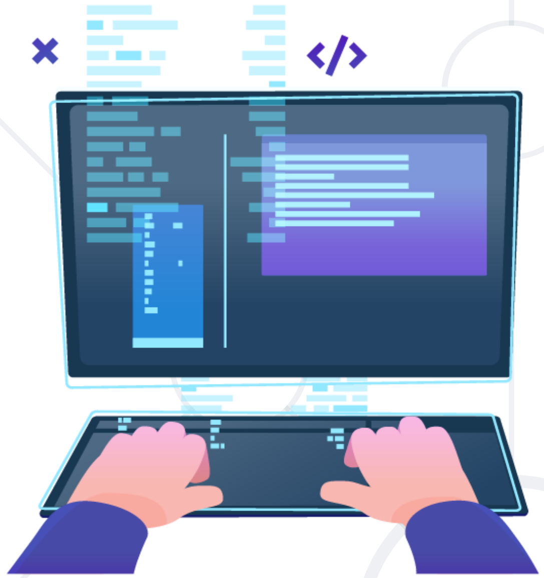
JS Fundamentals

Arrays, Strings, Objects and Associative Arrays



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

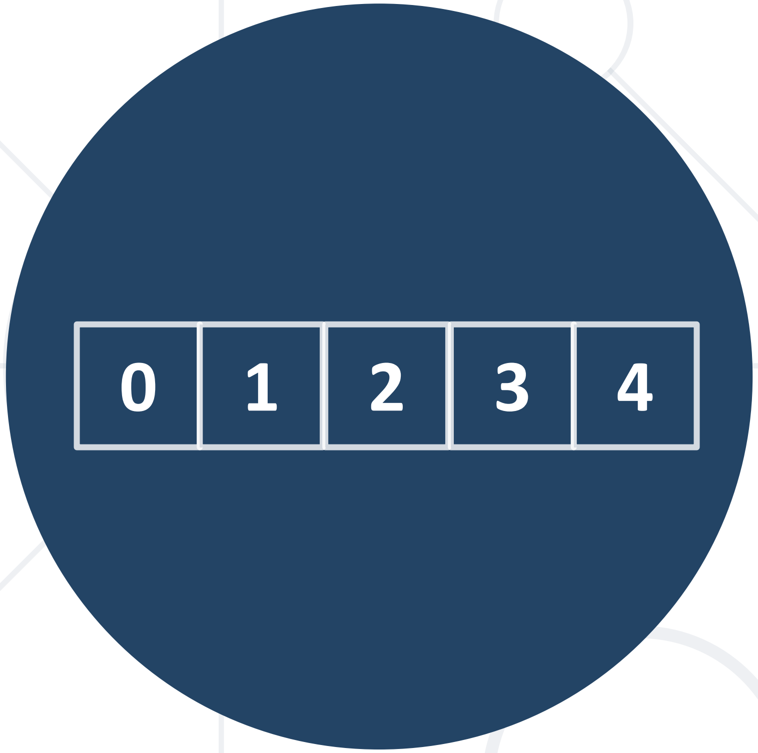
You Have Questions?

Software University

sli.do

#QA-Auto-BackEnd

Table of Contents



Working with Arrays of Elements

Arrays in JavaScript

Arrays in JavaScript

- Neither the **length** of a JavaScript array **nor** the **types** of its elements are **fixed**
- An array's **length can be changed** at any time
- Data can be stored at non-contiguous locations in the array
- JavaScript arrays are not guaranteed to be dense

Creating Arrays in JavaScript

- Array **literal**

```
let myArray = ["John Doe", 24, true];
```

```
let myArray = [];
myArray[0] = "John Doe";
myArray[1] = 24;
myArray[2] = true;
```

- Array **constructor**

```
let myArray = new Array("John Doe", 24, true);
```

Accessing Elements

- Array elements are accessed using their **index**

```
let cars = ['BMW', 'Audi', 'Opel'];
let firstCar = cars[0];   // BMW
let lastCar = cars[cars.length - 1];  // Opel
```

- Accessing indexes that do not exist in the array returns **undefined**

```
console.log(cars[3]);   // undefined
console.log(cars[-1]);  // undefined
```

Destructuring Syntax

- Expression that **unpacks values** from **arrays** or **objects**, into distinct **variables**

```
let numbers = [10, 20, 30, 40, 50];
let [a, b, ...elems] = numbers;

console.log(a) // 10
console.log(b) // 20
console.log(elems) // [30, 40, 50]
```

Rest operator

- The **rest operator** can also be used to collect function parameters into an array

Array's Methods

Modify the Array

Pop

- Removes the **last element** from an array and returns that element
- This method **changes** the **length** of the array

```
let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.pop());  // 70
console.log(nums.length); // 6
console.log(nums);        // [ 10, 20, 30, 40, 50, 60 ]
```

Push

- The **push()** method **adds one or more** elements to the **end** of an array and **returns** the new **length** of the array

```
let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length);    // 7
console.log(nums.push(80)); // 8 (nums.length)
console.log(nums); // [ 10, 20, 30, 40, 50, 60, 70, 80 ]
```

Shift

- The `shift()` method **removes** the **first element** from an array and **returns** that **removed element**
- This method **changes** the **length** of the array

```
let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.shift()); // 10 (removed element)
console.log(nums);  // [ 20, 30, 40, 50, 60, 70 ]
```

Unshift

- The **unshift()** method **adds one or more** elements to the **beginning** of an array and **returns** the new **length** of the array

```
let nums = [40, 50, 60];
console.log(nums.length);        // 3
console.log(nums.unshift(30));  // 4 (nums.length)
console.log(nums.unshift(10,20)); // 6 (nums.length)
console.log(nums);  // [ 10, 20, 30, 40, 50, 60 ]
```

Splice

- Changes the contents of an array by **removing** or **replacing** existing **elements** and / or **adding new** elements

```
let nums = [1, 3, 4, 5, 6];
nums.splice(1, 0, 2);        // inserts at index 1
console.log(nums);           // [ 1, 2, 3, 4, 5, 6 ]
nums.splice(4, 1, 19);       // replaces 1 element at index 4
console.log(nums);           // [ 1, 2, 3, 4, 19, 6 ]
let el = nums.splice(2, 1);  // removes 1 element at index 2
console.log(nums);           // [ 1, 2, 4, 19, 6 ]
console.log(el);             // [ 3 ]
```

Reverse

- Reverses the array
- The **first** array element becomes the **last**, and the **last** array element becomes the **first**

```
let arr = [1, 2, 3, 4];
arr.reverse();
console.log(arr); // [ 4, 3, 2, 1 ]
```

Join

- Creates and returns a **new string** by **concatenating** all of the elements in an array (or an array-like object), **separated** by commas or a **specified separator** string

```
let elements = ['Fire', 'Air', 'Water'];
console.log(elements.join());   // "Fire,Air,Water"
console.log(elements.join('')); // "FireAirWater"
console.log(elements.join('-')); // "Fire-Air-Water"
console.log(['Fire'].join(".")); // Fire
```

Slice

- The **`slice()`** method **returns** a shallow **copy** of a **portion** of an array into a **new array** object selected from begin to end (end not included)
- The **original array** will **not** be **modified**

```
let fruits = ['Banana', 'Orange', 'Lemon', 'Apple'];
let citrus = fruits.slice(1, 3);
let fruitsCopy = fruits.slice();
// fruits contains ['Banana', 'Orange', 'Lemon', 'Apple']
// citrus contains ['Orange', 'Lemon']
```

Includes

- Determines whether an array contains a certain element, returning **true** or **false** as appropriate

```
// array length is 3
// fromIndex is -100
// computed index is 3 + (-100) = -97
let arr = ['a', 'b', 'c'];
arr.includes('a', -100); // true
arr.includes('b', -100); // true
arr.includes('c', -100); // true
arr.includes('a', -2); // false
```

IndexOf

- The **indexOf()** method **returns** the **first index** at which a given **element** can be **found** in the array
  - Output is **-1** if element is **not present**

```
const beasts = ['ant', 'bison', 'camel', 'duck', 'bison'];

console.log(beasts.indexOf('bison')); // 1
// start from index 2
console.log(beasts.indexOf('bison', 2)); // 4
console.log(beasts.indexOf('giraffe')); // -1
```

ForEach

- The **forEach()** method **executes a provided function** once for each array element

```
const items = ['item1', 'item2', 'item3'];
const copy = [];

// For loop
for (let i = 0; i < items.length; i++) {
  copy.push(items[i]);
}

// ForEach
items.forEach(item => { copy.push(item); });
```

Map

- **Creates a new array** with the results of calling a **provided function** on every element in the calling array

```
let numbers = [1, 4, 9];
let roots = numbers.map(function(num, i, arr) {
  return Math.sqrt(num)
});
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

Find

- Returns the **first found value** in the array, if an **element** in the array **satisfies** the **provided** testing **function** or **undefined** if not found

```
let array1 = [5, 12, 8, 130, 44];
let found = array1.find(function(element) {
  return element > 10;
});
console.log(found); // 12
```

Filter

- Creates a **new array** with **filtered elements only**
- Calls a **provided** callback **function** once for each element in an array
- **Does not mutate** the **array** on which it is called

```
let fruits = ['apple', 'banana', 'grapes', 'mango', 'orange'];
 // Filter array items based on search criteria (query)
function filterItems(arr, query) {
  return arr.filter(function(el) {
      return el.toLowerCase().indexOf(query.toLowerCase()) !== -1;
  });
};
console.log(filterItems(fruits, 'ap')); // ['apple', 'grapes']
```

abc

Manipulating Strings

Concatenating

- Use the "**+**" or the "**+=**" operators

```
let text = "Hello" + ", ";
// Expected output: "Hello, "
text += "JS!"; // "Hello, JS!"
```

- Use the **concat()** method

```
let greet = "Hello, ";
let name = "John";
let result = greet.concat(name);
console.log(result); // Expected output: "Hello, John"
```

Searching for Substrings

- `indexOf(substr)`

```
let str = "I am JavaScript developer";
console.log(str.indexOf("Java")); // Expected output: 5
console.log(str.indexOf("java")); // Expected output: -1
```

- `lastIndexOf(substr)`

```
let str = "Intro to programming";
let last = str.lastIndexOf("o");
console.log(last); // Expected output: 11
```

Extracting Substrings

- `substring(startIndex, endIndex)`

```
let str = "I am JavaScript developer";
let sub = str.substring(5, 10);
console.log(sub); // Expected output: JavaS
```

String Operations

- `replace(search, replacement)`

```
let text = "Hello, john@softuni.bg, you have been
using john@softuni.bg in your registration.";
let replacedText = text.replace(".bg", ".com");
console.log(replacedText);
// Hello, john@softuni.com, you have been using
john@softuni.bg in your registration.
```

Splitting and Finding

- split(separator)

```
let text = "I love fruits";
let words = text.split(' ');
console.log(words); // Expected output: ['I', 'love', 'fruits']
```

- includes(substr)

```
let text = "I love fruits.";
console.log(text.includes("fruits")); // Expected output: True
console.log(text.includes("banana")); // Expected output: False
```

Repeating Strings

- `repeat(count)` - Creates a new string repeated count times

```
let n = 3;
for(let i = 1; i <= n; i++) {
  console.log('*'.repeat(i));
}
```

```
// *
// **
// ***
```

Trimming Strings

- Use **trim()** method to remove **whitespaces** (spaces, tabs, no-break space, etc.) from **both ends** of a string

```
let text = "   Annoying spaces       ";
console.log(text.trim()); // Expected output: "Annoying spaces"
```

- Use **trimStart()** or **trimEnd()** to remove whitespaces **only** at the beginning or at the end

```
let text = "   Annoying spaces       ";
text = text.trimStart();
text = text.trimEnd();
console.log(text); // Expected output: "Annoying spaces"
```

Starts With/Ends with

- Use **startsWith()** to determine whether a string **begins** with the characters of a specified substring

```
let text = "My name is John";
console.log(text.startsWith('My')); // Expected output: true
```

- Use **endsWith()** to determine whether a string **ends** with the characters of a specified substring

```
let text = "My name is John";
console.log(text.endsWith('John')); // Expected output: true
```

Padding at the Start and End

- Use **padStart()** to add to the current string **another substring** at the **start** until a **length** is reached

Receives **length** and **substring**

```
let text = "010";
console.log(text.padStart(8, '0')); // Expected output: 00000010
```

- Use **padEnd()** to add to the current string **another substring** at the **end** until a **length** is reached

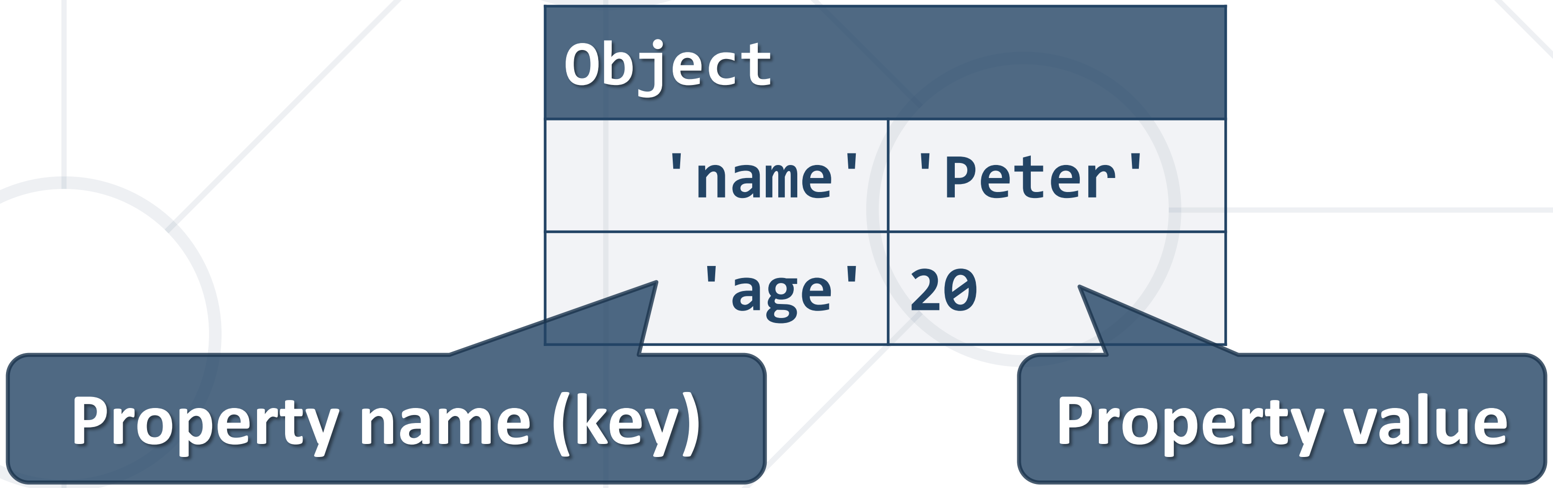
```
let sentence = "He passed away";
console.log(sentence.padEnd(20, '.'));
// Expected output: He passed away......
```

Objects

Definition, Properties and Methods

What Are Objects ?

- **Structure** of related data or functionality
- Contains **values** accessed by **string keys**
  - Data values are called **properties**
  - Function values are called **methods**



Object	
'name'	'Peter'
'age'	20

- You can **add** and **remove** properties **during runtime**

Object Definition

- We can create an object with an **object literal**

```
let person = { name:'Peter', age:20, height:183 };
```

- We can define an **empty object** and **add properties** later

```
let person = {};
person.name ='Peter';
person.age = 20;
person.hairColor = 'black';
```

```
person['lastName'] = 'Parker';
```

Methods of Objects

- Functions within a JavaScript object are called **methods**
- We can **define** methods using several syntaxes:

```
let person = {
  sayHello: function() {
     console.log('Hi, guys');
  }
}
```

```
let person = {
  sayHello() {
     console.log('Hi, guys');
  }
}
```

- We can **add** a method to an already defined object

```
let person = { name:'Peter', age: 20 };
person.sayHello = () => console.log('Hi, guys');
```

Built-in Method Library

- Get array of all property **names** (keys)

```
Object.keys(cat);
// ['name', 'age']
```

cat	
'name'	'Tom'
'age'	5

- Get array with of all property **values**

```
Object.values(cat);
// ['Tom', 5]
```

- Get and array of all properties as **key-value tuples**

```
Object.entries(cat);
// [['name', 'Tom'], ['age', 5]]
```

{a:1}

Associative Arrays

Storing Key-Value Pairs

What is an Associative Array ?

- Arrays indexed by **string keys**
- Hold a set of pairs **[key → value]**
  - The key is a **string**
  - The **value** can be of **any** type

Key	Value
`John Smith`	`+1-555-8976`
`Lisa Smith`	`+1-555-1234`
`Sam Doe`	`+1-555-5030`

Declaration

- An associative array in JavaScript is just an **object**
- We can declare it **dynamically**

```
let assocArr = {
   'one': 1,
   'two': 2,
   'three': 3,
   [key]: 6
};
```

```
assocArr['four'] = 4;
```

```
assocArr.five = 5;
```

```
let key = 'six';
assocArr[key] = 6;
```

Quotes are used if the key contains **special characters**

Valid ways to access **values** through **keys**

Using for – in

- We can use **for-in** loop to iterate through the keys

```
let assocArr = {};
assocArr['one'] = 1;
assocArr['two'] = 2;
assocArr['three'] = 3;

for(let key in assocArr) {
   console.log(key + " = " + assocArr[key]);
}
```

```
// one = 1
// two = 2
// three = 3
```

Manipulating Associative Arrays

- Check if a key is **present**

```
let assocArr = { /* entries */ };
if (assocArr.hasOwnProperty('John Smith')) { /* Key found */ }
```

- **Remove** entries

```
delete assocArr['John Smith'];
```

Sorting Associative Arrays

- Objects **cannot be sorted**, they must be converted first
  - Convert to **array** for **sorting**, **filtering** and **mapping**

```
let phonebook = { 'Tim': '0876566344',
                  'Bill': '0896543112' };

let entries = Object.entries(phonebook);
console.log(entries); // Array of arrays with two elements each
// [ ['Tim', '0876566344'],
//   ['Bill', '0896543112'] ]

let firstEntry = entries[0];
console.log(firstEntry[0]);  // Entry key -> 'Tim'
console.log(firstEntry[1]);  // Entry value -> '0876566344'
```

The entry is turned into an array of **[key, value]**

Sorting By Key

- The **entries** array can be **sorted**, using a **Compare function**
  - To **sort by key**, use the **first element** of each entry

```
entries.sort((a, b) => {
  keyA = a[0];
  keyB = b[0];
  // Perform comparison and return negative, 0 or positive
});
```

  - You can also **destructure** the entries

```
entries.sort(([keyA, valueA],[keyB, valueB]) => {
  // Perform comparison and return negative, 0 or positive
});
```

Array and Object Destructuring

- The **destructuring assignment** syntax makes it possible to unpack values from arrays, or properties from objects, into distinct variables
- On the left-hand side of the assignment to define what values to unpack from the sourced variable

```
const x = [1, 2, 3, 4, 5];
const [y, z] = x;
console.log(y); // 1
console.log(z); // 2
```

```
obj = { a: 1, b: 2 };
const { a, b } = obj;
// is equivalent to:
// const a = obj.a;
// const b = obj.b;
```

Sorting By Value

- To **sort by value**, use the **second element** of each entry

```
entries.sort((a, b) => {
  valueA = a[1];
  valueB = b[1];
  // Perform comparison and return negative, 0 or positive
});
```

- You can also **destructure** the entries

```
entries.sort(([keyA, valueA],[keyB, valueB]) => {
  // Perform comparison and return negative, 0 or positive
});
```

$f(x)$

Functions Overview

Definition and Objectives

Functions in JS

- A **function** is a **named subprogram** designed to perform a particular task
  - Functions are executed when they are called. This is known as **invoking** a function
  - Values can be **passed** into functions and used within the function

Use **camelCase**

Parameter

```
function printStars(count) {
  console.log("*".repeat(count));
}
```

Declaring Function

- Functions can be declared in two ways:
  - **Function declaration** (recommended way)

```
function printText(text) {
  console.log(text);
}
```

- Functions can have **parameters**
- Functions **always** return a value (custom or default)

Invoking a Function

- Functions are first **declared**, then **invoked** (many times)

```
function hLine() {
  console.log("-----------");
}
```

- Functions can be **invoked (called)** by their name

```
hLine();
```

- Invocation from another function

```
function printDocument() {
  printLabel();
}
```

Functions Without Parameters

- Does **not** receive arguments when invoked
- The result is **always the same** (unless it reads data from outside)

```
function printHeader() {
  console.log('~~~-   {@}   -~~~');
  console.log('~- Certificate -~');
  console.log('~~~-  ~---~  -~~~');
}
printHeader();   // Output is always the same
```

Functions With Parameters

- Can receive **any number** and **type** of arguments when invoked

```
function multiply(a, b) {
  console.log(a*b);
}
multiply(5, 7); // 35
```

Pass two numbers

```
function printName(nameArr) {
  console.log(nameArr[0] + ' ' + nameArr[1]);
}
printName(['John', 'Smith']); // John Smith
```

Pass array of strings

The Return Statement

- The **return** keyword immediately **stops the function's execution**
- **Returns** the specified value to the caller

```
function readFullName(firstName, lastName) {
  return firstName + " " + lastName;
}

const fullName = readFullName("John","Smith");
console.log(fullName) //John Smith
```

Using the Return Values

- Return value can be
  - **Assigned** to a variable

```
let max = getMax(5, 10);
```

  - **Used** in expression

```
let total = getPrice() * quantity * 1.20;
```

  - **Passed** to another function

```
multiply(getMax(5,10), 20)
```

Arrow Functions

- Special **shorthand syntax** for declaration
- They operate in the **context** of their **enclosing scope**
- Useful in **functional programming**

```
let increment = x => x + 1;
console.log(increment(5));  // 6
```

```
let increment = function(x) {
  return x + 1;
}
```

```
let sum = (a, b) => a + b;
console.log(sum(5, 6));  // 11
```

First-Class Functions

- **First-class functions** are treated like any other variable
  - Passed as an **argument**
  - **Returned** by another function
    - We can do that, because we treated functions in JavaScript as a **value**
  - Assigned as a **value** to a **variable**

```
const write = function () {
    return "Hello, world!";
}
```

Higher-Order Functions

- Can take **other functions as arguments**
- Can **return a function**

```
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(num => num * num);
console.log(squared); // Output: [1, 4, 9, 16, 25]
```

```
function greaterThan(n)
{
   return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11)); // Output: true
```

Summary

- Arrays in JS can hold mixed data
- Various methods for working with them
- Strings can be manipulated
- Objects == structure of related data
- Associative arrays == arrays indexed by key-value
- Functions in JS == named subprograms
  - Designed to perform particular tasks

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank

Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

License

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity