Aaron McCarley
Language Map for JavaScript
CS311

Variable Declaration
Is this language strongly typed or dynamically typed? Provide at least three examples (with different data types or keywords) of how variables are declared in this language.
	JavaScript is a dynamically typed language. All typed checks are performed when the code is ran/runtime execution. There are 4 ways to declare a JavaScript Variable. Using var, let, const, or using nothing
	
	Variables are containers for storing data (storing data values). An example would be using a, b, and c, as the variables, which are declared with the var keyword:
	
	var a = 5;
	var b = 6;
	var c = a + b;
	
	An example would be using a, b, and c, as the variables, which are declared with the let keyword:
	
	let a = 5;
	let b = 6;
	let c = a + b;
	
	An example for an declarable variable using example, a, b, and c.
	
	a = 5;
	b = 6;
	c = a + b;
	
	For a general rule: always declare variables with const. If the variable is going to change the use the let variable. In this example, price1, price2, and total, are variables:

	const price1 = 5;
	const price2 = 6;
	let total = price1 + price2;

Data Types List all of the data types (and ranges) supported by this language.	The concept of a datatype is to be able to operate on variables, it is important to know something about the type. Without data types, a computer cannot safely solve this. Example of adding a number and a string let x = 16 + "Volvo"; When adding a number and a string, JavaScript will treat the number as a string. JavaScript evaluates expressions from left to right. Different sequences can produce different results. <!DOCTYPE html> <html> <body> JavaScript

	When adding a number and a string, JavaScript will treat the number as a string.
	
	<script>
	
	let x = 16 + "Volvo";
	
	document.getElementById("demo").innerHTML = x;
	
	</script>
	
	</body>
	
	</html>
	
	Result: When adding a string and a number, JavaScript will treat the number as a string.
	
	Volvo16
	
	JavaScript has dynamic types. This means that the same variable can be used to hold different data types:
	
	let x; // Now x is undefined
	x = 5; // Now x is a Number
	x = "John"; // Now x is a String

	Strings are written with quotes. You can use single or double quotes:
	
	let carName1 = "Volvo XC60"; // Using double quotes
	let carName2 = 'Volvo XC60'; // Using single quotes
	
	You can use quotes inside a string, as long as they don't match the quotes surrounding the string:
	
	let answer1 = "It's alright"; // Single quote inside double quotes
	let answer2 = "He is called 'Johnny'"; // Single quotes inside double quotes
	let answer3 = 'He is called "Johnny"'; // Double quotes inside single quotes
	
	JavaScript has only one type of numbers. Numbers can be written with, or without decimals:
	
	let x1 = 34.00; // Written with decimals
	let x2 = 34; // Written without decimals
	
	Extra large or extra small numbers can be written with scientific (exponential) notation:
	
	let y = 123e5; // 12300000
	let z = 123e-5; // 0.00123
	
	An empty value has nothing to do with undefined.
	
	An empty string has both a legal value and a type.
	
	let car = ""; // The value is "", the typeof is "string"

Selection Structures Provide examples of all selection structures supported by this language (if, if else, etc.) **Don't just list them, show code samples of how each would look in a real program.**	**JavaScript provides two major constructs for making selections in code. The first of these constructs is the if statement. The other construct is the conditional operator.** **The Relational Operators** Before I can talk about selection structures, I need to cover how JavaScript programs compare things. Comparisons in JavaScript are done using the relational operators. The relational operators compare two values for different relations. The relational operators are: • `>` Greater than • `>=` Greater than or equal to • `<` Less than • `<=` Less than or equal to • `!=` Not equal to • `!==` Strict not equal to • `==` Equal to • `===` Strict equal to

	The two relational operators that may need some explanation are the strict operators, === and !==. These operators are necessary due to some quirks in the design of JavaScript. I can best explain these quirks with a demonstration using the shell:The Simple if Statement
	
	js> let number = 100;
	js> let strNumber = "100";
	js> number == strNumber
	true
	
	The first type of if statement in JavaScript is the simple if. In this form, a relational expression is tested and if the expression evaluates to true, a statement or set of statements are executed.
	
	JavaScript evaluates the number variable as equal to the string variable because they have the same surface value, although they are not really equal since they cannot be used interchangeably. If we compare the two variables using the strict equality operator, though, we get a different result:
	
	Here is the syntax template for the simple if:
	if (relational expression) {
	statement(s);
	}
	let grade = 0;
	putstr("Enter your grade: ");
	grade = parseInt(readline());
	if (grade >= 70) {

	print("You passed!");
	}
	Here are two runs of this program:
	Enter your grade: 78
	You passed!
	Enter your grade: 65
	The if-else Statement
	This form of the if statement allows a set of statements to execute if the relational expression is false, so it is much more useful than the simple if statement. Here is the syntax template for if-else:
	
	if (relational expression) {
	statement(s);
	}
	else {
	statement(s);
	}
	
	Now we can rewrite the example above so that the user is given a message if they don't score a passing grade:
	
	let grade = 0;
	putstr("Enter your grade: ");
	grade = parseInt(readline());
	if (grade >= 70) {
	print("You passed!");
	}
	else {
	print("Sorry. You did not pass.");
	}
	Here are two runs of this program:
	
	C:\js>js test.js
	Enter your grade: 78
	You passed!
	C:\js>js test.js

	Enter your grade: 68
	Sorry. You did not pass.
	
	The if-else if Statement
	The if-else if statement allows a long chain of relational expressions to be tied together to form one block of logic. A classic example of this type of decision is assigning a letter grade to a numeric score where the range of possible letter grades and the range of scores is:
	
	90–100 A
	80–89 B
	70–79 C
	60–69 D
	0–59 F
	
	Here is the syntax template for the if-else if statement:
	
	if (relational expression) {
	statement(s);
	}
	else if (relational expression) {
	statement(s);
	}
	else if (relational expression) {
	statement(s);
	}
	…
	else {
	statement(s);
	}
	
	The ellipses indicate that there can be more else if blocks in the statement.
	
	Every relational expression after the first one is optional, as is the else at the end of the statement.

	Let's see how the if-else if can be used to solve the letter grade problem I introduced above: let letterGrade = ""; let grade = 0; putstr("Enter the grade: "); grade = parseInt(readline()); if (grade >= 90) { letterGrade = "A"; } else if (grade >= 80) { letterGrade = "B"; } else if (grade >= 70) { letterGrade = "C"; } else if (grade >= 60) { letterGrade = "D"; } else if (grade >= 0) { letterGrade = "F"; } print(`A numeric grade of ${grade} earns a letter grade of   ${letterGrade}.`); There is a shortcut method of writing an if-else statement called the conditional operator. This operator is made up of two symbols $:. Here is the syntax template for the conditional operator: (relational expression) ? statement-to-evaluate-if-true : statement-to-evaluate-if-false; Here is the pass/fail program written using the conditional operator: let grade = 0; putstr("Enter the grade: "); grade = parseInt(readline());

	(grade >= 70) ? print("You passed!") : print("Sorry. You didn't pass.");

Repetition Structures
Provide examples of all repetition structures supported by this language (loops, etc.)
Don't just list them, show code samples of how each would look in a real program.
	There are three repetition structures in JavaScript, **the while loop, the for loop, and the do-while loop**.
	
	A repetition structure allows the programmer to specify that an action is to be repeated while some condition remains true.
	
	Use a for loop for counter-controlled repetition. Use a while or do-while loop for eventcontrolled repetition. Use a do-while loop when the loop must execute at least one time. Use a while loop when it is possible that the loop may never execute.
	
	The while Repetition Structure
	
	while (condition) {
	statement(s)
	}
	
	The braces are not required if the loop body contains only a single statement. However, they are a good idea and are required by the 104 Coding Standards.
	
	while (children > 0)
	{
	children = children - 1 ;
	cookies = cookies * 2 ;
	}
	
	Example finished code using a while loop.
	
	/* Get grades until user enters -1. Compute
	grade total and grade count */
	while (grade != -1)
	{
	total = total + grade;

	counter = counter + 1;
	grade = prompt("Enter another grade: ");
	
	grade = parseInt(grade);
	}
	/* Compute and display the average grade */
	average = total / counter;
	alert("Class average is: " + average + ".");
	
	var number;
	number = prompt("Enter a positive number: ");
	number = parseFloat(number);
	while (number <= 0)
	{
	alert("That's incorrect. Try again.\n");
	number = prompt("Enter a positive number: ");
	number = parseFloat(number);
	}
	alert ("You entered: " + number);
	
	The **for loop** handles details of the counter-controlled loop "automatically".
	The initialization of the loop control variable, the termination condition test, and control variable modification are handled in the for loop structure.
	
	Just as with a while loop, a for loop
	initializes the loop control variable before beginning the first loop iteration,
	modifies the loop control variable at the very end of each iteration of the loop, and
	performs the loop termination test before each iteration of the loop.
	The for loop is easier to write and read for counter-controlled loops.
	
	for(i = 0; i < 10; i = i + 1)
	{
	alert("i is " + i);
	}

	Loops may be nested (embedded) inside of each other. Any control structure (sequence, selection, or repetition) may be nested inside of any other control structure. It is common to see nested for loops.
	
	var i;
	for(i = 1; i < 10; i = i + 1)
	{ if
	(i == 5)
	{
	break;
	}
	document.write(i + " ");
	}
	document.write("Broke out of loop at i = " + i);
	
	the Do While Loop
	
	do {
	num = prompt("Enter a positive number: ");
	num = parseInt(num);
	if (num <= 0) { alert("That is not positive. Try again.");
	}
	}
	while (num <= 0);

Arrays If this language supports arrays, provide at least two examples of creating an array with a primitive or String data types (e.g. float, int, String, etc.)	Yes, JavaScript is supported by Arrays, and an example could be used if you wanted to loop through the cars and find a specific one. And what if you had more than 3 cars, say 300, the solution to describe how to package this data could be an array. An array can hold many values under a single name, and you can access the values by referring to an index number. If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

	let car1 = "Saab";
	let car2 = "Volvo";
	let car3 = "BMW";
	
	Using an array literal is the easiest way to create a JavaScript Array.
	
	const array_name = [item1, item2, ...];
	
	const cars = ["Saab", "Volvo", "BMW"];
	
	You can also create an array, and then provide the elements:
	
	const cars = [];
	cars[0]= "Saab";
	cars[1]= "Volvo";
	cars[2]= "BMW";
	
	The following example also creates an Array, and assigns values to it:
	
	const cars = new Array("Saab", "Volvo", "BMW");
	
	You access an array element by referring to the **index number**:
	
	const cars = ["Saab", "Volvo", "BMW"];
	let car = cars[0];
	
	This statement changes the value of the first element in cars:
	
	cars[0] = "Opel";

Data Structures If this language provides a standard set of data structures, provide a list of the data structures and their Big-Oh complexity.	A data structure in JavaScript is **a format to organize, manage and store data** in a way that allows **efficient access and modification**. More precisely, a data structure is a **collection of data values**, the **relationships** among them, and the functions or **operations** that can be applied to that data.

	// A collection of the values 1, 2 and 3
	const arr = [1, 2, 3]
	
	// Each value is related to one another, in the sense that each is indexed in a position of the array
	const indexOfTwo = arr.indexOf(2)
	console.log(arr[indexOfTwo-1]) // 1
	console.log(arr[indexOfTwo+1]) // 3
	
	// We can perform many operations on the array, like pushing new values into it
	arr.push(4)
	console.log(arr) // [1,2,3,4]
	
	JavaScript has **primitive (built in)** and **non-primitive (not built in)** data structures.
	
	Primitive data structures come by default with the programming language and you can implement them out of the box (like arrays and objects). Non-primitive data structures don't come by default and you have to code them up if you want to use them.
	
	Each item can be accessed through its **index** (position) number. Arrays always start at index 0, so in an array of 4 elements we could access the 3rd element using the index number 2.
	The **length** property of an array is defined as the number of elements it contains. If the array contains 4 elements, we can say the array has a length of 4.
	
	In some programming languages, the user can only store values of the same type in one array and the length of the array has to be defined at the moment of its creation and can't be modified afterwards.

	In JavaScript that's not the case, as we can store **values of any type** in the same array and the **length** of it can be **dynamic** (it can grow or shrink as much as necessary).
	
	Any data type can be stored in an array, and that includes arrays too. An array that has other arrays within itself is called a **multidimensional array**.
	
	const arr = [
	[1,2,3],
	[4,5,6],
	[7,8,9],
	]
	
	In JavaScript, an **object** is a collection of **key-value pairs**. This data structure is also called **map**, **dictionary** or **hash-table** in other programming languages.
	A typical JS object looks like this:
	
	const obj = {
	prop1: "I'm",
	prop2: "an",
	prop3: "object"
	}
	
	To access properties you can use two different syntaxes,
	either object.property or object["property"]. To access methods we call object.method().
	
	console.log(obj.prop1) // "Hello!"
	console.log(obj["prop1"]) // "Hello!"
	obj.prop3() // "I'm a property dude!"

	The syntax to assign new values is quite similar:
	
	obj.prop4 = 125
	obj["prop5"] = "The new prop on the block"
	obj.prop6 = () => console.log("yet another example")
	
	console.log(obj.prop4) // 125
	console.log(obj["prop5"]) // "The new prop on the block"
	obj.prop6() // "yet another example"
	
	Like arrays, in JavaScript objects come with many built-in methods that allow us to perform different operations and get information from a given object. A full list can be found here.
	Objects are a good way to group together data that have something in common or are somehow related. Also, thanks to the fact that property names are unique, objects come in handy when we have to separate data based on a unique condition.
	
	An example could be counting how many people like different foods:
	
	const obj = {
	pizzaLovers: 1000,
	pastaLovers: 750,
	argentinianAsadoLovers: 12312312312313123
	}
	
	Stacks are a data structure that store information in the form of a list. They allow only adding and removing elements under a **LIFO pattern (last in, first out)**. In stacks, elements can't be added or removed out of order, they always have to follow the LIFO pattern.

	Stacks are useful when we need to make sure elements follow the **LIFO pattern**. Some examples of stack usage are:
	• JavaScript's call stack.
	• Managing function invocations in various programming languages.
	• The undo/redo functionality many programs offer.
	
	// We create a class for each node within the stack
	class Node {
	// Each node has two properties, its value and a pointer that indicates the node that follows
	constructor(value){
	this.value = value
	this.next = null
	}
	}
	
	// We create a class for the stack
	class Stack {
	// The stack has three properties, the first node, the last node and the stack size
	constructor(){
	this.first = null
	this.last = null
	this.size = 0
	}
	// The push method receives a value and adds it to the "top" of the stack

```
    push(val){
        var newNode = new Node(val)
        if(!this.first){
            this.first = newNode
            this.last = newNode
        } else {
            var temp = this.first
            this.first = newNode
            this.first.next = temp
        }
        return ++this.size
    }
    // The pop method eliminates the element at the "top" of the stack and returns its
value
    pop(){
        if(!this.first) return null
        var temp = this.first
        if(this.first === this.last){
            this.last = null
        }
        this.first = this.first.next
        this.size--
        return temp.value
    }
}
```

```
const stck = new Stack

stck.push("value1")
stck.push("value2")
stck.push("value3")

console.log(stck.first) /*
        Node {
        value: 'value3',
        next: Node { value: 'value2', next: Node { value: 'value1', next: null } }
        }
    */
console.log(stck.last) // Node { value: 'value1', next: null }
console.log(stck.size) // 3

stck.push("value4")
console.log(stck.pop()) // value4




stck.push("value3")

console.log(stck.first) /*
        Node {
```

```
    value: 'value3',
    next: Node { value: 'value2', next: Node { value: 'value1', next: null } }
    }
  */
console.log(stck.last) // Node { value: 'value1', next: null }
console.log(stck.size) // 3

stck.push("value4")
console.log(stck.pop()) // value4
```

The big O of stack methods is the following:

- Insertion - O(1)
- Removal - O(1)
- Searching - O(n)
- Access - O(n)

Queues

Queues work in a very similar way to stacks, but elements follow a different pattern for add and removal. Queues allow only a **FIFO pattern (first in, first out)**. In queues, elements can't be added or removed out of order, they always have to follow the FIFO pattern.

```
// We create a class for each node within the queue
class Node {
  // Each node has two properties, its value and a pointer that indicates the node that follows
```

```
    constructor(value){
        this.value = value
        this.next = null
    }
}

// We create a class for the queue
class Queue {
    // The queue has three properties, the first node, the last node and the stack size
    constructor(){
        this.first = null
        this.last = null
        this.size = 0
    }
    // The enqueue method receives a value and adds it to the "end" of the queue
    enqueue(val){
        var newNode = new Node(val)
        if(!this.first){
            this.first = newNode
            this.last = newNode
        } else {
            this.last.next = newNode
            this.last = newNode
        }
        return ++this.size
```

```
    }
    // The dequeue method eliminates the element at the "beginning" of the queue and
returns its value
    dequeue(){
        if(!this.first) return null

        var temp = this.first
        if(this.first === this.last) {
            this.last = null
        }
        this.first = this.first.next
        this.size--
        return temp.value
    }
}

const quickQueue = new Queue

quickQueue.enqueue("value1")
quickQueue.enqueue("value2")
quickQueue.enqueue("value3")

console.log(quickQueue.first) /*
        Node {
            value: 'value1',
```

	next: Node { value: 'value2', next: Node { value: 'value3', next: null } }
	}
	*/
	console.log(quickQueue.last) // Node { value: 'value3, next: null }
	console.log(quickQueue.size) // 3
	
	quickQueue.enqueue("value4")
	console.log(quickQueue.dequeue()) // value1
	
	Linked lists are a type of data structure that store values in the form of a **list**. Within the list, each value is considered a **node**, and each node is connected with the following value in the list (or null in case the element is the last in the list) through a **pointer**. There are two kinds of linked lists, **singly linked lists** and **doubly linked lists.** Both work very similarly, but the difference is in singly linked lists each node has a **single pointer** that indicates the **next node** on the list. While in doubly linked lists, each node has **two pointers**, one pointing to the **next node** and another pointing to the **previous node**.
	
	// We create a class for each node within the list
	class Node{
	// Each node has two properties, its value and a pointer that indicates the node that follows
	constructor(val){
	this.val = val
	this.next = null
	}
	}

	// We create a class for the list
	class SinglyLinkedList{
	  // The list has three properties, the head, the tail and the list size
	  constructor(){
	  this.head = null
	  this.tail = null
	  this.length = 0
	  }
	  // The push method takes a value as parameter and assigns it as the tail of the list
	  push(val) {
	  const newNode = new Node(val)
	  if (!this.head){
	  this.head = newNode
	  this.tail = this.head
	  } else {
	  this.tail.next = newNode
	  this.tail = newNode
	  }
	  this.length++
	  return this
	  }
	  // The pop method removes the tail of the list
	  pop() {
	  if (!this.head) return undefined
	  const current = this.head

```
    const newTail = current
    while (current.next) {
        newTail = current
        current = current.next
    }
    this.tail = newTail
    this.tail.next = null
    this.length--
    if (this.length === 0) {
        this.head = null
        this.tail = null
    }
    return current
}
// The shift method removes the head of the list
shift() {
    if (!this.head) return undefined
    var currentHead = this.head
    this.head = currentHead.next
    this.length--
    if (this.length === 0) {
        this.tail = null
    }
    return currentHead
}
```

	// The unshift method takes a value as parameter and assigns it as the head of the list
	unshift(val) {
	const newNode = new Node(val)
	if (!this.head) {
	this.head = newNode
	this.tail = this.head
	}
	newNode.next = this.head
	this.head = newNode
	this.length++
	return this
	}
	// The get method takes an index number as parameter and returns the value of the node at that index
	get(index) {
	if(index < 0 || index >= this.length) return null
	const counter = 0
	const current = this.head
	while(counter !== index) {
	current = current.next
	counter++
	}
	return current
	}

```
    // The set method takes an index number and a value as parameters, and modifies the node value at the given index in the list
    set(index, val) {
        const foundNode = this.get(index)
        if (foundNode) {
            foundNode.val = val
            return true
        }
        return false
    }
    // The insert method takes an index number and a value as parameters, and inserts the value at the given index in the list
    insert(index, val) {
        if (index < 0 || index > this.length) return false
        if (index === this.length) return !!this.push(val)
        if (index === 0) return !!this.unshift(val)

        const newNode = new Node(val)
        const prev = this.get(index - 1)
        const temp = prev.next
        prev.next = newNode
        newNode.next = temp
        this.length++
        return true
    }
```

	// The remove method takes an index number as parameter and removes the node at the given index in the list
	remove(index) {
	if(index < 0 || index >= this.length) return undefined
	if(index === 0) return this.shift()
	if(index === this.length - 1) return this.pop()
	const previousNode = this.get(index - 1)
	const removed = previousNode.next
	previousNode.next = removed.next
	this.length--
	return removed
	}
	// The reverse method reverses the list and all pointers so that the head becomes the tail and the tail becomes the head
	reverse(){
	const node = this.head
	this.head = this.tail
	this.tail = node
	let next
	const prev = null
	for(let i = 0; i < this.length; i++) {
	next = node.next
	node.next = prev
	prev = node
	node = next

```
    }
    return this
  }
}
```

Trees are a data structures that link nodes in a **parent/child relationship**, in the sense that there're nodes that depend on or come off other nodes.

Trees are formed by a **root** node (the first node on the tree), and all the nodes that come off that root are called **children**. The nodes at the bottom of the tree, which have no "descendants", are called **leaf nodes**. And the **height** of the tree is determined by the number of parent/child connections it has.

Unlike linked lists or arrays, trees are **non linear**, in the sense that when iterating the tree, the program flow can follow different directions within the data structure and hence arrive at different values.

An example of a binary search tree

```
// We create a class for each node within the tree
class Node{
  // Each node has three properties, its value, a pointer that indicates the node to its left and a pointer that indicates the node to its right
  constructor(value){
    this.value = value
    this.left = null
    this.right = null
  }
}
```

```
// We create a class for the BST
class BinarySearchTree {
    // The tree has only one property which is its root node
    constructor(){
        this.root = null
    }
    // The insert method takes a value as parameter and inserts the value in its
corresponding place within the tree
    insert(value){
        const newNode = new Node(value)
        if(this.root === null){
            this.root = newNode
            return this
        }
        let current = this.root
        while(true){
            if(value === current.value) return undefined
            if(value < current.value){
                if(current.left === null){
                    current.left = newNode
                    return this
                }
                current = current.left
            } else {
                if(current.right === null){
```

```
                current.right = newNode
                return this
            }
            current = current.right
        }
    }
}
// The find method takes a value as parameter and iterates through the tree looking for that value
// If the value is found, it returns the corresponding node and if it's not, it returns undefined
find(value){
    if(this.root === null) return false
    let current = this.root,
        found = false
    while(current && !found){
        if(value < current.value){
            current = current.left
        } else if(value > current.value){
            current = current.right
        } else {
            found = true
        }
    }
    if(!found) return undefined
```

```
        return current
    }
    // The contains method takes a value as parameter and returns true if the value is found within the tree
    contains(value){
        if(this.root === null) return false
        let current = this.root,
            found = false
        while(current && !found){
            if(value < current.value){
                current = current.left
            } else if(value > current.value){
                current = current.right
            } else {
                return true
            }
        }
        return false
    }
}
```

Heaps are another type of tree that have some particular rules. There are two main types of heaps, **MaxHeaps** and **MinHeaps**. In MaxHeaps, parent nodes are always greater than its children, and in MinHeaps, parent nodes are always smaller than its children.

	Graphs are a data structure formed by a group of nodes and certain connections between those nodes. Unlike trees, graphs don't have root and leaf nodes, nor a "head" or a "tail". Different nodes are connected to each other and there's no implicit parent-child connection between them. We say a graph is undirected if there's no implicit direction in the connections between nodes. If we take the following example image, you can see that there's no direction in the connection between node 2 and node 3. The connection goes both ways, meaning you can traverse the data structure from node 2 to node 3, and from node 3 to node 2. Undirected means the connections between nodes can be used both ways. We say a graph is weighted if the connections between nodes have an assigned weight. In this case, weight just means a value that is assigned to a specific connection. It's information about the connection itself, not about the nodes. Following this example, we can see the connection between nodes 0 and 4, has a weight of 7. And the connection between nodes 3 and 1 has a weight of 4. When coding graphs, there're two main methods we can use: an **adjacency matrix** and an **adjacency list**. Let's explain how both work and see their pros and cons. An **adjacency matrix is a two dimensional structure** that represents the nodes in our graph and the connections between them. [[0, 1, 1, 0] [1, 0, 0, 1]

	[1, 0, 0, 1]
	[0, 1, 1, 0]
	]
	On the other hand, an **adjacency list** can be thought as **a key-value pair structure** where **keys represent each node** on our graph and t**he values are the connections** that that particular node has.
	Using the same example graph, our adjacency list could be represented with this object:
	
	{
	A: ["B", "C"],
	B: ["A", "D"],
	C: ["A", "D"],
	D: ["B", "C"],
	}
	
	While to do the same in a list, adding a value to B connections and a key-value pair to represent E is enough:
	
	{
	A: ["B", "C"],
	B: ["A", "D", "E"],
	C: ["A", "D"],
	D: ["B", "C"],
	E: ["B"],
	}

	A full implementation of a graph using an adjacency list:
	
	// We create a class for the graph
	class Graph{
	  // The graph has only one property which is the adjacency list
	  constructor() {
	  this.adjacencyList = {}
	  }
	  // The addNode method takes a node value as parameter and adds it as a key to the adjacencyList if it wasn't previously present
	  addNode(node) {
	  if (!this.adjacencyList[node]) this.adjacencyList[node] = []
	  }
	  // The addConnection takes two nodes as parameters, and it adds each node to the other's array of connections.
	  addConnection(node1,node2) {
	  this.adjacencyList[node1].push(node2)
	  this.adjacencyList[node2].push(node1)
	  }
	  // The removeConnection takes two nodes as parameters, and it removes each node from the other's array of connections.
	  removeConnection(node1,node2) {
	  this.adjacencyList[node1] = this.adjacencyList[node1].filter(v => v !== node2)
	  this.adjacencyList[node2] = this.adjacencyList[node2].filter(v => v !== node1)

```
    }
    // The removeNode method takes a node value as parameter. It removes all
connections to that node present in the graph and then deletes the node key from the
adj list.
    removeNode(node){
        while(this.adjacencyList[node].length) {
            const adjacentNode = this.adjacencyList[node].pop()
            this.removeConnection(node, adjacentNode)
        }
        delete this.adjacencyList[node]
    }
}

const Argentina = new Graph()
Argentina.addNode("Buenos Aires")
Argentina.addNode("Santa fe")
Argentina.addNode("Córdoba")
Argentina.addNode("Mendoza")
Argentina.addConnection("Buenos Aires", "Córdoba")
Argentina.addConnection("Buenos Aires", "Mendoza")
Argentina.addConnection("Santa fe", "Córdoba")

console.log(Argentina)
// Graph {
//     adjacencyList: {
```

	// 'Buenos Aires': ['Córdoba', 'Mendoza'], // 'Santa fe': ['Córdoba'], // 'Córdoba': ['Buenos Aires', 'Santa fe'], // Mendoza: ['Buenos Aires'] // } // }
Objects If this language supports object-orientation, provide an example of how you would write a simple object with a default constructor and then how you would instantiate it.	In real life, a car is an **object**. A car has **properties** like weight and color, and **methods** like start and stop: The property could be represented: car.name = Fiat car.model = 500 car.weight = 850kg car.color = white The method could be represented: car.start() car.drive() car.brake() car.stop() All cars have the same **properties**, but the property **values** differ from car to car. All cars have the same **methods**, but the methods are performed **at different times**. You have already learned that JavaScript variables are containers for data values.

	This code assigns a **simple value** (Fiat) to a **variable** named car: let car = "Fiat"; Objects are variables too. But objects can contain many values. This code assigns **many values** (Fiat, 500, white) to a **variable** named car: const car = {type:"Fiat", model:"500", color:"white"}; The values are written as **name:value** pairs (name and value separated by a colon). You define (and create) a JavaScript object with an object literal: const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}; Objects can also have **methods**. Methods are **actions** that can be performed on objects. Methods are stored in properties as **function definitions**. Example: **Properties** firstName LastName age eyeColor

	fullName
	
	Property Value
	
	John
	
	Doe
	
	50
	
	Blue
	
	function() {return this.firstName + " " + this.lastName;}
	
	Example:
	
	const person = {
	  firstName: "John",
	  lastName : "Doe",
	  id : 5566,
	  fullName : function() {
	  return this.firstName + " " + this.lastName;
	  }
	};
	
	In the example above, this refers to the **person object**.
	
	I.E. **this.firstName** means the **firstName** property of **this**.

	I.E. **this.firstName** means the **firstName** property of **person**. In JavaScript, the this keyword refers to an **object**. **Which** object depends on how this is being invoked (used or called). The this keyword refers to different objects depending on how it is used: In an object method, this refers to the **object**. Alone, this refers to the global object. In a function, this refers to the **global object**. Ina function, in strict mode, this is undefined. In an event, this refers to the **element** that received the event. **Methods like call(), apply(), and bind() can refer this to any object.** You access an object method with the following syntax: objectName.methodName() Example: name = person.fullName(); When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

	x = new String(); // Declares x as a String object
	y = new Number(); // Declares y as a Number object
	z = new Boolean(); // Declares z as a Boolean object
	
	Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

Runtime Environment What runtime environment does this language compile to? For example, Java compiles to the Java Virtual Machine. Do other languages also compile to this runtime?	JavaScript is an **interpreted programming language**. It means that source code isn't compiled into binary code prior to execution. **A JavaScript engine is a program responsible for translating source code into machine code and executing the translation result on a computer's central processing unit (CPU).** All JavaScript engines implement **specification of the language** provide by ECMAScript. Standardization facilitates the development of independent engines and ensures your scripts give the same results no matter where you run them.

	The JavaScript runtime environment provides access to built-in libraries and objects that are available to a program so that it can interact with the outside world and make the code work. In the context of a browser this is comprised of the following elements: 1. The JavaScript engine (which in turn is made up of the heap and the call stack) 2. Web APIs 3. The callback queue 4. The event loop
Libraries/Frameworks What are the popular libraries or frameworks used by programmers for this language? List at least three (3) and describe what they are used for..	JavaScript libraries contain various functions, methods, or objects to perform practical tasks on a webpage or JS-based application. You can even build a WordPress site with them. Think of them as a book library where you revisit to read your favorite books. You may be an author and enjoy other authors' books, get a new perspective or idea, and utilize the same in your life.

	Similarly, a JavaScript library has codes or functions that developers can reuse and repurpose. A developer writes these codes, and other developers reuse the same code to perform a certain task, like preparing a slideshow, instead of writing it from scratch. It saves them significant time and effort. They are precisely the motive behind creating JavaScript libraries, which is why you can find dozens of them for multiple use cases. They not only save you time but also bring simplicity to the entire development process. To use a JavaScript library in your app, add `<script>` to the `<head>` element using the `src` attribute that references the library source path or the URL. Read the JavaScript library's documentation you intend to use for more information and follow the steps provided there. As we've said, JavaScript libraries are used to perform specific functions. There are around 83 of them, each created to serve some purpose, and we are going to cover some of their usability in this section. You can use JavaScript libraries for: • Data Visualization in Maps and Charts Data visualization in applications is crucial for users to view the statistics clearly in the admin panel, dashboards, performance metrics, and more. Presenting these data in charts and maps helps you analyze that data easily and make informed business decisions. Examples: Chart.js, Apexcharts, Algolia Places

	• DOM Manipulation Document Object Model (DOM) represents a web page (a document) as objects and nodes that you can modify using JavaScript. You can change its content, style, and structure. Examples: jQuery, Umbrella JS • Data Handling With the enormous amounts of data that businesses now deal with daily, handling and managing them properly is essential. Using a JavaScript library makes it easier to handle a document following its content while adding more interactivity. Examples: D3.js • Database Effective database management is necessary to read, create, delete, edit, and sort data. You can also use sophisticated queries, auto-create tables, synchronize and validate data, and much more. Examples: TaffyDB, ActiveRecord.js • Forms Use JS libraries to simplify form functions, including form validation, synchronization, handling, conditional capabilities, field controls, transforming layouts, and more. Examples: wForms, LiveValidation, Validanguage, qForms • Animations

	People love animations, and you can leverage them to make your web page interactive and more engaging. Adding micro-interactions and animations is easy by using JavaScript libraries. Examples: Anime.js, JSTweener • Image Effects Users can add effects to images and make them stand out using JS libraries. Effects include blurring, lightening, embossing, sharpening, grayscale, saturation, hue, adjusting contrast, flipping, inverting, reflection, and so on. Examples: ImageFX, Reflection.js • Fonts Users can incorporate any font they wish to make their web page more compelling based on the content type. Examples: typeface.js • Math and String Functions Adding mathematical expressions, date, time, and strings can be tricky. For example, a date consists of many formats, slashes, and dots to make things complex for you. The same holds when it comes to matrices and vectors. Use JavaScript libraries to simplify these complexities in addition to manipulating and handling URLs effortlessly. Examples: Date.js, Sylvester, JavaScript URL Library

	• User Interface and Its Components You can provide a better user experience via web pages by making them more responsive and dynamic, decreasing the number of DOM operations, boosting page speed, and so forth. Examples: ReactJS, Glimmer.js And those are just the most common use cases. Other uses of JavaScript libraries include: • Creating a custom dialog box • Creating keyboard shortcuts • Switching platforms • Creating rounded corners • Affecting data retrieval/AJAX • Aligning page layouts • Creating navigation and routing • Logging and debugging • And many more Important Libraries. jQuery is a classic JavaScript library that's fast, light-weight, and feature-rich. It was built in 2006 by John Resig at BarCamp NYC. jQuery is free and open-source software with a license from MIT. React.js (also known as ReactJS or React) is an open-source, front-end JavaScript library. It was created in 2013 by Jordan Walke, who works at Facebook as a software engineer. Data-Driven Documents (D3) or D3.js is another famous JS library that developers use to document manipulation based on data. It was released in 2011 under the BSD license.

	Underscore is a JavaScript utility library that provides various functions for typical programming tasks. It was created in 2009 by Jeremy Askenas and release with an MIT license. Now, Lodash has overtaken it. Lodash is also a JS utility library that makes it easier to work with numbers, arrays, strings, objects, etc. It was released in 2013 and also uses functional programming design like Underscore.js. Algolia Places is a JavaScript library that provides an easy and distributed way of using address auto-completion on your site. It's a blazingly fast and wonderfully accurate tool that can help increase your site user experience. Algolia Places leverages the impressive open-source database of OpenStreetMap to cover worldwide places. If you want to add animations to your site or application, Anime.js is one of the best JavaScript libraries you can find. It was released in 2019 and is lightweight with a powerful yet simple API. Animate On Scroll works great for single-page parallax websites. This JS library is fully open-source and helps you add decent animations on your pages that look sweet as you scroll down or up. Chart.js is an excellent JavaScript library to use. Chart.js is a flexible and simple library for designers and developers who can add beautiful charts and graphs to their projects in no time. It is open-source and has an MIT license. Cleave.js offers an interesting solution if you want to format your text content. Its creation aims to provide an easier way to increase the input field's readability by formatting the typed data. Released in 2017, Glimmer features lightweight and fast UI components. It uses the powerful Ember CLI and can work with EmberJS as a component.

	JavaScript frameworks are application frameworks that allow developers to manipulate code to meet their unique requirements. Web application development is analogous to building a house. You have the option to create everything from scratch with construction materials. But it will consume time and may incur high costs. But if you use readymade materials such as bricks and assemble them based on the architecture, then construction becomes faster, saving you money and time. Application development works similarly. Instead of writing every code from scratch, you can use pre-written codes working as building blocks based on the application architecture. Frameworks can adapt to website design more quickly and make it easy to work with JavaScript. How to Use JavaScript Frameworks To use a JavaScript framework, read the JS framework's documentation you intend to use and follow the steps. What Are JavaScript Frameworks Used For? • To build websites • Front-end app development • Back-end app development • Hybrid app development • Ecommerce applications • Build modular scripts, for example, Node.js • Update DOM manually • Automate repetitive tasks using templating and 2-way binding • Develop video games • Create image carousels, • Testing codes and debugging • To bundle modules Example Frameworks:

	AngularJS by Google is an open-source JavaScript framework released in 2010. This is a front-end JS framework you can use to create web apps.It was created to simplify the development and testing of web applications with a framework for MVC and MVVM client-side architectures. Design fast and mobile responsive sites quickly using Bootstrap, one of the most popular open-source toolkits for front-end development. Released in 2016, Aurelia is a simple, unobtrusive, and powerful open-source, front-end JS framework to build responsive mobile, desktop, and browser applications. It aims to focus on aligning web specifications with convention instead of configuration and requires fewer framework intrusion. Vue.js was created in 2014 by Evan You while he worked for Google. It is a progressive JavaScript framework to build user interfaces.Vue.js is incrementally adoptable from its core and can scale between a framework and library easily based on various use cases. The open-source JS framework Ember.js is battle-tested and productive to build web applications with rich UIs, capable of working across devices. It was released in 2011 and was named SproutCore 2.0 back then.
Domains What industries or domains use this programming language? Provide specific examples of companies that use this language and what they use it for**. E.g. Company X uses C# for its line of business applications.**	JavaScript is everywhere, but just how are some of the world's largest tech companies using JavaScript? For these companies, JavaScript is immensely important, and that doesn't look like it's going to change any time soon.

	Companies such as PayPal, Netflix, Facebook, Google, Etc. use JavaScript. Often used for front-end development in the companies mentioned above. JavaScript is excellent technology to use on the web. It is not that hard to learn and it is very versatile. The speed and small memory footprint of JavaScript in comparison to other languages brings up more and more uses for it from automating repetitive tasks in programs like Illustrator, up to using it as a server-side language with a standalone parser. The future is wide open. JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.). There are also more advanced server side versions of JavaScript such as Node.js, which allow you to add more functionality to a website than downloading files (such as realtime collaboration between multiple computers). Inside a host environment (for example, a web browser), JavaScript can be connected to the objects of its environment to provide programmatic control over them. JavaScript contains a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements.

	Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example: • Client-side JavaScript extends the core language by supplying objects to control a browser and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation. • Server-side JavaScript extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server. This means that in the browser, JavaScript can change the way the webpage (DOM) looks. And, likewise, Node.js JavaScript on the server can respond to custom requests from code written in the browser.

	How JavaScript Makes Things Dynamic : HTML defines the structure of your web document and the content therein. CSS declares various styles for the contents provided on the web document. HTML and CSS are often called markup languages rather than programming languages, because they, at their core, provide markups for documents with very little dynamism. JavaScript, on the other hand, is a dynamic programming language that supports Math calculations, allows you to dynamically add HTML contents to the DOM, creates dynamic style declarations, fetches contents from another website, and lots more. Uses of JavaScript : JavaScript is one of the most used languages in the market these days. JavaScript stands second in the lineup. It is mainly used in building websites and web applications. The other application of JavaScript is listed below.

	• Adding interactive behavior to web pages. JavaScript allows users to interact with web pages. … • Creating web and mobile apps. Developers can use various JavaScript frameworks for developing and building web and mobile apps. … • Building web servers and developing server applications. … • Game development. Core Features of JavaScript : JavaScript is one of the most popular languages which includes numerous features when it comes to web development. It's amongst the top programming languages as per Github and we must know the features of JavaScript properly to understand what it is capable of. Some of the features are lightweight, dynamic, functional and interpreted. Now we are going to discuss some important features of JavaScript. • Light Weight Scripting language

	• Dynamic Typing • Object-oriented programming support • Functional Style • Platform Independent • Prototype-based • Interpreted Language • Async Processing • Client-Side Validation • More control in the browser • Detecting the User's Browser and OS • Handling Dates and Time Frameworks : 1. React

	2. Angular 3. Backbone 4. Vue 5. Node 6. Ember