



Programming Foundations - Module 2

Noroff Guide



**Noroff
Education**

Table of Contents

Module overview	3
Introduction	3
Learning outcomes	3
2.1. Lesson - Classes	5
Introduction	5
Learning outcomes	6
The concept of classes	6
Constructing the class	7
Encapsulation	10
Inheritance	12
Static methods	15
What did I learn in this lesson?	16
References	17
2.1. Lesson task - Classes	17
The task	17
2.2. Lesson - Algorithms and data structures	18
Introduction	18
Learning outcomes	19
Sets	19
Maps	21
Advanced data structures	23
Introduction to algorithms	25
Standard sorting algorithms	29
Built-in sorting in JavaScript	32
What did I learn in this lesson?	34
References	34
2.2. Lesson task - Algorithms and data structures	35
The task	35
2.3. Lesson - Basic asynchronous operations	37
Introduction	37
Learning outcomes	38
Function sequence	38
Callbacks	39

Timeouts	41
Intervals	42
Promises	44
Async/Await.....	46
What did I learn in this lesson?	48
References	48
2.3. Lesson task - Basic asynchronous operations	48
The task.....	48
2.4. Lesson - Revision	49
Introduction	49
Learning outcomes	50
The customer asks for a table	50
Sneaky customers.....	52
Free the table	54
The final code	56
What did I learn in this lesson?	60
References	60
2.4. Lesson task - Revision.....	60
The task.....	60
2.5. Lesson - Self-study	61
Introduction	61
Materials	62
2.5. Lesson task - Self-study.....	62
The task.....	62

Module overview

Introduction

Welcome to Module 2 of Programming Foundations.

If anything is unclear, check your progression plan and/or contact a tutor on Discord.

Module structure	Estimated workload
2.1. Lesson and task Classes	8 hours
2.2. Lesson and task Algorithms and data structures	8 hours
2.3. Lesson and task Basic asynchronous operations	8 hours
2.4. Lesson and task Revision	8 hours
2.5. Lesson and task Self-study	8 hours

Learning outcomes

In this module, we are covering the following knowledge learning outcomes:

- The candidate has knowledge of primary concepts, core ideas and general basic methods within programming.

- The candidate has knowledge of program syntax, program structure, control structures, data types, and variables as used in the JavaScript language.
- The candidate has knowledge of development and debugging within a browser.
- The candidate has knowledge of industry relevant software used for writing JavaScript code.
- The candidate can update their knowledge of basic programming concepts.

In this module, we are covering the following skill learning outcomes:

- The candidate can apply knowledge of program syntax, program structure, control structures, data types, and variables to complete simple tasks using the JavaScript language.
- The candidate can apply knowledge of browsers and development tools to set up a computer to develop and debug small JavaScript programs.
- The candidate masters the use of relevant tools for writing, editing and debugging JavaScript source code.

In this module, we are covering the following general competence learning outcome:

- The candidate can carry out simple programming tasks with the JavaScript language.

2.1. Lesson - Classes

Introduction



Object-Oriented Programming (OOP) is a popular paradigm fundamental in numerous programming languages, such as Java and C++. It helps us divide our application into smaller parts. We can then use these parts as many times as we want.

Fortunately, with ECMAScript 2015, we can implement it in JavaScript using Classes. In this lesson, we'll learn about some basic class concepts, such as encapsulation and inheritance.

Classes are usually stored in separate files. We'll also learn about modules, which enable us to connect functionalities from different JavaScript files.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of development and debugging within a web browser.

In this lesson, we are covering the following skill learning outcomes:

- The candidate can apply knowledge of web browsers and development tools to set up a computer for use in developing and debugging small JavaScript programs.
- The candidate masters the use of relevant tools for writing, editing, and debugging JavaScript source code.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out simple programming tasks with the JavaScript language.

The concept of classes

We can think of a class as the blueprint of a house. It has all the details: the floors, doors, windows, etc. We can build the house (the object) based on this information. Based on one class, we can build several houses.

Classes can have multiple properties and methods. Properties are set in the class constructor. We invoke the construction during the initialisation of the class object.

Classes are the central concept of Object-Oriented Programming. You can read more about this in the Mozilla documentation in the READ section.

Properties and methods of the class are public by default. After creating a class object, we can access them from anywhere in our code. However, sometimes we

don't want this enabled. Fortunately, we can also use private properties and methods. These can only be changed from inside the class.

READ

Page: [Object-oriented programming](#) by MDN Web Docs.

WATCH

Video: [Learn object-oriented design principles \(1m 27s\)](#) by Barron Stone and Olivia Chiu Stone on LinkedIn Learning.

Constructing the class

The class has the following syntax:

```
class ClassName {  
    constructor() { ... }  
    method_1() { ... }  
    method_2() { ... }  
    method_3() { ... }  
}
```

```
let newObject = new ClassName(args);
```

We can pass the arguments to the class constructor in the parenthesis. A constructor is a function that creates an instance of a class called an 'object'. The constructor is automatically executed after we complete the object with the 'new' keyword. We use it to initialise properties. Unlike many other programming languages, classes in JavaScript can only have one constructor.

Let's look at a working example of how to create a simple class:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.year;
  }
}
```

Our class is the car template. It has two properties: name and year. It also has the age method, which returns the age of our car based on today's date. We use the 'this' keyword to tell the computer we want to access the class property, not the input data.

Let's create a car object:

```
let newCar = new Car("Nissan", 2020);
```

We can access all properties and methods of our new object:

```
console.log(newCar.name); // Nissan
console.log(newCar.age()); // 2
console.log(newCar.year); // 2020
```

READ

Page: [JavaScript Classes](#) by W3 schools.

WATCH

1. Video: [Introduction to classes](#) (1m 49s) by Emmanuel Henri on LinkedIn Learning.
2. Video: [Objects: A practical introduction](#) (4m 53s) by Morten Rand-Hendriksen on LinkedIn Learning.

ACTIVITY

Create a class named Student. It should have the properties: name, surname and age. It should also have two methods: the first one – greetings(), should return a string: “Hello my name is _insert_name_property_ _insert_surname_property_”. The second one should be called birthYear() and return the student’s year of birth.

Ask three classmates for the necessary data (or create dummy data if you prefer) and create four objects of the Student class. The data of the fourth object should be your own. Print the results of greetings() and birthYear() methods for every object.

Source code: [Click here to reveal.](#)

Encapsulation

But what if we pass the wrong data? For example, a year greater than 2022. For this reason, we often don't want our properties to be accessible to everyone. The same could be said about certain methods. We can use the encapsulation concept to solve this. Encapsulation is the concept of hiding certain data, which can only be accessed from inside the class. This appears in every programming language that supports Object-oriented Programming.

Let's edit our class, so if the provided year is incorrect, it will be set to the current year by default:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  set year(newYear) {
    let date = new Date();
    if (newYear > date.getFullYear()) {
      this._year = date.getFullYear();
    }
    else {
      this._year = newYear;
    }
  }
  get year() {
    return this._year;
  }
  age() {
    let date = new Date();
    return date.getFullYear() - this._year;
  }
}

let newCar = new Car("Nissan", 2023);
console.log(newCar.year); // 2022
console.log(newCar._year); // 2022
```

We added getters and setters using the 'get' and 'set' keywords. The setter method checks whether our input data is correct. If not, it sets it to the current year. The

getter method isn't doing much right now, but we can use it to return the year as a date and keep storing it as a number within the class, for example.

However, after creating the class object, we can still edit our year to be greater than the current one. If someone knows we store our year in the `_year` variable, they can modify it because our `_year` property is still public. We can make it private by adding `#` before its name. When using private properties and methods, we have to add their declarations at the beginning of the class.

Let's make the year property private:

```
class Car {
  #_year
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  set year(newYear) {
    let date = new Date();
    if (newYear > date.getFullYear()) {
      this.#_year = date.getFullYear();
    }
    else {
      this.#_year = newYear;
    }
  }
  get year() {
    return this.#_year;
  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.#_year;
  }
}

let newCar = new Car("Nissan", 2023);
console.log(newCar.year = 2022);
console.log(newCar._year); // undefined
```

We can also see another use of the getters here. Now the year property isn't accessible from outside the class, so we need to use getters and setters to access it. Our setter is already checking whether the new value is correct.

READ

Page: [Private class fields](#) by MDN Web Docs.

WATCH

1. Video: [Encapsulation](#) (3m 38s) by Barron Stone and Olivia Stone on LinkedIn Learning.
2. Video: [JavaScript Beginners Tutorial 21 | OOP | Encapsulation with example](#) (6m 1s) by Automation Step by Step on YouTube.
3. Video: [Getting and setting class values](#) (3m 35s) by Eve Porcello on LinkedIn Learning.

Inheritance

As we learned, classes are like house templates. But there can be various houses, such as villas, detached houses, semi-detached houses and many more. All these houses have much in common, windows and doors, for example. But they can each have their own properties too. Fortunately, JavaScript and Object-oriented Programming provide a mechanism to implement this.

In the beginning, we can create a template for a basic house and then create other templates for all house types based on our first template. We'll have access to all house properties and methods from the new templates, but we can add our new

properties and methods only for this class. In this situation, we say that the villa class inherits from the house class, for example.

Let's look at the code example of the situation described above.

```
class House {
  constructor(name) {
    this.name = name;
  }
  info() {
    return 'This house is the: ' + this.name + '. ';
  }
}

class Villa extends House {
  constructor(name, prestige) {
    super(name);
    this.prestige = prestige;
  }
  info() {
    return super.info() + 'It has the prestige ' +
this.prestige + ' out of 10. ';
  }
}

class DetachedHouse extends House {
  constructor(name, hasGarden) {
    super(name);
    this.hasGarden = hasGarden;
  }
}

let villa = new Villa("Not so prestigious villa",1)
let detachedHouse = new DetachedHouse("Small house, but with
the garden", true)
console.log(villa.info()); // This house is the: Not so
prestigious villa. It has the prestige 1 out of 10.
console.log(detachedHouse.info()); // This house is the:
Small house, but with the garden.
```

Above we see the implementation of the base house class and two types of houses: villas and detached houses. The basic house class has the name property and info() method. The villa class has the additional prestige property and overrides our info method. We can use properties and methods from the base class with the help of the 'super' keyword. On the other hand, the detached house only adds a hasGarden property. Note that the info() class, in this case, works the same as for the house class.

Thanks to the inheritance mechanism, we don't have to implement the name keyword and info method in these classes.

READ

Page: [JavaScript Class Inheritance](#) by W3 schools.

WATCH

Video: [Inheritance](#) (4m) by Barron Stone and Olivia Stone on LinkedIn Learning.

ACTIVITY

Create a basic class Pizza. The pizza has a name and price. It should also have an info() method which says, "I am basic pizza: _name_. I cost _price_." Create two subclasses: ItalianPizza and AmericanPizza.

AmericanPizza has a sauce string property. ItalianPizza has a region property, which determines where it comes from. Their info() method should contain additional information about these properties.

Source code: [Click here to reveal.](#)

Static methods

Static methods are the methods defined in the class itself. We don't have to create a class object to use these methods. Let's look at a working example of the static method:

```
class House {
    constructor(name) {
        this.name = name;
    }
    info() {
        return 'This house is the: ' + this.name + '. ';
    }
    static greetings() {
        return "Hello, my dear landlord.";
    }
}
```



```
}  
let house = new House("My house");  
console.log(House.greetings()); // Hello, my dear landlord.  
console.log(house.greetings()); // Uncaught TypeError:  
house.greetings is not a function
```

We can see that invoking this method on the instance of the house object results in error. The correct way is to invoke it in the class itself.

READ

1. Page: [JavaScript Static Methods](#) by W3 Schools.
2. Article: [Building a Project with JavaScript Classes](#) by Elijah Trillionz.

WATCH

Video: [JavaScript Classes #3: Static Methods – JavaScript OOP Tutorial](#) (7m 31s) by dcode on YouTube.

What did I learn in this lesson?

This lesson provided the following insights:

- Object-oriented programming and why we should use it.
- How to create classes in JavaScript.
- Encapsulation and inheritance.

References

W3 schools (n.d.) *HTML Tutorial*. Available at: <https://www.w3schools.com/html/default.asp> [Accessed 25 May 2022].

MDN Web Docs (2022) *Web technology for developers*. Available at: <https://developer.mozilla.org/en-US/docs/Web> [Accessed 25 May 2022].

2.1. Lesson task - Classes

The task

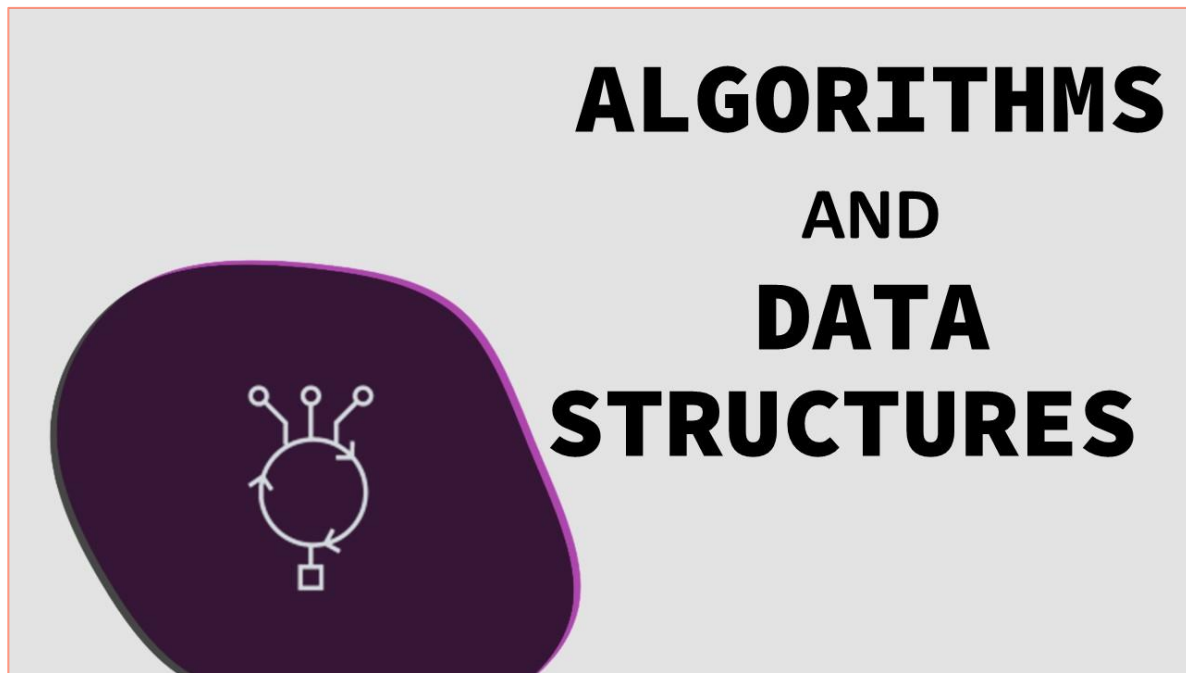
In this lesson, we learned some crucial concepts of Object-oriented Programming. We learned about classes, encapsulation and inheritance.

In this task, you have to create a menu for a restaurant. Every dish has a name and price. Our restaurant serves pizza, pancakes and pasta. In addition, pizza has an array of ingredients, and pancakes have the Boolean `sweet` property, which defines whether pancakes should be served sweet. Pasta has the Boolean property `isVegetarian`, which defines whether it contains meat.

Create a model in Object-oriented Programming simulating the menu. Think about which properties should be private and which not.

2.2. Lesson - Algorithms and data structures

Introduction



We learned about several data types that allow us to store our information efficiently. In this lesson, we'll learn about two more collections: sets and maps. These structures are less popular than arrays but, in some cases, much more convenient. We'll also go through more complicated data structures, which aren't implemented in JavaScript but might be helpful in future.

After getting to know data structures well, we'll learn some basics of algorithms. An algorithm is a sequence of well-defined instructions typically used to solve a problem. We can compare it to a cake recipe. On input, we have ingredients. Then, we execute some operations, and at the very end, during output, we get the expected value – a tasty cake (if we executed everything correctly).

This lesson will focus on applying algorithms to the sorting problem. The sorting problem is about putting all values from the array in order, usually from the smallest

to the greatest. We'll implement some standard sorting algorithms and compare them to the built-in solution.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of development and debugging within a web browser.

In this lesson, we are covering the following skill learning outcomes:

- The candidate can apply knowledge of web browsers and development tools to set up a computer for use in developing and debugging small JavaScript programs.
- The candidate masters the use of relevant tools for writing, editing, and debugging JavaScript source code.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out simple programming tasks with the JavaScript language.

Sets

So far, we know how to store multiple elements in the arrays. However, sometimes other collections might be a better choice. ECMAScript 6 in 2015 allowed us to use set and map collections. Sets are perfect when we don't want our collection to contain duplicates or elements to be in order.

We must pass an array as a parameter to create a new set. We can also create an empty set and then add new elements one by one:

```
const letters = new Set(["a", "b", "c"]);
```

```
// Create a Set
const letters = new Set();
// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
```

Remember, all elements within the set should be unique. For example, nothing will happen when we try to add element “b” to our letter set again. As we can see, we can add new elements with the method `add()`. We can also remove it with the method `delete()` or check if our set already contains a certain element: with the `has()` method. To check the current size of our set, we can use the `size` property:

```
// Create a Set
const letters = new Set();
// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
// Let's access "c" via variable
const c = "c"
// Remove c from the Values
letters.delete(c);
// Check if our set contains "c"
console.log(letters.has("c"));
// Check the size of our set
console.log(letters.size);
```

Like with arrays, we can use iteration methods on sets! Let’s print all elements from our set in the console:

```
const letters = new Set(["a", "b", "c"]);
letters.forEach (function(value) {
    console.log(value);
})
```

We can also access all values from the set with the set iterator. We can get it from the `values()` method. Let’s use the `for-of` loop and the set iterator to do the same thing as in the previous exercise:

```
const letters = new Set(["a", "b", "c"]);
for (const x of letters.values()) {
```

```
console.log(x);  
}
```

READ

Page: [JavaScript Sets](#) by W3 schools.

WATCH

1. Video: [Build your Map object with the set method](#) (4m 41s) by Jamie Pittman on LinkedIn Learning.
2. Video: [Add values to your set](#) (3m 17s) by Jamie Pittman on LinkedIn Learning.

Maps

Let's imagine we're creating a program for a restaurant and have to implement a menu where every dish has a price. For now, we would use two-dimensional arrays to do this. The first element would represent our dish name, while the second is its price. However, some operations on this array, such as adding and removing new values, would be problematic.

Fortunately, maps are the perfect data structures for our case. The map is like a dictionary. We need to specify the key and tell what value is behind it. Remember, the keys need to be unique, while values don't have to.

Let's look at a simple, practical example of the map:

```
const menu = new Map([
  ["Margarita", 9],
  ["Pepperoni", 10],
  ["Salami", 10]
]);
```

To create a map, we need to put our data as a two-dimensional array in the same format we referred to at the beginning of this section.

Note that it's completely fine that pepperoni and salami cost the same.

However, imagine our restaurant wants to create a special offer on the pepperoni and the new price should temporarily change to 9.5. We can easily do this with the `set()` method:

```
menu.set("Pepperoni", 9.5);
```

To get our price, we can use the `get()` method and pass our key name as the parameter:

```
menu.get("Pepperoni"); //9.5
```

We can also delete elements from the menu with the `delete()` method, check our map size with the `size` property or whether a position is already in our menu with the `has()` method. All these methods work the same way as sets.

The `for-each` method also works similarly, but we have access to the key this time. Let's print our menu:

```
const menu = new Map([
  ["Margarita", 9],
  ["Pepperoni", 10],
  ["Salami", 10]
]);
menu.forEach (function(value, key) {
  console.log("Dish: " + key + ", Price: " + value);
})
```

Instead of `values()` method, we have `entries()` method. This method provides access to the array containing keys and values from the map. Let's store all that information in a string:

```
// List all entries
let text = "";
for (const x of menu.entries()) {
  text += x;
}
```

READ


Page: [JavaScript Maps](#) by W3 schools.

Advanced data structures

Usually, collections provided by JavaScript are more than enough for our coding. However, more advanced data structures might sometimes be necessary with more complex problems. We'll need to implement these structures by ourselves with the provided schema. In this section, we'll discuss these structures briefly, but we highly recommend also reading the article in the READ section. This article explains the implementation and provides a more detailed description of each data structure.

1. Stack: Works like stacking books. We can only add or remove elements from the top.
2. Queue: Similar to stack, we can remove elements only from the top and add elements only to the end.
3. Linked List: Data is grouped in nodes, and nodes are connected with links. We start from the 'head' node and must go through all the links to reach the last node.
4. Set: We use it to create a set of unique elements, just like a built-in JavaScript set, but we can also create intersections and differences of sets or subsets.

5. Hash Table: This is a key-value data structure, just like JavaScript Map. Thanks to the hash function, we can quickly find a value corresponding to a key.
6. Tree: Data is grouped in nodes. We have access to the root node. Inserting new elements and searching for an element is highly efficient in this structure.
7. Graph: Data is grouped in nodes. Graphs are used widely to solve daily problems, such as finding the shortest route to school or recommending new friends on social media.

	<p>READ</p> <p>Article: 8 Common Data Structures in Javascript by Kingsley Tan.</p>
---	---

WATCH

1. Video: [How to Implement a Stack in JavaScript](#) (7m 48s) by dcode on YouTube.
2. Video: [How to Implement a Queue in JavaScript](#) (8m 22s) by dcode on YouTube.
3. Video: [Linked lists](#) (2m 36s) by Joe Marini on LinkedIn Learning.
4. Video: [What are sets?](#) (2m 39s) by Kathryn Hodge on LinkedIn Learning.
5. Video: [Understanding hash tables](#) (3m 15s) by Kathryn Hodge on LinkedIn Learning.
6. Video: [Introduction to tree data structures](#) (2m 38s) by Kathryn Hodge on LinkedIn Learning.
7. Video: [Graph – Data Structures in Javascript](#) (20m 27s) by Questionable Coding on YouTube.

Introduction to algorithms

Algorithms are tools to solve complicated problems. In this section, we'll focus on how to compare their quality. We'll also learn recursion, a powerful tool crucial for divide-and-conquer algorithms.

Some algorithms are more efficient than others. The following measures define the effectiveness of an algorithm:

- **Time complexity.** How much time is required to execute the algorithm? As computers differ, the time complexity isn't measured in seconds but

the number of standard operations. You can read more about this in the article in the READ section.

- **Memory complexity.** How much memory we need to execute the algorithm.

Usually, we focus more on time complexity, as memory isn't the problem nowadays. While memory is easily accessible, some algorithms might use so much memory that it becomes significant.

Based on these factors, we can define four performance cases:

- **Worst case time complexity:** A function defined as a result of a maximum number of steps taken on any instance of size n . It's usually expressed in Big O notation.
- **Average case time complexity:** A function defined as a result of the average number of steps taken on any instance of size n . It's usually expressed in Big theta notation.
- **Best case time complexity:** A function defined as a result of the minimum number of steps taken on any instance of size n . It's usually expressed in Big omega notation.
- **Space complexity:** A function defined as a result of additional memory space needed to carry out the algorithm. It's usually expressed in Big O notation.

There are two crucial strategies applied to many algorithms: recursion and divide-and-conquer. Recursion is a function that calls itself with slightly modified parameters. Let's imagine a function that counts down in the console, from the input value to 1:

```
const countdown = (value) => {  
  for(let i=value; i>0; i--) {  
    console.log(i);  
  }  
}  
countdown(10);
```

It's a standard function, and we should easily understand it. Let's write the same, using recursion.

```

Const  countdown = (value) => {
    console.log(value);
    const newValue = value - 1;

    if(newValue > 0) {
        countdown(newValue);
    }
}
countdown(10);

```

Instead of iterating through values, we only print the value once. If we're still above 0, we run our function for the 1 smaller value. The function will run 10 times until we don't meet the condition:

```
newValue > 0
```

We can easily reverse the counting:


```


const countDownRev = (value) => {
    const newValue = value - 1;
    if(newValue > 0) {
        countDownRev(newValue);
    }
    console.log(value);
}
countDownRev(10);


```

We only put the first line to the bottom, and now our function prints numbers 1 to 10 instead of 10 to 1.

Divide-and-conquer is a strategy to divide one complicated problem into many smaller, easier ones to solve. Based on the results of smaller problems, we get our final result.

	<p>READ</p> <ol style="list-style-type: none"> 1. Article: How to analyze time complexity: Count your steps by yourbasic.org. 2. Article: Analysis of Algorithms Set 3 (Asymptotic Notations) by Abhay Rathi.
---	--

	<p>WATCH</p> <ol style="list-style-type: none"> 1. Video: Learn Big O Notation In 12 Minutes (12m 17s) by Web Dev Simplified on YouTube. 2. Video: What are algorithms? (2m 44s) by Joe Marini on LinkedIn Learning. 3. Video course: JavaScript: Recursion (1h 7m) by Mustapha Rufai on LinkedIn Learning.
---	---

	<p>ACTIVITY</p> <p>Try to solve problems on JavaScript Recursion – Exercises, Practice, Solution by W3 schools.</p> <p>The last ones are hard, so don't worry if you can't do them. However, you should be able to do at least a few. In case of problems, you can check their solutions.</p>
---	--

Standard sorting algorithms

Sorting our data structures can help the effectiveness of our program. Imagine storing the restaurant's menu, but clients usually order only cheap dishes. If we sort our data, we can quickly find a cheap dish from the menu instead of searching an entire array.

Sorting algorithms are important to study because they can often reduce the time complexity of our problem since naïve sort is far from efficient.

In this section, we'll learn three basic sorting algorithms. The three basic ones sort data with $O(n^2)$ time complexity.

For the sake of our algorithms, let's define a helper method which swaps two elements within an array:

```
function swap(arr, a, b) {  
  let temp = arr[a];  
  arr[a] = arr[b];  
  arr[b] = temp;  
}
```

Let's analyse the bubble sort:

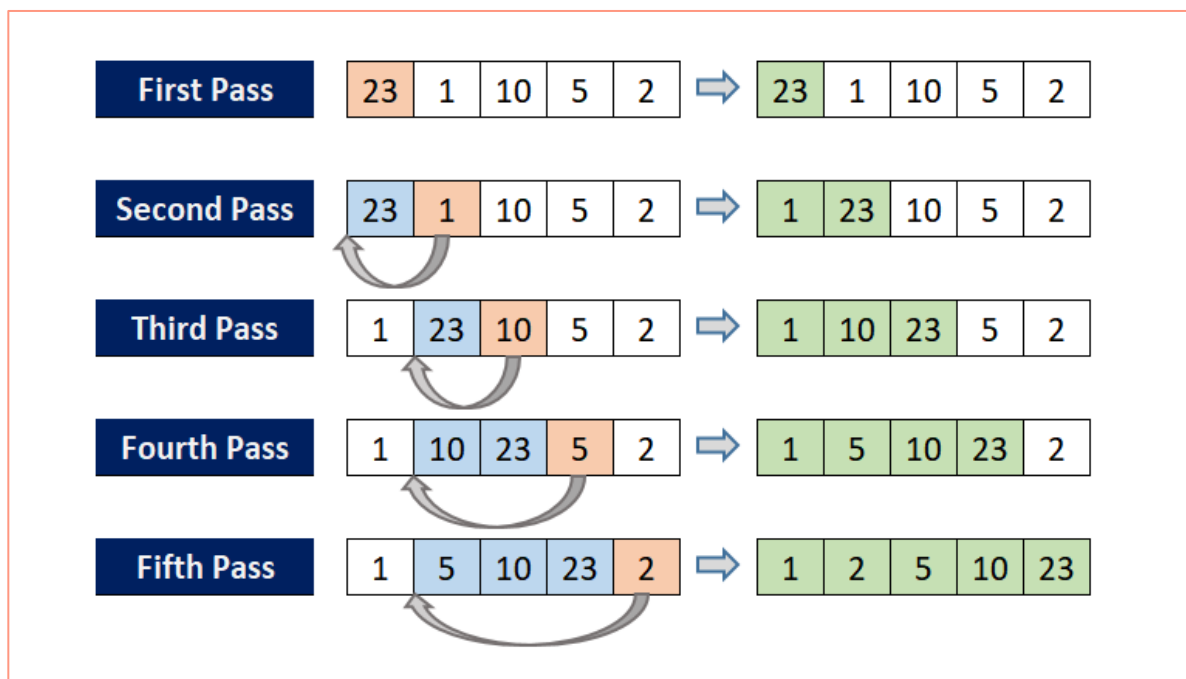
```
function bubbleSort(arr) {  
  for (let i = 0; i < arr.length; i++) {  
    for (let j = 0; j < arr.length - 1 - i; j++) {  
      if (arr[j] > arr[j + 1]) {  
        swap(arr, j, j + 1);  
      }  
    }  
  }  
  return arr;  
}
```

The general idea of the bubble sort is to check every pair of adjacent elements and swap their places if they're not in the correct order. This strategy seems very logical. With each iteration, we are closer to the solution even though each swap makes us only one step closer to the final result.

The other standard algorithm is the insertion sort:

```
function insertionSort(arr) {
  let temp;
  for (let i = 1; i < arr.length; i++) {
    let j = i;
    temp = arr[i];
    while (j > 0 && arr[j - 1] > temp) {
      arr[j] = arr[j - 1];
      j--;
    }
    arr[j] = temp;
  }
  return arr;
}
```

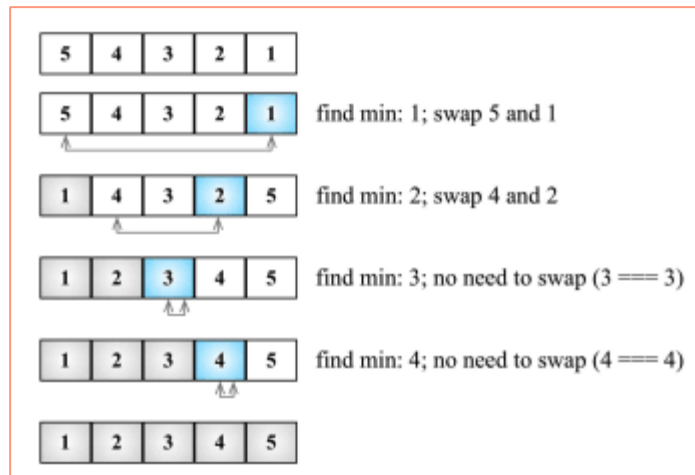
In the following example, you can see how it works:



We always take the first unsorted value (from the right) and put it in the proper place (on the left).

The last one is the selection sort. This algorithm is about finding the smallest value from all unsorted values and placing it at the end of the sorted part.


Let's look at how this works on graphics and study its code.




```
function selectionSort(arr) {
  let minIndex;
  for (let i = 0; i < arr.length - 1; i++) {
    minIndex = i;
    for (let j = i; j < arr.length; j++) {
      if (arr[minIndex] > arr[j]) {
        minIndex = j;
      }
    }
    if (i !== minIndex) {
      swap(arr, i, minIndex);
    }
  }
  return arr;
}
```

READ

Article: [Sorting Algorithms in JavaScript](#) by Kennedy Mwangi.

	<p>WATCH</p> <ol style="list-style-type: none">1. Video: Sorting Algorithm Visualization (Insertion sort, Selection sort, Bubble sort) (8m 45s) by Suvro Debnath on YouTube.2. Video: Order arrays with sort (4m 11s) by Jamie Pittman on LinkedIn Learning to learn more about sorting arrays.
---	---

	<p>ACTIVITY</p> <p>Modify the algorithms; instead of sorting from the smallest to the greatest, sort from the greatest to the smallest.</p> <p>Source code: Click here to reveal.</p>
--	---

Built-in sorting in JavaScript

We learned about some standard sorting algorithms. Fortunately, as long as we want to sort an array, we don't always have to implement our sorting. JavaScript provides us with the `sort()` method, which is more effective from the time complexity view. It has $O(n \log n)$ time complexity. Let's look at a working example of how to use the `sort()` method:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
console.log(fruits); // ['Apple', 'Banana', 'Mango', 'Orange']
```

The `sort()` method sort all elements starting from the smallest. To change this order, we can first sort all the elements and then use the `reverse()` method:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
console.log(fruits); // ['Apple', 'Banana', 'Mango',
'Orange']
fruits.reverse();
console.log(fruits); // ['Orange', 'Mango', 'Banana',
'Apple']
```

By default, the sort method sorts values as strings – alphabetically. So if we want to sort the numbers array, we get the following result:

```
const numbers = [0, 1, 11, 12, 2, 55, 4];
numbers.sort();
console.log(numbers); // [0, 1, 11, 12, 2, 4, 55]
```


We can fix this by passing our own compare function. The compare function should take two input parameters and return the number. From the returned value, we can estimate the relation between input parameters:


- Negative value: The first parameter will be sorted before the second.
- Positive value: The second parameter will be sorted before the first.
- Zero: No changes will be made to these two elements (elements are equal).

Let's implement the compare function and use it to fix our number sorting:

```
const numbers = [0, 1, 11, 12, 2, 55, 4];
numbers.sort((a, b) => a - b);
console.log(numbers); // [0, 1, 2, 4, 11, 12, 55]
```

We passed the compare function as the arrow function and got the expected result.

	<p>READ</p> <ol style="list-style-type: none">1. Page: JavaScript Sorting Arrays by W3 schools.2. Page: Array.prototype.sort() by MDN Web Docs.
---	---

	<p>WATCH</p> <p>Video: sort Array Method JavaScript Tutorial (5m 46s) by Florin Pop on YouTube.</p>
---	--

What did I learn in this lesson?

This lesson provided the following insights:

- How and when to use sets and maps.
- When we should consider using other, more advanced data structures.
- Algorithms and why they are important.
- The sorting and how to sort data efficiently.

References

W3 schools (n.d.) *HTML Tutorial*. Available at: <https://www.w3schools.com/html/default.asp> [Accessed 25 May 2022].

Section (n.d.) *Your App. We do the rest*. Available at: <https://www.section.io> [Accessed 7 June 2022].

Better Programming (n.d.) *Better Programming*. Available

at: <https://betterprogramming.pub/>[Accessed 7 June 2022].

Geeks for Geeks (n.d.) *Geeks for Geeks*. Available at: <https://www.geeksforgeeks.org/> [Accessed 7 June 2022].

yourbasic (n.d.) *Algorithms to Go*. Available at: <https://yourbasic.org/>[Accessed 7 June 2022].

2.2. Lesson task - Algorithms and data structures

The task

In this lesson, we learned about new data structures. We also learned what an algorithm is and gained insight into three basic sorting algorithms. We saw that JavaScript offers a good alternative – a `sort()` function.

In this task, you'll have to sort the numbers, but to make it more interesting, you won't sort them from smallest to greatest. First, you need to get two numbers from the user. You can use prompts to do that. Both need to be positive integers. The bigger one defines the size of your array.

We can generate the array of the provided size this way:

```
const numbers = Array.from(Array(10).keys()); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We want to sort all elements by the remainder when dividing by the second, smaller parameter. If the remainder is the same for two numbers, the smaller one should be before the greater one. We can get the remainder through the modulo operator:

```
console.log(11%4); // 3
```

Our sorted array will look this way for the size of the array equal to 10 and second parameter 3

```
[0, 3, 6, 9, 1, 4, 7, 2, 5, 8]
```

Please print the result in the console.

You should implement sorting in two ways:

1. Using any of the basic sorting algorithms mentioned in this lesson or any other sorting algorithm found on the Internet that is no less optimal.
2. Using the `sort()` method.

ACTIVITY

After completing this task, try your hand at finishing some [function exercises on the w3resource platform](#). You can find many interesting activities with their solutions. Completing some of them will help you become familiar with algorithmic thinking.

2.3. Lesson - Basic asynchronous operations

Introduction



Computers are asynchronous by design. By asynchronous, we mean they can perform numerous operations simultaneously. JavaScript is synchronous by default. All functions we call will execute from the first to the last order, and a single error will stop the entire sequence.

However, JavaScript also provides tools to make asynchronous functions. We can start the function, perform some operations and perform the rest when our function finishes the calculations.

In this lesson, we'll learn how to achieve this.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of development and debugging within a web browser.

In this lesson, we are covering the following skill learning outcomes:

- The candidate can apply knowledge of web browsers and development tools to set up a computer for use in developing and debugging small JavaScript programs.
- The candidate masters the use of relevant tools for writing, editing, and debugging JavaScript source code.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out simple programming tasks with the JavaScript language.

Function sequence

For the sake of our example, let's create the following files:

async.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
  </head>
  <body>
    <p id="displayer"></p>
    <script src="async.js"></script>
```

```
    </body>
</html>
```

async.js:

```
function print(text) {
    document.getElementById("displayer").innerHTML = text;
}
print("Hello");
```

Run the `async.html` file. We should see the Hello text. Our print functions put the text we want into the paragraph of id equal to “displayer”.

If we call multiple functions in JavaScript, they are executed in the sequence they are called. If we call our print function numerous times, only the last result will stay on the website:

```
function print(text) {
    document.getElementById("displayer").innerHTML = text;
}
print("Hello");
print("My");
print("Dear");
print("Friend");
```

If for some reason, operation print needs more time to execute, we should see the text changing after each iteration finishes.

Callbacks

What if our function doesn't execute properly? Do we want our entire sequence to stop? Or run it as if nothing happened? Both options are less than ideal. We don't want our program to stop, but some functions could be based on the first function result.

JavaScript solves this problem with callbacks. Remember, we can pass functions as parameters in JavaScript. The concept of callbacks is about passing functions based on the first one's result as the argument and invoking them if your first function is executed correctly.

Let's create a simple calculator which sums two numbers and prints the result:

```
function print(text) {  
    document.getElementById("displayer").innerHTML = text;  
}  
function calculator(a, b, callback) {  
    let result = a + b;  
    callback(result);  
}  
calculator(1, 2, print);
```

Instead of invoking the print function after summing the numbers, we pass it as a callback. It runs after the calculations and prints the final result.


This example is simple, but if we imagine the calculator function as asynchronous and taking much more time, it would save us so much time that we would be able to continue our program instead of waiting for the result to print.

READ

Page: [JavaScript Callbacks](#) by W3 schools.

WATCH

1. Video: [Callbacks](#) (5m 29s) by Morten Rand-Hendriksen on LinkedIn Learning.
2. Video: [Understanding what asynchronous means](#) (2m 18s) by Sasha Vodnik on LinkedIn Learning.

	<p>ACTIVITY</p> <p>Pass a second callback function to the calculator, which divides the result by two and invokes it between result calculation and result printing.</p> <p>Source code: Click here to reveal.</p>
---	--

Timeouts


JavaScript provides a tool to execute a function after the specified time. It enables us to simulate the long execution of some code fragments but can also be used to wait for other processes to finish. We'll use the `setTimeout()` method to do this. This takes the function to execute and the number of milliseconds before the time-out.

Let's modify our calculator to print the result only after the five seconds of 'calculations':

```
function print(text) {
    document.getElementById("displayer").innerHTML = text;
}
function calculator(a, b, callback) {
    let result = a + b;
    callback(result);
}
print("Waiting for the calculation to end...")
setTimeout(() => calculator(1, 2, print), 5000);
```

Note how we had to create a new function (arrow) to provide in the `setTimeout()` method. We need to pass the entire function and not only the function invocation, so we created a new function which invoked our function.

	<p>WATCH</p> <p>Video: 9.4: JavaScript setTimeout() Function – p5.js Tutorial (9m 19s) by The Coding Train on YouTube.</p>
--	---

	<p>ACTIVITY</p> <p>Change the time of the ‘calculator’ to 10 seconds.</p> <p>Source code: Click here to reveal.</p>
---	--

Intervals

Timeouts help manage the function sentence, but intervals give us even more possibilities. With intervals, we can invoke our function as many times as we want in specified time intervals. We’ll use the `setInterval()` method, which works similarly to the `setTimeout()` method.

Let’s make a dynamic timer, increasing every two seconds:

```
function print(text) {
    document.getElementById("displayer").innerHTML = text;
}
function timer(callback) {
    counter++;
    callback(counter);
}
print("Waiting...")
var counter = 0;
setInterval(() => timer(print), 2000);
```

Intervals are the perfect tool for creating advanced animations. They can also be used to refresh the website after the user has idled for a long time.

The `setInterval` method returns us an id of our interval. If we want to stop the interval, we can use the `clearInterval()` method and pass that id as the parameter.

Let's modify our code so the counter stops after 30 seconds:

```
function print(text) {  
    document.getElementById("displayer").innerHTML = text;  
}  
function timer(callback) {  
    counter++;  
    callback(counter);  
}  
print("Waiting...")  
var counter = 0;  
const id = setInterval(() => timer(print), 2000);  
setTimeout(() => clearInterval(id), 30000);
```

READ

Page: [Asynchronous JavaScript](#) by W3 schools.

WATCH

Video: [9.5: JavaScript setInterval\(\) Function – p5.js Tutorial](#)
(11m 40s) by The Coding Train on YouTube.

Promises

In this section, we'll learn how to execute some code only if our function is successfully executed. We'll also learn what to do when it fails.

We can divide our code into producing and consuming. Producing code can take time, while consuming code must wait for the result. JavaScript's Promises enable us to link these codes. Thanks to them, we can finally write asynchronous code.

Promises have the following syntax:

```
let promise = new Promise(function(resolve, reject) {  
    // "Producing Code" (May take some time)  
  
    resolve(); // when successful  
    reject();  // when error  
});  
  
// "Consuming Code" (Must wait for a fulfilled Promise)  
promise.then(  
    function(value) { /* code if successful */ },  
    function(error) { /* code if some error */ }  
);
```

Promises support two properties: state and result. There are three possible states:

- **Pending:** Our producing code is still executing, and we don't have a result yet – `promise.result` equals `undefined`.
- **Fulfilled:** The producing code is executed correctly, and the `resolve` function will execute – `promise.result` is a result value.
- **Rejected:** The producing code produced an error, and the `reject` function will execute – `promise.result` is an error object.

We can't access state and result properties ourselves, as they change without our knowledge. We must use a promise method to handle this.

We use a promise method by specifying callbacks for success and another for

failure within `Promise.then()`. Regardless of parameter names, the first function will execute after success, while the second after failure. Both are optional. The same applies to resolve and reject functions.

Let's look at a basic example showing that our Promises are asynchronous:

```
console.log( 'a' );
new Promise( function ( ) {
    console.log( 'b' );
    setTimeout( function ( ) {
        console.log( 'D' );
    }, 0 );
} );
// Other synchronous stuff, that possibly takes a very long
time to process
console.log( 'c' );
```

Unlike other programming languages, JavaScript offers only one thread. Because of this, our code won't be truly asynchronous. However, it can change the order of code execution. In this case, we have our logs in order: a, b, c, D. It shows that after seeing the timer, our program did other things instead of waiting.

Let's look at a more detailed example, including handling the result:

```
function print(text) {
    document.getElementById("displayer").innerHTML = text;
}
let promise = new Promise(function(resolve, reject) {
    let x = 0;
    if (x == 0) {
        resolve("OK");
    } else {
        reject("Error");
    }
});
promise.then(
    function(value) {print(value);},
    function(error) {print(error);}
);
```

Our browser prints “OK” for now, but if we change the x to any other value, the text will change to “Error”.

READ

1. Page: [JavaScript Promises](#) by W3 schools.
2. Page: [Understanding JavaScript Promises](#) by Node.js.

WATCH

1. Video: [Understanding promises](#) (2m 46s) by Sasha Vodnik on LinkedIn Learning.
2. Video: [JavaScript Promises In 10 Minutes](#) (11m 30s) by Web Dev Simplified on YouTube.

Async/Await

Promises are a great tool, but their code is usually pretty complicated. Fortunately, we can implement them easier by using `async` and `await` keywords.

The `async` keyword makes a function return a Promise. `Await` makes a function wait for a Promise.

Let's see how it works:

```
function print(text) {  
    document.getElementById("displayer").innerHTML = text;  
}  
  
async function display() {  
    let myPromise = new Promise(function(resolve) {  
        setTimeout(function() {resolve("Hello!");}, 3000);  
    });  
}
```

```
});  
const result = await myPromise;  
print(result);  
console.log(result); // "Hello!"  
console.log(myPromise); // Promise: ...  
}  
display();
```

Instead of using the `then()` method, we can access the result quickly with the `await` keyword. It will be accessible after our promise resolves. Remember, we can use the `await` keyword only inside an `async` function.

`Async` and `await` are truly powerful, but we can't always do the same thing as with promises. For example, we can't use them to recreate the example from the previous section. `Await` on errors will always throw the rejected value, but in our promises example, we could print it.

READ

1. Page: [JavaScript Async by W3 schools](#).
2. Page: [Modern Asynchronous JavaScript with Async and Await by Node.js](#).

WATCH

Video: [Understanding the async/await model \(1m 46s\)](#) by Sasha Vodnik on LinkedIn Learning.

What did I learn in this lesson?

This lesson provided the following insights:

- Manipulating the function sentence with callbacks, timeouts and intervals.
- Creating asynchronous code with the use of promises.
- Reducing the promises code with the use of `async` and `await` keywords.

References

W3 schools (n.d.) *HTML Tutorial*. Available at: <https://www.w3schools.com/html/default.asp> [Accessed 25 May 2022].

Node.js (n.d.) *Introduction to Node.js*. Available at: <https://nodejs.dev/learn> [Accessed 7 June 2022].

2.3. Lesson task - Basic asynchronous operations

The task

In this lesson, we learned about basic asynchronous operations in JavaScript. Now we know what callbacks, timeouts and intervals are. We can also create asynchronous code with promises.

In this task, you'll create a number guessing game. You have to get the number from the user (via prompt) and check if the number is the same as the one you hardcoded in the JavaScript file. If the values differ, you should ask the user again after 10 seconds. If the values are the same, you should end the game and congratulate the user with an alert.

2.4. Lesson - Revision

Introduction



REVISION

We already know how to program in JavaScript. This lesson will use this knowledge to create a more extensive application. We'll create an application simulating customers entering a restaurant and sitting at a table. The table can't have a smaller size than the number of customers visiting the restaurant.

Customers can sometimes be added automatically, without our knowledge. We should also be able to ask the customers to leave specific tables.

This application will show the practical use of knowledge from previous modules and lessons.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of development and debugging within a web browser.

In this lesson, we are covering the following skill learning outcomes:

- The candidate can apply knowledge of web browsers and development tools to set up a computer for use in developing and debugging small JavaScript programs.
- The candidate masters the use of relevant tools for writing, editing, and debugging JavaScript source code.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out simple programming tasks with the JavaScript language.

The customer asks for a table

In the first part, we'll add new customers to our restaurant. The customer gives their name and asks about a table of a specific size. If we have a table of this size, we should tell them its identifier and add them to our customer's array. Otherwise, we should ask them about providing a smaller size until we get the correct one.

Let's look at coding this:

project.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
```

```

        <script src="project.js"></script>
    </head>
    <body>
        <button style="width: 100px; height: 100px;"
onclick="newCustomer()">
            New customer
        </button>
    </body>
</html>

```

project.js:

```

function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}
const tableCount = 15;
let tableSizes = Array.from(Array(tableCount).keys()).map( ()
=> getRandomInt(2, 10)); //declares how many people can seat
at table of certain index
customers = [];
function newCustomer() {
    const name = prompt("Hello, what is your name?");
    let size = 0;
    let askAboutTableSize = "";
    while(true) {
        if (size == 0) {
            askAboutTableSize = "Hello " + name + "! Table
for how many people?"
        }
        else {
            askAboutTableSize = "Unfortunately we don't have
a table for that many people. Please ask the smaller table
size."
        }
        size = parseInt(prompt(askAboutTableSize));
        if(size != 0 && Math.max(...tableSizes) >= size) {
            break;
        }
    }
}

```

```

    }
    let tableNumber = 0;
    if (tableSizes.indexOf(size) >= 0) {
        tableNumber = tableSizes.indexOf(size);
    }
    else {
        tableNumber =
tableSizes.indexOf(Math.max(...tableSizes));
    }
    alert("You got a table of number: " + tableNumber + ",
which by default can fit " + tableSizes[tableNumber] + "
people.")
    customers.push([name, tableNumber]);
    console.log(customers);
}

```

First, copy this code and analyse how it works yourself.

The `getRandomInt()` function returns the random integer from the provided range.

We bind the `newCustomer()` method to our “New customer” button. We use the infinite while-loop to ask the user about the correct table size value multiple times. Note we use the `parseInt()` method to convert our string value to the integer. This is necessary since we’ll use this value as the `indexOf` parameter later.

Lastly, we add our customers to the customers’ list and show their current value in the console.

Sneaky customers

In this section, we’ll enable the attack of sneaky customers. A sneaky customer sits at a random table without us noticing. We’ll add two buttons: one to enable and one to disable the sneaky customer’s attack.

If enabled, one random customer with a random name will sit at a random table every 30 seconds. Every time a new sneaky customer is added, we should print “Shhhhhhhh” in the console.

Let's see how to solve this:

project.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
    <script src="project.js"></script>
  </head>
  <body>
    <button style="width: 100px; height: 100px;"
onclick="sneakyAttack()">
      Start sneaky attack
    </button>
    <button style="width: 100px; height: 100px;"
onclick="stopSneakyAttack()">
      Stop sneaky attack
    </button>
  </body>
</html>
```

project.js:

```
function getRandomInt(min, max) {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min)) + min;
}
function randomName() {
  return Math.random().toString(36).replace(/^[a-z]+/g,
  '').substr(0, 5);
}
const tableCount = 15;
let tableSizes = Array.from(Array(tableCount).keys()).map( ()
=> getRandomInt(2, 10)); //declares how many people can seat
at table of certain index
customers = [];
let sneakyAttackId = 0;
function sneakyAttack() {
  sneakyAttackId = setInterval(() => {
```

```

        customers.push([randomName(), getRandomInt(0,
tableSizes.length)]);
        console.log("Shhhhhhhh");
        return;
    }, 30000);
    console.log()
}
function stopSneakyAttack() {
    if(sneakyAttackId != 0) {
        clearInterval(sneakyAttackId);
        console.log("Sneaky attack stopped.")
    }
}

```

First, copy this code and analyse how it works yourself.

Intervals are a great tool to solve this problem. We can bind functions starting and clearing the interval to our buttons. We can execute everything responsible for a sneaky attack within the interval parameter function.

Free the table

If we only add new customers, our restaurant will get too crowded. In this section, we'll implement a tool to free a table. We want to remove all customers from the table of a provided index. We get the index via prompt as often as necessary until it's in the correct format.

Let's see how to solve this:

project.html:

```

<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
    <script src="project.js"></script>
  </head>
  <body>

```

```

        <button style="width: 100px; height: 100px;"
onclick="freeTable()">
            Free table
        </button>
    </body>
</html>

```

project.js:

```

function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}
const tableCount = 15;
let tableSizes = Array.from(Array(tableCount).keys()).map( ()
=> getRandomInt(2, 10)); //declares how many people can seat
at table of certain index
customers = [];
let sneakyAttackId = 0;
function freeTable() {
    let index = parseInt(prompt("Provide an index to free its
table. It should be greater or equal to 0, but less than " +
tableCount + "."));
    while (index < 0 || index >= tableCount) {
        index = parseInt(prompt("Wrong index. Provide an
index to free its table. It should be greater or equal to 0,
but less than " + tableCount + "."));
    }
    console.log(customers);
    for(let i=customers.length-1; i>=0; i--) {
        if(customers[i][1] == index)
            customers.splice(i,1);
    }
    console.log(customers);
}

```

First, copy this code and analyse how it works yourself.

We can see how powerful the tool's splice function is, even though we must be

careful about using it. If we remove elements before our iterator, it will affect our loop. While executing such operations, it's best to print a state of the array before and after the deletion of the elements to ensure everything went the way we expected.

The final code

You can find our final application code here:

project.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
    <script src="project.js"></script>
  </head>
  <body>
    <button style="width: 100px; height: 100px; vertical-align: top;" onclick="newCustomer()">
      New customer
    </button>
    <button style="width: 100px; height: 100px; vertical-align: top;" onclick="sneakyAttack()">
      Start sneaky attack
    </button>
    <button style="width: 100px; height: 100px; vertical-align: top;" onclick="stopSneakyAttack()">
      Stop sneaky attack
    </button>
    <button style="width: 100px; height: 100px; vertical-align: top;" onclick="freeTable()">
      Free table
    </button>
  </body>
</html>
```

project.js:

```

function randomName() {
    return Math.random().toString(36).replace(/^[^a-z]+/g,
    '').substr(0, 5);
}
function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}
const tableCount = 15;
let tableSizes = Array.from(Array(tableCount).keys()).map( ()
=> getRandomInt(2, 10)); //declares how many people can seat
at table of certain index
customers = [];
let sneakyAttackId = 0;
function freeTable() {
    let index = parseInt(prompt("Provide an index to free its
table. It should be greater or equal to 0, but less than " +
tableCount + "."));
    while (index < 0 || index >= tableCount) {
        index = parseInt(prompt("Wrong index. Provide an
index to free its table. It should be greater or equal to 0,
but less than " + tableCount + "."));
    }
    console.log(customers);
    for(let i=customers.length-1; i>=0; i--) {
        if(customers[i][1] == index)
            customers.splice(i,1);
    }
    console.log(customers);
}
function sneakyAttack() {
    sneakyAttackId = setInterval(() => {
        customers.push([randomName(), getRandomInt(0,
tableSizes.length)]);
        console.log("Shhhhhhhh");
        return;
    }, 30000);
    console.log()
}

```

```

}
function stopSneakyAttack() {
    if(sneakyAttackId != 0) {
        clearInterval(sneakyAttackId);
        console.log("Sneaky attack stopped.")
    }
}
function newCustomer() {
    const name = prompt("Hello, what is your name?");
    let size = 0;
    let askAboutTableSize = "";
    while(true) {
        if (size == 0) {
            askAboutTableSize = "Hello " + name + "! Table
for how many people?"
        }
        else {
            askAboutTableSize = "Unfortunately we don't have
a table for that many people. Please ask the smaller table
size."
        }
        size = parseInt(prompt(askAboutTableSize));
        if(size != 0 && Math.max(...tableSizes) >= size) {
            break;
        }
    }
    let tableNumber = 0;
    if (tableSizes.indexOf(size) >= 0) {
        tableNumber = tableSizes.indexOf(size);
    }
    else {
        tableNumber =
tableSizes.indexOf(Math.max(...tableSizes));
    }
    alert("You got a table of number: " + tableNumber + ",
which by default can fit " + tableSizes[tableNumber] + "
people.")
    customers.push([name, tableNumber]);
    console.log(customers);
}

```

```

}
function freeTable() {
    let index = parseInt(prompt("Provide an index to free its
table. It should be greater or equal to 0, but less than " +
tableCount + "."));
    while (index < 0 || index >= tableCount) {
        index = parseInt(prompt("Wrong index. Provide an
index to free its table. It should be greater or equal to 0,
but less than " + tableCount + "."));
    }
    console.log(customers);
    for(let i=customers.length-1; i>=0; i--) {
        if(customers[i][1] == index)
            customers.splice(i,1);
    }
    console.log(customers);
}

```

WATCH

1. Video: [The Easiest Javascript Game Ever](#) (8m 33s) by KnifeCircus on YouTube.
2. Video: [Javascript Falling Ball Game Tutorial](#) (11m 50s) by KnifeCircus on YouTube.
3. Video: [How To Code The Snake Game In Javascript](#) (46m 26s) by Web Dev Simplified on YouTube.

What did I learn in this lesson?

This lesson provided the following insights:

- Using many concepts learned during this course to create a more significant application.
- The revision of information learned in past lessons.

References

W3 schools (n.d.) *HTML Tutorial*. Available at:

<https://www.w3schools.com/html/default.asp> [Accessed 25 May 2022].

2.4. Lesson task - Revision

The task

We created a more complex application in this lesson using our knowledge from numerous previous lessons. Before starting this task, ensure you understand the application well.

Right now, our application works well, but it isn't perfect. Many customers can sit at the same table. Your task is to fix this. Before adding a new customer, you must ensure nobody is sitting at that table. You shouldn't add a new customer if there are no correct tables.

What is the best way to do this? Is the customer array enough, or should you create a new one storing the state of every table? The decision is up to you.

2.5. Lesson - Self-study

Introduction



This is a self-study lesson that consolidates knowledge of the second module. It contains videos reminding parts of programming in JavaScript, such as data structures and algorithms, classes and asynchronous operations. It also includes various online exercises.

Materials

WATCH

1. Video course: [Programming Foundations: Algorithms](#) (1h 45m) by Joe Marini on LinkedIn Learning.
2. Video course: [JavaScript: Classes](#) (37m) by Emmanuel Henri on LinkedIn Learning.
3. Video course: [JavaScript: Async](#) (1h 31m) by Sasha Vodnik on LinkedIn Learning.

2.5. Lesson task - Self-study

The task

Attempt the following tasks:

1. Create a 'Hello' function, that returns 'Hello world'.
2. Define two functions. The first function a should return 'Hello a!' and the second function should return 'Hello b!'
3. Write a function 'Greet' that takes one parameter and returns the 'Hello <parameter>!'
4. Write a function 'Log' that takes a parameter and logs it.
5. Write a function 'MyStringFunction' that takes a string and returns the number of characters in that string. (Make use of string.length).
6. Write a function 'Upper' that takes a string and returns the uppercase of the string.
7. Write a function 'Add' that takes two numbers and returns their sum.

8. Write a function 'Modulo' that takes a natural number and a dividend and returns the remainder of the modulo (%) of the two numbers.
9. Write a function 'MinMax' that finds the min and max numbers of an array of numbers.
10. Write a function 'Parse' that takes a string ('20') and a number (5) and returns the multiplication of the string and the number (use the parseInt function to read the string as a number).
11. Write a function that accepts two numbers and checks if the numbers are equal. After the condition is checked, log an appropriate message to the console.
12. Write a function 'toArray' that accepts four values and returns an array of these values.
13. Write a function 'ArrayLength' that takes an array of values and returns the array's length.
14. Write a function 'ArraySort' that takes an array of values and returns a sorted array. (Make use of the sort() function).
15. Create a js file containing a line comment and a multiline comment/block comment.
16. Write a function 'MyForLoop' that loops until a variable equals 10. The variable should be incremented by '1' in each iteration. (Make use of the for-loop).
17. Write a function 'MyWhileLoop' that declares a variable set to '10' and loops until that variable is equal to '0'. The variable should be decremented by '1' in each iteration. (Make use of the while-loop).
18. Write a function that checks if a given value is a number. (Make use of the isNaN() function). Return a Boolean of the result.
19. Write a function 'SplitFunction' that accepts the string '7+12+100' and splits it into individual values, then summing these values. (Make use of the split() and parseInt() functions). Return the summed result.
20. Write a function 'Factorial' that accepts a positive integer and calculates the factorial of that integer. Return the result.



**Noroff
Education**