Class-2: Callbacks and HOF

Topics Covered

- HOF & Callbacks
- map , forEach
- filter
- reduce
- sort

Callbacks, HOF and Sorting

Basics of Functions:

Instructor Task:

Function Statement:

- The function statement declares a function. A declared function is “saved for later use”, and will be executed later, when it is invoked (called).

```
function javascript(){
        console.log("Welcome to JS")
}

javascript()
```

Function expression:

- Functions are like heart to JavaScript, beautiful feature of a function is that you can assign it to a variable.

```
var b = function(){
        console.log("Welcome to JS")
}

b()
```

Anonymous function :

- A function without a name is called anonymous function
- Anonymous functions are used as values , i.e. you can use it to assign it to some variables. In the above snippet the function which we assign to variable b is an anonymous function

Difference Between Parameters and Arguments :

- *Parameters* are variables listed as a part of the function definition.
- *Arguments* are values passed to the function when it is invoked.

```
function foo( a, b, c ) {}; // a, b, and c are the parameter
s

foo( 1, 2, 3 ); // 1, 2, and 3 are the arguments
```

Think of vaccination scenario

- Many People are queued up and as a doctor what would you tell your staff to do:
  1. Everyone has a token of their number in the line. You go with a token 0 and match it with the first person and vaccinate him, then you scratch the 0 on your token and make it 1, then go to next person and vaccinate, and so on.
  2. Go and vaccinate everyone in the line one by one.
- Lets take array of persons

```
var persons = ['Chandra', 'Varun', 'Nrupul', 'Prateek', 'Ama
n'];
```

- Let's write a function for vaccination

```
function vaccinate(person) {
    console.log(person + 'has been vaccinated.')
}
```

- Instead of going to each and every person, lets use for loop

```
for (var i = 0; i < persons.length; i++) {
vaccinate(persons[i]);
}
```

- You can also use `map` function instead of `for` loop

```
persons.map(vaccinate)
```

- **The surprising thing that happened is we passed a function name as an argument!**
- What is `vaccinate` here ? it's a callback function

Callback functions:

- **Callback is an (uninvoked) function passed to another (invoked) function as an argument.**

Instructor Task:

- Before explaining them what actually callback function is, let's ask them few basic questions on what we discussed.

```
function eatBreakfast(item){
    console.log("I will eat"+" "+item +" "+ "as my breakfast")
}

eatBreakfast("idly")
```

Students Task:

- Ask output of the above code snippet

Instructor Task:

- In the above code snippet we have passed string as argument.
- Now let's to pass number as argument along with string

```
function eatBreakfast(item,time){
    console.log("I will eat"+" "+item +" "+ "as my breakfas
t"+"at"+" "+time)
}

eatBreakfast("idly",9)
```

Students Task:

- Ask output of the above code snippet
- Also ask them can they pass functions as argument along with strings and numbers

```
function eatBreakfast(item,time){

    console.log("I will eat"+" "+item +" "+ "as my breakfas
t"+"at"+" "+time)
}


function doBrush(){
```

```
    console.log("First brush your teeth")
}

eatBreakfast("idly",9,doBrush)
```

- Ask students can we pass function as an argument?
- If yes, how can we access that as function parameter?

Instructor Task:

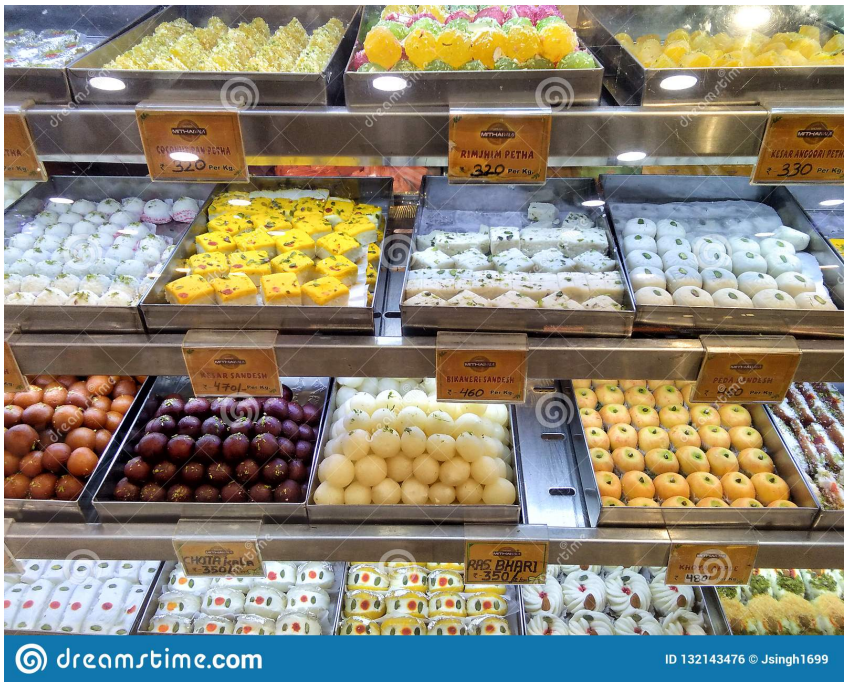
- Explain them on how to access callback function

```
function eatBreakfast(item,time,doBrush){
    doBrush()
    console.log("I will eat"+" "+item +" "+ "as my breakfas
t"+"at"+" "+time)
}


function doBrush(){
    console.log("First brush your teeth")
}

eatBreakfast("idly",9,doBrush)
```

HOF - Sweets Analogy

- Suppose you go to a sweet shop to buy some sweets

- Sweetshop has so many variety of sweets



- You will be doing one kind among these
  - One sweet from all varieties available for eg: 1 kova, 1 laddu, 1 gulabjamun, etc.

- Only one kind of sweet eg: kova

- Mixing all kinds of sweets in shop itself and your stomach will be like this 😀

forEach:

Instructor Task:

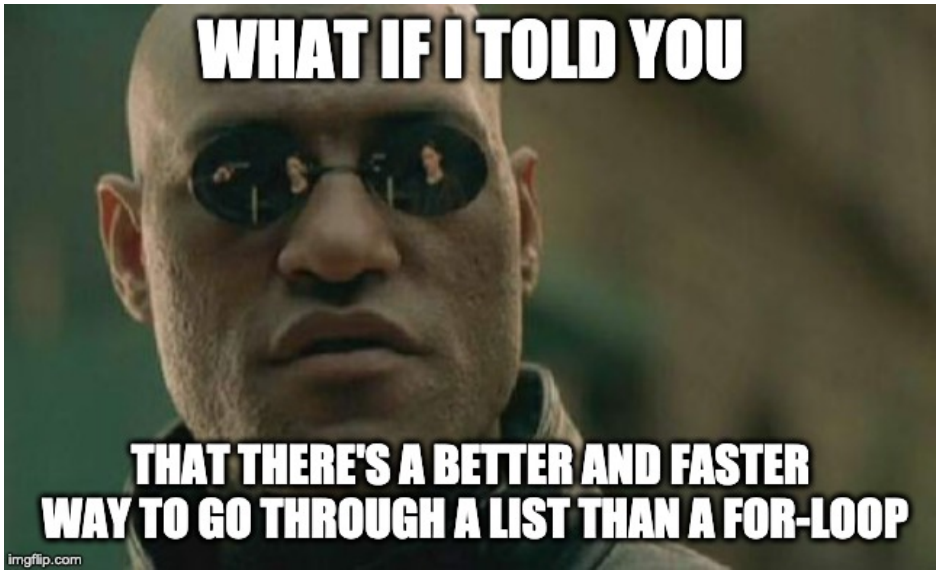
- Lets take example of this sweets menu

```
var sweets = ["kova","gulabjamun","laddu","mysorepak","badsha
w"]
```

- As a person I wanted all of these items in my plate, so I will go to each and every dish and pick it up, and also I can use for loop for this

```
var sweets = ["kova","gulabjamun","laddu","mysorepak","badsha
w"]

for (var i = 0; i < sweets.length; i++){


```

```
  console.log(food_menu[i])
}
```



- Explain syntax of forEach

```
var sweets = ["kova","gulabjamun","laddu","mysorepak","badsha
w"]

sweets.forEach(function (elem,index) {
    console.log(elem)
})

Output :
kova
gulabjamun
laddu
mysorepak
badshaw
```

- here elem is each sweet individually
- index is index number.

Warning :

- forEach has extra charges

- To pack those sweets in a box, we need to pay extra charges

```
var sweets = ["kova","gulabjamun","laddu","mysorepak","badsha
w"]
var box = []
sweets.forEach(function (elem,index) {
    box.push(elem)
})

console.log(box) //[ 'kova', 'gulabjamun', 'laddu', 'mysorepa
k', 'badshaw' ]
```

- Here creating extra array is extra charge.

map:

- map is similar to forEach, only difference is map doesnt have any additional charges


```
var sweets = ["kova","gulabjamun","laddu","mysorepak","badsha
w"]
```

```
var output= sweets.map(function (elem,index) {
    return elem
})

console.log(output)
```

- map method will return you a box(array) along with sweets

Difference between forEach and map

forEach

map

filter:

- If you want only one sweet for eg:kova, then we will use filter
- Filter also doesnt have any additional charges
- if will return you a box(array) along with sweets

```
var sweets = ["kova","gulabjamun","laddu","mysorepak","badsha
w"]
var output= sweets.filter(function (elem,index) {
    return elem=="kova"
})

console.log(output) // ["kova"]
```

forEach

Instructor Activity

Example 1: Simple Array Iteration

```
javascriptCopy code
const fruits = ['apple', 'banana', 'mango', 'orange'];

fruits.forEach((fruit) => {
    console.log(fruit);
});
```

Explanation: This example iterates over an array of fruits and logs each fruit to the console.

Example 2: Using Index

```
javascriptCopy code
const fruits = ['apple', 'banana', 'mango', 'orange'];

fruits.forEach((fruit, index) => {
    console.log(`${index + 1}. ${fruit}`);
});
```

Explanation: This example also iterates over an array of fruits but includes the index in the output, displaying a numbered list.

Example 3: Modifying Array Elements

```
javascriptCopy code
const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = [];

numbers.forEach((number) => {
    doubledNumbers.push(number * 2);
});

console.log(doubledNumbers);
```

Explanation: This example demonstrates how to create a new array by doubling each number from the original array.

Example 4: Iterating Over an Array of Objects

```
javascriptCopy code
const users = [
    { name: 'John', age: 25 },
    { name: 'Jane', age: 30 },
    { name: 'Mike', age: 20 }
];

users.forEach((user) => {
    console.log(`Name: ${user.name}, Age: ${user.age}`);
});
```

Explanation: This example iterates over an array of user objects and logs each user's name and age.

Student activity

Codepen - https://codepen.io/vchandu111/pen/oNrBEYK?editors=1010

Given the following array:

```
const arr = [3, 4, 5, 6, 7];
```

Use the `forEach` method and a conditional statement to output a string of only the odd numbers from the array, separated by a hyphen (" - ").

Expected Output:

```
"3 - 5 - 7"
```

Hint:

- Use `forEach` to iterate through the array.

- Use a conditional statement to check for odd numbers.

```
const arr = [3, 4, 5, 6, 7];
const oddNumbers = [];

// Iterate through the array using forEach
arr.forEach((num) => {
  // Check if the number is odd
  if (num % 2 !== 0) {
    // Add the odd number to the oddNumbers array
    oddNumbers.push(num);
  }
});

// Join the odd numbers with a hyphen and print the result
const result = oddNumbers.join(" - ");
console.log(result); // Output: "3 - 5 - 7"
```

Map

Example 1: Simple Array Transformation

```
javascriptCopy code
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = numbers.map(function(number) {
    return number * number;
});

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

Explanation: This example squares each number in the array

Example 7: Doubling Numbers

```
javascriptCopy code
const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map(function(number) {
    return number * 2;
});

console.log(doubled); // [2, 4, 6, 8, 10]
```

Explanation: This example creates a new array by doubling each number from the original array.

Example 2: Extracting Properties from Objects

```
javascriptCopy code
const users = [
    { name: 'John', age: 25 },
    { name: 'Jane', age: 30 },
    { name: 'Mike', age: 20 }
];

const userNames = users.map(function(user) {
    return user.name;
});

console.log(userNames); // ['John', 'Jane', 'Mike']
```

Explanation: This example extracts the `name` property from each user object in the array and returns a new array containing only the names.

Student activity

Example 5: Creating a New Array from Existing Data

Codepen: https://codepen.io/vchandu111/pen/poXRaOZ?editors=0010

```
javascriptCopy code
const products = [
    { name: 'Laptop', price: 1000 },
    { name: 'Phone', price: 500 },
    { name: 'Tablet', price: 700 }
];

const productDescriptions = products.map(function(product) {
    return product.name + ' costs $' + product.price;
});

console.log(productDescriptions);
/*
[
  'Laptop costs $1000',
  'Phone costs $500',
  'Tablet costs $700'
]
*/
```

Explanation: This example creates an array of strings describing each product by combining its `name` and `price`.

Filter

Instructor Activity

Example 1: Filtering Even Numbers

```
javascriptCopy code
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

const evenNumbers = numbers.filter(function(number) {
    return number % 2 === 0;
```

```
});

console.log(evenNumbers); // [2, 4, 6, 8]
```

Explanation: This example filters out even numbers from the array.

Example 2: Filtering Objects by Property

```
javascriptCopy code
const users = [
    { name: 'John', age: 25 },
    { name: 'Jane', age: 30 },
    { name: 'Mike', age: 20 }
];

const adults = users.filter(function(user) {
    return user.age >= 21;
});

console.log(adults);
/*
[
    { name: 'John', age: 25 },
    { name: 'Jane', age: 30 }
]
*/
```

Explanation: This example filters out users who are 21 or older.

Example 3: Filtering Strings by Length

```
javascriptCopy code
const words = ['apple', 'banana', 'kiwi', 'mango'];

const longWords = words.filter(function(word) {
```

```
    return word.length > 5;
});

console.log(longWords); // ['banana']
```

Explanation: This example filters out words that have more than 5 characters.

Student activity

Codepen: https://codepen.io/vchandu111/pen/jOjyJKo?editors=1010

Example 4: Filtering Array with Multiple Conditions

```
javascriptCopy code
const products = [
    { name: 'Laptop', price: 1000, inStock: true },
    { name: 'Phone', price: 500, inStock: false },
    { name: 'Tablet', price: 700, inStock: false }
];

const inStockProducts = products.filter(function(product) {
    return product.inStock && product.price > 600;
});

console.log(inStockProducts);
/*
[
    { name: 'Laptop', price: 1000, inStock: true },
    { name: 'Tablet', price: 700, inStock: true }
]
*/
```

Explanation: This example filters products that are in stock and have a price greater than 600.

Reduce

Let's consider the sum of elements in an array, like so

```
var nums = [1,2,3]
If I want to calculate the sum, how will I do it?

var sum = 0
for(var i = 0; i < nums.length; i++) {
    sum = sum + nums[i]
}

console.log(sum) gives the sum of the numbers in the array, right
```

Reduce is similar to this.

- The `.reduce()` method iterates through an array and returns a single value.
- There are two scenearios
  - Without inital value
  - With inital value
- **How reduce() works without an initial value**
- The code below shows what happens if we call `reduce()` with an array and no initial value.

```
const array = [15, 16, 17, 18, 19];

array.reduce(function (acc, el) {
  return acc+el;
});
```

callback iteration	acc	current value(el)	acc+el (stores in acc)
first call	15	16	31
second call	31	17	48
third call	48	18	66

callback iteration	acc	current value(el)	acc+el (stores in acc)
fourth call	66	19	85

- **How reduce() works with an initial value**
- Here we reduce the same array using the same algorithm, but with an *initialValue* of `10` passed the second argument to `reduce()`

```
let array = [15, 16, 17, 18, 19];

let addNums=

array.reduce(function (acc, cv) {
  return acc+cv;
},10);
```

callback iteration	acc	current value(cv)	acc+cv (stores in acc)
first call	10	15	25
second call	25	16	41
third call	41	17	58
fourth call	58	18	76
fifth call	76	19	95

Example 1: Total Price of Products

Codepen: Example Activity

Problem Statement:

Given the following array of product objects:

```
const products = [
    { name: 'Laptop', price: 1000 },
    { name: 'Phone', price: 500 },
    { name: 'Tablet', price: 700 }
];

// Use reduce to calculate the total price
const totalPrice = products.reduce(function(accumulator, curren
tProduct) {
    return accumulator + currentProduct.price;
}, 0);

console.log(totalPrice); // 2200
```

Use the `reduce` method to calculate the total price of all products.

Expected Output:

```
javascriptCopy code
2200
```

Hint:

- Use `reduce` to iterate through the array and accumulate the sum of the `price` properties.

Student Activity

codepen: https://codepen.io/vchandu111/pen/RwzKQXp?editors=1010

```
Given input
let data = [
  { name: "John", subject: "Javascript" },
  { name: "John", subject: "HTML" },
  { name: "John", subject: "CSS" },
  { name: "Pete", subject: "Java" },
```

```
  { name: "Pete", subject: "English" },
  { name: "Pete", subject: "Maths" },
  { name: "Mary", subject: "Rust" },
  { name: "Mary", subject: "Elm" }
];


Expected output
{
  John: ["Javascript", "HTML", "CSS"],
  Pete: ["Java", "English", "Maths"],
  Mary: ["Rust", "Elm"]
}
```

```
let data = [
  { name: "John", subject: "Javascript" },
  { name: "John", subject: "HTML" },
  { name: "John", subject: "CSS" },
  { name: "Pete", subject: "Java" },
  { name: "Pete", subject: "English" },
  { name: "Pete", subject: "Maths" },
  { name: "Mary", subject: "Rust" },
  { name: "Mary", subject: "Elm" }
];

let subjectHash = data.reduce(function(acc, item) {
  // Check if the name already exists as a key in the accumulator
  if (!acc[item.name]) {
    // If not, initialize it with an empty array
    acc[item.name] = [];
  }

  // Add the subject to the array for the respective name
  acc[item.name].push(item.subject);

  return acc;
```

```
}, {});

console.log(subjectHash);
/*
{
  John: ["Javascript", "HTML", "CSS"],
  Pete: ["Java", "English", "Maths"],
  Mary: ["Rust", "Elm"]
}
*/
```

Chaining

Till now, we've seen a few higher order functions which give us power to apply a logic using less lines of code, right?

What if I told you that is not all that JS provides us. We can even combine these higher order functions. Let's solve a few problems and learn the concept of chaining through the problems.

method	Input	return value
forEach	array	undefined
map	array	array
filter	array	array
reduce	array	single value

forEach.map

map.filter

filter.reduce

forEach.filter

forEach.reduce

map.reduce

Problem Statement:

Given an array of numbers, find the sum of the squares of the even numbers

```
// Function expression to check if a number is even
var isEven = function(num) {
    return num % 2 === 0;
};

// Function expression to square a number
var square = function(num) {
    return num * num;
};

// Function expression to sum numbers
var sum = function(accumulator, currentValue) {
    return accumulator + currentValue;
};

// Array of numbers
var numbers = [2, 3, 4, 5, 6, 7, 8, 9];

// Using chaining with function expressions
var sumOfSquaresOfEvens = numbers
    .filter(isEven)      // Filter out even numbers
    .map(square)         // Square the filtered numbers
    .reduce(sum, 0);     // Sum the squared numbers

// Output the result
console.log(sumOfSquaresOfEvens); // Output: 120
```

Sort - https://www.javascripttutorial.net/javascript-array-sort/

Introduction to JavaScript Array sort() method

```
let numbers = [0, 2, 5, 3, 10];
numbers.sort();
console.log(numbers);
```

When you sort an array of numbers, the `sort()` method converts these numbers to strings and compares the strings to determine the order. For example:

```
[ 0, 10, 2, 3, 5 ]
```

In this example, the `sort()` method places 10 before 2 because the string “10” comes before the string “2”.

To change this behavior, you need to pass a comparator function to the `sort()` method. The `sort(`) method will use the comparator function to determine the order of elements.

The following illustrates the syntax of the comparator function:

```
function compare(a,b) {
    return a-b
}
```

The `compare()` function accepts two arguments `a` and `b`. The `sort()` method will sort elements based on the return value of the `compare()` function with the following rules:

Return Value of `compare(a, b)`	**Action Taken by `sort()`**
Negative Number	`a` is placed before `b`

Positive Number	`b` is placed before `a`
Zero	`a` and `b` remain in their current positions

Example Array

```
const numbers = [3, 1, 5, 0];
```

Compare Function

We'll use a compare function to sort the array in ascending order:

```
numbers.sort(function(a,b){
    return a-b
})

console.log(numbers)
```

Steps of Sorting

1. **Initial Array:** `[3, 1, 5, 0]`
2. **First Comparison:**
   - **a = 3, b = 1**
   - Calculation: `a - b = 3 - 1 = 2`
   - Since `2 > 0`, `a` should come after `b`, so swap `3` and `1`.
   - **Array after swap:** `[1, 3, 5, 0]`
3. **Second Comparison:**
   - **a = 3, b = 5**
   - Calculation: `a - b = 3 - 5 = -2`
   - Since `2 < 0`, `a` should come before `b`, so no swap is needed.

- **Array remains:** `[1, 3, 5, 0]`

4. **Third Comparison:**
   - **a = 5, b = 0**
   - Calculation: `a - b = 5 - 0 = 5`
   - Since `5 > 0`, `a` should come after `b`, so swap `5` and `0`.
   - **Array after swap:** `[1, 3, 0, 5]`
5. **Fourth Comparison:**
   - **a = 3, b = 0**
   - Calculation: `a - b = 3 - 0 = 3`
   - Since `3 > 0`, `a` should come after `b`, so swap `3` and `0`.
   - **Array after swap:** `[1, 0, 3, 5]`
6. **Fifth Comparison:**
   - **a = 1, b = 0**
   - Calculation: `a - b = 1 - 0 = 1`
   - Since `1 > 0`, `a` should come after `b`, so swap `1` and `0`.
   - **Array after swap:** `[0, 1, 3, 5]`

Final Sorted Array

After all comparisons and swaps, the final sorted array is `[0, 1, 3, 5]`.

Summary of Steps

1. Compare 3 and 1: Swap → `[1, 3, 5, 0]`
2. Compare 3 and 5: No swap → `[1, 3, 5, 0]`
3. Compare 5 and 0: Swap → `[1, 3, 0, 5]`
4. Compare 3 and 0: Swap → `[1, 0, 3, 5]`
5. Compare 1 and 0: Swap → `[0, 1, 3, 5]`

By following these steps, you can see how the `sort()` method processes and arranges elements in ascending order using the provided compare function.

Steps of Sorting in Descending Order

1. **Initial Array:** [3, 1, 5, 0]
2. **First Comparison:**
   - **a = 3, b = 1**
   - Calculation: b - a = 1 - 3 = -2
   - Since -2 < 0, a should come before b, so no swap is needed.
   - **Array remains:** [3, 1, 5, 0]
3. **Second Comparison:**
   - **a = 1, b = 5**
   - Calculation: b - a = 5 - 1 = 4
   - Since 4 > 0, b should come before a, so swap 1 and 5.
   - **Array after swap:** [3, 5, 1, 0]
4. **Third Comparison:**
   - **a = 1, b = 0**
   - Calculation: b - a = 0 - 1 = -1
   - Since -1 < 0, a should come before b, so no swap is needed.
   - **Array remains:** [3, 5, 1, 0]
5. **Fourth Comparison:**
   - **a = 3, b = 5**
   - Calculation: b - a = 5 - 3 = 2
   - Since 2 > 0, b should come before a, so swap 3 and 5.
   - **Array after swap:** [5, 3, 1, 0]
6. **Fifth Comparison:**
   - **a = 3, b = 1**
   - Calculation: b - a = 1 - 3 = -2

- Since -2 < 0, a should come before b, so no swap is needed.
- **Array remains:** [5, 3, 1, 0]

7. **Sixth Comparison:**
   - **a = 1, b = 0**
   - Calculation: b - a = 0 - 1 = -1
   - Since -1 < 0, a should come before b, so no swap is needed.
   - **Array remains:** [5, 3, 1, 0]

Since no more swaps are needed, the array is now sorted in descending order:

Final Sorted Array: [5, 3, 1, 0]

Sorting an array of strings

Suppose you have an array of strings named `animals` as follows:

```
let animals = ['cat', 'dog', 'elephant', 'bee', 'ant'];Code lan
guage: JavaScript (javascript)
```

To sort the elements of the `animals` array in ascending order alphabetically, you use the `sort()` method without passing the compare function as shown in the following example:

```
let animals = ['cat', 'dog', 'elephant', 'bee', 'ant'];
animals.sort();

console.log(animals);Code language: JavaScript (javascript)
```

Output:

```
[ 'ant', 'bee', 'cat', 'dog', 'elephant' ]
```

To sort the `animals` array in descending order, you need to change the logic of the comparator function and pass it to the `sort()` method as the following example.

```
let animals = ['cat', 'dog', 'elephant', 'bee', 'ant'];

animals.sort((a, b) => {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

console.log(animals);Code language: JavaScript (javascript)
```

Output:

```
[ 'elephant', 'dog', 'cat', 'bee', 'ant' ]
```

Sorting an array of objects by a property

The following is an array of `employee` objects, where each object contains three properties: `name`, `salary` and `hireDate`.

```
let employees = [
    {name: 'John', salary: 90000, hireDate: "July 1, 2010"},
    {name: 'David', salary: 75000, hireDate: "August 15, 200
9"},
    {name: 'Ana', salary: 80000, hireDate: "December 12, 2011"}
]
```

Sorting objects by a numeric property

The following example shows how to sort the employees by `salary` in ascending order.

```
let employees = [
  { name: 'John', salary: 90000, hireDate: 'July 1, 2010' },
  { name: 'David', salary: 75000, hireDate: 'August 15, 2009'
},
```

```
  { name: 'Ana', salary: 80000, hireDate: 'December 12, 2011'
},
];

// sort by salary
employees.sort(function (x, y) {
  return x.salary - y.salary;
});

console.table(employees);
```