+6

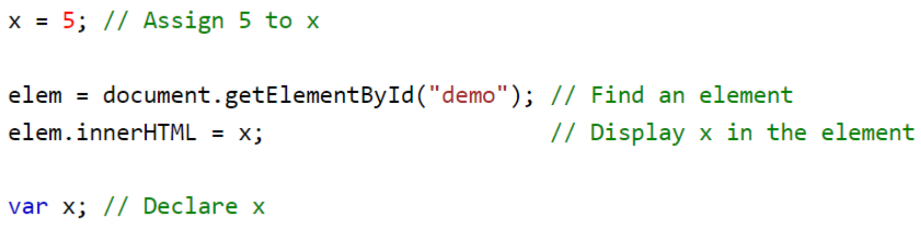
Lecture

Introduction to JavaScript Basic:

- JavaScript is a widely used and most popular programming language.
- It is a programming language of the Web and operates within web browsers to make web pages interactive and dynamic.
- JavaScript is known for its ability to control the Document Object Model (DOM), which represents the structure of a web page.
- **Difference between var, let and const in JavaScript.**

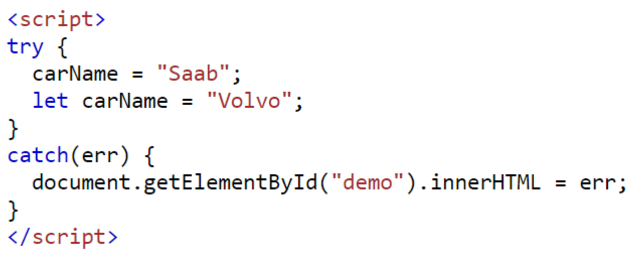
  In JavaScript, var, let, and const are used for declaring variables, but they have different scoping rules and behavior. Here are the key differences, along with coding examples:

Hoisting: It is JavaScript's default behavior of moving declarations to the top.



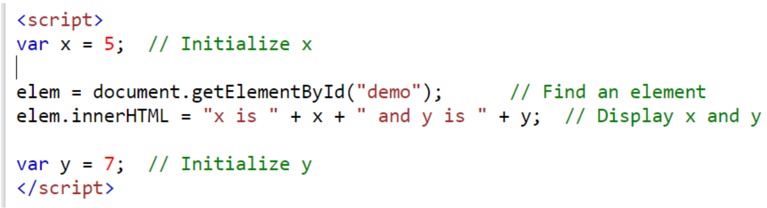
```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
```

Variables defined with **let** and **const** are hoisted to the top of the block, but not initialized.



```
<script>
try {
  carName = "Saab";
  let carName = "Volvo";
}
catch(err) {
  document.getElementById("demo").innerHTML = err;
}
</script>
```

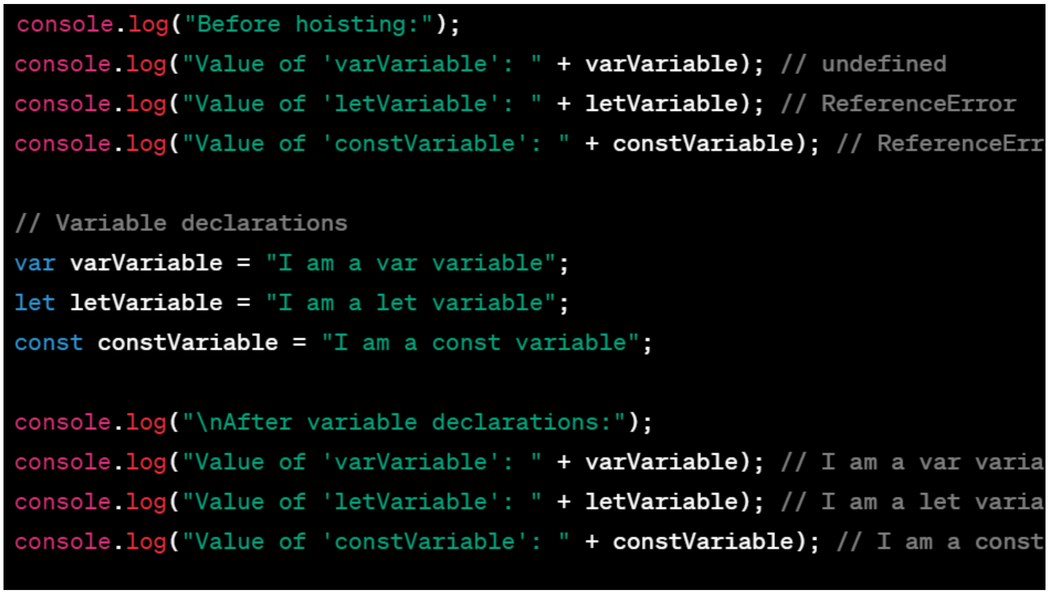
Note: JavaScript only hoists declarations, not initializations.



```
<script>
var x = 5;  // Initialize x

elem = document.getElementById("demo");      // Find an element
elem.innerHTML = "x is " + x + " and y is " + y;  // Display x and y

var y = 7;  // Initialize y
</script>
```

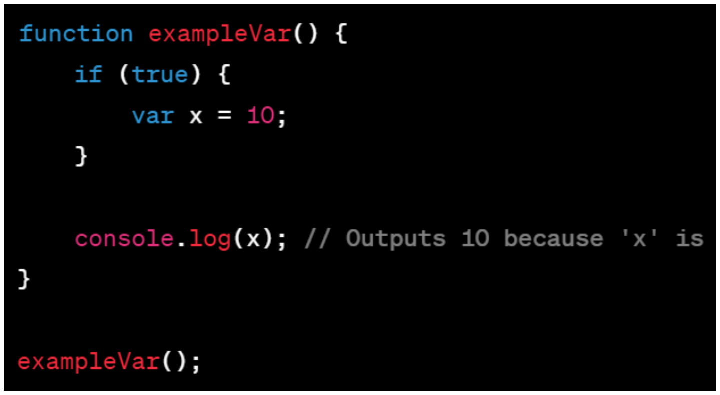
Output: x is 5 and y is undefined.

Example:



```
console.log("Before hoisting:");
console.log("Value of 'varVariable': " + varVariable); // undefined
console.log("Value of 'letVariable': " + letVariable); // ReferenceError
console.log("Value of 'constVariable': " + constVariable); // ReferenceErr

// Variable declarations
var varVariable = "I am a var variable";
let letVariable = "I am a let variable";
const constVariable = "I am a const variable";

console.log("\nAfter variable declarations:");
console.log("Value of 'varVariable': " + varVariable); // I am a var varia
console.log("Value of 'letVariable': " + letVariable); // I am a let varia
console.log("Value of 'constVariable': " + constVariable); // I am a const
```

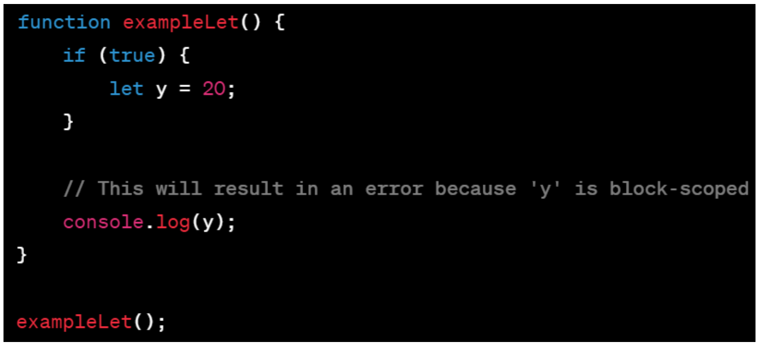
1. **var: var** declarations are function scoped. This means that variables declared with **var** are accessible throughout the entire function where they are declared.
    a. **Variables declared with var are hoisted**, which means their declarations are moved to the top of the function or global scope during execution.



```
function exampleVar() {
    if (true) {
        var x = 10;
    }

    console.log(x); // Outputs 10 because 'x' is
}

exampleVar();
```

2. **let: let** declarations are block scoped. This means that variables declared with let are only accessible within the block where they are defined (inside curly braces {}).

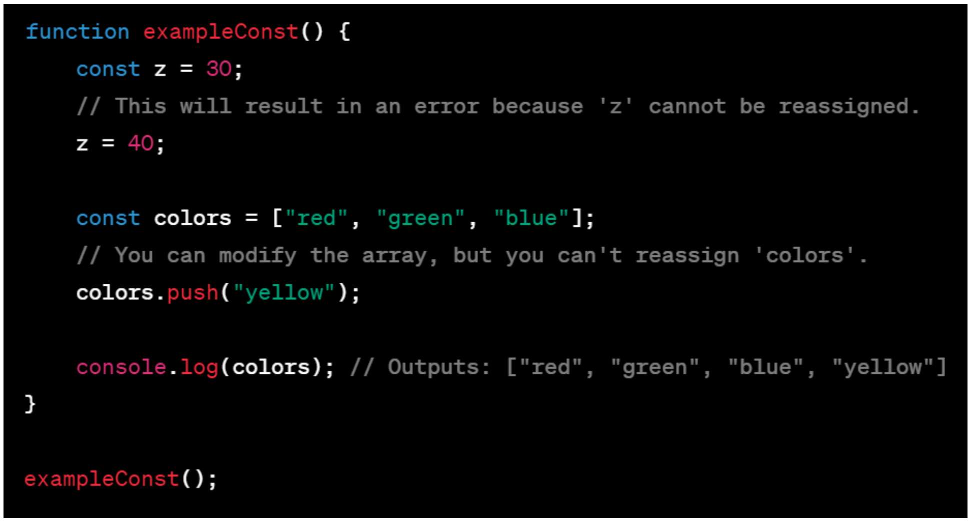
    a. **Variables** declared with let are also hoisted but not initialized. They are in a "temporal dead zone" until they are defined, which prevents you from accessing them before their declaration.



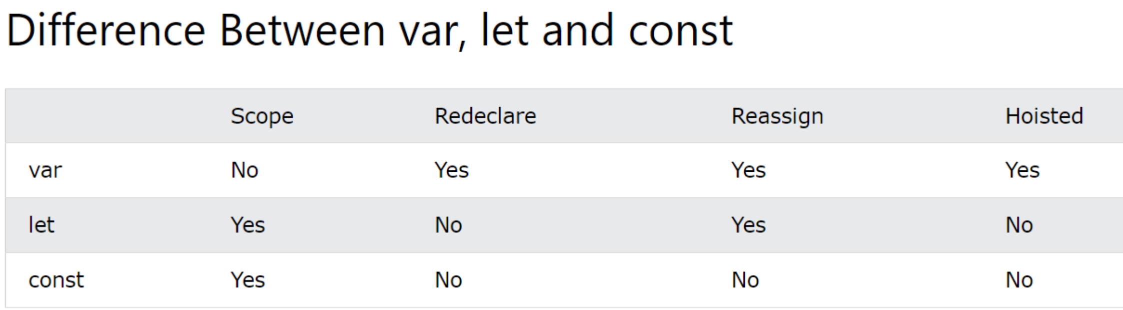
```
function exampleLet() {
    if (true) {
        let y = 20;
    }

    // This will result in an error because 'y' is block-scoped
    console.log(y);
}

exampleLet();
```

3. **const: const** declarations are also block-scoped like let.

    a. **Variables** declared with const are constants, which means their values cannot be reassigned after their initial assignment. However, the value itself can be mutable for objects and arrays.



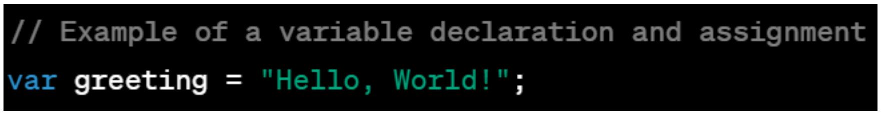
```
function exampleConst() {
    const z = 30;
    // This will result in an error because 'z' cannot be reassigned.
    z = 40;

    const colors = ["red", "green", "blue"];
    // You can modify the array, but you can't reassign 'colors'.
    colors.push("yellow");

    console.log(colors); // Outputs: ["red", "green", "blue", "yellow"]
}

exampleConst();
```



Difference Between var, let and const

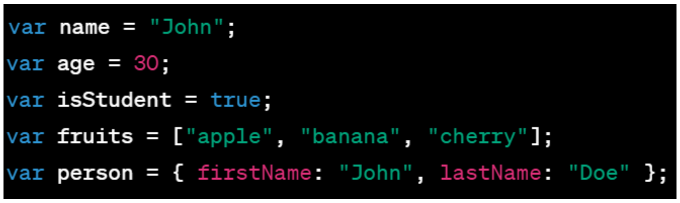
	Scope	Redeclare	Reassign	Hoisted
var	No	Yes	Yes	Yes
let	Yes	No	Yes	No
const	Yes	No	No	No

1. **Syntax:**
    a. JavaScript terminates statements with semicolons (;). It is case-sensitive and uses variables to store data.
    b. JavaScript accepts both double and single quotes:



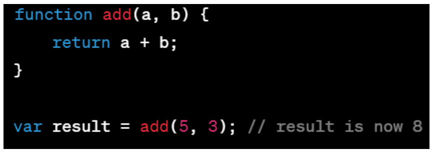
```
// Example of a variable declaration and assignment
var greeting = "Hello, World!";
```

2. **Data Types:**
    a. JavaScript has several data types, including strings, numbers, booleans, arrays, and objects.



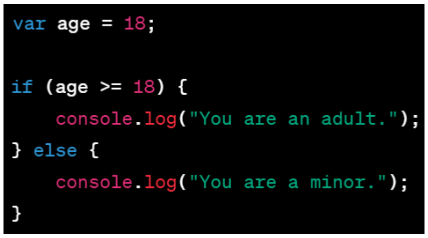
```
var name = "John";
var age = 30;
var isStudent = true;
var fruits = ["apple", "banana", "cherry"];
var person = { firstName: "John", lastName: "Doe" };
```

3. **Functions:**
    a. Functions in JavaScript allow you to group code into reusable blocks. They can take parameters and return values.



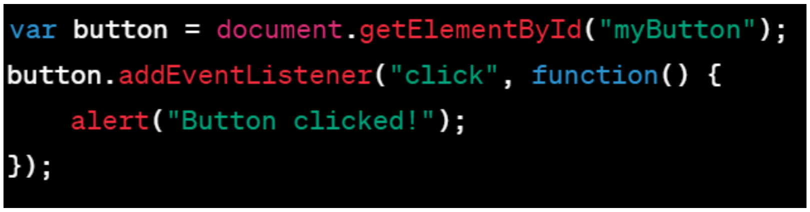
```
function add(a, b) {
    return a + b;
}

var result = add(5, 3); // result is now 8
```

4. **Conditional Statements:**
    a. JavaScript supports if statements for conditional execution of code.



```
var age = 18;

if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

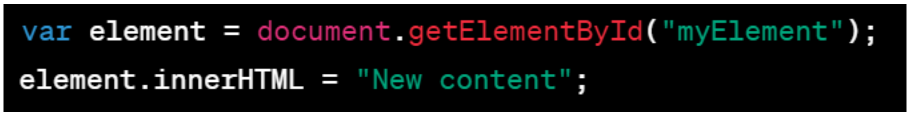
5. **Event Handling:**
    a. JavaScript handles user interactions, like button clicks and form submissions. Event listeners are used to responding to these events.



```
var button = document.getElementById("myButton");
button.addEventListener("click", function() {
    alert("Button clicked!");
});
```

6. **DOM Manipulation:**

   a. JavaScript can modify the DOM, allowing you to change the content and structure of web pages dynamically.

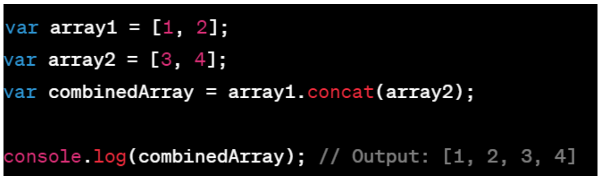


```
var element = document.getElementById("myElement");
element.innerHTML = "New content";
```

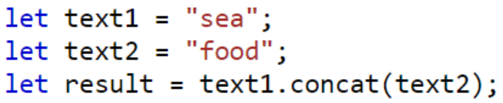
7. **Arrays:**

   a. Arrays in JavaScript are used to store collections of data. They come with a variety of built-in methods for manipulation. Let's explore some of these methods:

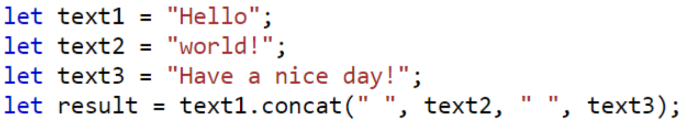
i. **Concat:** The concat method combines two or more arrays into a new array.



```
var array1 = [1, 2];
var array2 = [3, 4];
var combinedArray = array1.concat(array2);

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

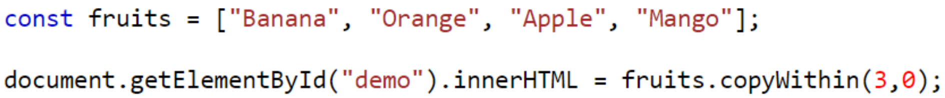


```
let text1 = "sea";
let text2 = "food";
let result = text1.concat(text2);
```



```
let text1 = "Hello";
let text2 = "world!";
let text3 = "Have a nice day!";
let result = text1.concat(" ", text2, " ", text3);
```

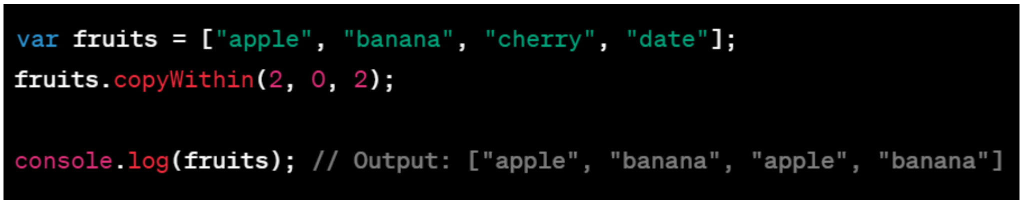
b. **CopyWithin:** This method copies a portion of an array to another location within the same array.



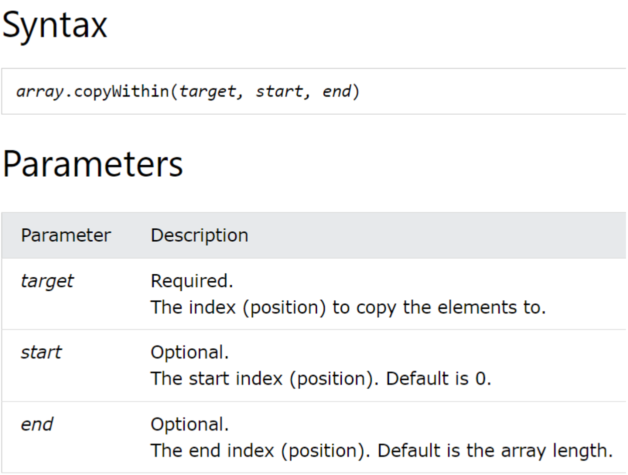
```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

document.getElementById("demo").innerHTML = fruits.copyWithin(3,0);
```

Output:

Banana,Orange,Apple,Banana



```
var fruits = ["apple", "banana", "cherry", "date"];
fruits.copyWithin(2, 0, 2);

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



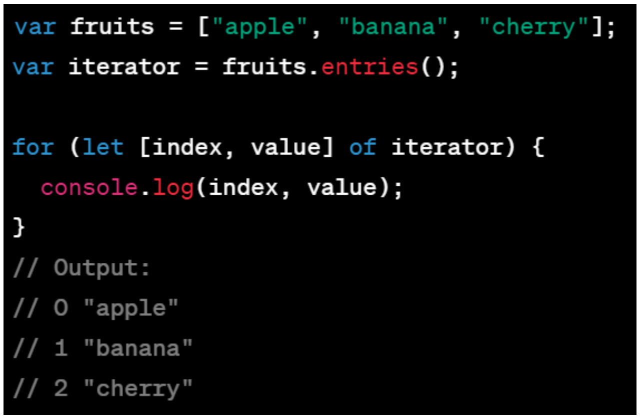
Syntax

```
array.copyWithin(target, start, end)
```

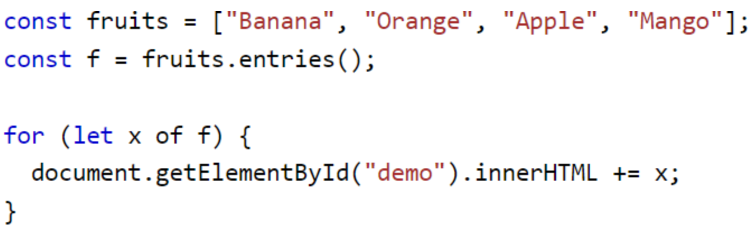
Parameters

Parameter	Description
target	Required. The index (position) to copy the elements to.
start	Optional. The start index (position). Default is 0.
end	Optional. The end index (position). Default is the array length.

c. **Entries:** The entries method returns an iterator of key-value pairs for each element in an array. The entries() method creates an Array Iterator and then iterates over the key/value pairs:

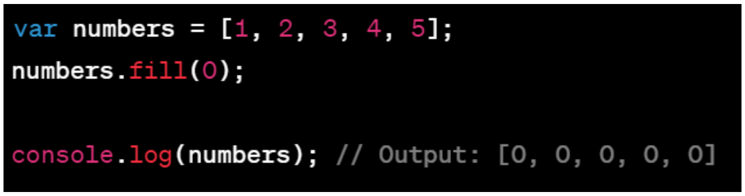


```
var fruits = ["apple", "banana", "cherry"];
var iterator = fruits.entries();

for (let [index, value] of iterator) {
  console.log(index, value);
}
// Output:
// 0 "apple"
// 1 "banana"
// 2 "cherry"
```

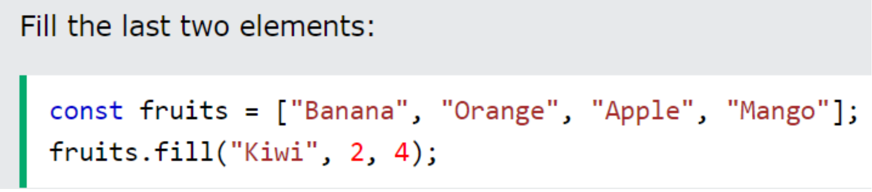


```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const f = fruits.entries();

for (let x of f) {
  document.getElementById("demo").innerHTML += x;
}
```

d. **Fill:** The fill method changes all elements in an array with a provided value.

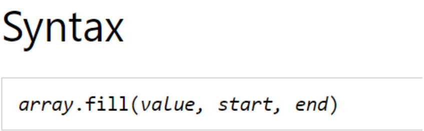


```
var numbers = [1, 2, 3, 4, 5];
numbers.fill(0);

console.log(numbers); // Output: [0, 0, 0, 0, 0]
```



Fill the last two elements:

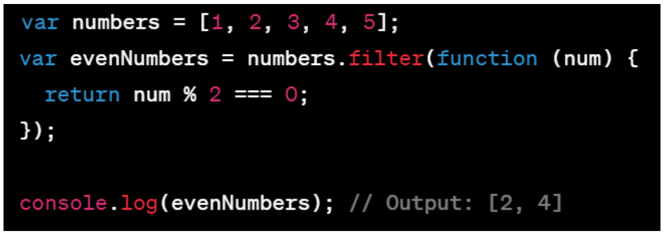
```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.fill("Kiwi", 2, 4);
```



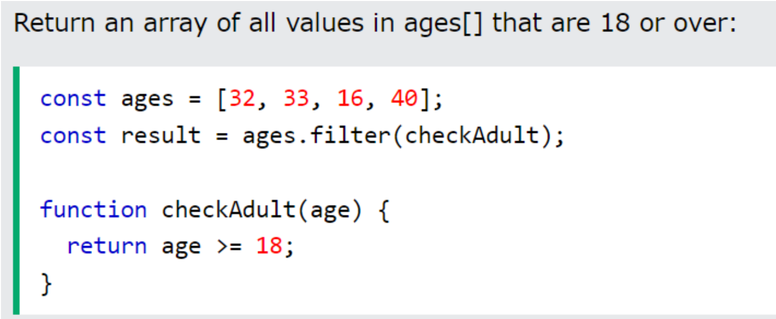
Syntax

```
array.fill(value, start, end)
```

e. Filter: The filter method creates a new array with elements that pass a given test.



```
var numbers = [1, 2, 3, 4, 5];
var evenNumbers = numbers.filter(function (num) {
  return num % 2 === 0;
});

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

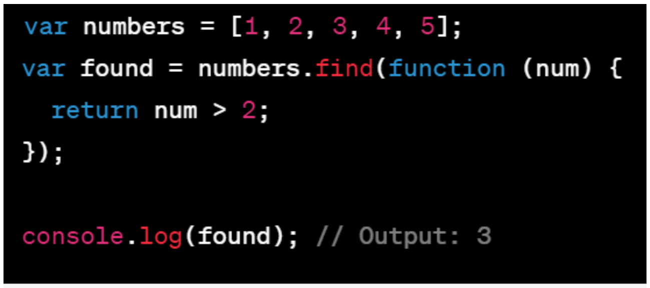


Return an array of all values in ages[] that are 18 or over:

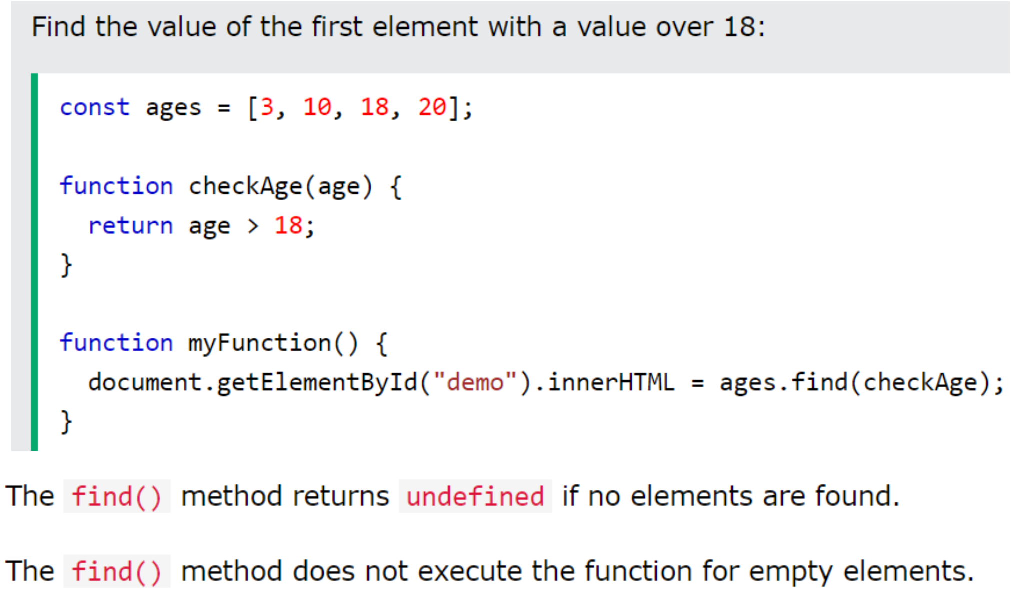
```
const ages = [32, 33, 16, 40];
const result = ages.filter(checkAdult);

function checkAdult(age) {
  return age >= 18;
}
```

The **filter()** method does not execute the function for empty elements.

f. Find: The find method returns the first element in an array that satisfies a provided test function.



```
var numbers = [1, 2, 3, 4, 5];
var found = numbers.find(function (num) {
  return num > 2;
});

console.log(found); // Output: 3
```



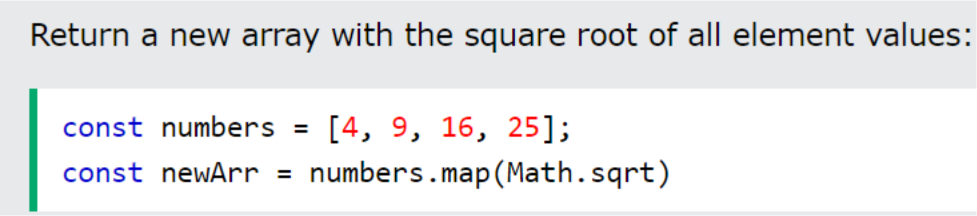
Find the value of the first element with a value over 18:

```
const ages = [3, 10, 18, 20];

function checkAge(age) {
  return age > 18;
}

function myFunction() {
  document.getElementById("demo").innerHTML = ages.find(checkAge);
}
```

The `find()` method returns `undefined` if no elements are found.

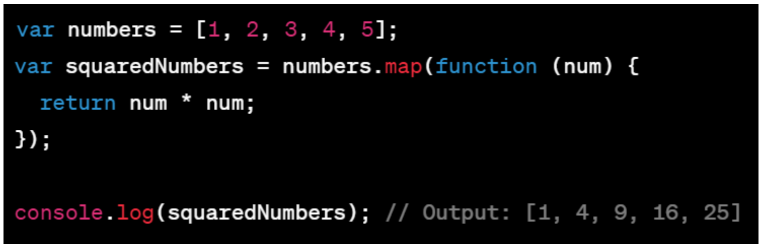
The `find()` method does not execute the function for empty elements.

g. **Map:** The map method creates a new array by applying a provided function to each element of the original array.

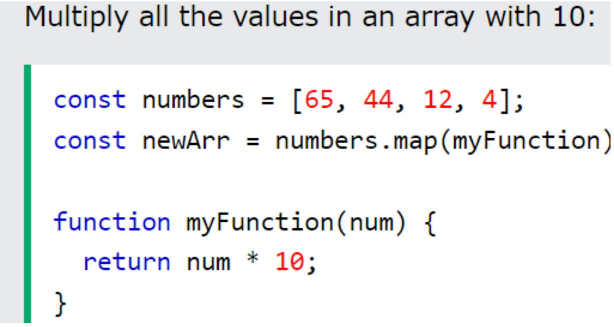


Return a new array with the square root of all element values:

```
const numbers = [4, 9, 16, 25];
const newArr = numbers.map(Math.sqrt)
```



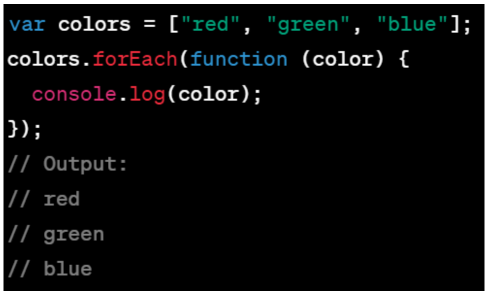
```
var numbers = [1, 2, 3, 4, 5];
var squaredNumbers = numbers.map(function (num) {
  return num * num;
});

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



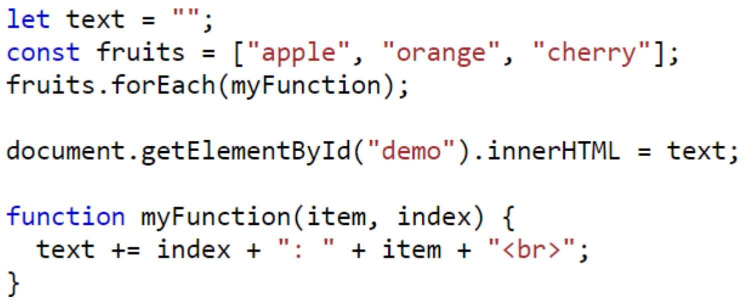
Multiply all the values in an array with 10:

```
const numbers = [65, 44, 12, 4];
const newArr = numbers.map(myFunction)

function myFunction(num) {
  return num * 10;
}
```

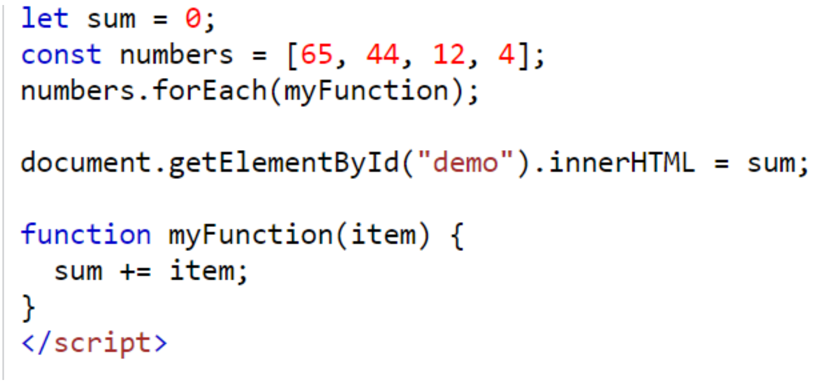
h. forEach: The forEach method executes a provided function once for each array element**. It has no return.**



```
var colors = ["red", "green", "blue"];
colors.forEach(function (color) {
  console.log(color);
});
// Output:
// red
// green
// blue
```

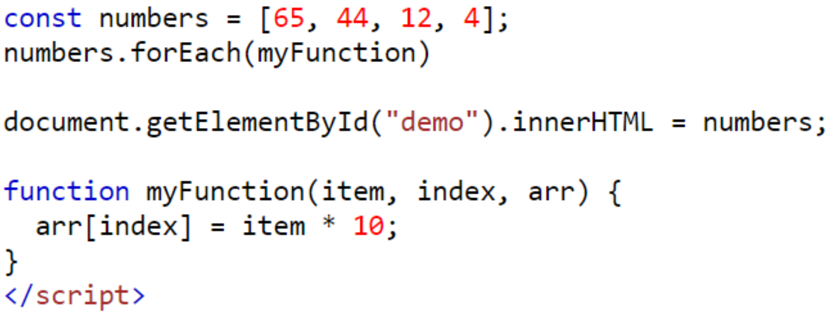


```
let text = "";
const fruits = ["apple", "orange", "cherry"];
fruits.forEach(myFunction);

document.getElementById("demo").innerHTML = text;

function myFunction(item, index) {
  text += index + ": " + item + "<br>";
}
```



```
let sum = 0;
const numbers = [65, 44, 12, 4];
numbers.forEach(myFunction);

document.getElementById("demo").innerHTML = sum;

function myFunction(item) {
  sum += item;
}
</script>
```

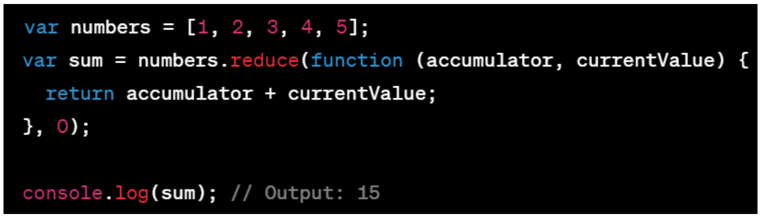
Output: 125



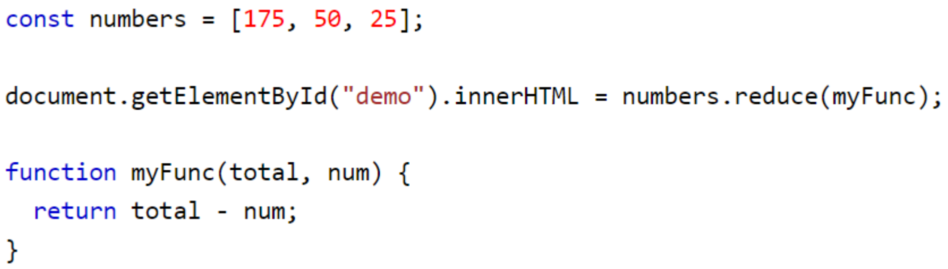
```
const numbers = [65, 44, 12, 4];
numbers.forEach(myFunction)

document.getElementById("demo").innerHTML = numbers;

function myFunction(item, index, arr) {
  arr[index] = item * 10;
}
</script>
```

Output: 650,440,120,40

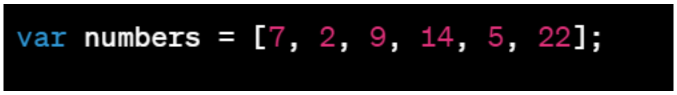
i. Reduce: The reduce method applies a function to an accumulator and each element in the array to reduce it to a single value. It considers by default the first index as first parameter.



```
var numbers = [1, 2, 3, 4, 5];
var sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);

console.log(sum); // Output: 15
```



```
const numbers = [175, 50, 25];

document.getElementById("demo").innerHTML = numbers.reduce(myFunc);

function myFunc(total, num) {
  return total - num;
}
```



```
var numbers = [7, 2, 9, 14, 5, 22];
```

Now, we want to find the maximum value in this array using the reduce function.

```
var maxNumber = numbers.reduce(function (accumulator, currentValue,
currentIndex, array) {
  if (currentValue > accumulator) {
    return currentValue; // Update accumulator with the new maximum value
  } else {
    return accumulator; // Keep the current maximum value
  }
}, numbers[0]); // Start with the first element as the initial accumulator value
```

Explanation:

The callback function is executed for each element in the array. It receives four parameters:

accumulator: This is the accumulated result of the reduction process. It starts with the initial value (the first element of the array) and is updated with each iteration.

currentValue: This is the current element in the array that is being processed in the current iteration.

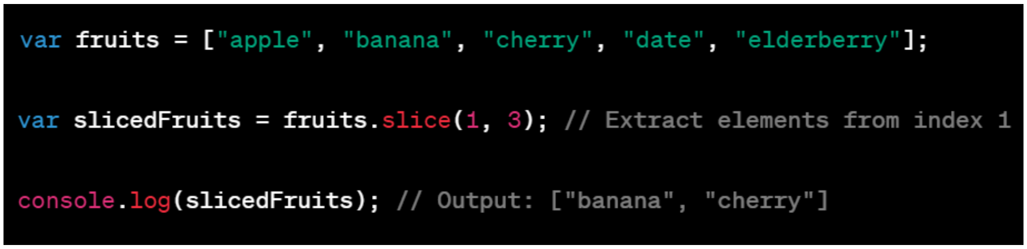
currentIndex: This is the index of the current element in the array.

array: This is the original array (numbers in this case).

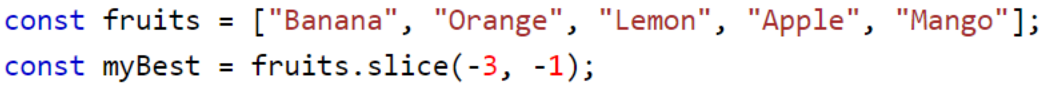
Lecture

Slice, Pop, Push, Flat, and Sort Operations on Array:

a. **slice:** The slice method is used to extract a portion of an array into a new array without modifying the original array. This method selects from a given start (inclusive), up to a given end (exclusive).

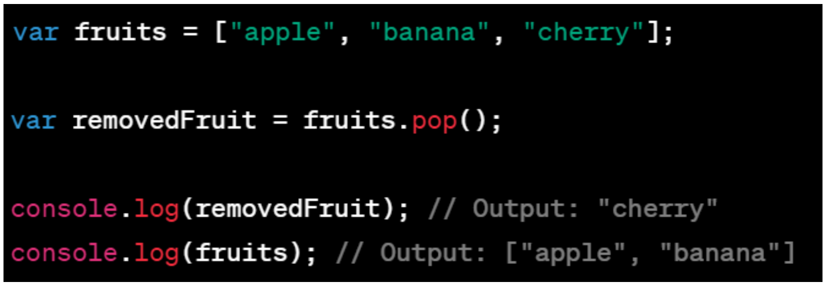


```
var fruits = ["apple", "banana", "cherry", "date", "elderberry"];

var slicedFruits = fruits.slice(1, 3); // Extract elements from index 1

console.log(slicedFruits); // Output: ["banana", "cherry"]
```



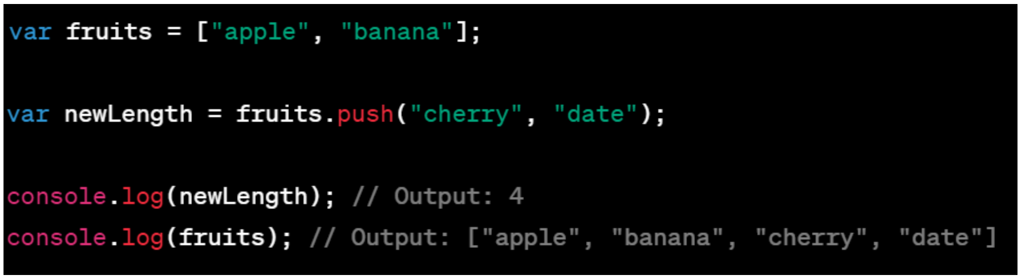
```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const myBest = fruits.slice(-3, -1);
```

b. **pop:** The pop method removes and returns the last element from an array. This method changes the original array and returns the removed item.



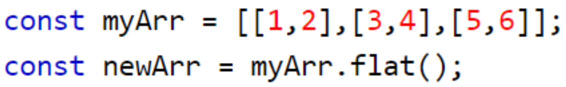
```
var fruits = ["apple", "banana", "cherry"];

var removedFruit = fruits.pop();

console.log(removedFruit); // Output: "cherry"
console.log(fruits); // Output: ["apple", "banana"]
```

c. **push:** The push method adds one or more elements to the end of an array and returns the new length of the array. This method changes the length of the array.

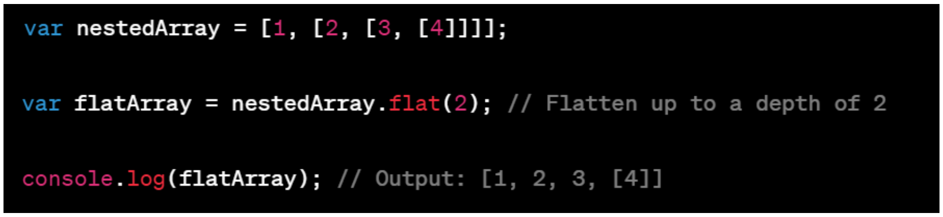


```
var fruits = ["apple", "banana"];

var newLength = fruits.push("cherry", "date");

console.log(newLength); // Output: 4
console.log(fruits); // Output: ["apple", "banana", "cherry", "date"]
```

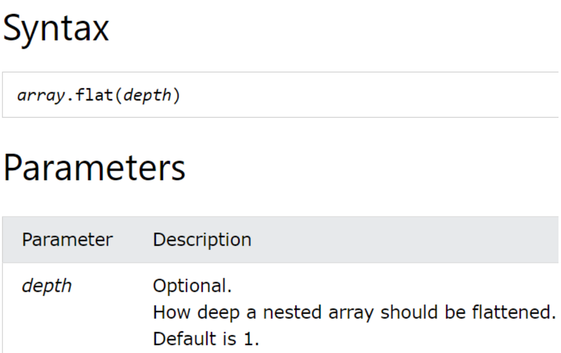
d. flat: The flat method creates a new array and concatenates sub-array elements in the new array.



```
const myArr = [[1,2],[3,4],[5,6]];
const newArr = myArr.flat();
```



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

var flatArray = nestedArray.flat(2); // Flatten up to a depth of 2

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



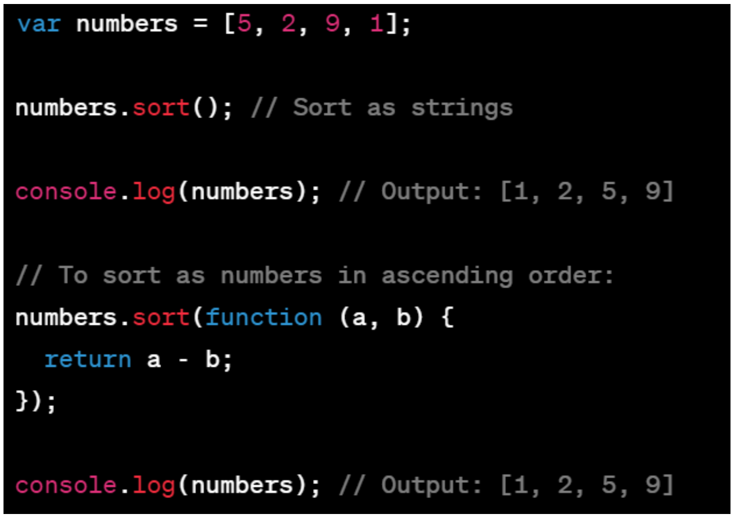
Syntax

```
array.flat(depth)
```

Parameters

Parameter	Description
depth	Optional. How deep a nested array should be flattened. Default is 1.

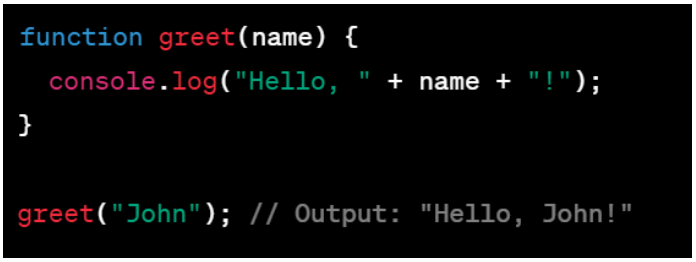
e. sort: This method sorts the elements of an array and returns the sorted array. It overwrites the original array. By default, it sorts elements as strings.



```
var numbers = [5, 2, 9, 1];

numbers.sort(); // Sort as strings

console.log(numbers); // Output: [1, 2, 5, 9]

// To sort as numbers in ascending order:
numbers.sort(function (a, b) {
  return a - b;
});

console.log(numbers); // Output: [1, 2, 5, 9]
```

JavaScript Functions & ES6 Arrow Functions

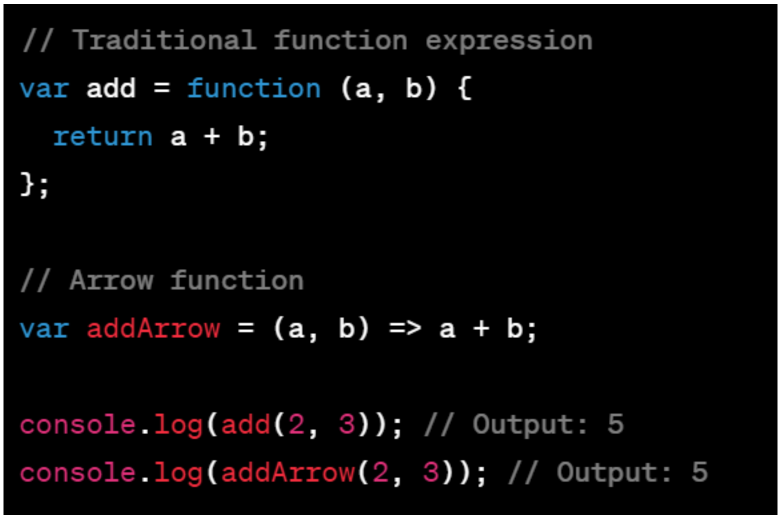
- JavaScript functions are blocks of code that can be called and executed.



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

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

ES6 Arrow Functions:

- Arrow functions are a concise way to define functions in JavaScript introduced in ES6 (ECMAScript 2015). They have a shorter syntax compared to traditional function expressions.

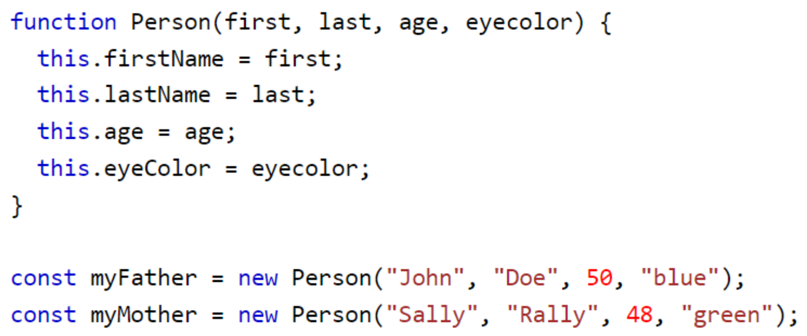


```
// Traditional function expression
var add = function (a, b) {
  return a + b;
};

// Arrow function
var addArrow = (a, b) => a + b;

console.log(add(2, 3)); // Output: 5
console.log(addArrow(2, 3)); // Output: 5
```

Lecture:

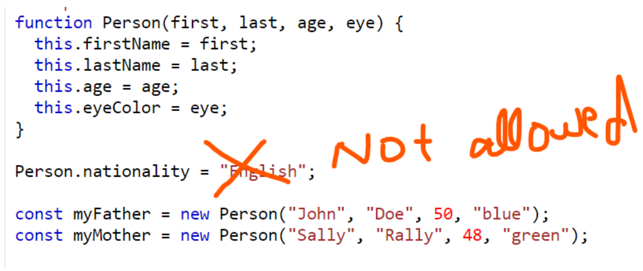
JavaScript prototype-based approach:

- There are two main ways to create classes in JavaScript:
  - the prototype-based approach
  - the class-based approach.
- All JavaScript objects inherit properties and methods from a prototype.
- In JavaScript, a prototype is a fundamental concept that is related to how objects inherit properties and methods from other objects.
- Every object in JavaScript has a prototype, which is essentially a reference to another object.



```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");
```

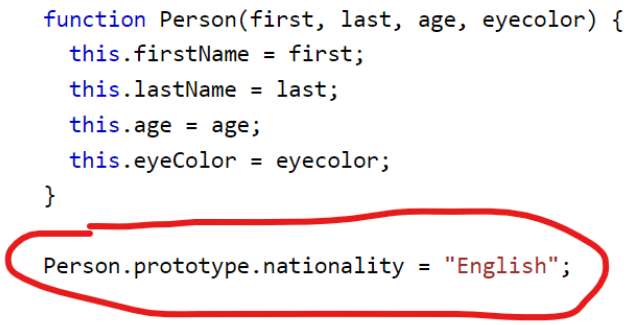
- We cannot add a new property to an existing object constructor:



```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

Person.nationality = "~~English~~";

const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");
```

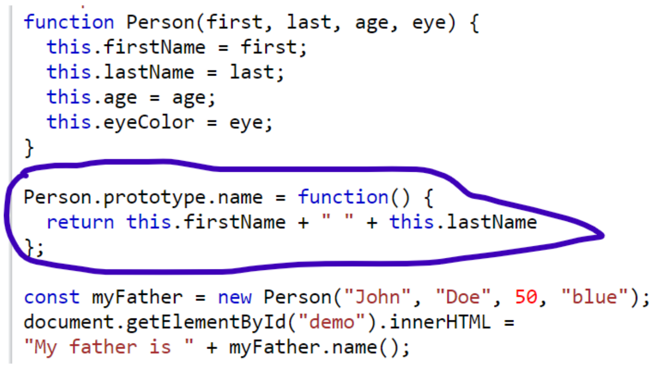
Not allowed!

- Using the **prototype** Property, we can add to the object constructor.
- The JavaScript prototype property allows you to add new properties to object constructors:



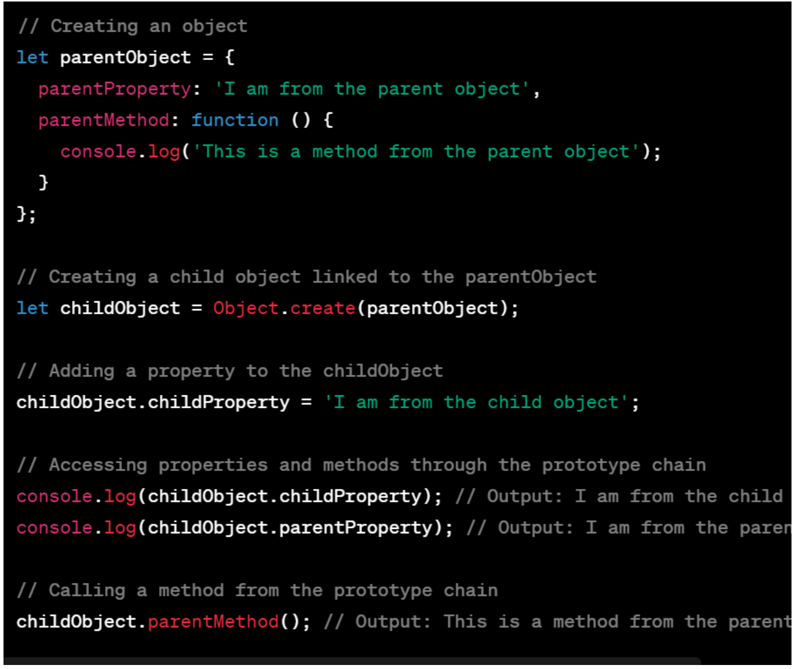
```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

Person.prototype.nationality = "English";
```

- The JavaScript prototype property also allows you to add new **methods** to objects constructors:



```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}
Person.prototype.name = function() {
  return this.firstName + " " + this.lastName
};

const myFather = new Person("John", "Doe", 50, "blue");
document.getElementById("demo").innerHTML =
"My father is " + myFather.name();
```

- ➢ In JavaScript, objects can be linked to other objects, forming a prototype chain.

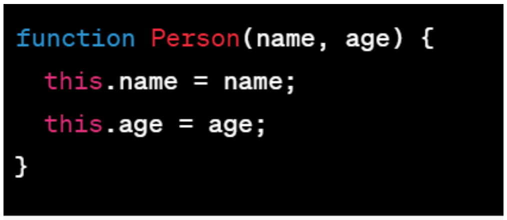


```
// Creating an object
let parentObject = {
  parentProperty: 'I am from the parent object',
  parentMethod: function () {
    console.log('This is a method from the parent object');
  }
};

// Creating a child object linked to the parentObject
let childObject = Object.create(parentObject);

// Adding a property to the childObject
childObject.childProperty = 'I am from the child object';

// Accessing properties and methods through the prototype chain
console.log(childObject.childProperty); // Output: I am from the child
console.log(childObject.parentProperty); // Output: I am from the paren
// Calling a method from the prototype chain
childObject.parentMethod(); // Output: This is a method from the parent
```

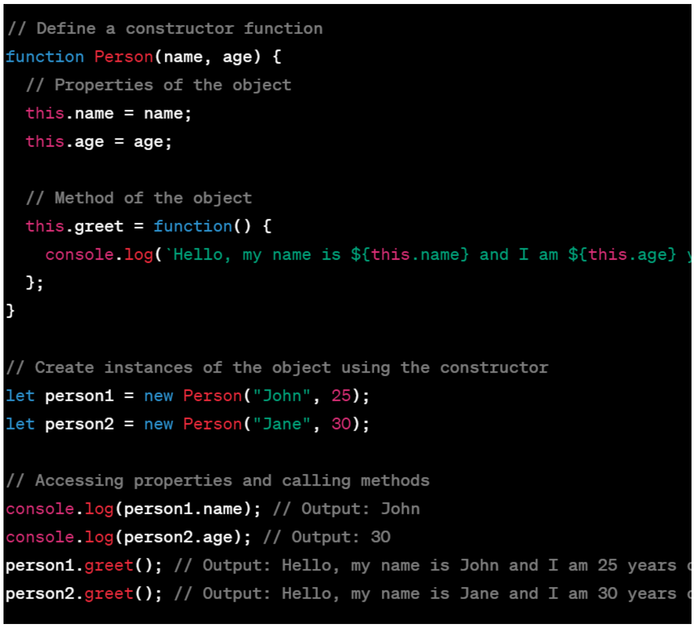
➢ More Explanation of Example

Constructor Functions:

- In JavaScript, we can create objects using constructor functions. These constructor functions are like blueprints for creating objects of a specific type.
- For example, you can create a constructor function for creating "Person" objects:



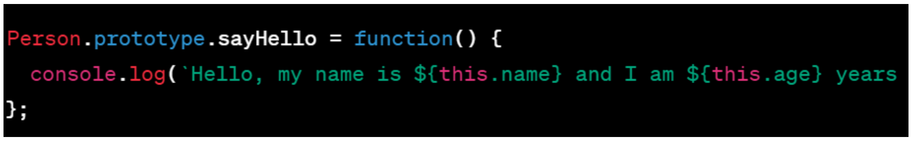
```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```



```javascript
// Define a constructor function
function Person(name, age) {
  // Properties of the object
  this.name = name;
  this.age = age;

  // Method of the object
  this.greet = function() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} y
  };
}

// Create instances of the object using the constructor
let person1 = new Person("John", 25);
let person2 = new Person("Jane", 30);

// Accessing properties and calling methods
console.log(person1.name); // Output: John
console.log(person2.age); // Output: 30
person1.greet(); // Output: Hello, my name is John and I am 25 years o
person2.greet(); // Output: Hello, my name is Jane and I am 30 years o
```

Prototype Object:

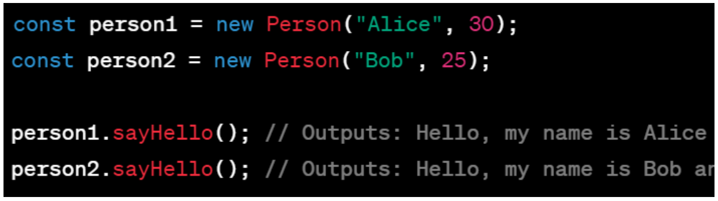
- Each constructor function has a **prototype** object associated with it. You can add methods and properties to this prototype object.



```javascript
Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name} and I am ${this.age} years
};
```

Inheritance:

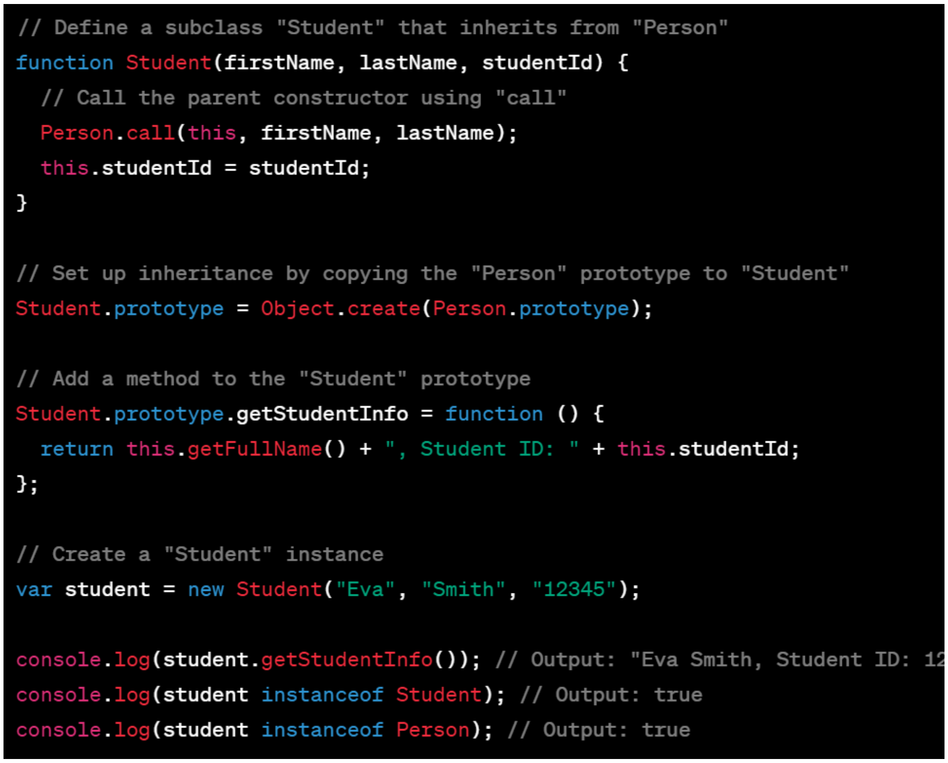
- When we create an instance of an object using the constructor function, that instance inherits the properties and methods from the constructor's prototype.
- For example:



```javascript
const person1 = new Person("Alice", 30);
const person2 = new Person("Bob", 25);

person1.sayHello(); // Outputs: Hello, my name is Alice
person2.sayHello(); // Outputs: Hello, my name is Bob ar
```

- **Creating Prototype-Based Classes**
- In JavaScript, we can use constructor functions and prototypes to define classes.
- JavaScript uses prototypes to achieve inheritance.

- We can also create subclasses. Also, we can inherit properties and methods from a parent class by extending the prototype chain:

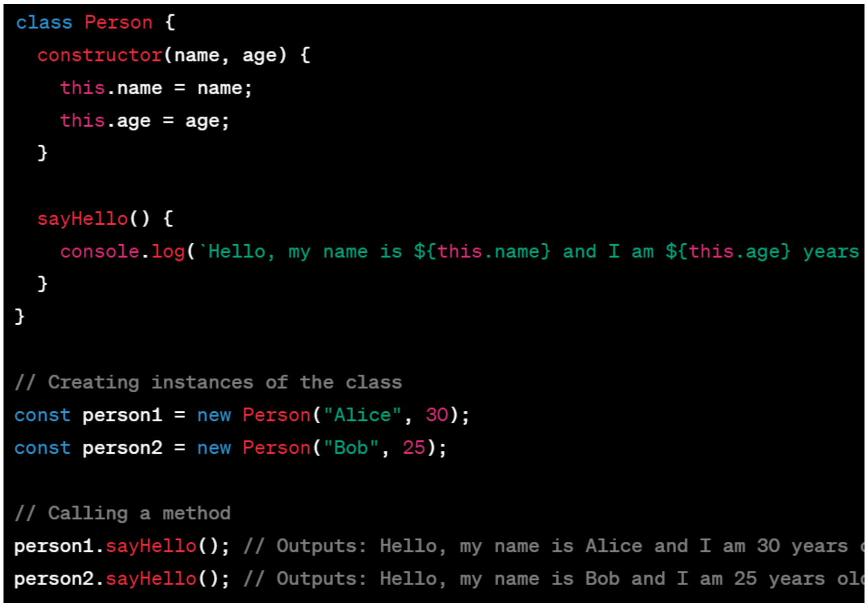


```
// Define a subclass "Student" that inherits from "Person"
function Student(firstName, lastName, studentId) {
  // Call the parent constructor using "call"
  Person.call(this, firstName, lastName);
  this.studentId = studentId;
}

// Set up inheritance by copying the "Person" prototype to "Student"
Student.prototype = Object.create(Person.prototype);

// Add a method to the "Student" prototype
Student.prototype.getStudentInfo = function () {
  return this.getFullName() + ", Student ID: " + this.studentId;
};

// Create a "Student" instance
var student = new Student("Eva", "Smith", "12345");

console.log(student.getStudentInfo()); // Output: "Eva Smith, Student ID: 12
console.log(student instanceof Student); // Output: true
console.log(student instanceof Person); // Output: true
```

- The Object.create method is used to set up the inheritance by creating a new object with the Person.prototype as its prototype.

ES6 Classes

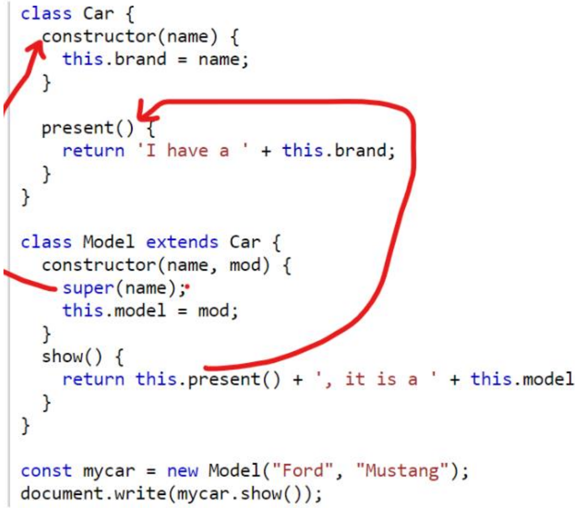
- ES6 (ECMAScript 2015) introduced a new way to create and work with classes in JavaScript.
- Prior to ES6, JavaScript used constructor functions and prototypes for object-oriented programming.
- A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.
- Example:



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

  sayHello() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years
  }
}

// Creating instances of the class
const person1 = new Person("Alice", 30);
const person2 = new Person("Bob", 25);

// Calling a method
person1.sayHello(); // Outputs: Hello, my name is Alice and I am 30 years
person2.sayHello(); // Outputs: Hello, my name is Bob and I am 25 years ol
```

➢ Class Inheritance

- To inherit a class, use the extends keyword.

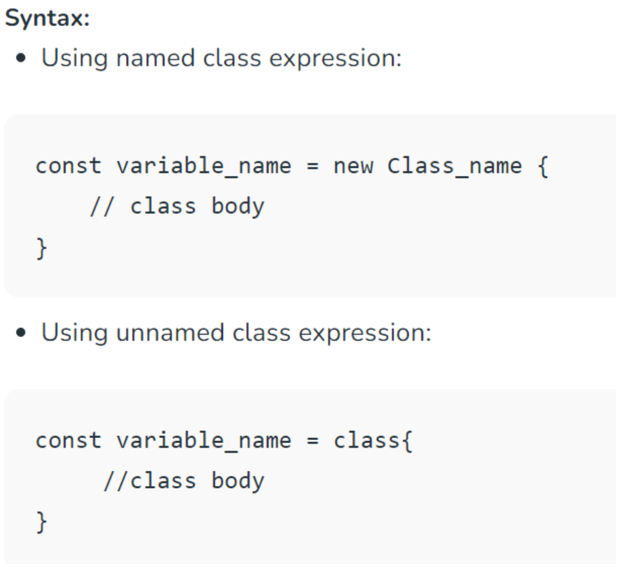


```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

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

const mycar = new Model("Ford", "Mustang");
document.write(mycar.show());
```

➢ Class Expression:

- A class expression in JavaScript is a way to define a class using an expression rather than a declaration.
- It creates anonymous classes that can be assigned to variables.

- Class expressions are like function expressions in that they give us more flexibility in defining and using classes in our code.



Syntax:

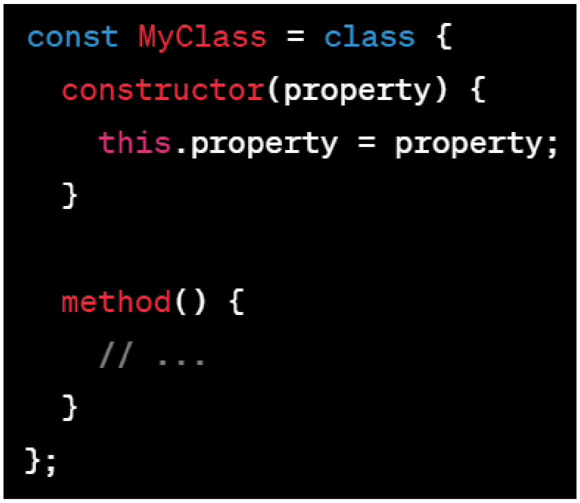
- Using named class expression:

```
const variable_name = new Class_name {
    // class body
}
```

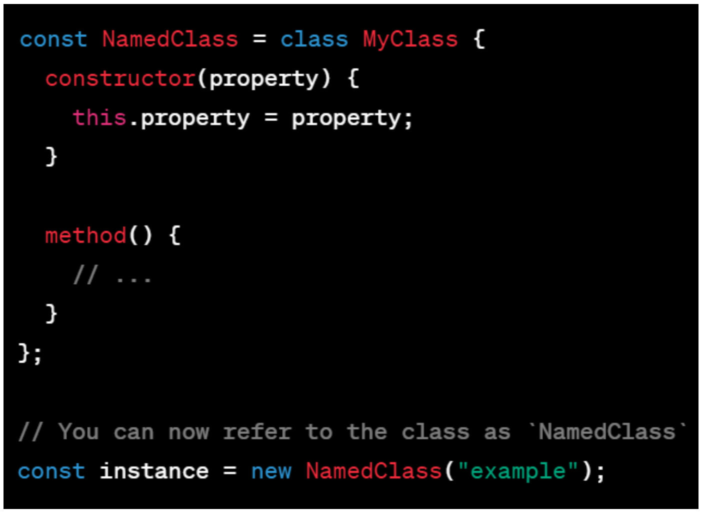
- Using unnamed class expression:

```
const variable_name = class{
     //class body
}
```

More Detailed Syntax

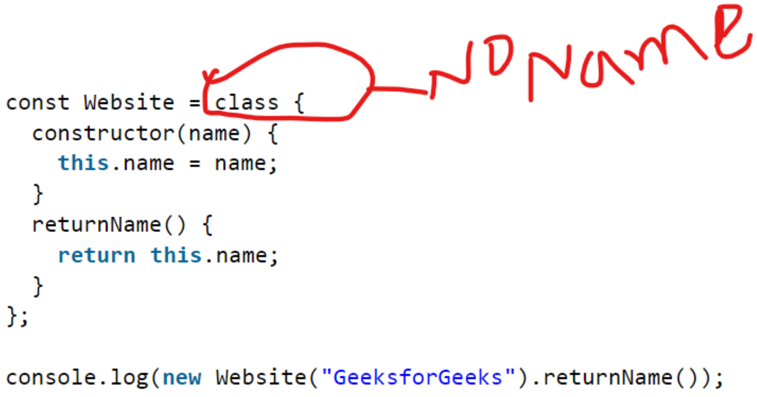


```
const MyClass = class {
  constructor(property) {
    this.property = property;
  }

  method() {
    // ...
  }
};
```



```
const NamedClass = class MyClass {
  constructor(property) {
    this.property = property;
  }

  method() {
    // ...
  }
};

// You can now refer to the class as `NamedClass`
const instance = new NamedClass("example");
```

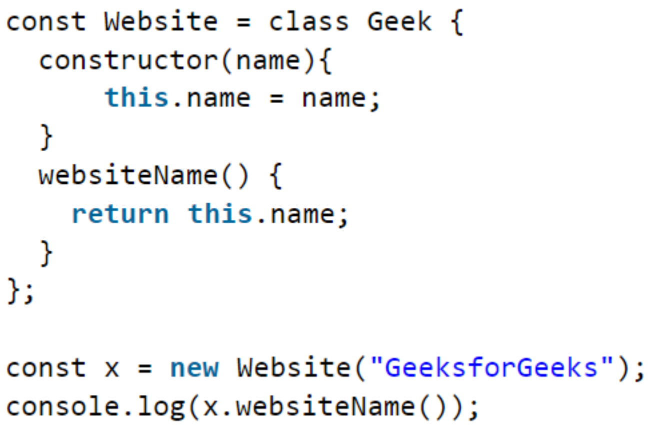
- **Example:**
  - **Without name (Anonymous Class)**



```
const Website = class {
  constructor(name) {
    this.name = name;
  }
  returnName() {
    return this.name;
  }
};

console.log(new Website("GeeksforGeeks").returnName());
```

NO Name

  - **With Geek name**



```
const Website = class Geek {
  constructor(name){
      this.name = name;
  }
  websiteName() {
    return this.name;
  }
};

const x = new Website("GeeksforGeeks");
console.log(x.websiteName());
```

Installation of React

First check node version through cmd. Node -v

Install node.js if not installed

Npm create vite@latest

Give project name

Then go to project folder and install third party deficiencies npm i or(install)

Npm run dev.