
    const data = this.data;
    return {
      next() {
        if (index < data.length) {
          return { value: data[index++], done: false };
        }
        return { done: true };
      }
    };
  }
```

```
 };

 for (const item of myIterable) {
  console.log(item);
 }
 // Output:
 // "apple"
 // "banana"
 // "cherry"

//  Custom iterator with infinite sequence:
const infiniteSequence = {
  [Symbol.iterator]() {
   let value = 1;
   return {
    next() {
     value++;
     return { value, done: false };
    }
   };
  }
 };

 let count = 0;
 for (const num of infiniteSequence) {
  console.log(num);
  if (++count >= 5) {
   break;
  }
 }
 // Output:
 // 2
 // 3
 // 4
 // 5
 // 6
```

//   Using generator functions:

```
function* fibonacciSequence() {
  let prev = 0;
  let curr = 1;
  while (true) {
   yield curr;
   [prev, curr] = [curr, prev + curr];
  }
 }

 const fibonacci = fibonacciSequence();
 for (let i = 0; i < 10; i++) {
   console.log(fibonacci.next().value);
 }
 // Output:
 // 1
 // 1
 // 2
 // 3
 // 5
 // 8
 // 13
 // 21
 // 34
 // 55
//===========END===============
```

//   115 Array like vs iterables- real examples all Example in Javascript

// Array-like objects and iterables are two different concepts in JavaScript, each with its own characteristics.

// Array-like objects:

```
function sum() {
   let total = 0;
   for (let i = 0; i < arguments.length; i++) {
     total += arguments[i];
   }
```

```
  return total;
 }

 console.log(sum(1, 2, 3)); // Output: 6

//  Iterables:
const mySet = new Set([1, 2, 3]);

for (const item of mySet) {
 console.log(item);
}
// Output:
// 1
// 2
// 3

// Differences between array-like and iterable objects:
function printItems(collection) {
  for (let i = 0; i < collection.length; i++) {
   console.log(collection[i]);
  }
 }

 const arrayLike = { 0: 'a', 1: 'b', 2: 'c', length: 3 };
 const iterableArray = ['x', 'y', 'z'];


 printItems(arrayLike); // Output: "a", "b", "c"
 printItems(iterableArray); // Output: "x", "y", "z"
//===========END===============
```

//   116 Generators- real examples all Example in Javascript

// Generators in JavaScript are functions that can be paused and resumed, allowing for the generation of a sequence of values over time. They provide a powerful mechanism for controlling iteration and asynchronous programming.

// Generating a sequence of numbers:

```
function* numberSequence() {
  let num = 1;
  while (true) {
   yield num++;
  }
 }


 const generator = numberSequence();
 console.log(generator.next().value); // Output: 1
 console.log(generator.next().value); // Output: 2
 console.log(generator.next().value); // Output: 3
```

//   Customizing iteration using yield and yield*:

```
function* teamMembers() {
  yield 'John';
  yield 'Jane';

  yield* ['Mike', 'Sarah'];

  yield 'Tom';

}


const team = teamMembers();

for (const member of team) {

  console.log(member);

}

// Output:

// "John"

// "Jane"

// "Mike"

// "Sarah"

// "Tom"
```

//   Asynchronous programming with generators:

```
function fetchData(url) {

    return new Promise((resolve) => {

      setTimeout(() => {

        resolve(`Data from ${url}`);

      }, 2000);

    });

  }
```

```
function* dataGenerator() {
  yield fetchData('https://api.example.com/data1');
  yield fetchData('https://api.example.com/data2');
  yield fetchData('https://api.example.com/data3');
}

const generator = dataGenerator();

function handleResult(result) {
  if (!result.done) {
    result.value.then((data) => {
      console.log(data);
      handleResult(generator.next());
    });
  }
}

handleResult(generator.next());
```

//===========END===============

//   117 Protypical inheritance- real examples all Example in Javascript

// Prototype-based inheritance is a feature in JavaScript that allows objects to inherit properties and methods from other objects. It is a fundamental concept in JavaScript's object-oriented programming paradigm.

// Inheriting from a base object:

```
// Base object

const animal = {

  sound: 'No sound',

  makeSound() {

   console.log(this.sound);

  }

 };


 // Derived object

 const cat = Object.create(animal);

 cat.sound = 'Meow';


 cat.makeSound(); // Output: "Meow"

//   Constructor functions and prototypal inheritance:

// Constructor function

function Person(name) {

  this.name = name;

 }


 Person.prototype.greet = function() {

  console.log(`Hello, my name is ${this.name}`);

 };


 // Derived constructor function
```

```
function Employee(name, job) {
 Person.call(this, name);
 this.job = job;
}

Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;

Employee.prototype.work = function() {
 console.log(`${this.name} is working as a ${this.job}`);
};

const john = new Employee('John', 'Developer');
john.greet(); // Output: "Hello, my name is John"
john.work(); // Output: "John is working as a Developer"
```

// Using class syntax for prototypal inheritance:

```
class Vehicle {
  constructor(type) {
   this.type = type;
  }

  drive() {
   console.log(`Driving a ${this.type}`);
  }
}

class Car extends Vehicle {
  constructor(brand) {
    super('Car');
    this.brand = brand;
  }

  displayBrand() {
    console.log(`Brand: ${this.brand}`);
  }
}

const myCar = new Car('Toyota');
myCar.drive(); // Output: "Driving a Car"
myCar.displayBrand(); // Output: "Brand: Toyota"
```

//===========END===============

// 118 Constructors, .prototype and methods- real examples all Example in Javascript

// Constructors, .prototype, and methods are important concepts in JavaScript that are used to create and define objects with shared properties and behaviors. Here are some real-world examples that demonstrate the use of constructors, .prototype, and methods in JavaScript:

// Using a constructor function and methods:

```
function Person(name, age) {
  this.name = name;
  this.age = age;

  this.greet = function() {
   console.log(`Hello, my name is ${this.name}`);
  };

  this.introduce = function() {
   console.log(`I am ${this.name} and I am ${this.age} years old`);
  };
 }

 const john = new Person('John', 30);
 john.greet(); // Output: "Hello, my name is John"
 john.introduce(); // Output: "I am John and I am 30 years old"

//  Using the .prototype property and methods:
function Car(brand) {
  this.brand = brand;
 }

 Car.prototype.drive = function() {
  console.log(`Driving the ${this.brand}`);
 };
 Car.prototype.honk = function() {

  console.log(`Honking the horn of ${this.brand}`);

 };


 const myCar = new Car('Toyota');

 myCar.drive(); // Output: "Driving the Toyota"

 myCar.honk(); // Output: "Honking the horn of Toyota"
```

// Using ES6 class syntax and methods:

```
class Animal {

  constructor(name) {

   this.name = name;

  }


  eat() {

   console.log(`${this.name} is eating`);

  }


  sleep() {

   console.log(`${this.name} is sleeping`);

  }

 }
```

```
const cat = new Animal('Kitty');

cat.eat(); // Output: "Kitty is eating"

cat.sleep(); // Output: "Kitty is sleeping"


//===========END===============
```

//   119 Primitives as Objects- real examples all Example in Javascript

// In JavaScript, primitive values (such as numbers, strings, booleans) have corresponding object wrapper types (Number, String, Boolean) that allow you to access properties and methods associated with those primitive values.

// Using string methods with a string object:

```
const greeting = 'Hello, World';

console.log(greeting.length); // Output: 12

console.log(greeting.toUpperCase()); // Output: "HELLO, WORLD"

console.log(greeting.charAt(7)); // Output: "W"
```

// Using number methods with a number object:

```
const number = 42;

console.log(number.toString()); // Output: "42"

console.log(number.toFixed(2)); // Output: "42.00"

console.log(number.toExponential(2)); // Output: "4.20e+1"
```

// Using boolean methods with a boolean object:

```
const flag = true;

console.log(flag.toString()); // Output: "true"

//===========END===============
```

//   120 Polyfills- real examples all Example in Javascript

// Polyfills are code snippets or scripts that provide modern functionality in older browsers or environments that do not natively support that functionality. They fill the gaps by implementing the missing features using existing language constructs.

// Array.prototype.includes():

// The Array.prototype.includes() method was introduced in ECMAScript 2016 and is used to check if an array includes a certain value. However, it is not supported in older browsers like Internet Explorer 11. Here's a polyfill that adds support for Array.prototype.includes():

```
if (!Array.prototype.includes) {
  Array.prototype.includes = function(searchElement, fromIndex) {
   var array = Object(this);
   var len = array.length >>> 0;

   if (len === 0) {
    return false;
   }

   var n = fromIndex || 0;
   var k;

   if (n >= 0) {
    k = n;
   } else {
    k = len + n;
    if (k < 0) {
     k = 0;
    }
   }

   while (k < len) {
    var currentElement = array[k];
    if (searchElement === currentElement || (searchElement !== searchElement &&
currentElement !== currentElement)) {
     return true;
    }
    k++;
   }

   return false;
  };
 }
```

// Object.assign():

// The Object.assign() method is used to copy the values of all enumerable properties from one or more source objects to a target object. It was introduced in ECMAScript 2015. For older browsers that do not support it, you can use the following polyfill:

```
if (typeof Object.assign !== 'function') {
  Object.assign = function(target) {
   if (target === null || target === undefined) {
    throw new TypeError('Cannot convert undefined or null to object');
   }
   var output = Object(target);

   for (var index = 1; index < arguments.length; index++) {

    var source = arguments[index];

    if (source !== null && source !== undefined) {

     for (var nextKey in source) {

      if (Object.prototype.hasOwnProperty.call(source, nextKey)) {

       output[nextKey] = source[nextKey];

      }

     }

    }

   }

   return output;

  };

 }
```

//===========END===============

//   121 Static properties - real examples all Example in Javascript

// Static properties in JavaScript are properties that belong to a class itself rather than individual instances of the class. They are accessed directly on the class itself, without the need for an instance.

// Math class with static property:

```
console.log(Math.PI); // Output: 3.141592653589793
```

// Custom class with static property:

```
class Counter {

  static count = 0;
  static increment() {
    Counter.count++;
  }

  static getCount() {
    return Counter.count;
  }
}

Counter.increment();
Counter.increment();
console.log(Counter.getCount()); // Output: 2
```

```
//  ES5 syntax for static properties:
function Circle(radius) {
  this.radius = radius;
}

Circle.PI = 3.1416;

Circle.prototype.getArea = function() {
  return Circle.PI * this.radius * this.radius;
};
```

```
 console.log(Circle.PI); // Output: 3.1416


 const circle = new Circle(5);

 console.log(circle.getArea()); // Output: 78.54

//===========END===============
```

//   122 Class syntax- real examples all Example in Javascript

// Creating a class and instantiating objects:

```
class Person {

  constructor(name, age) {

   this.name = name;

   this.age = age;

  }


  greet() {

   console.log(`Hello, my name is ${this.name}`);

  }


  introduce() {

   console.log(`I am ${this.name} and I am ${this.age} years old`);

  }

 }


 const john = new Person('John', 30);
 john.greet(); // Output: "Hello, my name is John"

 john.introduce(); // Output: "I am John and I am 30 years old"


//  Inheritance using class syntax:

class Animal {

  constructor(name) {

   this.name = name;

  }


  eat() {

   console.log(`${this.name} is eating`);

  }

 }


 class Dog extends Animal {

  bark() {

   console.log(`${this.name} is barking`);

  }

 }


 const dog = new Dog('Buddy');

 dog.eat(); // Output: "Buddy is eating"

 dog.bark(); // Output: "Buddy is barking"
```

```
//  Static methods in a class:
class MathUtils {
  static add(x, y) {
    return x + y;
  }

  static subtract(x, y) {
    return x - y;
  }
 }

 console.log(MathUtils.add(5, 3)); // Output: 8
 console.log(MathUtils.subtract(10, 4)); // Output: 6
//===========END===============
//   123 Getters/ setters- real examples all Example in Javascript
// Using getters and setters for computed properties:
class Circle {
  constructor(radius) {
    this.radius = radius;
  }

  get diameter() {
    return this.radius * 2;
  }
  set diameter(diameter) {
    this.radius = diameter / 2;
  }

  get area() {
    return Math.PI * this.radius ** 2;
  }
}

const circle = new Circle(5);
console.log(circle.diameter); // Output: 10
console.log(circle.area); // Output: 78.53981633974483

circle.diameter = 12;
console.log(circle.radius); // Output: 6
console.log(circle.area); // Output: 113.09733552923255
```

// Using getters and setters for data validation:

```
class Person {
  constructor(name, age) {
    this._name = name;
    this._age = age;
  }
  get name() {
    return this._name.toUpperCase();
  }

  set age(age) {
    if (age > 0 && age < 150) {
      this._age = age;
    } else {
      console.log('Invalid age value');
    }
  }

  get age() {
    return this._age;
  }
}

const john = new Person('John', 30);
console.log(john.name); // Output: "JOHN"
console.log(john.age); // Output: 30

john.age = 200; // Output: "Invalid age value"
console.log(john.age); // Output: 30
```

//===========END===============

//   124 Computer property names- real examples all Example in Javascript

// Computed property names in JavaScript allow you to dynamically set the property name of an object using an expression or a variable. They provide a way to create object properties with dynamic or computed names.

// Using variables as property names:

```
const key = 'name';
const person = {
  [key]: 'John',
  age: 30
};

console.log(person.name); // Output: "John"
console.log(person['name']); // Output: "John"
```

// Using expressions as property names:

```
const prefix = 'prop';
const obj = {
  [`${prefix}1`]: 'Value 1',
  [`${prefix}2`]: 'Value 2'
};

console.log(obj.prop1); // Output: "Value 1"
console.log(obj.prop2); // Output: "Value 2"
```

```
// Using functions to compute property names:
function getPropertyKey(prefix) {
  return `${prefix}_key`;
 }


 const config = {
  [getPropertyKey('prefix')]: 'Value'
 };


 console.log(config.prefix_key); // Output: "Value"
//===========END===============
```

//   125 this" binding issues- real examples all Example in Javascript

// In JavaScript, "binding issues" can refer to problems related to the scope and value of variables, as well as the context in which functions are executed.

```
// Incorrect use of the "this" keyword:
const obj = {
 value: 42,
 getValue: function() {
  return this.value;
 }
};


const getValue = obj.getValue;
```

```
console.log(getValue()); // Error: Cannot read property 'value' of undefined


// Event handler context:
const button = document.querySelector('button');
button.addEventListener('click', function() {
  console.log(this.textContent); // Error: Cannot read property 'textContent' of null
});


// Function binding and arrow functions:
const obj = {
  value: 42,
  getValue: function() {
    return this.value;
  }
};


const getValue = obj.getValue.bind(obj);
console.log(getValue()); // 42


const arrowGetValue = () => obj.getValue();
console.log(arrowGetValue()); // Error: Cannot read property 'value' of undefined
//==========END==============
```

//   126 Inheritance- real examples all Example in Javascript

```
// Inheritance is a fundamental concept in object-oriented programming (OOP) that allows objects to inherit properties and behaviors from other objects. In JavaScript, inheritance can be achieved through prototype-based inheritance.
// Prototype-based inheritance:
// Parent class
function Animal(name) {
  this.name = name;
}

Animal.prototype.sound = function() {
  console.log("Making a sound...");
};

// Child class inheriting from Animal
function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
  console.log("Barking...");
};
```

```
// Creating instances
const myDog = new Dog("Max", "Labrador");
console.log(myDog.name); // Max
myDog.sound(); // Making a sound...
myDog.bark(); // Barking...

// class keyword and extends keyword:
// Parent class
class Shape {
  constructor(color) {
    this.color = color;
  }

  draw() {
    console.log("Drawing a shape...");
  }
}

// Child class extending Shape
class Circle extends Shape {
  constructor(color, radius) {
    super(color);
    this.radius = radius;
```

```
 }

 calculateArea() {
  return Math.PI * this.radius * this.radius;
 }
}

// Creating instances
const myCircle = new Circle("red", 5);
console.log(myCircle.color); // red
myCircle.draw(); // Drawing a shape...
console.log(myCircle.calculateArea()); // 78.53981633974483
//===========END===============
```

//   127 Static methods in Class- real examples all Example in Javascript

// Static methods in JavaScript classes are methods that are bound to the class itself rather than the class instances. They are called on the class itself, without the need to create an instance of the class.

```
// Math utility class with static method:
class MathUtils {
 static add(a, b) {
  return a + b;
 }

 static multiply(a, b) {
    return a * b;
  }
}

console.log(MathUtils.add(5, 3)); // 8
console.log(MathUtils.multiply(4, 6)); // 24

// Utility class for formatting strings:
class StringUtils {
  static capitalize(str) {
    return str.charAt(0).toUpperCase() + str.slice(1);
  }

  static reverse(str) {
    return str.split('').reverse().join('');
  }
}

console.log(StringUtils.capitalize('hello')); // Hello
console.log(StringUtils.reverse('JavaScript')); // tpircSavaJ

// Date utility class with a static method to check leap year:
class DateUtils {
  static isLeapYear(year) {
    return (year % 4 === 0 && year % 100 !== 0) || year % 400 === 0;
  }
}

console.log(DateUtils.isLeapYear(2020)); // true
console.log(DateUtils.isLeapYear(2021)); // false
//==========END==============
```

// 128 Private and protected members- real examples all Example in Javascript

// In JavaScript, there is no built-in support for private and protected members like in some other programming languages. However, there are conventions and techniques that can be used to achieve similar encapsulation and access control.

// Private members using closures:

```
function Counter() {
  let count = 0; // Private member

  this.increment = function() {
    count++;
    console.log(count);
  };
}

const counter = new Counter();
counter.increment(); // 1
counter.increment(); // 2
```

```
console.log(counter.count); // undefined (private)

// Protected members using naming conventions:
class Shape {
 constructor() {
  this._color = 'red'; // Protected member
 }

 _draw() {
  console.log('Drawing a shape');
 }
}

class Circle extends Shape {
 constructor() {
  super();
  this._radius = 5; // Protected member
 }

 drawCircle() {
  this._draw(); // Accessing protected method
  console.log(`Color: ${this._color}`); // Accessing protected property
  console.log(`Radius: ${this._radius}`);
 }
}

const circle = new Circle();
circle.drawCircle();
//===========END===============
```

//   129 Asynchronous Javascript - Callbacks- real examples all Example in Javascript

// Asynchronous JavaScript involves executing code non-sequentially, allowing other operations to continue while waiting for certain tasks to complete. Callbacks are a common way to handle asynchronous operations in JavaScript.

// Asynchronous file reading with callbacks:

```
const fs = require('fs');

function readFileAsync(path, callback) {
  fs.readFile(path, 'utf8', (err, data) => {
    if (err) {
      callback(err, null);
    } else {
      callback(null, data);
    }
  });
}

readFileAsync('file.txt', (err, data) => {
  if (err) {
    console.error('Error:', err);
  } else {
    console.log('File content:', data);
  }
});

// Asynchronous API request with callbacks:
function fetchUserData(userId, callback) {
  // Simulating API request delay
  setTimeout(() => {
    const user = { id: userId, name: 'John Doe' };
    callback(null, user);
  }, 2000);
}

fetchUserData(123, (err, user) => {
  if (err) {
    console.error('Error:', err);
  } else {
    console.log('User:', user);
  }
});

// Asynchronous event handling with callbacks:
const button = document.querySelector('button');
```

```
function handleClick(event, callback) {
  event.preventDefault();
  callback();
}

button.addEventListener('click', (event) => {
  handleClick(event, () => {
    console.log('Button clicked!');
  });
});
//===========END===============
```

//   130 Callback hell- real examples all Example in Javascript

// Callback hell, also known as the pyramid of doom, occurs when multiple asynchronous operations are nested within each other using callbacks. This can lead to deeply nested and hard-to-read code.

// Nested file reading with callbacks:

```
const fs = require('fs');

fs.readFile('file1.txt', 'utf8', (err, data1) => {
  if (err) {
    console.error('Error:', err);
  } else {
    fs.readFile('file2.txt', 'utf8', (err, data2) => {
      if (err) {
        console.error('Error:', err);
      } else {
        fs.readFile('file3.txt', 'utf8', (err, data3) => {
          if (err) {
            console.error('Error:', err);
          } else {
            console.log('Data:', data1, data2, data3);
          }
        });
      }
    });
  }
});

// Nested API requests with callbacks:
function fetchUserData(userId, callback) {
  setTimeout(() => {
    const user = { id: userId, name: 'John Doe' };
    callback(null, user);
  }, 2000);
}

function fetchUserPosts(user, callback) {
  setTimeout(() => {
    const posts = ['Post 1', 'Post 2', 'Post 3'];

    callback(null, posts);

  }, 2000);

}


fetchUserData(123, (err, user) => {

  if (err) {

    console.error('Error:', err);

  } else {

    fetchUserPosts(user, (err, posts) => {

      if (err) {

        console.error('Error:', err);

      } else {

        console.log('User:', user);

        console.log('Posts:', posts);

      }

    });

  }

});

//==========END==============
```

//   131 Promises & Chaining- real examples all Example in Javascript

// Promises and chaining provide a more structured and readable way to handle asynchronous operations in JavaScript. Promises represent the eventual completion or failure of an asynchronous operation and allow for more organized handling of asynchronous code.

// Promise-based file reading and chaining:

```
const fs = require('fs');

function readFileAsync(path) {
  return new Promise((resolve, reject) => {
    fs.readFile(path, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}

readFileAsync('file1.txt')
  .then((data1) => {
    console.log('File 1:', data1);
    return readFileAsync('file2.txt');
  })
  .then((data2) => {
    console.log('File 2:', data2);
    return readFileAsync('file3.txt');
  })
  .then((data3) => {
```

```
  console.log('File 3:', data3);
})
.catch((err) => {
  console.error('Error:', err);
});

// Promise-based API request and chaining:
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const user = { id: userId, name: 'John Doe' };
      resolve(user);
    }, 2000);
  });
}

function fetchUserPosts(user) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const posts = ['Post 1', 'Post 2', 'Post 3'];
      resolve(posts);
    }, 2000);
  });
}
```

```
 fetchUserData(123)
  .then((user) => {
   console.log('User:', user);
   return fetchUserPosts(user);
  })
  .then((posts) => {
   console.log('Posts:', posts);
  })
  .catch((err) => {
   console.error('Error:', err);
  });
//===========END===============
```

//   132 Promise API- real examples all Example in Javascript

// The Promise API in JavaScript provides several methods to work with promises, allowing for more advanced handling of asynchronous operations.

// Promise.all():

```
function fetchUserDetails() {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
   const userDetails = { name: 'John Doe', age: 30 };
   resolve(userDetails);
  }, 2000);
 });
```

```
}

function fetchUserPosts() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const userPosts = ['Post 1', 'Post 2', 'Post 3'];
      resolve(userPosts);
    }, 3000);
  });
}

Promise.all([fetchUserDetails(), fetchUserPosts()])
  .then(([userDetails, userPosts]) => {
    console.log('User Details:', userDetails);
    console.log('User Posts:', userPosts);
  })
  .catch((err) => {
    console.error('Error:', err);
  });

  // Promise.race():
  function fetchData(url, timeout) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
```

```
      resolve(`Data received from ${url}`);
    }, timeout);
  });
}

const promise1 = fetchData('api.example.com/endpoint1', 2000);
const promise2 = fetchData('api.example.com/endpoint2', 3000);

Promise.race([promise1, promise2])
  .then((result) => {
    console.log('First to resolve:', result);
  })
  .catch((err) => {
    console.error('Error:', err);
  });

  // Promise.resolve() and Promise.reject():
  function checkIfEven(number) {
    return new Promise((resolve, reject) => {
      if (number % 2 === 0) {
        resolve(`${number} is even.`);
      } else {
        reject(`${number} is not even.`);
      }
```

```
   });
  }


  Promise.resolve('Start')
   .then(() => checkIfEven(4))
   .then((result) => {
    console.log(result);
    return checkIfEven(7);
   })
   .catch((err) => {
    console.error('Error:', err);
   })
   .finally(() => {
    console.log('Promise chain completed.');
   });
//===========END===============
```

//   133 Async Await- real examples all Example in Javascript

// Async/await is a modern JavaScript feature that allows for more concise and synchronous-looking code when working with asynchronous operations. It provides a way to write asynchronous code that resembles synchronous code, making it easier to understand and maintain.

// Async/await with Promises:

```
function fetchUserDetails() {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
      const userDetails = { name: 'John Doe', age: 30 };
      resolve(userDetails);
    }, 2000);
  });
}

function fetchUserPosts() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const userPosts = ['Post 1', 'Post 2', 'Post 3'];
      resolve(userPosts);
    }, 3000);
  });
}

async function fetchData() {
  try {
    const userDetails = await fetchUserDetails();
    const userPosts = await fetchUserPosts();
    console.log('User Details:', userDetails);
    console.log('User Posts:', userPosts);
  } catch (err) {
    console.error('Error:', err);
  }
```

```
}

fetchData();
```

// Async/await with Fetch API:

```
async function fetchWeather(city) {
  try {
    const response = await fetch(`https://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=${city}`);
    if (!response.ok) {
      throw new Error('Failed to fetch weather data.');
    }
    const data = await response.json();
    console.log('Weather:', data);
  } catch (err) {
    console.error('Error:', err);
  }
}

fetchWeather('London');
//===========END===============
```

//   134 Async Iterators & Async Generators - real examples all Example in Javascript

// Async iterators and async generators are advanced features in JavaScript that allow for asynchronous iteration over collections and generation of asynchronous sequences. They provide a powerful mechanism for working with asynchronous data streams.

```
// Async iterator over an asynchronous collection:
const asyncCollection = {
  data: ['A', 'B', 'C'],
  async *[Symbol.asyncIterator]() {
    for (const item of this.data) {
      await new Promise((resolve) => setTimeout(resolve, 1000));
      yield item;
    }
  }
};

(async () => {
  for await (const item of asyncCollection) {
    console.log(item);
  }
})();
// Asynchronous data stream using async generators:
async function* generateData() {
  let value = 1;
  while (true) {
    await new Promise((resolve) => setTimeout(resolve, 1000));
    yield value++;
  }
}
```

```
(async () => {
  for await (const item of generateData()) {
    console.log(item);
    if (item >= 5) {
      break;
    }
  }
})();
//===========END===============
```

//   135 Native prototypes- real examples all Example in Javascript

// Native prototypes in JavaScript allow you to extend and modify the behavior of built-in objects and their prototypes. While it's generally recommended to avoid modifying native prototypes directly,

// Extending the Array prototype:

```
Array.prototype.first = function() {
  return this[0];
};


const arr = [1, 2, 3];
console.log(arr.first()); // 1
```

// Extending the String prototype:

```
String.prototype.capitalize = function() {
  return this.charAt(0).toUpperCase() + this.slice(1);
};


const str = 'hello';

console.log(str.capitalize()); // Hello


// Extending the Number prototype:

Number.prototype.square = function() {

  return this * this;

};


const num = 5;

console.log(num.square()); // 25




  //  "emmet.includeLanguages": { "javascript": "javascriptreact", "typescript": "typescriptreact" },
```