**Language Map for JavaScript**

|  |  |
| --- | --- |
| **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 dynamically typed  Code:  var num = 1;  let msg;  msg = “Hello JavaScript”;  const num = 100; |
| **Data Types**  *List all of the data types (and ranges) supported by this language.* | Strings, numbers, Bigint, Booleans, undefined, null, symbol, object  let myString = "Hello World"; // String let myNumber = 123; // Number let myBoolean = true; // Boolean let myNull = null; // Null let myUndefined; // Undefined let mySymbol = Symbol("mySymbol"); // Symbol (ES6) let myBigInt = 123n; // BigInt (ES2020)  let myObject = { name: "John", age: 30 }; // Object let myArray = [1, 2, 3]; // Array |
| **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.*** | If statement  if (condition) {  // code to be executed if condition is true }  if/else statement  if (condition) {  // code to be executed if condition is true } else {  // code to be executed if condition is false }  if/else/if/else statement  if (condition1) {  // code to be executed if condition1 is true } else if (condition2) {  // code to be executed if condition2 is true } else {  // code to be executed if all conditions are false }  switch statement  switch (expression) {  case value1:  // code to execute if expression matches value1  break;  case value2:  // code to execute if expression matches value2  break;  default:  // code to execute if no case matches } |
| **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.*** | for loop  for (let i = 0; i < 5; i++) {  console.log("Iteration: " + i); }  for/of loop  const numbers = [1, 2, 3, 4, 5]; for (const number of numbers) {  console.log(number); }  for/in loop  const person = { name: "Alice", age: 30 }; for (const key in person) {  console.log(key + ": " + person[key]); }  while loop  let count = 0; while (count < 5) {  console.log("Count: " + count);  count++; }  do-while loop  let i = 0; do {  console.log("Index: " + i);  i++; } while (i < 5); |
| **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.)* | const cars = ["GMC", "Chevrolet", "Cadillac"];  Add array element  const cars = ["GMC", "Chevrolet", "Cadillac"]; fruits.push("Buick");  // Adds a new element (Buick) to cars  new array  const points = new Array(40, 100, 1, 5, 25, 10); const points = [40, 100, 1, 5, 25, 10]; |
| **Data Structures**  *If this language provides a standard set of data structures, provide a list of the data structures and their Big-Oh complexity.* | **Array**  const arr = ['a', 'b', 'c', 'd']  console.log(arr[2]) // c  **Object**  const obj = {  prop1: "I'm",  prop2: "an",  prop3: "object"  }  **Stack – Insertion O(1), Removal O(1), Searching O(n), Access O(n)**  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  **Queue - Insertion O(1), Removal O(1), Searching O(n), Access O(n)**  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 queue 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 List - Insertion O(1), Removal O(n), Searching O(n), Access O(n)**  class Node{  // Each node has three properties, its value, a pointer that indicates the node that follows and a pointer that indicates the previous node  constructor(val){  this.val = val;  this.next = null;  this.prev = null;  }  }  // We create a class for the list  class DoublyLinkedList {  // 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.length === 0){  this.head = newNode  this.tail = newNode  } else {  this.tail.next = newNode  newNode.prev = this.tail  this.tail = newNode  }  this.length++  return this  }  // The pop method removes the tail of the list  pop(){  if(!this.head) return undefined  const poppedNode = this.tail  if(this.length === 1){  this.head = null  this.tail = null  } else {  this.tail = poppedNode.prev  this.tail.next = null  poppedNode.prev = null  }  this.length--  return poppedNode  }  // The shift method removes the head of the list  shift(){  if(this.length === 0) return undefined  const oldHead = this.head  if(this.length === 1){  this.head = null  this.tail = null  } else{  this.head = oldHead.next  this.head.prev = null  oldHead.next = null  }  this.length--  return oldHead  }  // 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.length === 0) {  this.head = newNode  this.tail = newNode  } else {  this.head.prev = newNode  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  let count, current  if(index <= this.length/2){  count = 0  current = this.head  while(count !== index){  current = current.next  count++  }  } else {  count = this.length - 1  current = this.tail  while(count !== index){  current = current.prev  count--  }  }  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){  var foundNode = this.get(index)  if(foundNode != null){  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 === 0) return !!this.unshift(val)  if(index === this.length) return !!this.push(val)  var newNode = new Node(val)  var beforeNode = this.get(index-1)  var afterNode = beforeNode.next  beforeNode.next = newNode, newNode.prev = beforeNode  newNode.next = afterNode, afterNode.prev = newNode  this.length++  return true  }}}  **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**  **Graphs** |
| **Objects**  *If this language support object-orientation, provide an example of how you would write a simple object with a default constructor and then how you would instantiate it.* | const truck = {type:"GMC", model:"Sierra 1500", color:"black"};  function car {  this.type = type;  this.model = model;  this.color = color;  } |
| **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 Engine  V8: Chrome and Node.js  SpiderMonkey: Firefox  JavaScriptCore: Safari  Web APIs (Browser Environment)  COM Manipulation - Access and modify the HTML structure of a webpage.  Fetch API: Make network requests to fetch data from servers.  WebSockets: Establish real-time communication channels with servers.  Canvas API: Draw graphics on the webpage.  Example:  // Fetch API fetch('https://api.example.com/data')  .then(response => response.json())  .then(data => console.log(data));  Node.js (Node.js Environment)  File System Access: Read and write files on the server.  Network Operations: Create servers and handle network requests.  Child Processes: Spawn new processes to run other programs.  Example:  // File System Access const fs = require('fs'); fs.readFile('data.txt', 'utf8', (err, data) => {  if (err) {  console.error(err);  return;  }  console.log(data); }); |
| **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..* | jQuery – helps in DOM manipulation and event handling in web applications. Provides cross-browser compatibility and it simplifies the code.  ReactJS – create user interfaces in a declarative and efficient way. Creates simple views for each state in the application and efficiently updates and renders the right component as your data changes.  React Reactstrap – bootstrap-based React UI used to make good-looking webpages. Provides Bootstrap components and increases code reusability.  jQuery Mobile – HTML5-based user interface system designed to make responsive websites and apps. Accessible on smartphones, tablets, and desktop devices. Creates responsive and Ajax-based user interface system.  Angular js – build dynamic and interactive web applications. Creates application based on MVC architecture that leads to cleaner and more organized code.  Bootstrap – enable developers to build responsive, mobile-first websites with ease. Creates responsive web designs, mobile-first development, efficient prototyping, consistent cross-browser compatibility. |
| **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.*** | Web Development: JavaScript is the foundation of front-end web development, enabling interactive user interfaces and dynamic content.  Media and Entertainment: Streaming platforms like Netflix, Hulu, and Spotify rely on JavaScript for their user interfaces and video playback functionality.  E-commerce: JavaScript powers online shopping experiences, from product displays and shopping carts to payment processing and order tracking.  Social Media: Social networking platforms like Facebook, Twitter, and Instagram are heavily reliant on JavaScript for their dynamic content and real-time interactions. |