Data Structure & Algorithm:

Data Structure:

The term means a value or set of values. It specifies either the value of a variable or a constant. While a data item that does not have subordinate data items is categorized and an elementary item, the one that is composed of one or more subordinate data items is called a group item.

Data is defined as collection of raw facts which do not have any defined context. We can say that Data is collection of numbers and text values which do not have any direct benefit to the organization that collects the data since it is unorganized and unprocessed . For example data collection done by government regarding health, income, dietary habits, family etc. does not have any benefit if it is stored as it is. The reason is that it is very voluminous and do not give any information. The Data thus collected needs to be processed before it can be used for decision making or reporting tasks.

A field is defined as a unit of meaningful information about an entity like date of flight, name of passenger, address etc. A record is a collection of data items. Record is a collection of units of information about a particular entity. Passenger of an airplane, an employee of an organization or an article sold from a store. A file is a collection of related records. A collection of records involving a set of entities with certain aspects in common and organized for some particular purpose is called a file. for example collection of records of all passengers.

A data structure is defined as mathematical or logical model used to organize and manipulate data. The management of data is done through various operations like traversal, insertion, deletion, searching, sorting etc. The logical organization of data as a data structure is done to make the data organization easier, allow access to individual elements of a data structure, defining the association among data elements and provide various operations to process the data to derive information.

To create a computer program and solve a specific problem, the programmer needs some structure to hold the data values. So, data structures are an important aspect of problem solving through computer programming. It must be easy to understand and implement and must exhibit the relationship among data elements required to provide solution. For example to implement a transportation or network problem a graph like data structure is needed. To implement sequential execution of submitted tasks a queue like data structure is the best option.

Linear Data Structure:

Data elements are organized in a linear fashion in a linear data structure. Traversal can only be done sequentially. All the previous elements must be followed first to reach a particular element in linear data structures. It also means that only one data element can be directly reached from the previous element. Every data element in a linear data structure have a direct relation with its prior and following element Examples of linear data structure are Arrays, Linked Lists, Stacks and Queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

Non-Linear Data Structure:

In a non-linear data structure data elements are arranged non-linearly and traversal cannot be done sequentially. Every data element may be linked to more than one data elements. The linkages of the data elements reflect a particular relationship. The relationship between the elements can be hierarchical or random. Examples of linear data structure are Trees and Graphs. if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure.

Primitive Data Structure:

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.

Non-Primitive Data Structure:

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

Linear Data Structure Elements:

Array:

An array is an indexed collection of fixed number of homogeneous data elements. The main advantage of arrays is we can represent huge number of values by using single variable. So that readability of the code will be improved. But the main disadvantage of array is fixed in size that is one's we create an array there is no chance of increasing or decreasing in the size based on our requirement.

Hence to use arrays concept compulsory we should know the size in advance, which may not possible always. An array is a homogeneous and linear data structure that is stored in contiguous memory locations. An element in an array can be accessed, inserted or removed by specifying its position or index or subscript along with the name of the array.

Linked List:

A linked list is a fundamental data structure in computer science, used to organize and manage collections of elements. Unlike arrays, which store elements in contiguous memory locations, a linked list consists of nodes, each containing both data and a reference (or pointer) to the next node in the sequence. This flexible structure enables dynamic memory allocation, making it efficient for inserting and deleting elements anywhere in the list. However, traversal through a linked list can be slower compared to arrays, as it requires following the links sequentially.

Linked lists come in various forms, including singly linked lists with one-way connections, doubly linked lists with both next and previous references, and circular linked lists where the last node points back to the first. Each variation offers specific advantages depending on the use case, making linked lists a versatile and essential component of data structuring and manipulation in programming. These are the following types of Linked List.

Singly Linked List:

Singly Linked List is a type of linked list when each node contains data and a reference to the next node. It's a simple structure that allows traversal in only one direction, from the head (start) to the tail (end) of the list. Singly linked lists are efficient for insertions and deletions at the beginning and middle of the list, but accessing elements further down the list requires linear traversal.

A linear or singly linked list is a linear data structure consisting of a sequence of nodes. Each node consists of two parts – Data and Link to the next node. The address of first node is stored in a pointer Start.

Each node stores the address of its next node in its link part. The linked list is non-contiguous collection of nodes that allows insertion and deletion at any specific location.

Doubly Linked List:

A doubly linked list is a linear data structure consisting of a sequence of nodes. Each node consists of three parts – Data, Forward Pointer to the next node and Backward Pointer to the previous node. The address of first node is stored in a pointer called the Header and address of last node is stored in pointer called the Trailer. The doubly linked list allows traversal in forward and backward direction.

Doubly Linked List is a type of linked list where each node has references to both the next and the previous nodes. This bidirectional linkage allows for easier traversal in both directions, making insertions and deletions at any position more efficient. However, doubly linked lists require more memory due to the additional references in each node.

Circular Linked List:

A circular linked list forms a closed loop, where the last node points back to the first node, creating a circular structure. This type can be implemented as either singly or doubly linked. Circular linked lists are useful for applications like implementing circular buffers or managing processes in a round-robin scheduling algorithm.

Queue:

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. Elements are added to the back (enqueue) and removed from the front (dequeue). It resembles a real-world queue, like people waiting in line. Queues are useful for managing tasks in order, such as in breadth-first searches and print job scheduling.

Queue is a linear data structure that follows the first-in first-out scheme. Insertions are done at the rear of the queue and removals at the front of the queue. Two variables needed for these operations are the front and rear.

Circular Queue:

A Circular Queue is queue following “First In First Out” (FIFO) principle with the difference that the last element is associated back with the first element of the queue.

Insertion will be done at end called the REAR. Deletion will take place at beginning called the REAR. The elements that are inserted earlier will be deleted first. The elements added later will be deleted later. An item is added at the end and removed from the beginning and the first follows the last element. It is used to maintain sequence of events in a round robin fashion.

Double-Ended Queue:

A variation of Queue is Dequeue in which deletions and insertions can be done at both ends. A deque is a versatile linear structure that supports insertion and deletion at both ends. It can function as a queue or a stack, providing flexibility for various applications. Deques are efficient for tasks like implementing algorithms that require simultaneous access to both ends, such as palindrome checking and implementing sliding window problems.

Priority Queue:

A priority queue is a data structure where elements have assigned priorities, and the highest-priority element is accessed first. It doesn't follow a strict order like queues. It's often implemented as a binary heap, enabling efficient insertion and retrieval of the highest-priority element. Priority queues are crucial in scenarios where tasks must be processed based on urgency or importance, such as in Dijkstra's algorithm for shortest path finding and Huffman coding in data compression.

priority queue is a linear data structure that behaves like a queue where each element has a priority associated with it on the basis of which deletions are done. The element with maximum (minimum) priority will be deleted from the queue. It can also be defined as a data structure that supports operations of min or max insert, delete or search.

Non-Linear Data Structure Element:

Heap:

A max(min) heap is a complete binary tree with the property that the value of each node is at least as large(small) as the values at its children. If the elements are distinct then the root node will be the largest (smallest).

The insertion will always be done as a leaf node first and then the heap will be “happified” to bring the newly inserted node at its correct position. Heapify means that the node will be compared with its parent and if it is larger(smaller) than its parent then there will be an interchange between parent node and the newly inserted child node. This process will end up in a heap after insertion.

Deletion will be done at the root node. The last node of the complete binary tree will replace the deleted root node and then the process of Heapify will be done starting from the new root node.

Tree:

A tree is a Non-Linear Data Structure that is an abstract model of a hierarchical structure consisting of nodes with a parent-child relation. Its applications are Organization charts, File systems, Programming environments. There are four things associated with any tree -Distinction between nodes, Value of nodes, orientation structure and the number of levels.

Root is starting point of the tree is a node called root characterized by a node without parent. External/Leaf Node is a node without any children. Internal node is all the nodes other than the root leaf nodes. It can be said to be a node with at least one child. Ancestors of a node parent, grand-parent and any other nodes which lie on the path from root to that node.

Depth of a node is the number of ancestors for any give node. Height of a tree is maximum depth among all the leaf nodes. Descendant of a node is child grandchild and all the other nodes lying on path from a node to a leaf node. Subtree is a tree consisting of a node and its descendants.

Graph:

A graph G(V,E) is a defined with two sets. V , a set of vertices, and E the set of edges between two pair of vertices from the set V. When the edges of the graph have to be defined with the direction, it is called a directed graph or digraph, and the edges are called directed edges or arcs. In a digraph edge (a,b is not the same as edge (b, a). In a directed edge (a,b) a is called the source/starting point and b is called the sink/destination/ end point. In an undirected graph the direction of edge is not specified.

When the edges of the graph have to be defined with the direction, it is called a directed graph or digraph, and the edges are called directed edges or arcs. In a digraph edge (a,b is not the same as edge (b, a). In a directed edge (a,b) a is called the source/starting point and b is called the sink/destination/ end point. In an undirected graph the direction of edge is not specified. Two vertices that are connected to each other with an edge are called neighbors. They are also said to be adjacent to each other.

Data Structure Operations:

Traversal:

Traversal of a data structure can be defined as the process of visiting every element of a data structure at least once. This operation is most commonly used for printing, searching, displaying or reading the elements stored in the data structure. It is usually done by using a variable or pointer that indicates the current element of the data structure being processed.

This variable or pointer is updated after visiting an element so that the location or address of next element can be found. When the last element of the data structure is reached the traversal process ends. Traversal can be useful when all the elements of the data structure have to be manipulated in similar way.

In an array traversal begins from the first element. A variable stores the index of first element of the array and it is incremented after visiting each node. When the value of this variable is equal to the index of last node, the traversal of the array ends.

Insertion:

Once a data structure is created, it can be extended by adding new elements. The process of adding new elements in an existing data structure is called insertion. Where the element can be added depends upon the data structure that a user is dealing with. Insertion operation always ends up in increasing the size of the data structure.

A linked list and an array allow a user to insert a new element at any location. A stack and a queue allow a user to insert a new element only at a specific end. New nodes can be added to a graph or tree in a random fashion.

For all the data structure the insertion can be done till the data structure has enough space to store new elements either due to its defined size or memory availability. A condition when a user tries to insert a new element in a data structure that does not have the needed space for new element is called Overflow

Deletion:

Once a data structure is created, a user can remove any of the existing elements and free up the space occupied by it. The process of deleting an existing element from a data structure is called deletion. Which element can be deleted depends upon the data structure that a user is dealing with. Deletion operation always ends up in reducing the size of the data structure.

A linked list and an array allow a user to delete an existing element at any location i.e. start, mid or end. A stack and a queue allow a user to delete an element only at a specific end. Nodes can be deleted from a graph or tree in a random fashion. For all the data structure the deletion can be done till the data structure has elements. A condition when a user tries to delete an element from a data structure that does not have any element is called Underflow

Search:

The process of locating an element in data structure and returning its index or address is called Search. This is the most commonly used operation in a data structure. Since a data structure stores data in an organized fashion for convenient processing, it is important that an element could be easily located in a data structure.

When performing search in a data structure a key value is needed, this is matched with the values stored in the data structure. When the value of the element matches the key value successful search is done and the search operation returns the location of that element. If the key value does not match any element in the data structure and reaches end, it is unsuccessful search and a null location is returned as a result of search operation.

Sorting:

The process of arranging the data elements in a data structure in a specific order (ascending or descending) by specific key values is called sorting. The students records stored in an array can be sorted by their registration numbers, names or the scores. In each sorting the criteria will be different to fulfill the processing needs of a user.

Sorting may result in physical relocation of elements or it can be just a rearrangement of key values as index without changing the physical address of complete data element so that a subsequent traversal operation displays the data in sorted manner.

Merge:

Merging can be defined as the process of combining elements of two data structures. In a simpler form merging can be treated as appending set of elements of one data structure after elements of another data structure of same element structure. For example two arrays containing student data of two different classes with same fields to form one list. The final data structure may or may not be sorted.

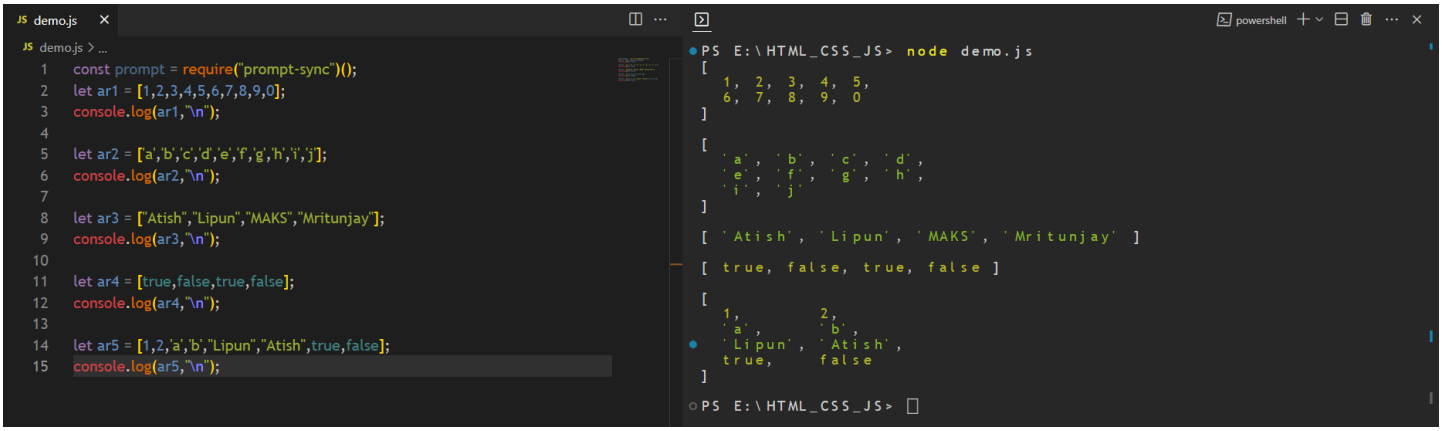
Another form of merging can be to merge two data structure of different constructions to create totally new data structure. For example one data structure with student registration number and name can be merged with another data structure containing course and result data of same set of students to form one list(or file)

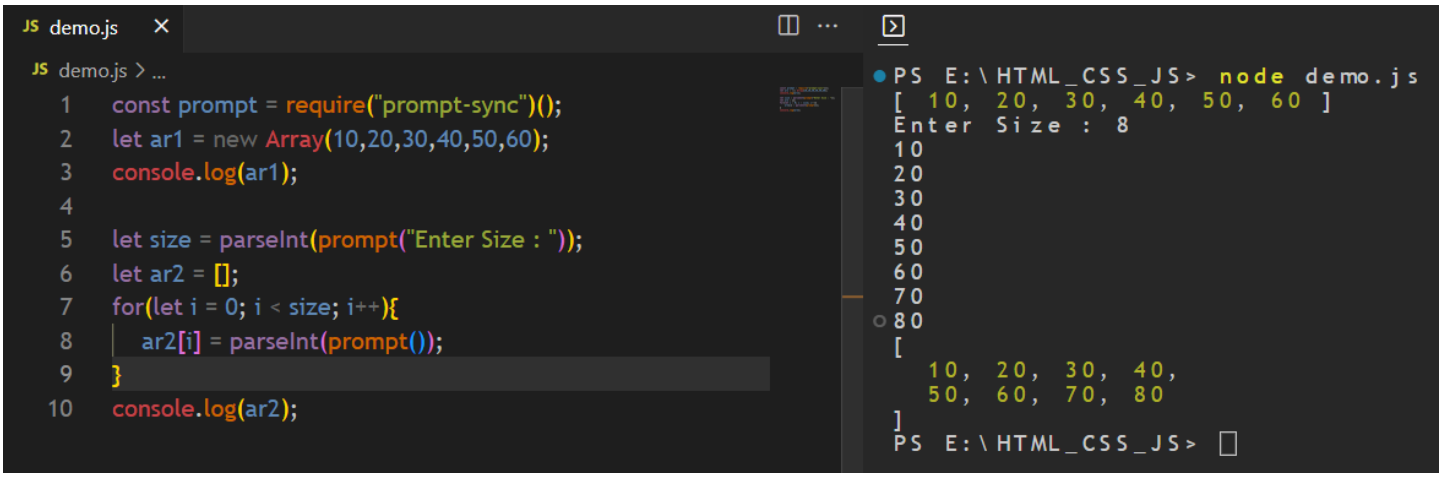
Copy:

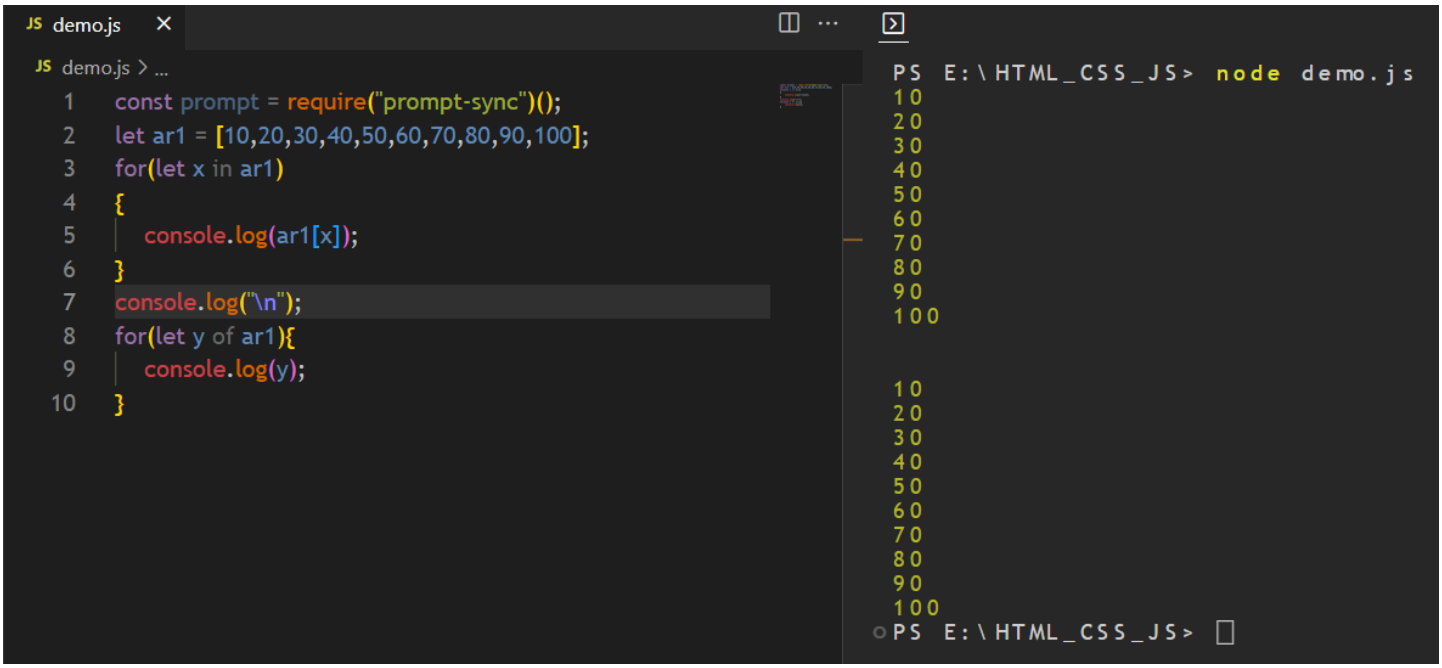
The process of copying is the operation that makes a new data structure from an existing data structure. In the process of copying the original data structure retains its structure and data elements. The new data structure has same data elements. Both these data structure can be used to perform all the operations discussed previously. Changes in one data structure will not affect the elements or count of elements of the other data structure.

Array & Array Method:

In Java Script array is a linear and homogeneous data structure which is used for to store data in a linear and sequential way. The homogeneous means in an array all the elements data type are same type of data type.



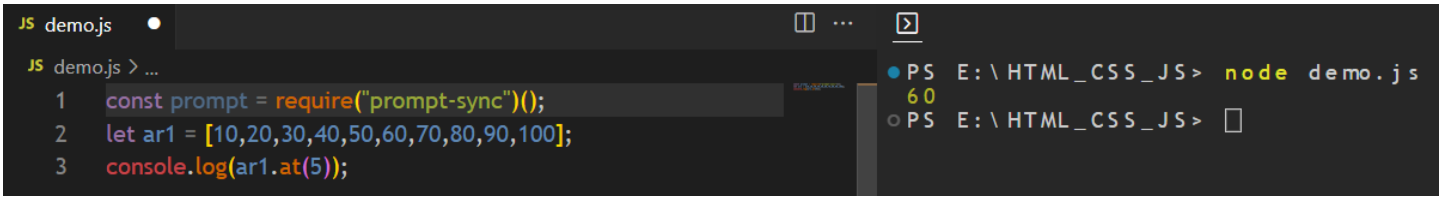




Array Method:

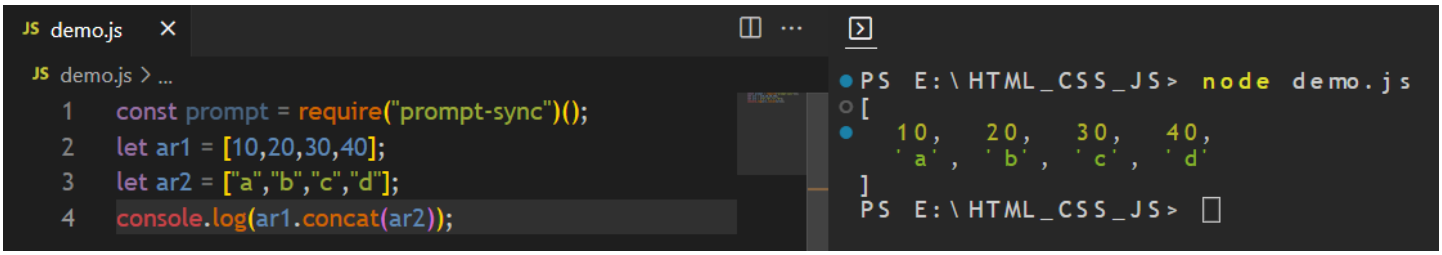
at():

at() this method is used for to display the given index number's element and count starts from 0.



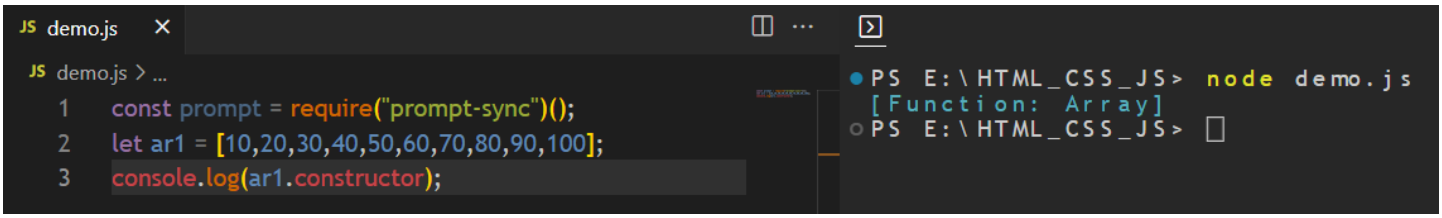
concat():

this method concatenates(joins) two or more arrays. The concat() method returns a new array, containing the joined arrays. The concat() method does not change the existing arrays.



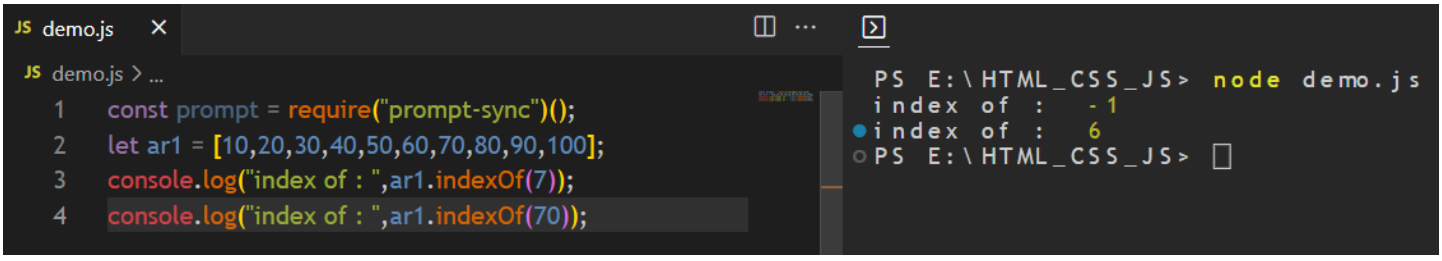
constructor:

this property returns the function that created the array prototype.



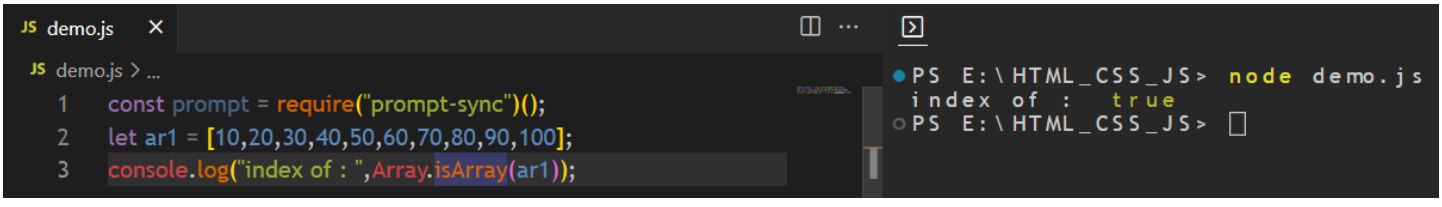
indexOf()

this method returns the first index position of a specified value. this function returns -1 if the value is not found. This method starts at a specified index and searches from left to right.



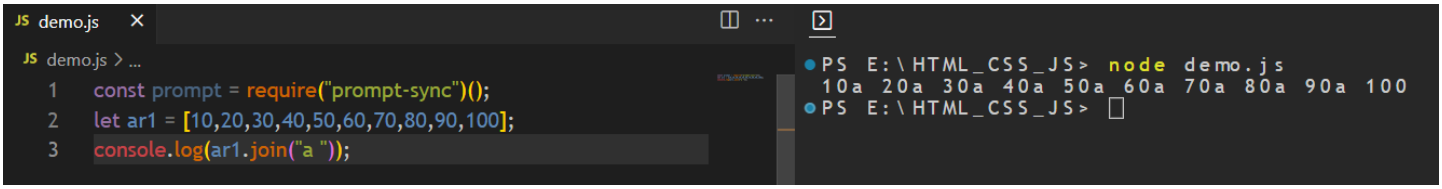
isArray()

this function is used for to check whether the array is a valid array or not.



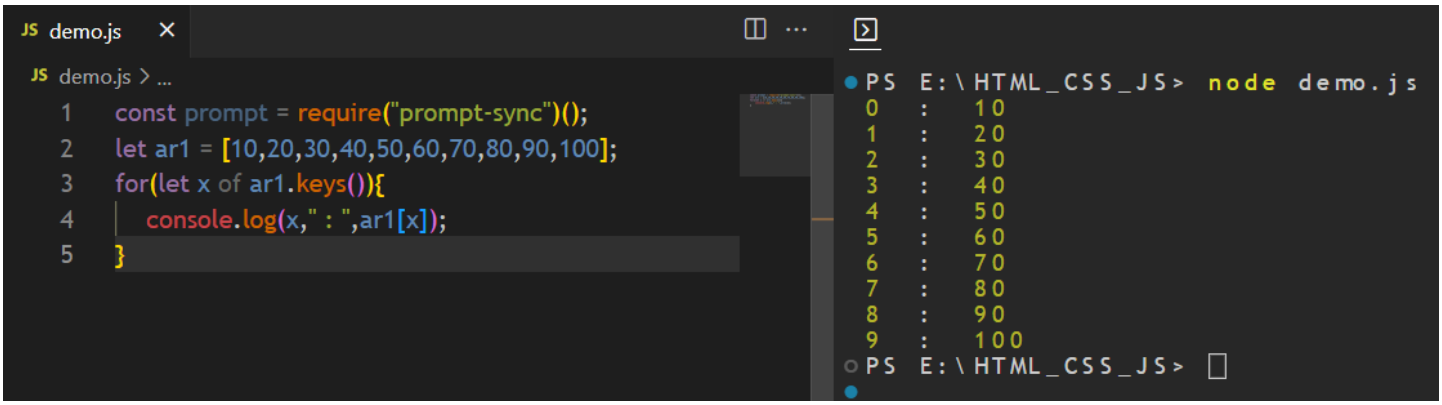
join():

this method returns an array as a String. this function does not change the original array. any separator can be specified. The default is comma.



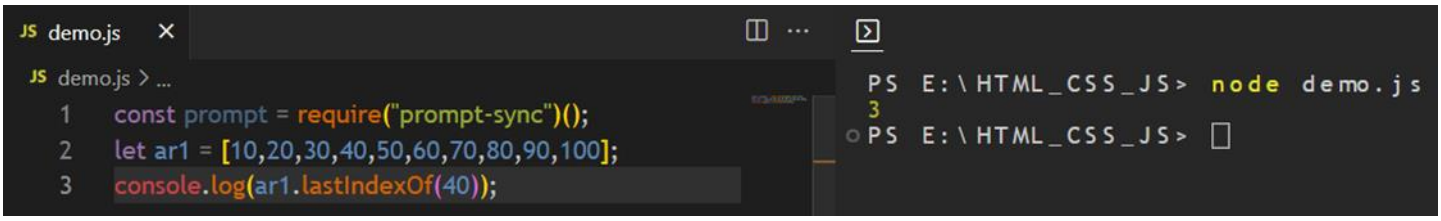
keys():

this function return an array iterator object with the keys of an array. this function method does not change the original array.

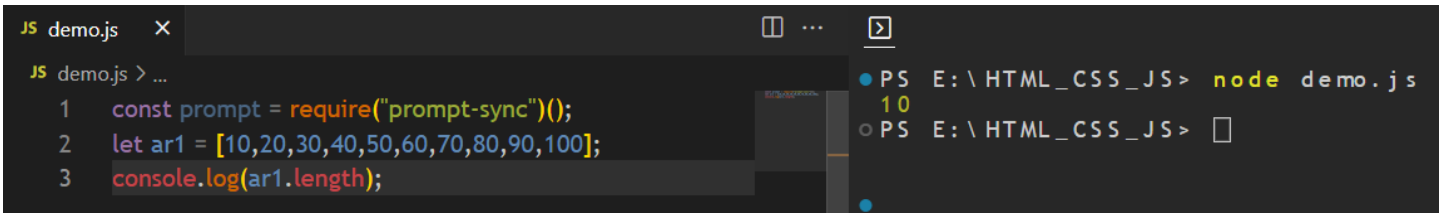


lastIndexOf():

this method returns the lastIndex(position) of a specified value. the lastIndexOf() method returns -1 if the value is not found. The lastIndexOf() starts at a specified index and searches from right to left. By default the search starts at the last element and ends at the first. Negative start values counts from the last element(but still searches from right to left).

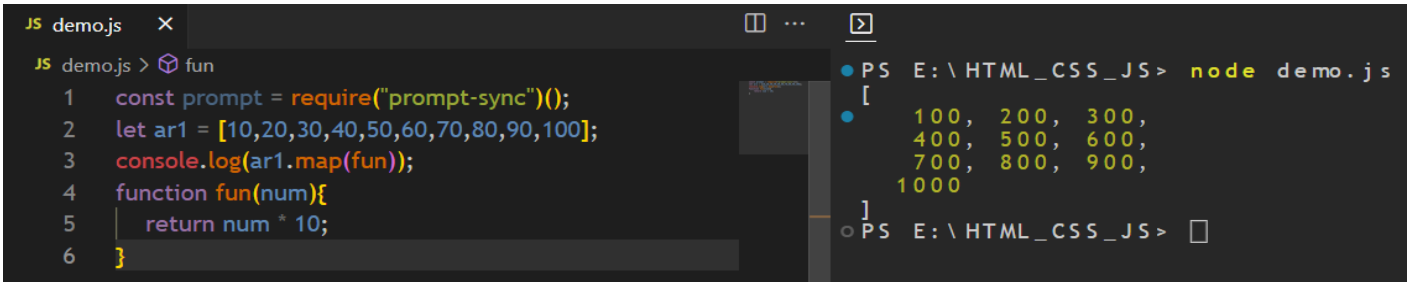


Length: this is used for to find the size of array.



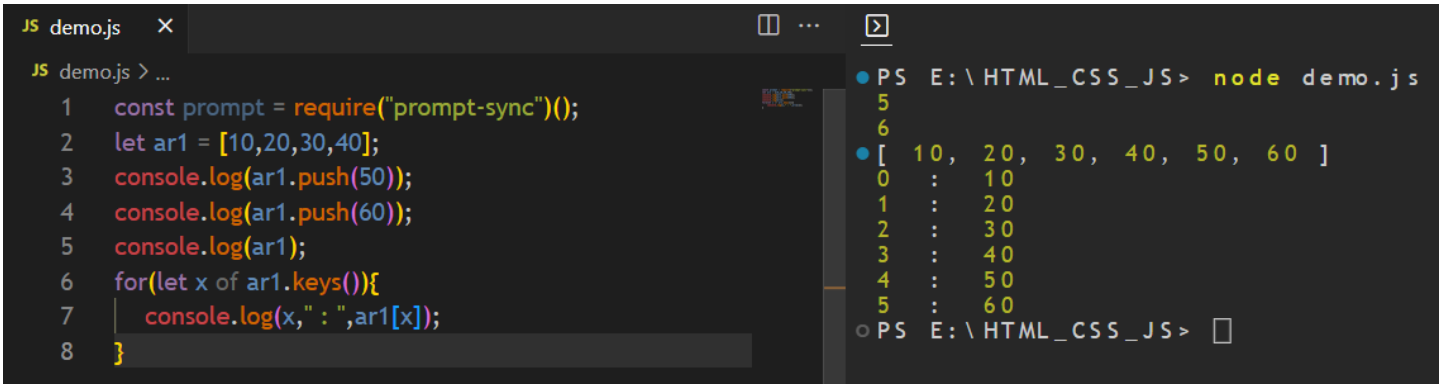
map():

this function creates a new array from calling a function for every array element. This function does not execute the function for empty element. Map() doesn't change the original array.



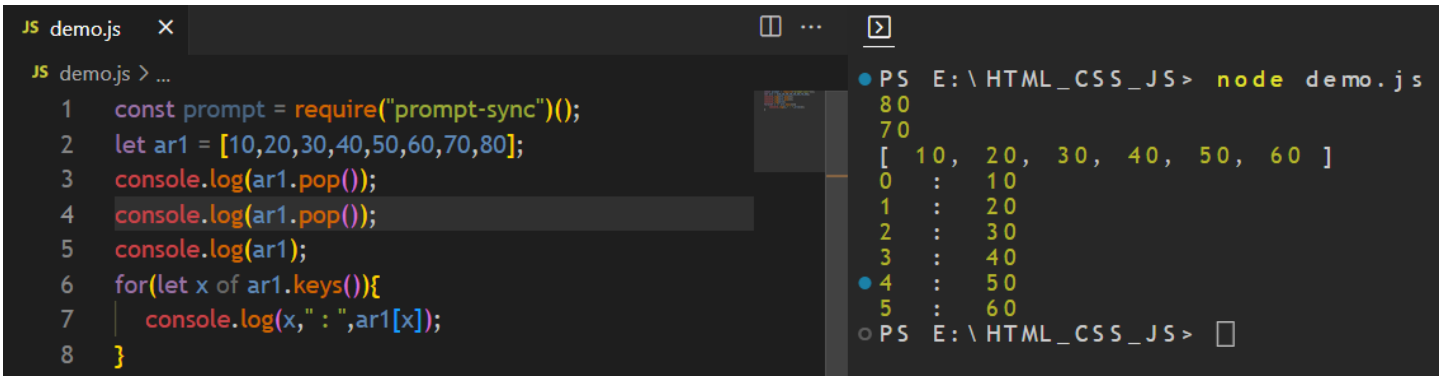
push():

this method adds new items to the end of an array. the push() method changes the length of the array. the push() method returns the new length. If you print then count start from 1.



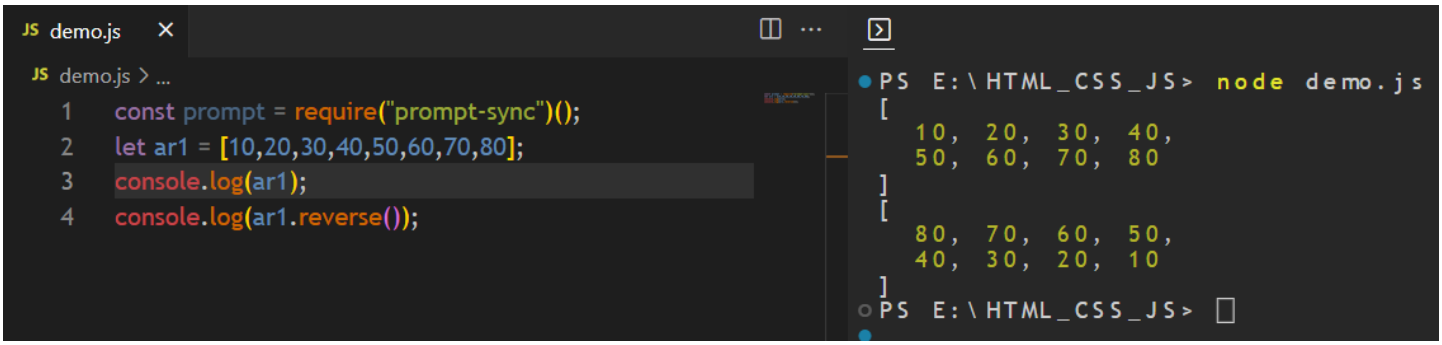
pop():

this method removes the last element of an array. the pop method changes the original array. the pop() method returns the removed element.



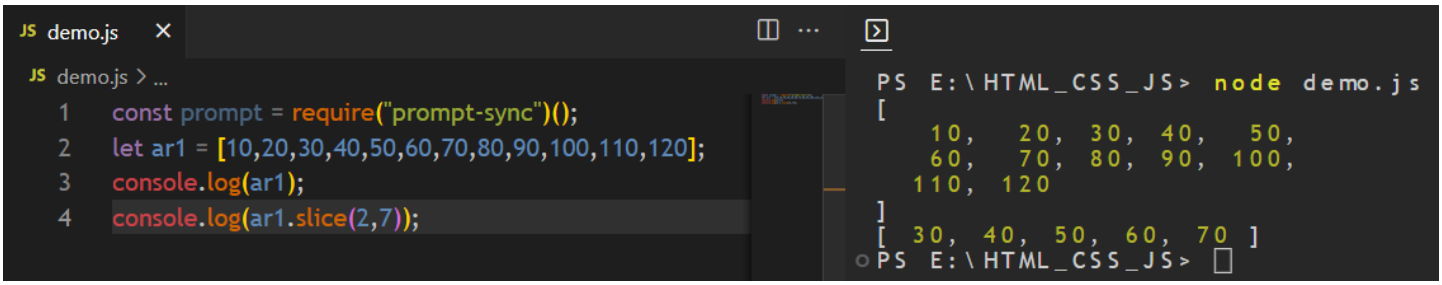
reverse():

this method reverse the order of the elements in an array. the reverse() method overwrites the original array.



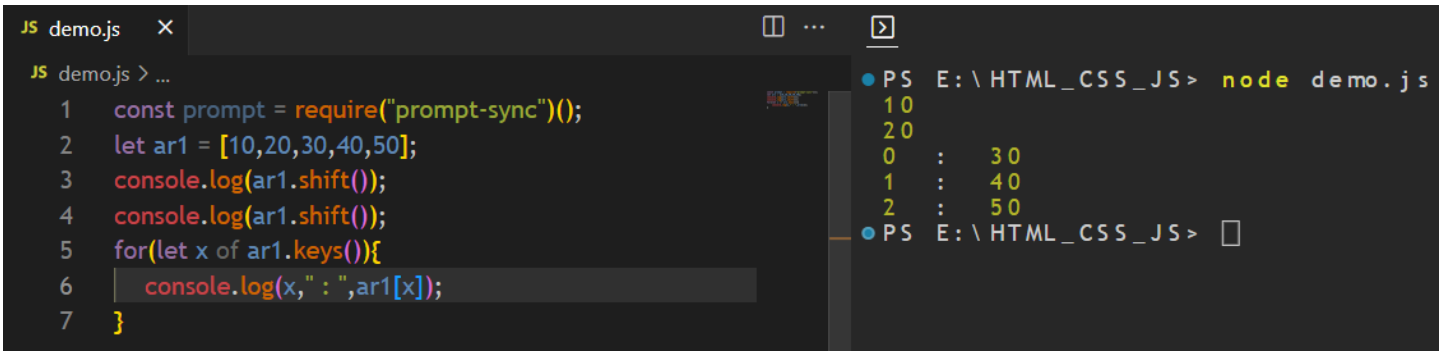
slice():

this method returns selected elements in an array. as a new array. the slice method selects from a given start up to a (not inclusive) given end. The slice() method does not change the original array. print from 0 to n – 1



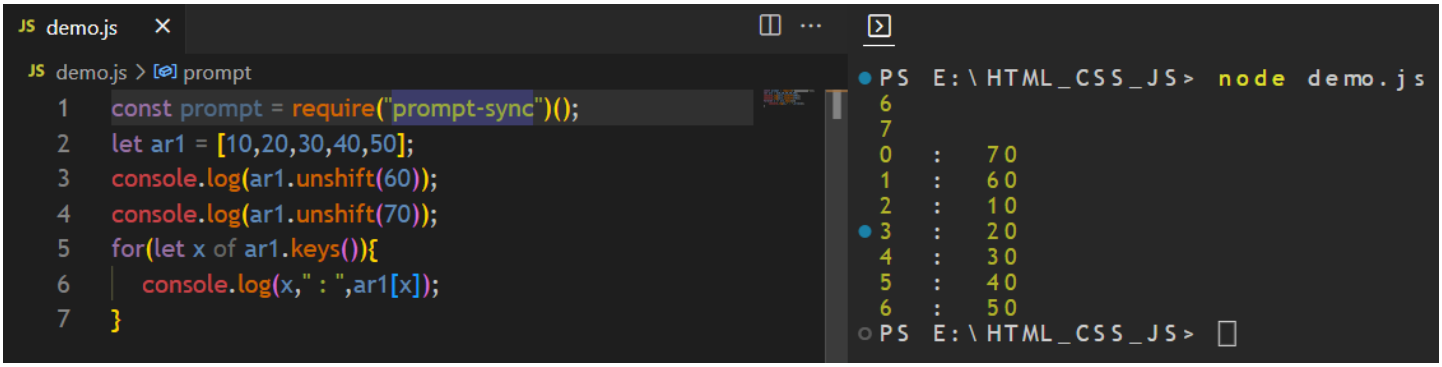
shift():

this method removes the first item of an array. the shift() method changes the original array. the shift() method returns the shifted element.



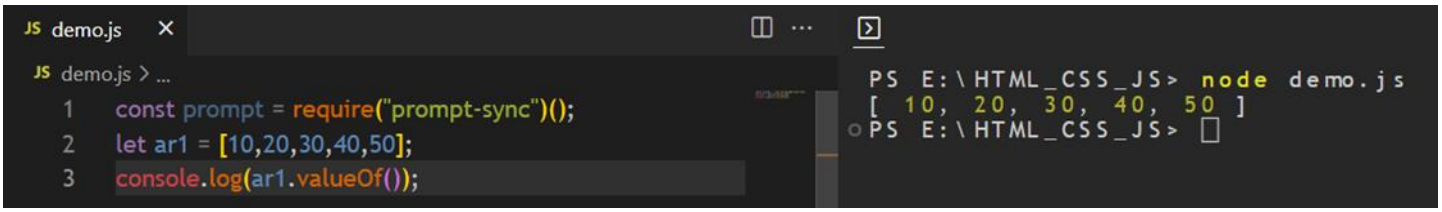
unshift():

this method adds new elements to the beginning of an array. the unshift() method overwrites the original array. if you print the method the array count starts from 1.



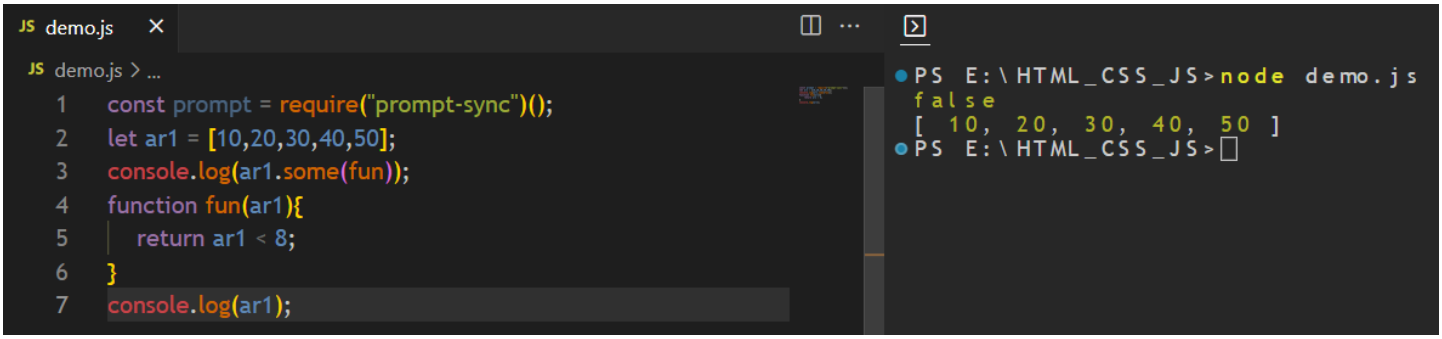
valueOf():

this method returns the itself. This method does not change the original array.



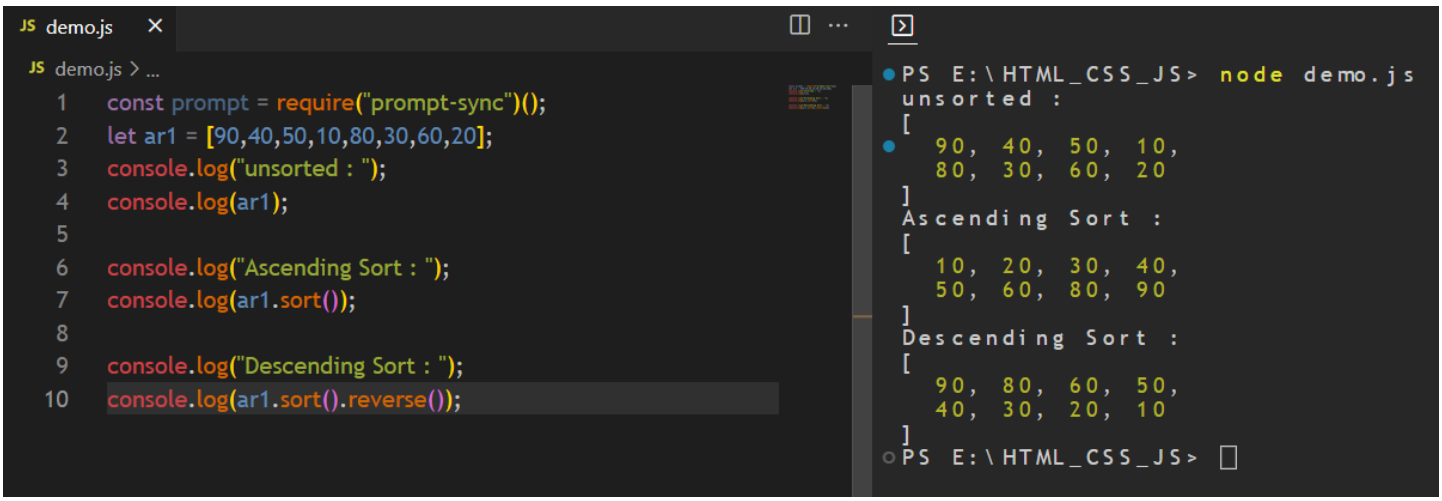
some():

this function checks if any array elements pass a test provides as a callback function. this function method executes the callback function once for each array element. The some() method returns true of the function returns true for one of the array elements. This function returns false. If the function returns false for all of the array elements. The some() method does not execute the function for empty array elements. The some() method does not change the original array.



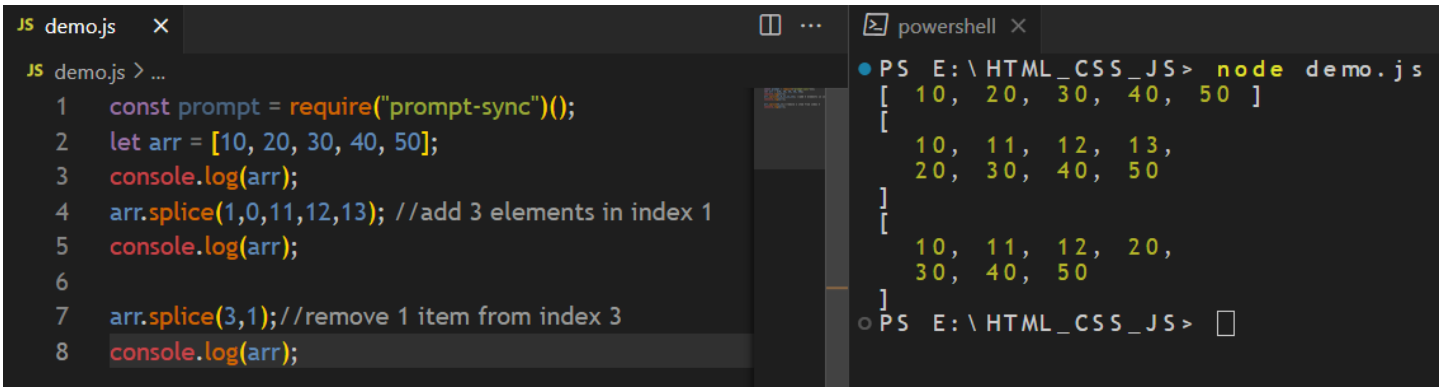
sort():

the method sort the element of array. this method overwrites the original array. the sort() sorts the elements as strings in alphabetical and ascending order.



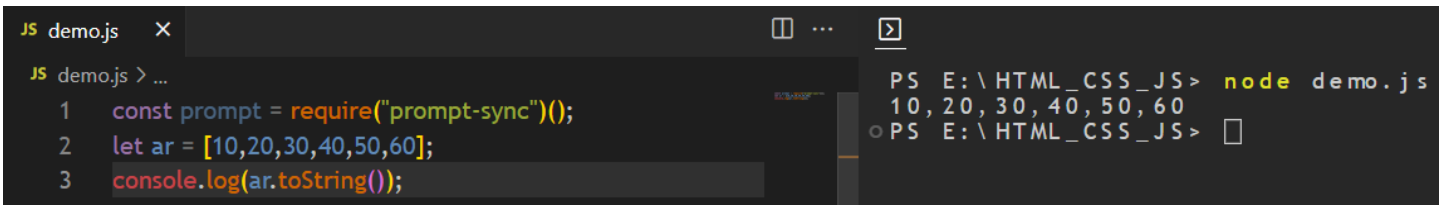
splice():

this method adds and/or removes array elements. The splice() method overwrites the original array.

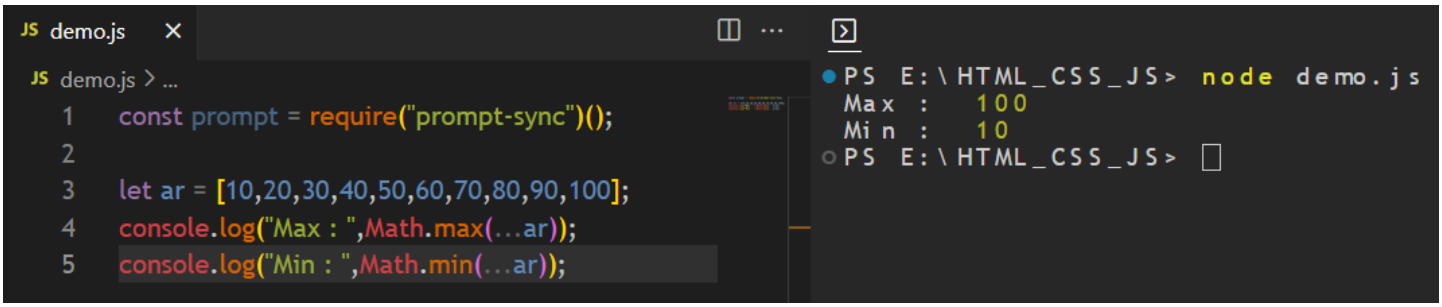


toString():

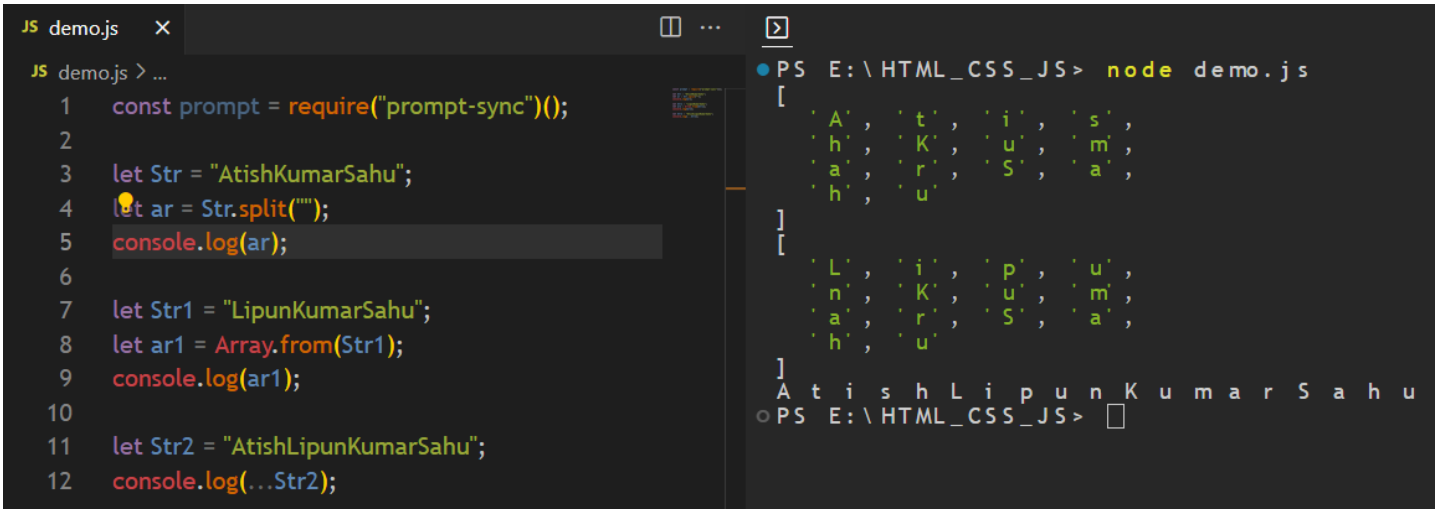
this method returns as string with array values separated by commas, this method does not change the original array.



Max & Min In Array:

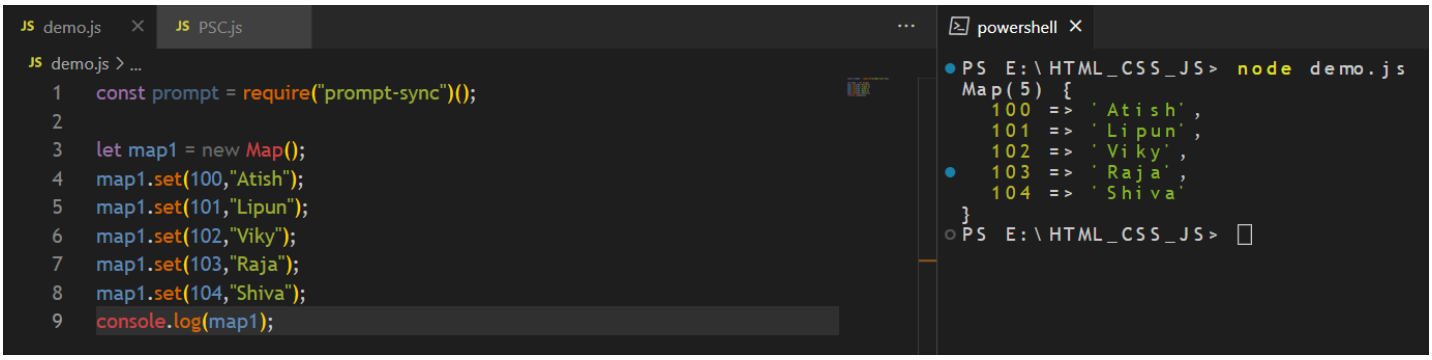


Array & String Conversion:



Map In JavaScript:

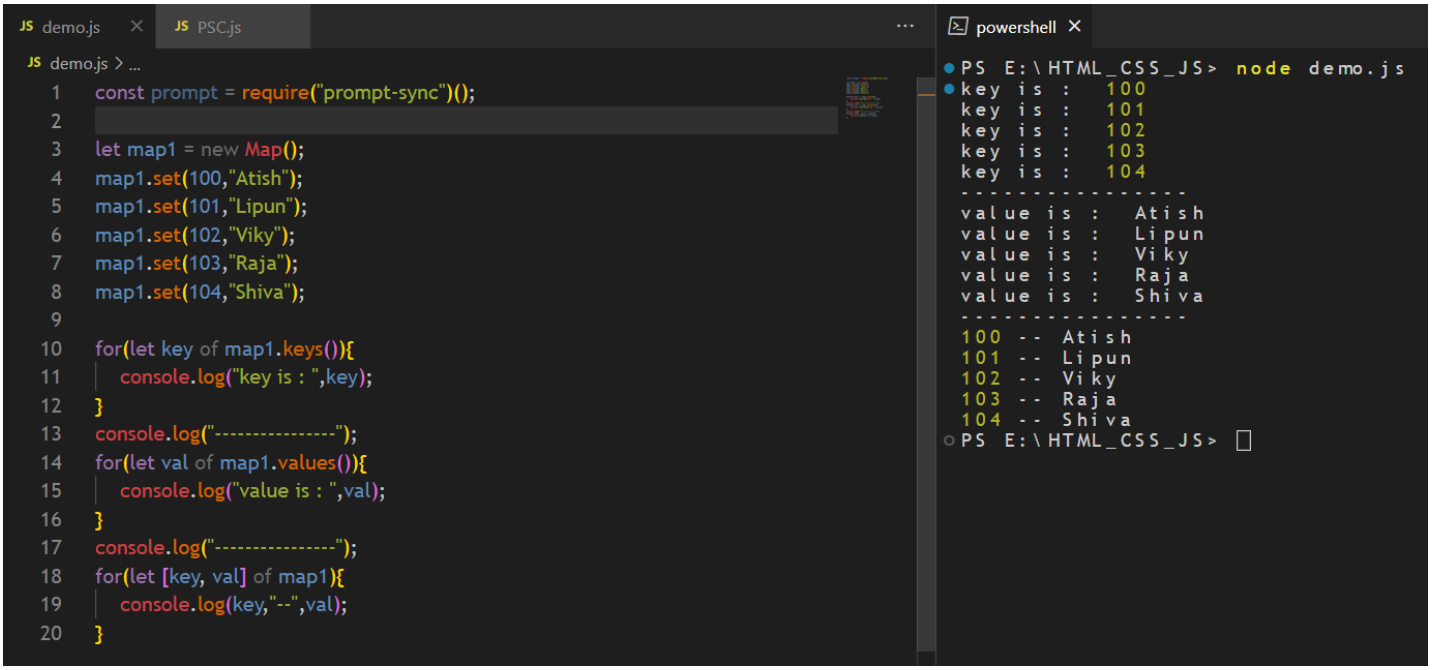
A Map holds key-value pairs where the keys can be any datatype. A Map remembers the original insertion order of the keys. A Map has a property that represents the size of the map.



```
const prompt = require("prompt-sync")();

let map1 = new Map();
map1.set(100,"Atish");
map1.set(101,"Lipun");
map1.set(102,"Viky");
map1.set(103,"Raja");
map1.set(104,"Shiva");
console.log(map1);
```

```
PS E:\HTML_CSS_JS> node demo.js
Map(5) {
  100 => 'Atish',
  101 => 'Lipun',
  102 => 'Viky',
  103 => 'Raja',
  104 => 'Shiva'
}
PS E:\HTML_CSS_JS>
```

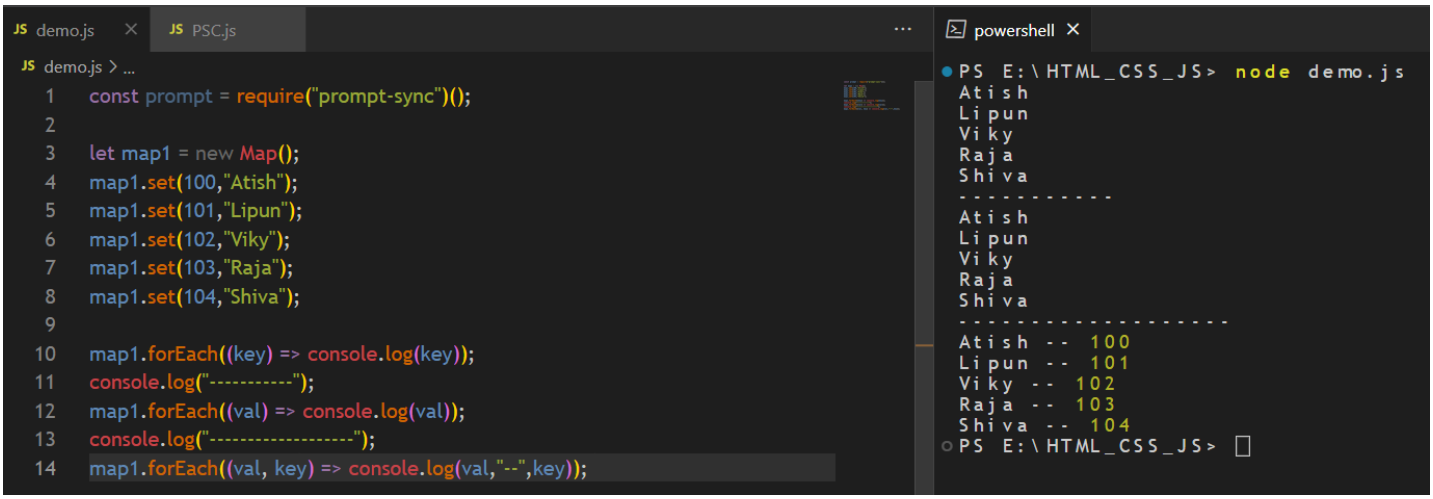


```
const prompt = require("prompt-sync")();

let map1 = new Map();
map1.set(100,"Atish");
map1.set(101,"Lipun");
map1.set(102,"Viky");
map1.set(103,"Raja");
map1.set(104,"Shiva");

for(let key of map1.keys()){
    console.log("key is : ",key);
}
console.log("----------------");
for(let val of map1.values()){
    console.log("value is : ",val);
}
console.log("----------------");
for(let [key, val] of map1){
    console.log(key,"--",val);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
key is :  100
key is :  101
key is :  102
key is :  103
key is :  104
----------------
value is :  Atish
value is :  Lipun
value is :  Viky
value is :  Raja
value is :  Shiva
----------------
100 -- Atish
101 -- Lipun
102 -- Viky
103 -- Raja
104 -- Shiva
PS E:\HTML_CSS_JS>
```



```
const prompt = require("prompt-sync")();

let map1 = new Map();
map1.set(100,"Atish");
map1.set(101,"Lipun");
map1.set(102,"Viky");
map1.set(103,"Raja");
map1.set(104,"Shiva");

map1.forEach((key) => console.log(key));
console.log("----------");
map1.forEach((val) => console.log(val));
console.log("--------------------");
map1.forEach((val, key) => console.log(val,"--",key));
```

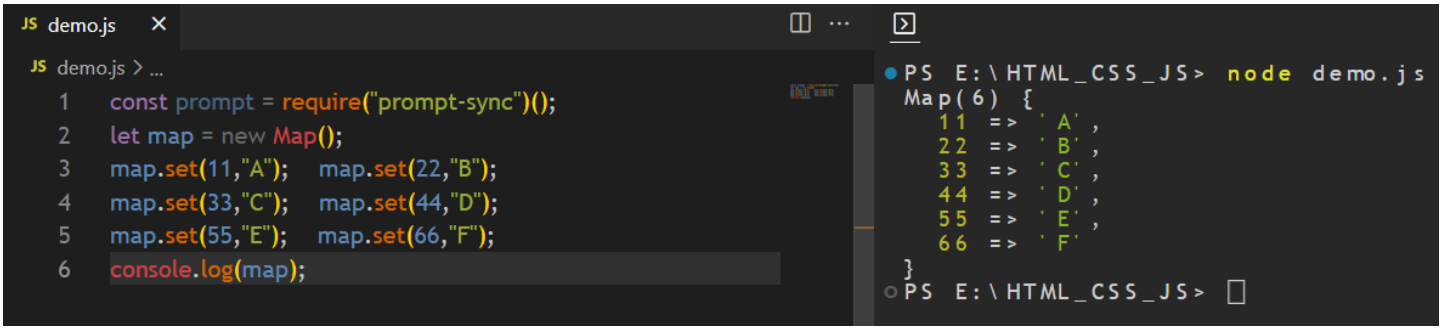
```
PS E:\HTML_CSS_JS> node demo.js
Atish
Lipun
Viky
Raja
Shiva
----------
Atish
Lipun
Viky
Raja
Shiva
--------------------
Atish -- 100
Lipun -- 101
Viky -- 102
Raja -- 103
Shiva -- 104
PS E:\HTML_CSS_JS>
```

Map Methods:

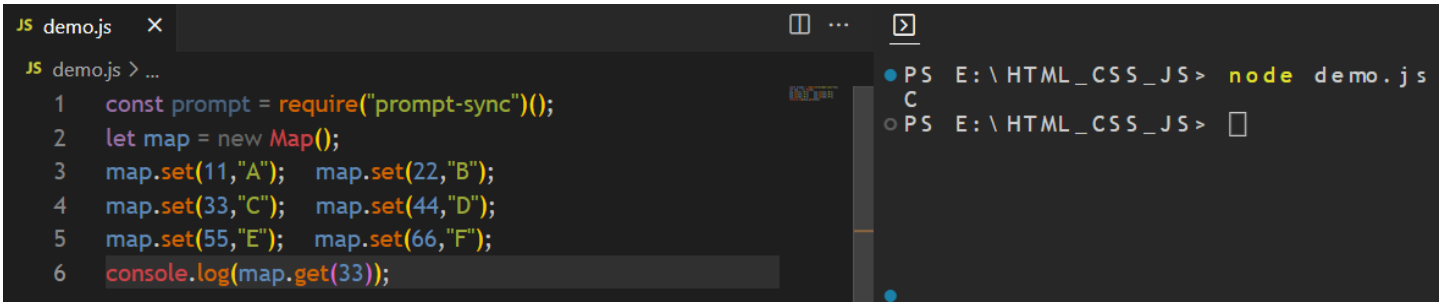
new Map(): this is used for to create a new Map Object.

```
JS demo.js > ...
1   const prompt = require("prompt-sync")();
2   let map = new Map();
```

Set(): it sets the value for a key in a map.



gets(): get the value for a key in a map.

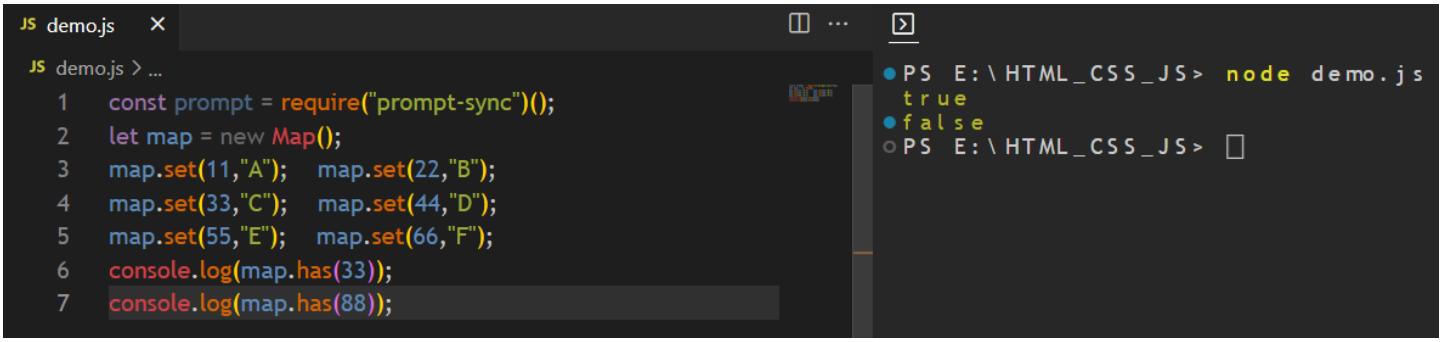


delete(): it removes a map elemssent specified by a key.

```
JS demo.js
JS demo.js > ...
1   const prompt = require("prompt-sync")();
2   let map = new Map();
3   map.set(11,"A");   map.set(22,"B");
4   map.set(33,"C");   map.set(44,"D");
5   map.set(55,"E");   map.set(66,"F");
6   console.log(map.delete(55));
7   console.log(map);
```

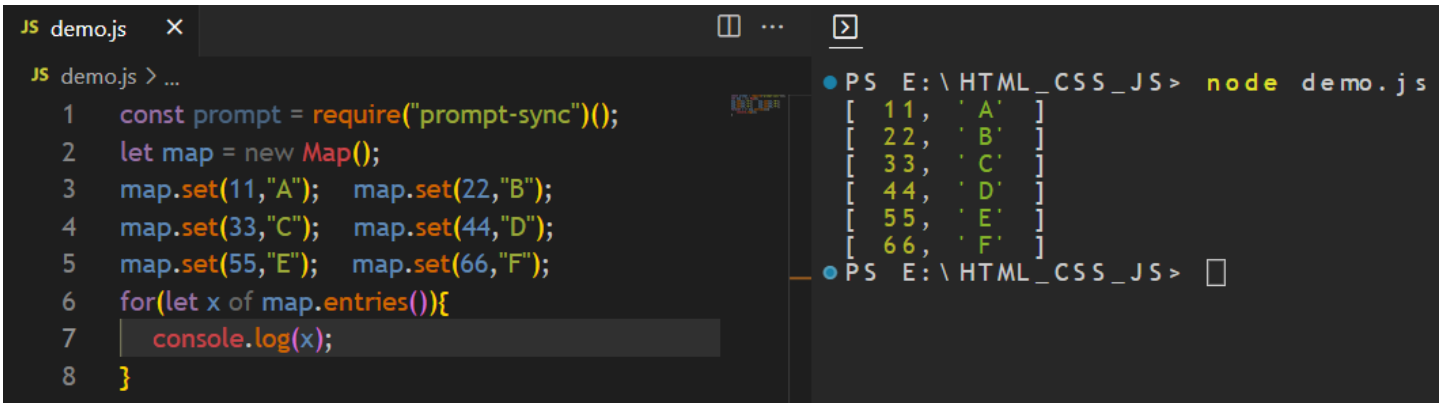
```
PS E:\HTML_CSS_JS> node demo.js
true
Map(5) { 11 => 'A', 22 => 'B', 33 => 'C', 44 => 'D', 66 => 'F' }
PS E:\HTML_CSS_JS>
```

has(): it returns true if a key is exist in map.

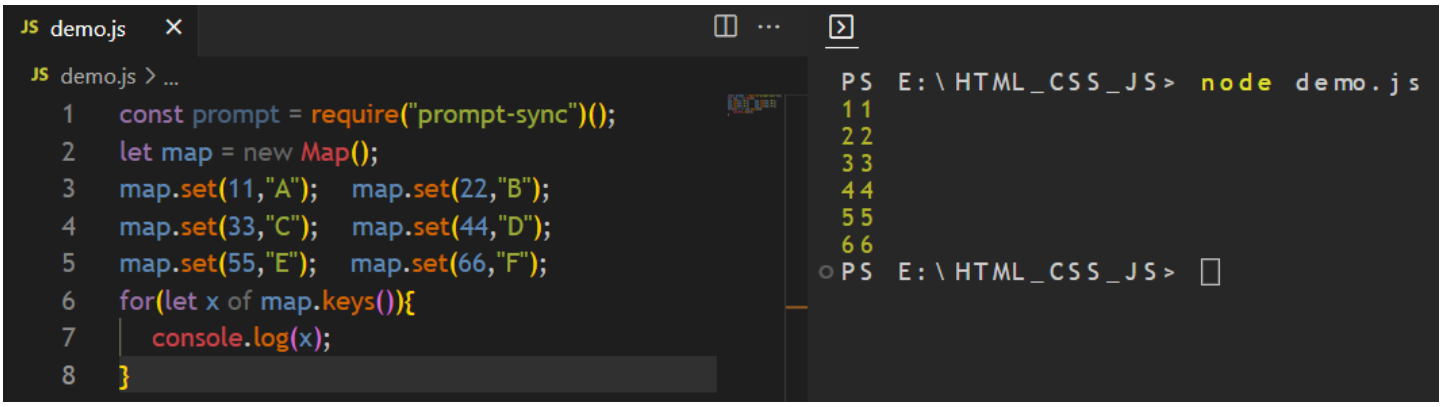


forEach(): it invokes a callback for each key/value pain in a map.

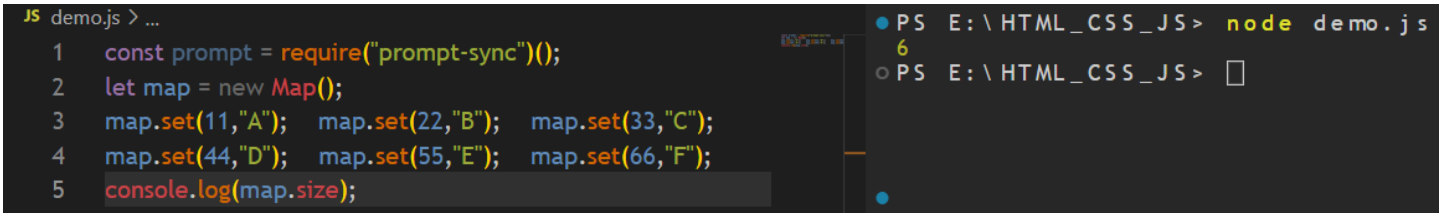
entries(); it return an iterator object with [key, value] pairs in map.



keys() this method returns an iterator object with the keys in a map.

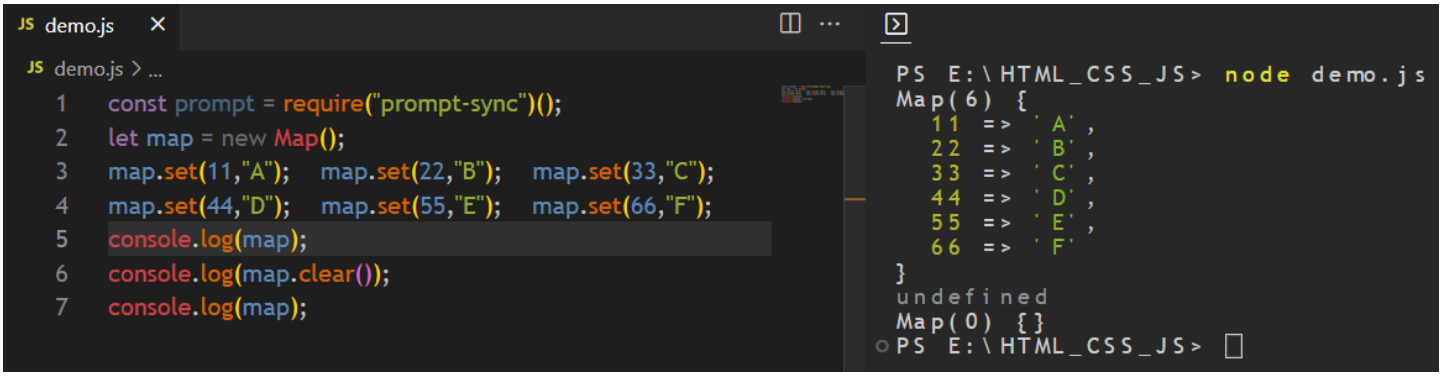


size: to know the size of map.



values(): this method to return an iterator with the values in map.

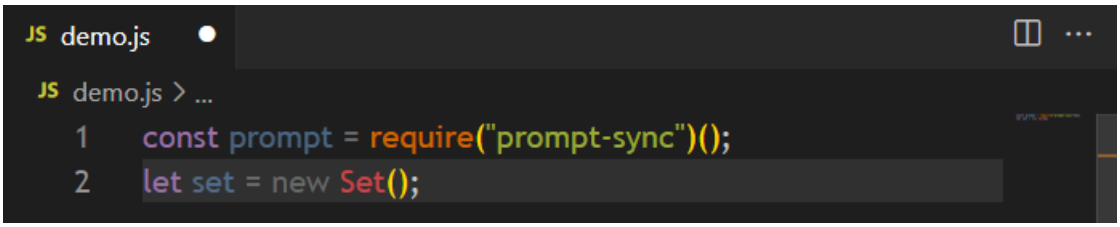
clear(): this method used to remove all the elements from the map.



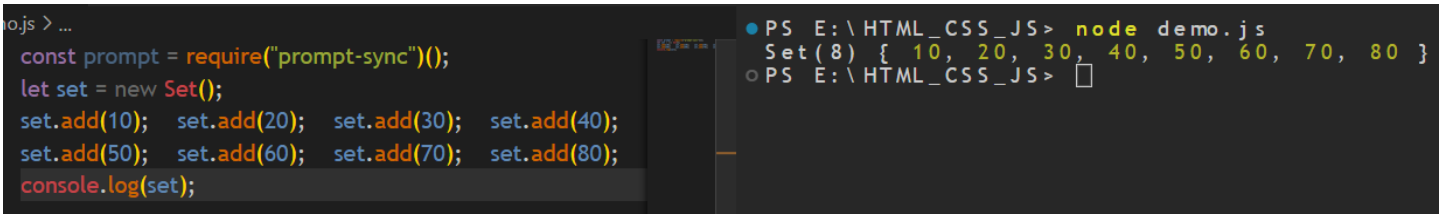
Set In JavaScript:

A JavaScript Set is a collection of unique values. Each value can only occur once in a Set. A Set can hold any value of any data type.

new Set() this is used to create a new Set Object.

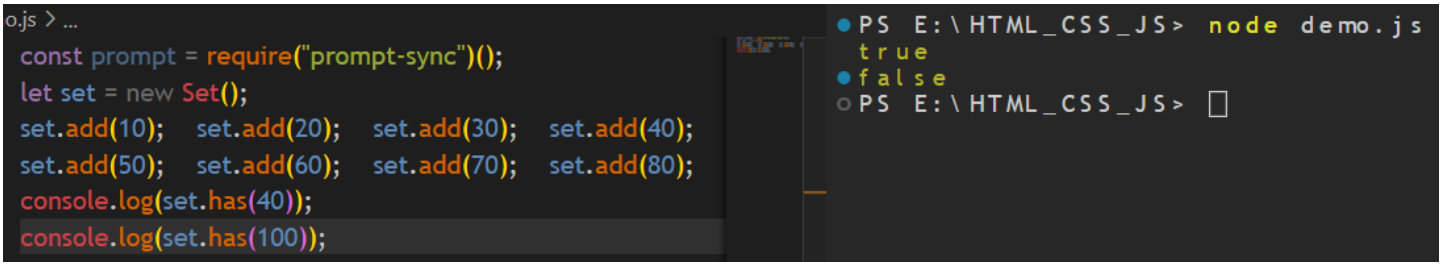


add() this is used for to add a new element to set.

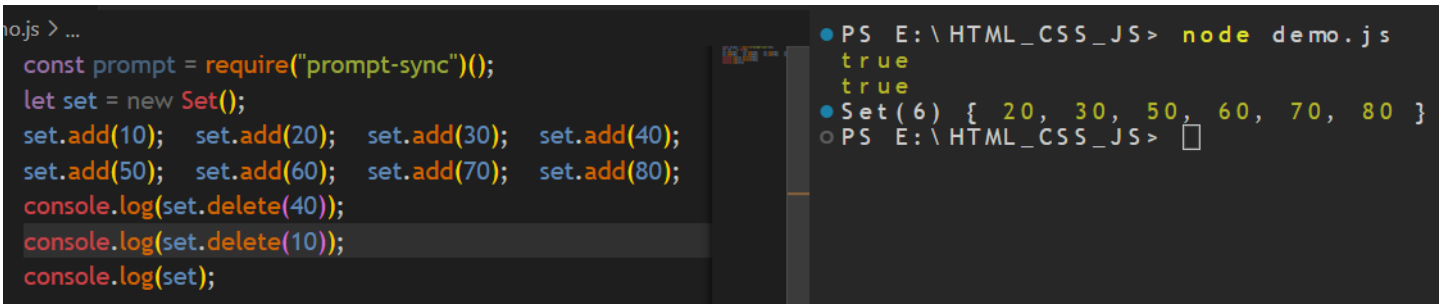


size(): it returns the number elements in a set.

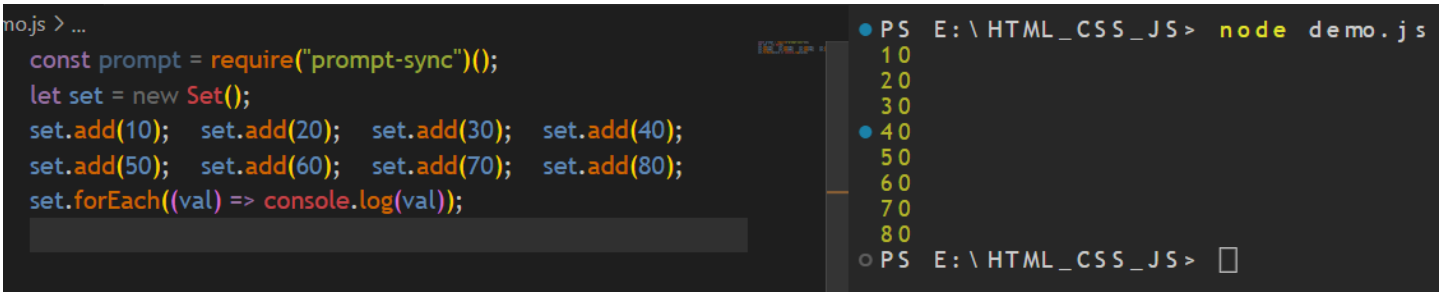
has(): it return true if a value exists.



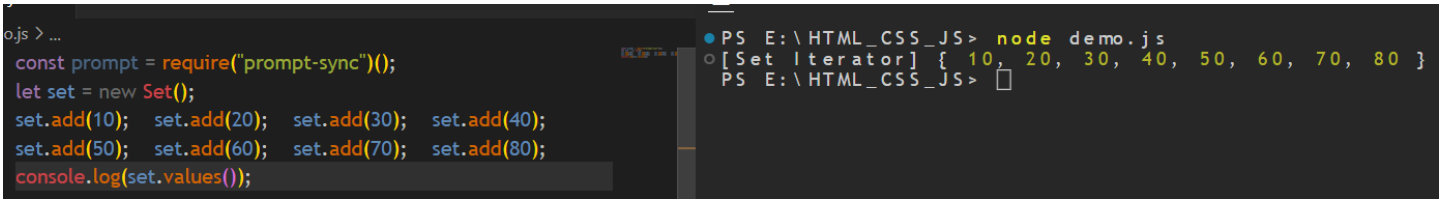
delete(): removes an element from a set.



forEach(): it invokes a callback for each element



values(): returns as an iterator with all the values in a set.



keys(): same as values.

entries(): it returns an iterator with the [value value] pairs from a set.

```
.js > ...
const prompt = require("prompt-sync")();
let set = new Set();
set.add(10);   set.add(20);   set.add(30);   set.add(40);
set.add(50);   set.add(60);   set.add(70);   set.add(80);
console.log(set.entries());
```

```
PS E:\HTML_CSS_JS> node demo.js
[Set Entries] {
  [ 10, 10 ],
  [ 20, 20 ],
  [ 30, 30 ],
  [ 40, 40 ],
  [ 50, 50 ],
  [ 60, 60 ],
  [ 70, 70 ],
  [ 80, 80 ]
}
PS E:\HTML_CSS_JS>
```

clear(): removes all elements from a set.

```
.js > ...
const prompt = require("prompt-sync")();
let set = new Set();
set.add(10);   set.add(20);   set.add(30);   set.add(40);
set.add(50);   set.add(60);   set.add(70);   set.add(80);
console.log(set.clear());
console.log(set);
```

```
PS E:\HTML_CSS_JS> node demo.js
undefined
Set(0) {}
PS E:\HTML_CSS_JS>
```

Array Destructuring:

```
JS PSC.js   JS demo.js
JS demo.js > ...
1   const prompt = require("prompt-sync")();
2   let arr1 = ["BookName",100, 20, 5000, "India"];
3   console.log(arr1);
4   console.log(arr1[0]);
5   let [banme, page, price, num, place,value1] = arr1;
6   console.log(page);
7   console.log(value1);
8
9   let raa2 = ["book",,100,200];
10  console.log(raa2);
11  let[name, val1 = 1000, val2, val3] = raa2;
12  console.log(val1);
13  console.log(val3);
14
15  let raa3 = [100, 200, 300, ["Atish","Lipun", "manish"], 500];
16  console.log(raa3);
17  let[v1, v2, v3,[s1, s2, s3],v5] = raa3;
18  console.log(v3);
19  console.log(s2);
20
21  function ar(){
22      return [100, 200];
23  }
24  let[al1, al2] = ar();
25  console.log(al2);
```

```
PS E:\HTML_CSS_JS> node demo.js
[ 'BookName', 100, 20, 5000, 'India' ]
BookName
100
undefined
[ 'book', <1 empty item>, 100, 200 ]
1000
200
[ 100, 200, 300, [ 'Atish', 'Lipun', 'manish' ], 500 ]
300
Lipun
200
PS E:\HTML_CSS_JS>
```

```
JS demo.js
JS demo.js > ...
1   const prompt = require("prompt-sync")();
2   //insert element in an array and print the array
3   let size = parseInt(prompt("Enter Size Value : "));
4   let ar = [];
5   for(let i = 0; i < size; i++){
6       let ele = parseInt(prompt("Enter Element : "));
7       ar.push(ele);
8   }
9   for(let j of ar.keys()){
10      console.log(j," : ",ar[j]);
11  }
```

```
powershell
PS E:\HTML_CSS_JS> node demo.js
Enter Size Value : 5
Enter Element : 1
Enter Element : 20
Enter Element : 3
Enter Element : 40
Enter Element : 5
0   :   1
1   :   20
2   :   3
3   :   40
4   :   5
PS E:\HTML_CSS_JS>
```

Map Function:

```
589  /*Map Function*/
590  let ar11 = [10,20,30,40,50,60,70,80,90,100];
591  console.log(ar11);
592  let ar12 = ar11.map(function(val){
593      return val * 10;
594  });//anonymous function
595  console.log(ar12);
596  let ar13 = ar11.map((val) => val * 20);//arrow function.
597  console.log(ar13);
```

Filter Function:

```
599  /*Filter Function*/
600  let ar20 = [2,22,13,5,6,61,23,7];
601  console.log(ar20);
602  let ar21 = ar20.filter(val => val > 10);
603  console.log(ar21);
604  let ar22 = ar20.filter(function(val){
605      return val < 10;
606  });
607  console.log(ar22);
```

```
609  let ar33 = [
610      {
611          name : "name1",
612          position: "developer"
613      },
614      {
615          name : "name2",
616          position: "Manager"
617      },
618      {
619          name : "name3",
620          position: "CEO"
621      },
622      {
623          name : "name4",
624          position: "developer"
625      },
626      {
627          name : "name5",
628          position: "developer"
629      },
630      {
631          name : "name6",
632          position: "Manager"
633      },
634      {
635          name : "name7",
636          position: "H.R."
637      },
638      {
639          name : "name8",
640          position: "Manager"
641      }
642  ];
643  console.log(ar33);
644  let ar44 = ar33.filter(val => val.position == "developer");
645  console.log(ar44);
```

Specific Max & Min:

```
demo.js
const prompt = require("prompt-sync")();

let ar = [20, 50, 10 , 80 , 60 , 30 , 70, 40, 90];
let ar1 = ar.sort();
console.log(ar1);
console.log(ar1[ar1.length - 2]);


let ar2 = [20, 50, 10 , 80 , 60 , 30 , 70, 40, 90];
let ar3 = ar.sort().reverse();
console.log(ar3);
console.log(ar3[ar3.length - 2]);
```

```
PS E:\HTML_CSS_JS> node demo.js
[
  10, 20, 30, 40, 50,
  60, 70, 80, 90
]
80
[
  90, 80, 70, 60, 50,
  40, 30, 20, 10
]
20
PS E:\HTML_CSS_JS>
```

Stack Implementation:

```
demo.js > Stack > push
const prompt = require("prompt-sync")();

class Stack {
  constructor() {
    this.items = [];
  }
  // Add an element to the top of the stack
  push(element) {
    this.items.push(element);
  }
  // Remove and return the top element of the stack
  pop() {
    if (this.isEmpty()) {
      return "Stack is empty";
    }
    return this.items.pop();
  }
```

```
demo.js > Stack > constructor
  // Return the top element of the stack without removing it
  peek() {
    if (this.isEmpty()) {
      return "Stack is empty";
    }
    return this.items[this.items.length - 1];
  }
  // Check if the stack is empty
  isEmpty() {
    return this.items.length === 0;
  }
  // Display the elements of the stack
  display() {
    if (this.isEmpty()) {
      return "Stack is empty";
    }
    return this.items.join(" -> ");
  }
}
```

```
// Create a stack instance
const stack = new Stack();
// Function to take user input using prompt
function getUserInput() {
  while (true) {
    console.log( "Choose an operation:\n1. Push\n2. Pop\n3. Peek\n4. Is Empty?\n5. Display\n6. Exit\n");
    let choice = prompt("");
    switch (choice) {
      case "1":
        const element = prompt("Enter the element to push:");
        stack.push(element);
        break;
      case "2":
        const poppedElement = stack.pop();
        console.log(`Popped element: ${poppedElement}`);
        break;
      case "3":
        const topElement = stack.peek();
        console.log(`Top element: ${topElement}`);
        break;
      case "4":
        const isEmpty = stack.isEmpty();
        console.log(`Is the stack empty? ${isEmpty ? "Yes" : "No"}`);
        break;
      case "5":
        const stackElements = stack.display();
        console.log(`Stack: ${stackElements}`);
        break;
      case "6":
        return; // Exit the program
      default:
        console.log("Invalid choice. Please try again.");
    }
  }
}
// Start taking user input
getUserInput();
```

Queue Implementation In JS:

JS demo.js > ...

```
class Queue {
  constructor() {
    this.items = [];
  }

  // Add an element to the back of the queue
  push(element) {
    this.items.push(element);
  }

  // Remove and return the front element of the queue
  pop() {
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    return this.items.shift();
  }

```

```
    // Return the front element of the queue without removing it
    peek() {
      if (this.isEmpty()) {
        return "Queue is empty";
      }
      return this.items[0];
    }

    // Check if the queue is empty
    isEmpty() {
      return this.items.length === 0;
    }

    // Display the elements of the queue
    display() {
      if (this.isEmpty()) {
        return "Queue is empty";
      }
      return this.items.join(" -> ");
    }
  }
```

demo.js > getUserInput

```
// Create a queue instance
const queue = new Queue();

// Function to take user input using prompt
function getUserInput() {
  while (true) {
    const choice = prompt(
      "Choose an operation:\n1. Enqueue (Push)\n2. Dequeue (Pop)\n3. Peek\n4. Is Empty?\n5. Display\n6. Exit"
    );

    switch (choice) {
      case "1":
        const element = prompt("Enter the element to enqueue:");
        queue.push(element);
        break;
      case "2":
        const dequeuedElement = queue.pop();
        alert(`Dequeued element: ${dequeuedElement}`);
        break;
      case "3":
        const frontElement = queue.peek();
        alert(`Front element: ${frontElement}`);
        break;
      case "4":
        const isEmpty = queue.isEmpty();
        alert(`Is the queue empty? ${isEmpty ? "Yes" : "No"}`);
        break;
      case "5":
        const queueElements = queue.display();
        alert(`Queue: ${queueElements}`);
        break;
      case "6":
        return; // Exit the program
      default:
        alert("Invalid choice. Please try again.");
    }
  }
}
// Start taking user input
getUserInput();
```

Linked List:

demo.js > ...

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
  }

  // Add an element to the end of the linked list
  push(element) {
    const newNode = new Node(element);
    if (!this.head) {
      this.head = newNode;
    } else {
      let current = this.head;
      while (current.next) {
        current = current.next;
      }
      current.next = newNode;
    }
  }

  // Remove and return the last element of the linked list
  pop() {
    if (!this.head) {
      return "Linked list is empty";
    }
    if (!this.head.next) {
      const data = this.head.data;
      this.head = null;
      return data;
    }
    let current = this.head;
    let prev = null;
    while (current.next) {
      prev = current;
      current = current.next;
    }
    prev.next = null;
    return current.data;
  }

  // Return the last element of the linked list without removing it
  peek() {
    if (!this.head) {
      return "Linked list is empty";
    }
    let current = this.head;
    while (current.next) {
      current = current.next;
    }
    return current.data;
  }
```

```
    // Check if the linked list is empty
    isEmpty() {
      return !this.head;
    }

    // Display the elements of the linked list
    display() {
      if (!this.head) {
        return "Linked list is empty";
      }
      const elements = [];
      let current = this.head;
      while (current) {
        elements.push(current.data);
        current = current.next;
      }
      return elements.join(" -> ");
    }
  }
```

```
  // Create a linked list instance
  const linkedList = new LinkedList();
  // Function to take user input using prompt
  function getUserInput() {
    while (true) {
      const choice = prompt(
        "Choose an operation:\n1. Push\n2. Pop\n3. Peek\n4. Is Empty?\n5. Display\n6. Exit"
      );
      switch (choice) {
        case "1":
          const element = prompt("Enter the element to push:");
          linkedList.push(element);
          alert(`Pushed element: ${element}`);
          break;
        case "2":
          const poppedElement = linkedList.pop();
          alert(`Popped element: ${poppedElement}`);
          break;
```

```
        case "3":
          const lastElement = linkedList.peek();
          alert(`Last element: ${lastElement}`);
          break;
        case "4":
          const isEmpty = linkedList.isEmpty();
          alert(`Is the linked list empty? ${isEmpty ? "Yes" : "No"}`);
          break;
        case "5":
          const listElements = linkedList.display();
          alert(`Linked List: ${listElements}`);
          break;
        case "6":
          return; // Exit the program
        default:
          alert("Invalid choice. Please try again.");
      }
    }
  }
  // Start taking user input
  getUserInput();
```