JavaScript (ES6)
for Beginners

JavaScript and jQuery for Beginners

You can reach the author at info@csharpschool.com.

Disclaimer - Who is this book for?

It's important to mention that this book isn't meant to be a *get-to-know-it-all* book; it's more on the practical and tactical side where you learn as you progress through the examples and build a real website in the process. Because I dislike reading irrelevant fluff (filler material) not relevant to the tasks at hand, I assume that we agree on this and will, therefore, only include relevant information pertinent for the tasks at hand, making the book shorter and more condensed saving you time and effort in the process. The goal is for you to have created several JavaScript and jQuery applications upon finishing this book. You can always look into details at a later time when you have a few projects under your belt.

About the author

Jonas started a company back in 1994, focusing on education in Microsoft Office and the Microsoft operating systems. While still studying at the university in 1995, he wrote his first book about Windows 95 as well as course materials.

In the year 2000, after working as a Microsoft Office consultant for a couple of years, he wrote his second book about Visual Basic 6.0.

Between 2000 and 2004, he worked as a Microsoft instructor with two of the largest education companies in Sweden. First teaching Visual Basic 6.0, and when Visual Basic.NET and C# were released, he started teaching these languages as well as the .NET Framework—teaching classes on all levels for beginner to advanced developers.

From the year 2005, Jonas shifted his career towards consulting once again, working hands-on with the languages and framework he taught.

Jonas wrote his third book *C# programming* aimed at beginners to intermediate developers in 2013 and has published several other books since then.

Contents

01—Introduction

History of JavaScript

Install Server in VS Code

In this course, you use a server called Live Server to load your application and provide real-time updates in the browser when you save your code. You can use the F12 Developer Tools in the browser to debug your application either through breakpoints or logging information to the console tab.

1. Click the **Extensions** icon in the left toolbar.
2. Search for Live Server and click the **Install** button for that extension.

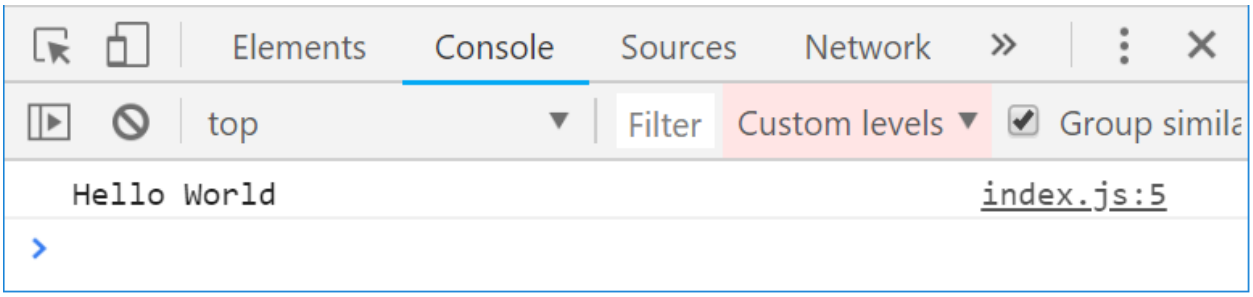
Useful Shortcuts in VS Code

You can move a line of code in Visual Studio Code by holding down the **Alt** key and hitting the **arrow-up** or **arrow-down** keys on the keyboard.

Developer Tools (F12)

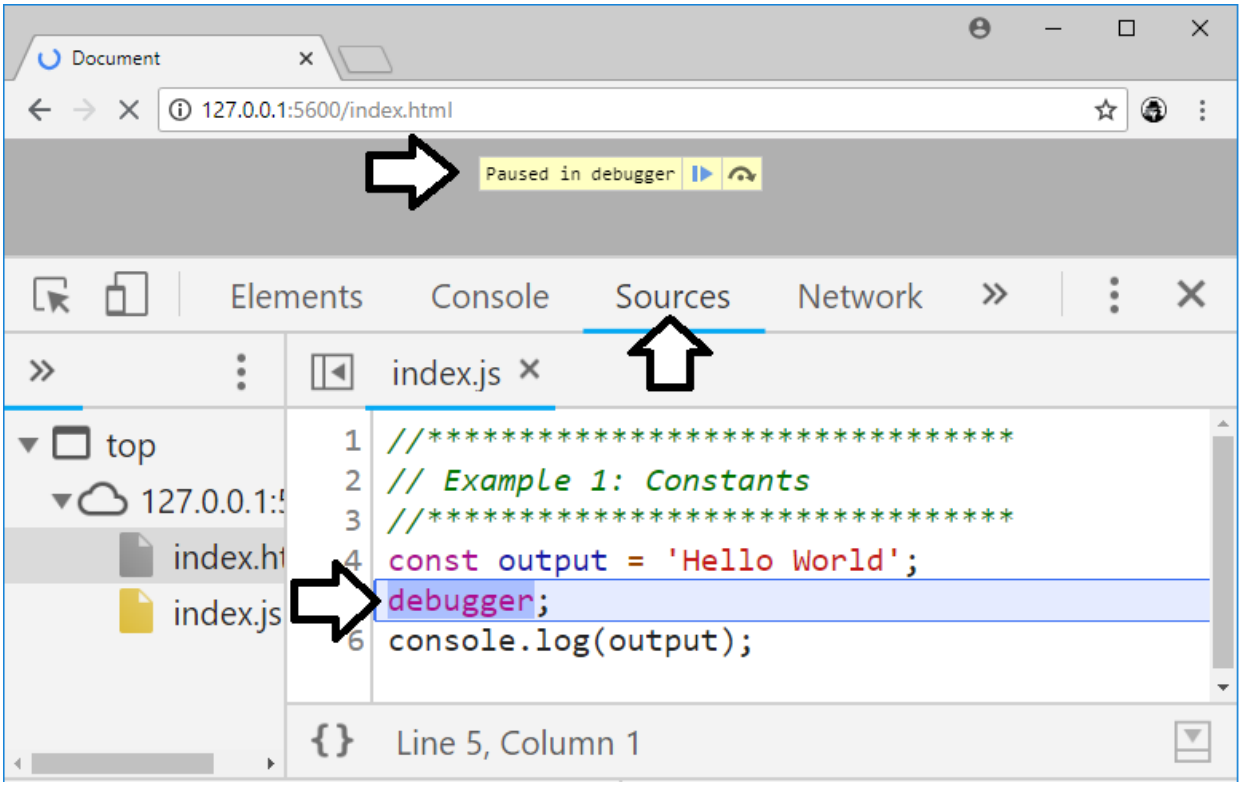
Throughout this course, you debug your code in different ways using the web browser's developer tools that you can open by pressing **F12** on the keyboard when the browser is active.

In the beginning, you use the `console.log()` function to emit information to the Console tab. You place the value, variable, constant, or object you want to inspect inside the parenthesis.



Later, you use the `debugger` keyword to halt the execution of the script at a certain point in the program flow to inspect values. You can also place breakpoints directly in the source code in the Sources tab in the F12 window by clicking on the line number where you want the code to halt.

A toolbar appears at the top of the browser, where you can resume execution, or step over the next function call.



02—Language Features

When coding in JavaScript, pretty much everything is declared using camel-casing—the first word in the name begins with a lower-case letter, and each subsequent word in the name begins with an upper-case letter.

So, if you, for instance, declare a variable for storing an id, you could name it `id`—note that the name is in all lower-case because the first (and only) word in the name should begin with a lower-case letter. If you, on the other hand, declare a variable for a first name, you could name it `firstName`—note that the first word in the variable name is in lower-case, whereas the second word begins with an upper-case letter.

Comments

When you write code, you want to document your code to make it easier for other developers to understand what the code does, and sometimes you want to comment out parts of your code to test some other code in its place.

You can add comments to your code in two ways, single-line comment or multi-line comment.

You create a single-line comment by adding two slashes at the beginning or end of the code. Placing it at the beginning of the code line comments out the whole line of code while placing it at the end makes it possible to add comments to the code without commenting out the code itself.

```
output = 'A new world'; // Will generate an error
//console.log(output);
```

Multi-line comments are just what they sound like; you comment out a block of code (several lines). You add a block-comment by beginning the block with a slash and asterisk and ending with an asterisk and a slash. You can add a block-comment by writing the slashes and asterisk manually, selecting **Edit-Toggle Block Comment** in the menu, or by selecting all the rows that you want to comment out and press **Shift+Alt+A** on the keyboard.

```
/* const output = 'Hello World';
console.log(output); */
```

It can also create more detailed comments about the code.

```
/*********************************
  Example 3: Changing a Constant
 *********************************/
```

Selecting an HTML Element by Id with JavaScript (getElementById)

Now, the fun begins. In the following sections, you create a simple web page and interact with elements on the page through what's called the DOM (Document Object Model). The DOM is an object representation of the web page's content, such as HTML5 tags and metadata about the page.

To write HTML5, you must first add a *.html* page to the project, if there isn't already one that you can use. To add a new HTML document, click on the **New File** button in the desired section of the **Explorer** window, which you can open with the **Explorer** button to the left of the code editor. The most used name for the main page of a site is *index.html*.

In Visual Studio Code, there is a code snippet that you can use to add the barebones of a page to an HTML file. Write an exclamation mark (**!**) and press the **Tab** key once on the keyboard, this should add code like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
</body>
</html>
```

Let's add some HTML elements to the page. We can start with an <h1> heading, which is the largest heading, and give it the id **title**. Then add a <p> paragraph with the id **text-area** below the heading, a with the id **content-area** below the paragraph, and a with the id **example-area** below the **content-area** . You will use the last to output results during the initial exercises.

Later, you use the **id** attributes to target the elements from JavaScript.

```
<body>
```

```
    <h1 id="title"></h1>
    <p id="text-area">Some text</p>
    <div id="content-area"></div>
    <div id="example-area"></div>
</body>
```

Adding a JavaScript File to the HTML Document

To add a JavaScript file, click on the **New File** button in the desired section of the **Explorer** window, which you can open with the **Explorer** button to the left of the code editor. It's important to add the *.js* extension to the file for the file to act as a JavaScript file. The name of the file usually denotes its purpose; you can create as many files as needed and link them in the same way described below. If the JavaScript file is used with the *index.html* file, then it is usually named *index.js*.

To get access to JavaScript files (.js) from the HTML page, you need to add a <script> tag pointing to your JavaScript file. If the .js files are in the same folder as the HTML file, you add **./** before the name to specify that the file is in the same folder. It is important to add the <script> tag at the very end of the <body> element, immediately above the </body> closing tag. All other HTML you add to the page should be added above the <script> tag.

```
<body>
    <!-- The elements you added earlier should be here -->
    <script src="index.js"></script>
</body>
```

The reason you add the <script> tag at the end of the <body> element is that all the HTML elements must be loaded before the JavaScript code is executed for the code to find the elements on the page.

Selecting an HTML Element by Id with JavaScript (getElementById)

To target an element by its id, you use a built-in function called **getelementById** on the **document** object and pass in the id of the desired element. The **document** object contains the page's DOM.

The following JS code would store (cache) the <p> element in a variable called **paragraph** for easy reuse and to avoid looking in the DOM every time this paragraph is used. Note that it the complete <p> element that is stored, including its content.

Example 2-1: Reading and writing to elements

```
const paragraph = document.getElementById("text-area");
```

Let's store the elements in constants as well.

```
const div = document.getElementById("content-area");
const exampleDiv = document.getElementById("example-area");
```

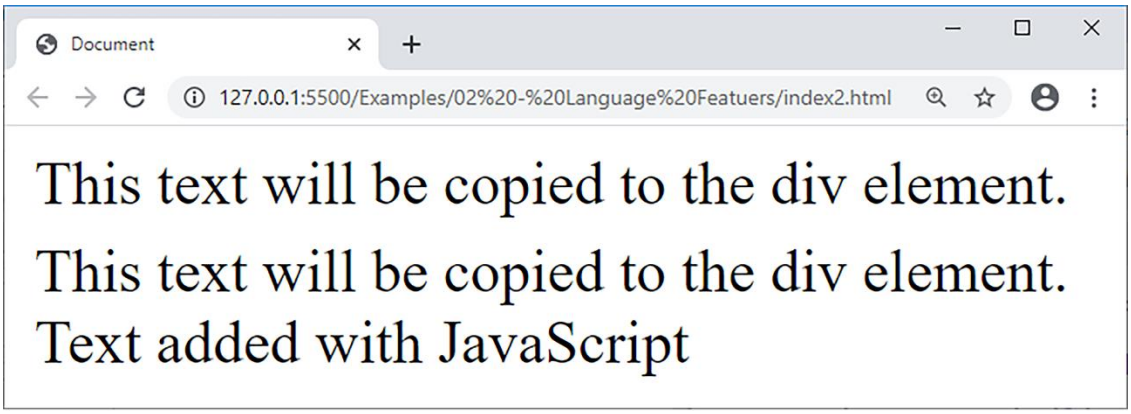
To read an element's content, you use one of its **innerHTML** or **innerText** properties; depending on if you want to read the content as HTML or text.

Let's read the HTML from the <p> element and place it in the element, note that **innerHTML** works with plain text as well. You can begin by reading the <p> element's content placing it in a constant named **paragraphHTML**.

```
const paragraphHTML = paragraph.innerHTML;
```

To write the text to the , you assign the text in the **paragraphHTML** constant to the **innerText** property on the element's object stored in the **div** constant. When you save the page, the changes should be reflected on the webpage.

```
div.innerHTML = paragraphHTML;
exampleDiv.innerHTML = 'Text added with JavaScript';
```



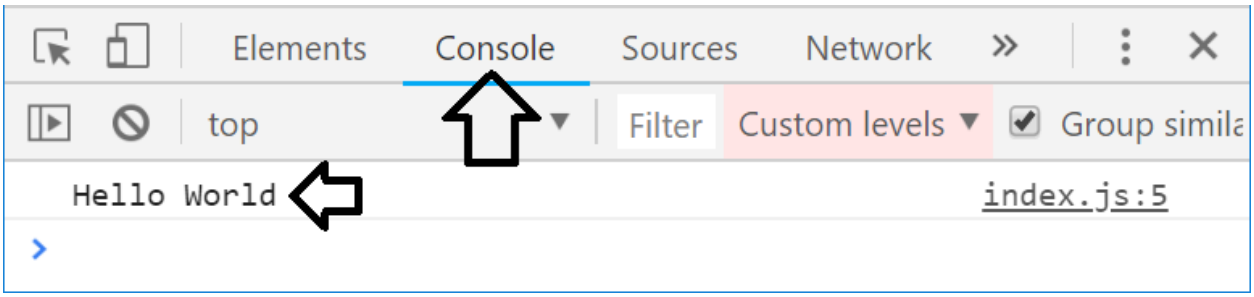
Constants

Sometimes you want to store a value that is immutable—you set it once and use it as many times as you need. To create a constant, you use the `const` keyword.

You want to use constants to avoid programmer errors in the future. If the value never should change, use a constant, otherwise use a variable.

Example 2-2: The following example creates a constant named **output** that holds the value *Hello World* and prints it to the Console debugger window in the browser developer tools (F12). A constant must be assigned a value when you create it; you cannot declare it as a name only: `const output;`

```
const output = 'Hello World';
console.log(output);
```

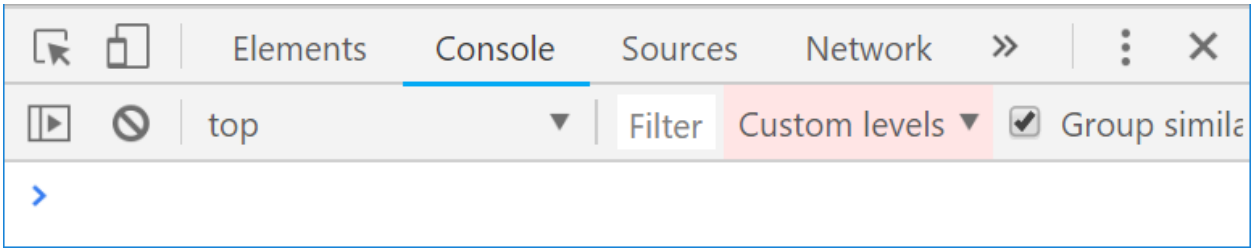


Now target the **example-area** —like in the previous example—and write the text from the **output** constant to it.

```
const div = document.getElementById("example-area");
const output = 'Hello World';
console.log(output); // Logs to F12
div.innerHTML = output;
```

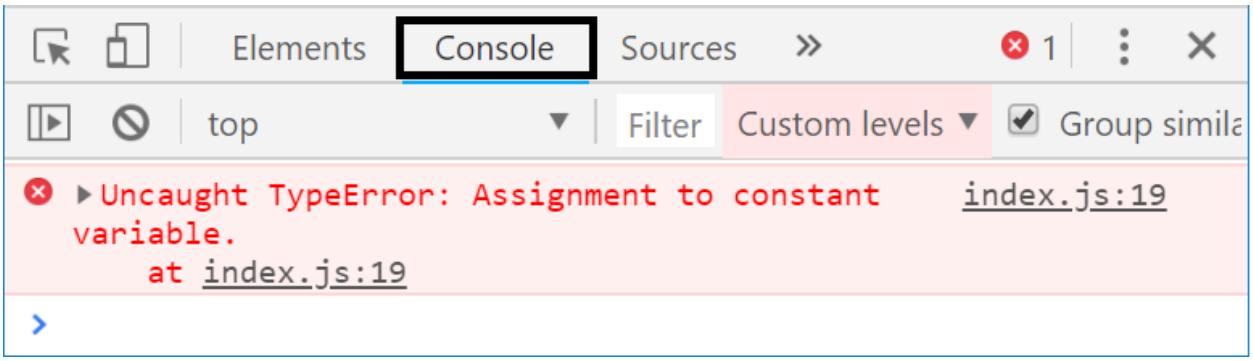
Example 2-3: Adding the debugger command between the two previously added code lines, the execution halts before the code executes, and the F12 Console tab displays the value. To continue the execution and display the value in the Console, you press **F8** on the keyboard.

```
const output = 'Hello World';
debugger;
console.log(output);
```



Example 2-4: If you try to change the value of a constant, you get an error message in the Console window, directing you to the line where the error occurred.

```
const output = 'Hello World';
output = 'A new world'; // Will generate an error
console.log(output);
```



Exercise 2-1: Create a constant named `id` and assign the value 100 to it; then print it to the Console window and inspect the value.

Exercise 2-2: Change the value of the `id` constant to 200 on a new row between the two previously added rows. You should get an error when you save the code.

Exercise 2-3:

1. Comment out the line where you change the value.
2. Add the `debugger` command below the comment.
3. Save the code. The execution should halt at that line in the F12 Sources tab.
4. Switch to the Console tab and verify that the value isn't displayed in it (the execution hasn't reached that line of code yet; it's below the `debugger` command).
5. Press **F8** on the keyboard or click the **Resume code execution** button at the top of the browser.
6. Verify that the Console window displays the value of the `id` constant.

Variables: let vs. var

What is a variable? A simple analogy would be a warehouse that stores the products in containers, some small, some large. Depending on the type of item, the container has to have room to accommodate it. However, if you were to shove items into containers without labels, there would be chaos.

When programming, you also need to store items of different sizes; it could be a string of characters, like a name, or a true/false value to keep track of if something is selected. It could also be numerical values for calculations, or complete objects, such as a Car or a Cat that group all information internally for easy access. To store values in an application, you use variables; the name you give it acts as the label on a container in the warehouse—making it possible to acquire its contents. The computer's memory stores the data for quick access.

You can define a variable with either the `let` or `var` keywords, but they have different scope behavior. The scope defines where a variable should be reachable. The `let` keyword behave more like a variable does in traditional languages such as C#, Java, and C++. Therefore most developers prefer the `let` keyword, which also is best practice. Due to its behavior, `let` also lets you catch errors earlier.

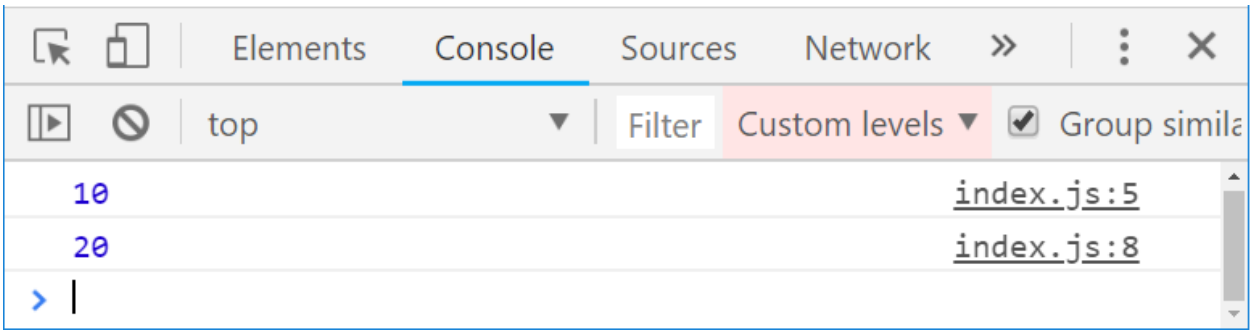
JavaScript is a dynamic language, which makes it possible to change the content from one data type to another; you could, for instance, store a string in a variable at the beginning of the program and then change it to a numerical or Boolean value later without getting an error. Dynamic variables differ between JS and other languages, such as C# and Java, where a variable's data type isn't allowed to change once it is declared.

Variable declaration

If you use the same code, only changing the keyword declaring the variable, you can see the differences. Let's look at some examples.

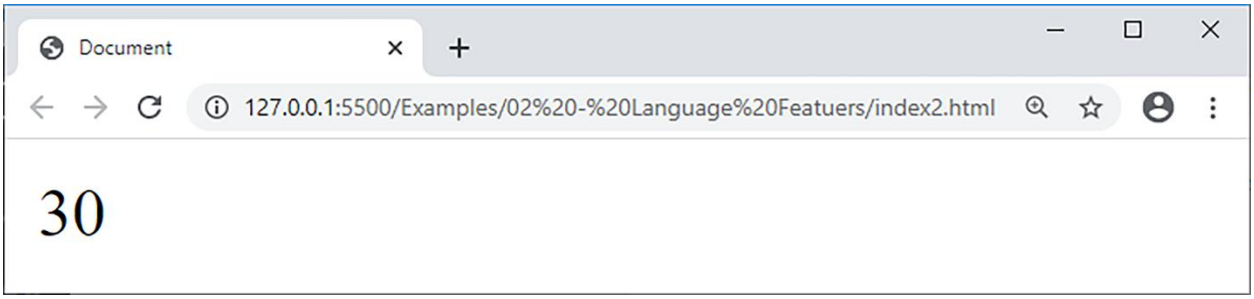
Example 2-5: If you declare variables with the `let` and `var` keyword and write them to the console, everything looks fine—the console displays the values.

```
let id1 = 10;
console.log(id1);

var id2 = 20;
console.log(id2);
```



Now, add the two variables and print the result in **example-area** .

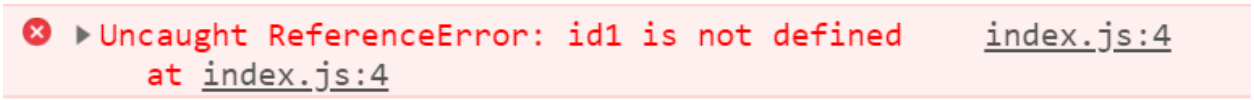
```
const div = document.getElementById("example-area");
div.innerHTML = id1 + id2;
```



Variable access

Example 2-6: If you declare a variable with the `let` keyword and try to access it before it is declared, you get an error, but if you declare the variable with the `var` keyword, the value is **undefined**.

```
console.log(id1); // error
let id1 = 10;

console.log(id2); // undefined
var id2 = 20;
```



```
undefined                                                    index.js:7
>
```

Variable Scope

Another difference is the reach of a variable—its scope. If you declare a `let` variable inside a code-block—like an if-block or a for-loop—the variable is only available inside that block. If you do the same with a variable declared with the `var` keyword, the value is available outside that block.

Example 2-7: Variable block scope.

```
if(true){
    let id1 = 10;
}
console.log(id1); // error
```

```
Uncaught ReferenceError: id1 is not defined    index.js:7
    at index.js:7
```

```
if(true){
    var id2 = 20;
}
console.log(id2); // 20
```

```
20                                                           index.js:12
>
```

Exercise 2-4: Declare a variable with the `let` keyword called *name* and assign your name to it. You can use either single- or double quotation marks around strings. Print the value in the variable to the console window and verify that it is displayed.

Exercise 2-5: Move the code that prints the value to the console above the variable declaration and then save the code. The Console window displays an error. You can move a line of code by holding down the **Alt** key and hitting the **arrow-up** or **arrow-down** keys on the keyboard.

Exercise 2-6: Change the `let` keyword to `var`. Your name should be displayed in the console window because it has a global scope.

Exercise 2-7:

1. Delete the previous code.
2. Add an if-block with **true** inside its parenthesis (to ensure that it always is executed).
3. Declare a variable with the `let` keyword called *year* inside the if-block and assign the current year to it.
4. Print the value of the variable to the console inside the if-block, below the *year* variable.
5. Save the code and verify that the Console window displays the year.

Exercise 2-8: Move the line of code that prints to the console below the closing curly-brace if the if-block and save the code. The Console window now displays an error.

Exercise 2-9: Change the let keyword to var and save the code. Verify that the Console window displays the year.

Strings and numbers

When working with numbers, it's essential to know that a number within quotes are not the same as a regular numeric value. In other words, "2.00" is not the same as 2.00 when assigned to a variable; this becomes very clear when using the plus operator to add values together.

When working with strings, you can use either single- or double quotation marks to define the beginning and end of it.

Examples 2-8: Strings and numbers.

```
let num1 = 2,
    num2 = 3.00,
    num3 = 0.55,
    num4 = 2000000,
    num5 = "2.01"; // Not considered a number

let name1 = "John Doe", // Only double quotes
    name2 = "John's",   // Mix of single- and double quotes
    name3 = 'John's',   // error: can't use only single quotes
    name4 = 'John\'s',  // Escaped single quote works
    name5 = 'John Doe'; // Only single quotes
```

You use the **+**-operator when adding numerical values, but the same operator is also used to concatenate strings; this can lead to unexpected results if you assign a numerical value as a string. When adding two float values, you can get a rounding error (see example 2-8 below). Later you use the **Math** object to round values.

Example 2-9: The **+**-operator

```
let num1 = 2.00,
    num2 = 3.00,
    num3 = 4.1,
    num4 = 5.3,
    num5 = "4.00"; // Considered a string

let firstName = "John",
    lastName = "Doe";

console.log(num1 + num2); // 5
console.log(num1 + num5); // 24.00 Converted into a string and concatenated
console.log(num1 + num2); // 5
console.log(num3 + num4); // 9.399999999999999 You would expect 9.4
```

```
console.log(firstName + lastName); // JohnDoe
console.log(firstName + ' ' + lastName); // John Doe
```

```
5                                    index.js:13
24.00                                index.js:14
5                                    index.js:15
9.399999999999999                    index.js:16
JohnDoe                              index.js:18
John Doe                             index.js:19
>
```

Example 2-10: The **/**-operator

Division works as expected in most scenarios, but dividing by zero gives the unexpected value *Infinity*, where you would expect an error.

If you divide zero by zero (0/0), you get NaN as a result, although the data type is a number.

When you calculate values, keep these quirks in mind because you will most likely run into them.

```
let num1 = 10, num2 = 0;

console.log(num1 / num2); // Infinity (you would expect an error)
```

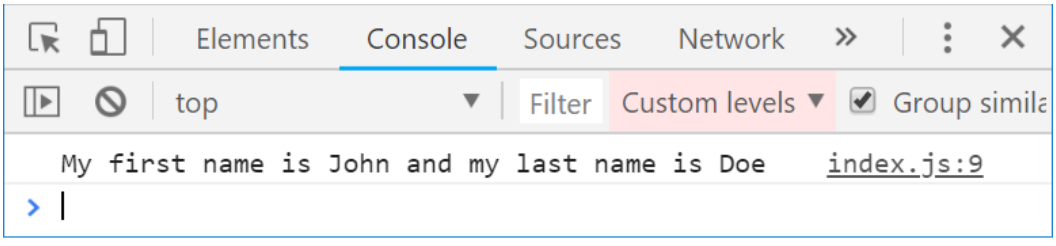
```
Infinity                             index.js:8
-Infinity                            index.js:9
NaN "number"                         index.js:10
```

Backtick Strings

You can also use backticks to inject values into a string using the ${} syntax.

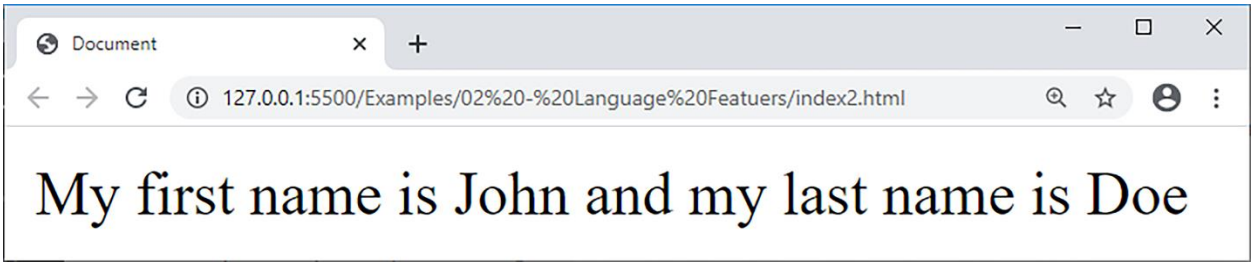
Example 2-11: Backticks and injected values

```
let firstName = "John", lastName = "Doe";
let text = `My first name is ${firstName} and my last name is ${lastName}`;
console.log(text);
```

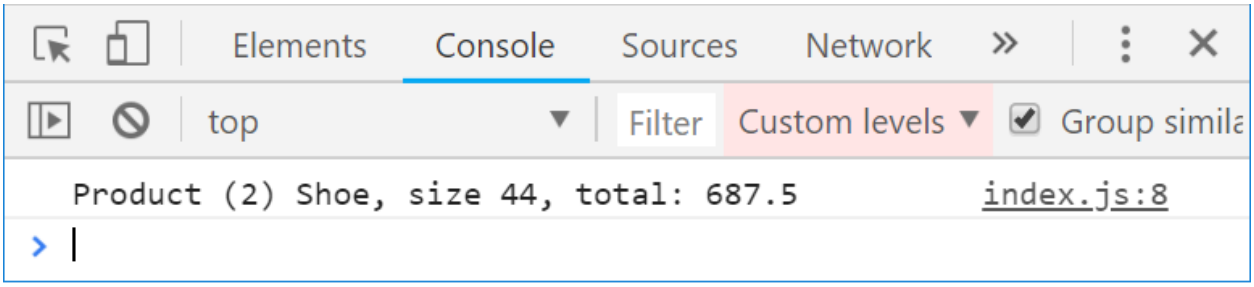


Now, display the string in the **text** variable in the **example-area** .

```
let firstName = "John", lastName = "Doe";
let text = `My first name is ${firstName} and my last name is ${lastName}`;

console.log(text);

const div = document.getElementById("example-area");
div.innerHTML = text;
```



Exercise 2-12: Use backticks and injection to create the output displayed in the image below. Declare a string variable named **product** and assign the value *shoe* to it. Declare numerical variables for the **id** (2), **size** (44), **price** (550), **vat** (0.25), and calculate the total price, including vat in a variable called **total**.

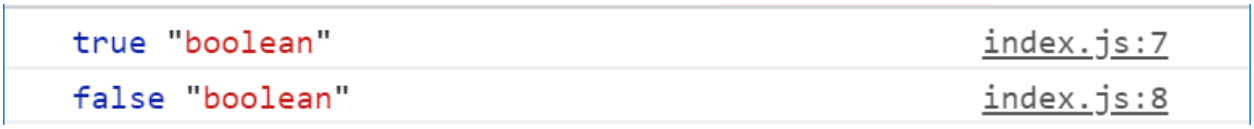


Boolean Values

Boolean values can be **true** or **false** and are usually used as flags in the program to determine if a user takes a specific action, or a condition is met, for instance: **isMultiplayer**.

Example 2-13: Boolean values.

```
let isMultiplayer = true;
let isMamal = false;

console.log(isMultiplayer, typeof(isMultiplayer));
console.log(isMamal, typeof(isMamal));
```



undefined and null

If you don't assign values to a variable, it has the value **undefined**. As a rule, you never assign **undefined** explicitly to a variable. If you want to specify that a variable shouldn't have a value, you assign it the value **null**.

Both **undefined** and **null** means that the variable contains nothing, but **undefined** specifies that the variable never was initialized, whereas **null** specifies that you intentionally blanked out the variable.

Example 2-14: undefined and null

```
let notInitialized;
let nullValue = null;

console.log(notInitialized, typeof(notInitialized));
console.log(nullValue, typeof(nullValue));
```

```
undefined "undefined"                    index.js:7
null "object"                            index.js:8
```

Arrays

An array is a variable that can hold multiple values of any type, even of different data types; this means that you can store a string, Boolean, numerical, or any other type of data in one array, but that is a bad practice. Stick to one data type per array.

Let's say that you want to store the 52 cards in a deck of cards. To create 52 variables, one for each card, would be impractical. Instead, you could declare an array variable that holds the cards.

You use the square brackets to initialize an array; you can declare an empty array by assigning empty brackets, this tells other programmers your intent that the variable should be used as an array later. You can also initialize an array when it is declared by adding a comma-separated list inside the brackets.

You access an array's values by using square brackets directly after the variable name and specify the index of the value you want to fetch. Array indices are always zero-based.

If you try to access a value from an array with an index that doesn't exist, the array returns **undefined**.

You can assign values to an array by using the array's name followed by the index inside square brackets and then assign the value like you would any value.

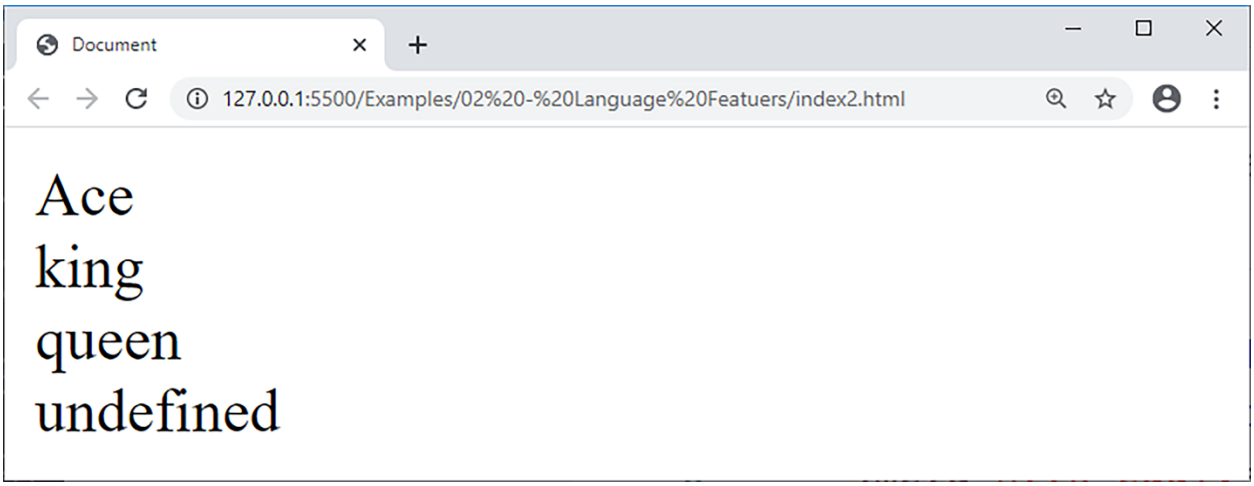
Example 2-15: Arrays

```
let emptyArray = [];    // Declares an empty array
let values = [4, 5, 10]; // Declares an array containing three values
let value = values[0];
let cards = ['Ace', 'king', 'queen'];
values[2] = 6; // Assigning a new value to the third position in the array

console.log(value);     // 4
```

```
console.log(values[0]); // 4
console.log(values[1]); // 5
console.log(values[2]); // 6
console.log(values[3]); // undefined

console.log(cards[0], cards[1], cards[2], cards[3]); // Ace king queen undefined
```

```
4                                          index.js:9
4                                          index.js:10
5                                          index.js:11
6                                          index.js:12
undefined                                  index.js:13
Ace king queen undefined                   index.js:15
```

Now, display the cards on separate rows in the **exemple-area** . You can add a </ br> element to create a new row; you might recall that **innerHTML** can handle HTML elements as well as text.

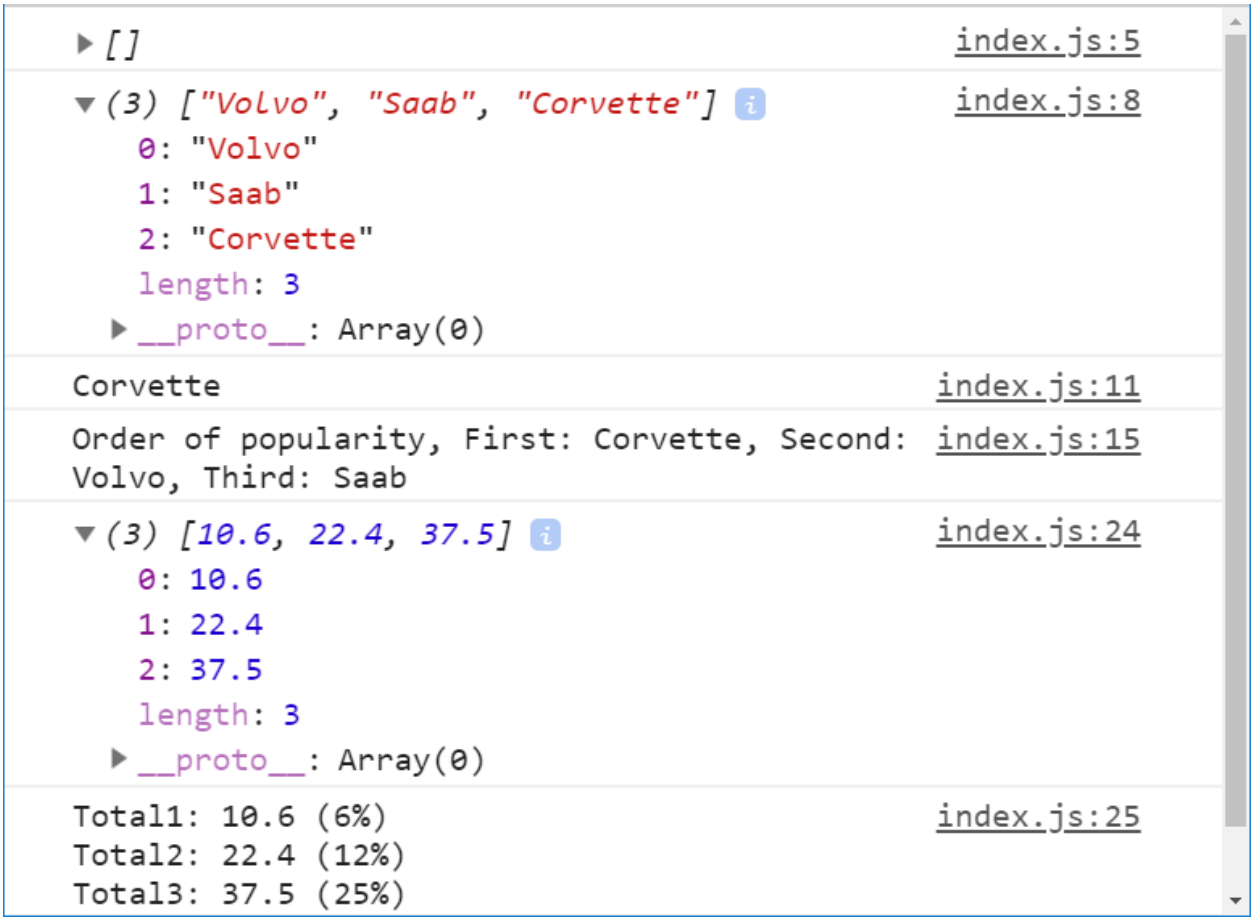


Exercise 2-11: Arrays

See image below for guidance.

1. Create an empty array named **emptyArray** and print it to the console.
2. Create an array named **cars** with three cars, choose the car models you like best. Print the array content to the console. In the console, click the chevron to the left of the array to expand the values in it. Listed inside are the values with their respective indices.
3. Use the array index to pick one car and store it in a variable called **myFavoriteCar**. Print the variable's content to the console.
4. Use backticks to create the following result by injecting the three values from the array into the string (the car models could differ depending on what you added to your array):
   Order of popularity, First: Corvette, Second: Volvo, Third: Saab.

5. Now, let's do some calculations and display the result in the console.
    a. Add an array called **price** with values 10, 20 and 30.
    b. Add an array called **vat** with the values 0.06, 0.12, and 0.25.
    c. Add an empty array called **total** that stores the calculated totals.
    d. Calculate the total for the three prices and store the values in the **total** array at the same indices as the values in the **price** and **vat** arrays. Use the price and vat at the same indices and the formula: *total = price + price * vat*.
    e. Print the **total** array to the console and expand it to inspect its values.
    f. Create a backtick string that prints each total with its vat on a separate line.
    Total1: 10.6 (6%)
    Total2: 22.4 (12%)
    Total3: 37.5 (25%)



Array Features

When iterating over an array, you need to know the number of elements in the array. To find out the number of elements, you use the **length** property of the array.

To add values to an array, you can assign values as you did in the previous section, or you can use the **push()** method in the array variable.

To remove the last element in an array (and store it in a variable), you use the **pop()** method.

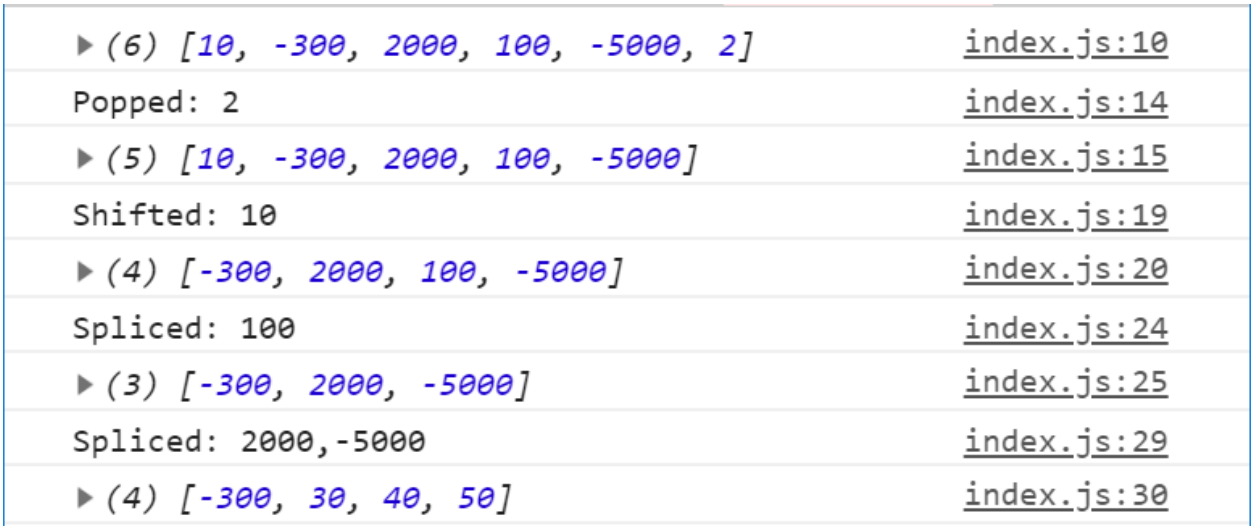
To remove the first element in an array (and store it in a variable), you use the **shift()** method.

To remove one or more values inside an array (and store it in a variable), you use the **splice(startIdx, numberToRemove)** method. You can use **splice** to add items after items have been removed by specifying the values as a comma-separated list after the **numberToRemove** value.

If you like to read more about methods you can use with arrays, you can Google *Javascript Array methods* and select the link for W3Schools. You learn more about some of these methods later in the course.

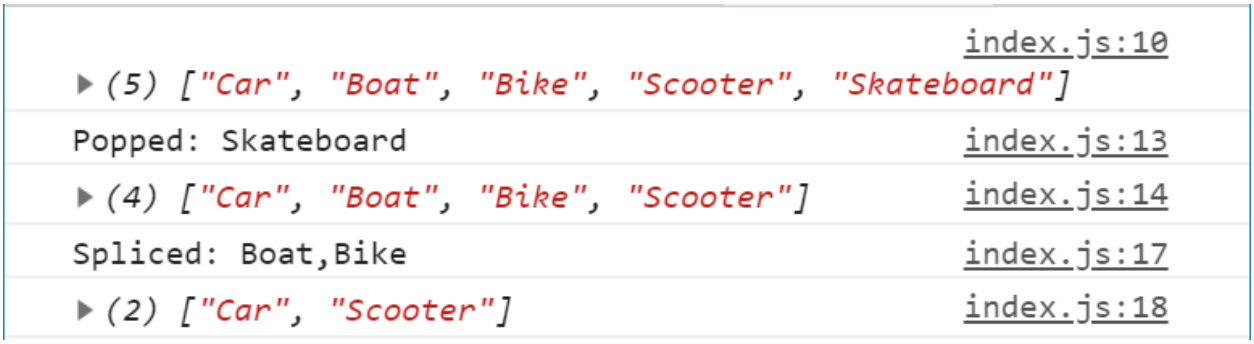
Example 2-15: Manipulating arrays.

```
let values = [10, -300, 2000];

// Add values to the array
values.push(100);
values.push(-5000);
values.push(2);
console.log(values);

// Remove the last value
let result = values.pop();
console.log(`Popped: ${result}`);
console.log(values);

// Remove the first value
result = values.shift();
console.log(`Shifted: ${result}`);
console.log(values);

// Remove values inside an array
result = values.splice(2, 1);
console.log(`Spliced: ${result}`);
console.log(values);

// Remove values and add new values
result = values.splice(1, 2, 30, 40, 50);
console.log(`Spliced: ${result}`);
console.log(values);
```



Exercise 2-12: Manipulating Arrays.

1. Add an array named **vehicles** and add the values *Car* and *Boat* to it. Print out the array to the console and inspect the values.
2. Add the values *Bike*, *Scooter,* and *skateboard* to the array using the **push()** method. Print out the array to the console and inspect the values.
3. Fetch the last value in the array with the **pop()** method and store it in a variable named **vehicle**. Print the value to the console with backtick syntax, and the text *Popped:* in front of the value and print the array itself on a separate line in the console.
4. Use the **splice()** to remove two values beginning at index *1* and store the result in the **vehicle** variable. Print the value to the console with backtick syntax, and the text *Spiced:* in front of the value and print the array itself on a separate line in the console.



Slice

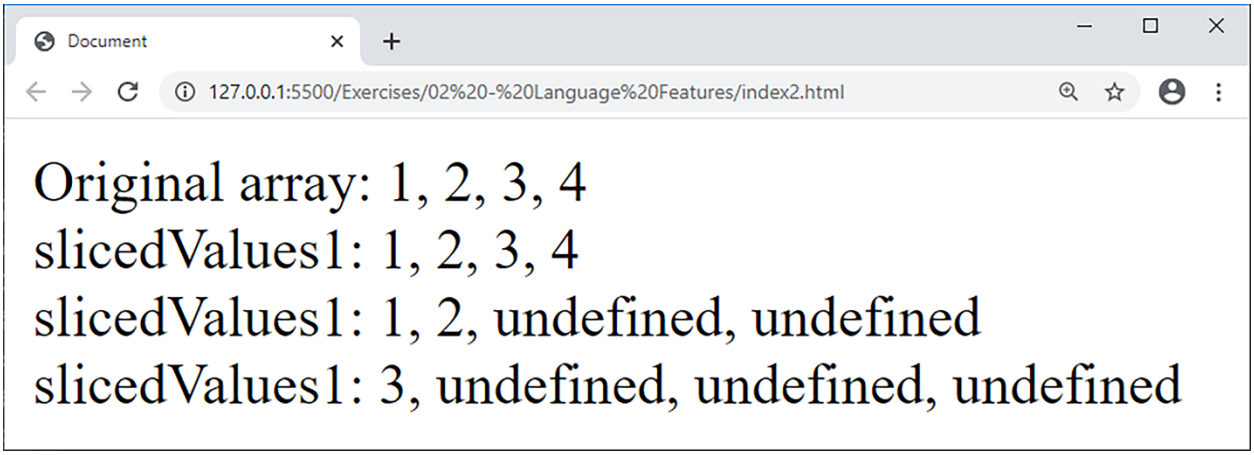
You can use the **slice()** function on an array to copy the entire array or a segment of the array. The function's first parameter is the index where to start copying from the array; the second is the index of the element where the copying ends. You can leave out the second index if you want to copy from the start index to the end of the array.

```
let values = [1, 2, 3, 4];
let slicedValues1 = values.slice(0);    // Copies the entire array
let slicedValues2 = values.slice(0, 2); // [1, 2]
```

```
let slicedValues3 = values.slice(2, 3); // [3]
```

Exercise 2-13: Copy parts of an array.

Add the JavaScript above to a JS file and display the content from the four arrays in a with the id **exercise-area** using a backtick string. Note that **undefined** displays for the values that don't exist in the 3rd and 4th arrays; they only contain two and one values respectively, not four like the two first arrays.



Spread (...)

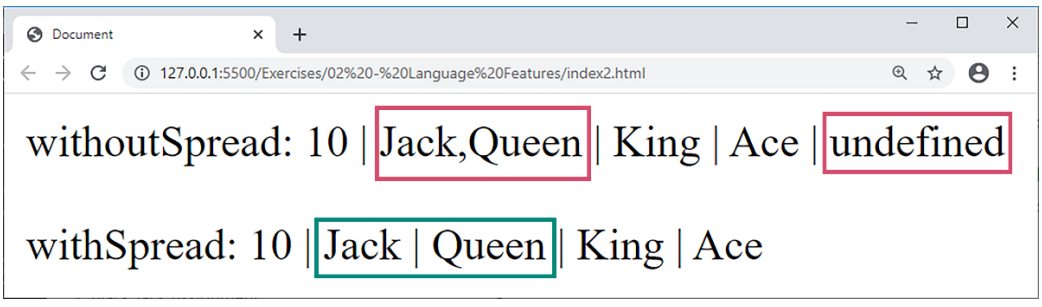
You can use the spread operator (...) to convert an array into its values; this can be useful if you want to add an array into another array, or to get the value from an array with one element.

```
var mid = [3, 4];
var arr1 = [1, 2, mid, 5, 6];    // [1, 2, [3, 4], 5, 6]
var arr2 = [1, 2, ...mid, 5, 6]; // [1, 2, 3, 4, 5, 6]
```

Exercise 2-14: Flatten an array into its items

Add an array named **cardsToInsert** that has two string elements with the text, *Jack and Queen*, to the JS file. Add two new arrays named **withoutSpread** and **withSpread** that both have three strings with the text *10*, *King*, and *Ace*. Insert the **cardsToInsert** array between the *10* and *King* values in the **withoutSpread** array without using the **spread** (...) operator. Insert the **cardsToInsert** array between the *10* and *King* values in the **withSpread** array using the **spread** (...) operator. Display the content from the **withoutSpread** and **withSpread** arrays in a with the id **exercise-area** using a backtick string.

Note that *Jack* and *Queen* store as one element in the **withoutSpread** array, and the last element is **undefined** as a result. The values added as individual values in the **withSpread** array.



Splice

You can use the **splice()** array function to remove elements from an array. The first argument is the start index, and the second, the number of elements to remove from the start index; this is useful if you want to remove an element at a given position in an array, for instance, when shuffling a deck of cards.

```
var values = [1, 2, 3, 4, 5, 6];
let result = values.splice(0, 1); // [1]

var values = [1, 2, 3, 4, 5, 6];
let result = values.splice(2, 3); // [3, 4, 5]

var values = [1, 2, 3, 4, 5, 6];
let result = values.splice(1, 2); // [2, 3]
```

Exercise 2-15: Remove items from an array

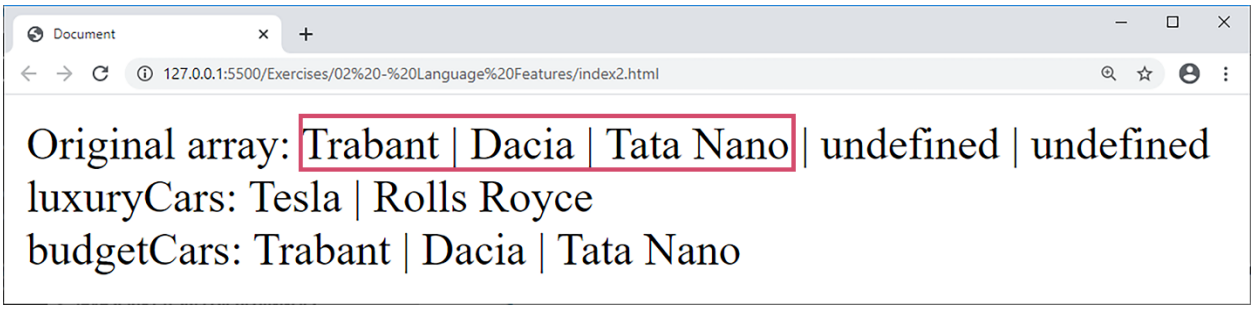
Add an array named **cars** that contains the following cars: *Trabant*, *Tesla*, *Dacia*, *Tata Nano*, and *Rolls Royce*. Add two empty arrays named **luxuryCars** and **budgetCars**.

Use the **splice** function to remove the *Tesla* and *Rolls Royce* cars and add them to the **luxuryCars** array using the **push** function.

Add the remaining cars to the **budgetCars** array using the spread (...) operator.

Print the content of the three arrays in the **exercise-area** using a backtick string.

Note that the luxury cars are removed from the original array while the budget cars are copied; the two removed values are displayed as **undefined** when trying to access all five original values because only three values remain in the array.



Find

You can use the **find()** array function to locate and return the first element in the array matching the fat arrow (=>) condition. The function returns **undefined** if no matching element is found. Here we use the fat arrow syntax to make the function call as compact as possible; **value** contains one value from the array at a time and is compared with the comparison value.

```
let values = [1, 2, 3, 4, 5, 6];
let result1 = values.find(value => value === 3);  // 3
let result2 = values.find(value => value === 10); // undefined
```

Sometimes you might need a Boolean; as a result, to evaluate if a value is present in the array. To achieve this, you can check that the result isn't **undefined**; if a value is found, the result is that value; therefore the result of checking if the result isn't **undefined** results in **true**—any value apart from **undefined** is **true**, and conversely, if no match is found the find function returns **undefined**, which results in **false** because **undefined !== undefined** is always **false**—they are equal.

```
let values = [1, 2, 3, 4, 5, 6];
let result3 = values.find(value => value === 3) !== undefined;  // true
let result4 = values.find(value => value === 10) !== undefined; // false
```

Exercise 2-16: Locating an element in an array

Add an array named **cars** that contains the following cars: *Trabant*, *Tesla*, *Dacia*, *Tata Nano*, and *Rolls Royce*.

Call the **find** method and locate the *Tesla* and store it in a variable named **tesla**.

Call the **find** method and locate the *Rolls Royce* and store it in a variable named **rollsRoyce**.

Call the **find** method and try to locate a *Volvo* and store it in a variable named **volvo**.

Call the **find** method and locate the *Tesla*; return **true** if it exists and **false** if it doesn't. Store the result in a variable named **exists**.

Call the **find** method and try to locate a *Saab*; return **true** if it exists and **false** if it doesn't. Store the result in a variable named **missing**.

Print the content in the variables in the **exercise-area** using a backtick string.

03—Program Flow

You can use conditional statements (if and switch) to alter the program flow depending on a condition. You can also use loops (for and while) to iterate over a set of values (in an array).

If

When evaluating whether an if-block or an else-block executes, you use comparison operators. Note the difference between == and ===; triple equal signs evaluates the value based on value and data type, whereas the double only evaluate on value.

In JavaScript, the expression in if and else if blocks work with **falsy** and **truthy** values and not strictly **true** or **false**; this means any value that is not **falsy** is **truthy** and evaluates as **true**. These values are considered **falsy** and evaluate as **false**: false, 0, empty strings (“” or ‘’), null, undefined and NaN. Here are some examples of **truthy** values: true, 0.5 and “0”.

==, !=	Compares two values but not on a type-specific basis, so “1” == 1 evaluates to **true** and “1” != 1 **false**.
===, !==	Compares two values on a type-specific basis, so “1” === 1 is considered **false** because one value is a string and the other is a number. 1 === 1, however, would return **true** because both are numbers.
>=, <=	Determines if two values are greater than or less than each other. Type-specificity is not an issue here.

Example 3-1: Comparisons using == and ===

```
"1" == 1:  true          "1" != 1:  false
"1" === 1: false         "1" !== 1: true
1 == 1:    true          1 != 1:    false
1 === 1:   true          1 !== 1:   false
"1" >= 1:  true          "1" != 2:  true
"1" <= 1:  true          "1" !== 2: true
1 >= 1:    true          1 != 2:    true
1 <= 1:    true          1 !== 2:   true
```

The is statement takes an expression that must evaluate to **true** for its code-block to be executed.

```
if(expression)
{
    console.log('Execute if expression is true');
}
```

If the expression only executes one line of code when the expression is **true**, then you can omit the curly-braces.

```
if(expression)
    console.log('Execute if expression is true');
```

Or.

```
if(expression) console.log('Execute if expression is true');
```

Example 3-2: If statement.

```
let value1 = 100
    value2 = '100';
```

```
let result = null;

if(value1 == value2)
{
    result = 'same';
}

console.log(result); // same

result = null;
if(value1 === value2)
    result = 'same';

console.log(result); // null
```

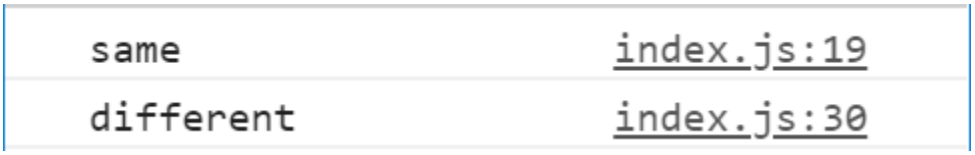
If…else

You can add an else-block that executes if the expression evaluates to **false**. As with if-blocks, you can omit the curly-braces if the else-block only executes one line of code.

Example 3-3: if…else Statements.

```
let value1 = 100
    value2 = '100';
let result = null;

if(value1 !== value2)
    result = 'same';
else
    result = 'different';

console.log(result); // same

if(value1 === value2)
{
    result = 'same';
}
else
{
    result = 'different';
}
```

```
console.log(result); // different
```



If…else if…else

You can add as many **else if**-blocks as you need with expressions that need to evaluate to **true** for the block to execute. Only one block executes in an if…else if…else statement, and the else-block executes if all expressions evaluate to **false**. As with if-blocks, you can omit the curly-braces if only one line of code should be executed in the **else if**-block.

Example 3-4: if…else if…else Statements.

```
let value1 = 100
    value2 = 200;
let result = null;

if(value1 === value2)
    result = 'same as';
else if(value1 < value2)
    result = 'smaller than';
else if(value1 > value2)
    result = 'greater than';
else result = 'not the same type as';

console.log(`Value 1 is ${result} Value 2`);
```

Exercise 3-1: Play around with if…else if…else statements and try comparing values of the same data type and then with different data types. Try both types of equality comparers (== and ===, != and !===) as well as greater than and smaller than. Continue when you feel that you have a good grasp of how f…else if…else statements work.

Switch and case

Another way to implement conditional logic is to use switch/case statements. A switch takes one value to evaluate, not an expression, and compares that value to the constants in the case-blocks. To stop the execution from falling through to another case-block, you add the **break** command.

You can also add a **default**-block in the switch, which handles all values that don't have a case-block. It's comparable to how the else-block executes when no expression evaluates to **true** for if or else if. Note that the **default**-block doesn't have to include a **break** command since it is the last case in the switch.

Example 3-5: In this example, the displayed text would be: *The book has a 6% sales tax*.

```
var product = 'book';
var salesTax;

switch(product)
{
    case 'spoon':
        salesTax = 0.25;
        break;
    case 'book':
        salesTax = 0.06;
        break;
    default:
        salesTax = 0.25;
}

console.log(`The ${product} has a ${salesTax*100}% sales tax`);
```

Exercise 3-2: Create a switch that compares vehicles and assigns the top speed and brand to two variables called **topSpeed** and **brand**. Use an if statement to determine what is output to the console. If the **topSpeed** variable is unassigned, then print out: *No vehicle was selected*, otherwise print out the brand and top speed like this: *Koenigsegg Agera RS has a top speed of 403 km/h.*

For

As a developer, you use loops a lot to iterate over arrays, let's look at the for-loop.

Example 3-6: for-loop

```
for(variable; expression; increment variable)
{
    // Code to execute until the expression is false
}

for(let i = 0; i < 5; i++)
{
    console.log(i)
}
```

The for-loop has three parts:

- A variable that is created and assigned a start value before the loop starts.
- An expression that determines when the loop ends.
- And variable increments/decrements.

The variable value increments until the expression returns **false**. In the example above, the values 0, 1, 2, 3, and 4 are displayed in the console because the expression states that it should iterate for as long as **i** is less than five.

You can also use the values and length of an array when creating a for-loop. Let's say that you have an array of playing cards that you want to iterate over and display in the console. You could then instantiate a variable named **i** in the loop and assign it an initial value of 0 (the first index of the array) and then check that the loop iterates the for the length of the array minus one. To display the cards, you use the array variable's square brackets with the iterator variable **i** inside.

Example 3-7: Iterating over an array.

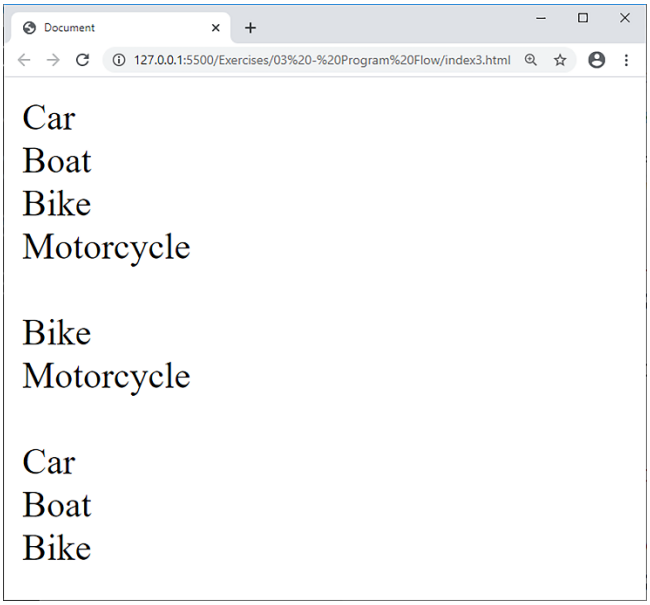
```
let cards = ['Ace of spades', 'King of spades', 'Queen of spades'];

for(let i = 0; i < cards.length; i++)
{
    console.log(cards[i]);
}
```

```
Ace of spades                    index.js:8
King of spades                   index.js:8
Queen of spades                  index.js:8
```

Exercise 3-3: Create an array called *vehicles* and add four vehicles: Car, Boat, Bike and Motorcycle.

1. Create a loop that displays all values in the **vehicles** array.
2. Create a loop that skips the first two elements in the array and displays the rest in the console.
3. Create a loop that displays all except the last element of the array in the console.
4. Repeat steps 1-3 and display the content on the web page. Use the **+=** operator to concatenate the output string; you can store it in a variable named **output** or concatenate directly to the **innerHTML** property of the **exercise-area** .

```
Loop 1                           index.js:6
Car                              index.js:9
Boat                             index.js:9
Bike                             index.js:9
Motorcycle                       index.js:9
                                 index.js:12
Loop 2
Bike                             index.js:15
Motorcycle                       index.js:15
                                 index.js:18
Loop 3
Car                              index.js:21
Boat                             index.js:21
Bike                             index.js:21
```



While

The while-loop is another popular way to iterate. The loop continues for as long as the expression is **true**; this means that you could create an infinite loop that freezes the browser if you aren't careful. Always remember to terminate the loop at some point; it could be when a counter variable reaches a specific value, or when meeting another Boolean expression.

Example 3-8: while-loop

```
let counter = 0;
while(expression)
{
    // Code to execute
    counter++;
}
```

Let's use the card array from the for-loop section and iterate over the cards win a while-loop and display them in the console.

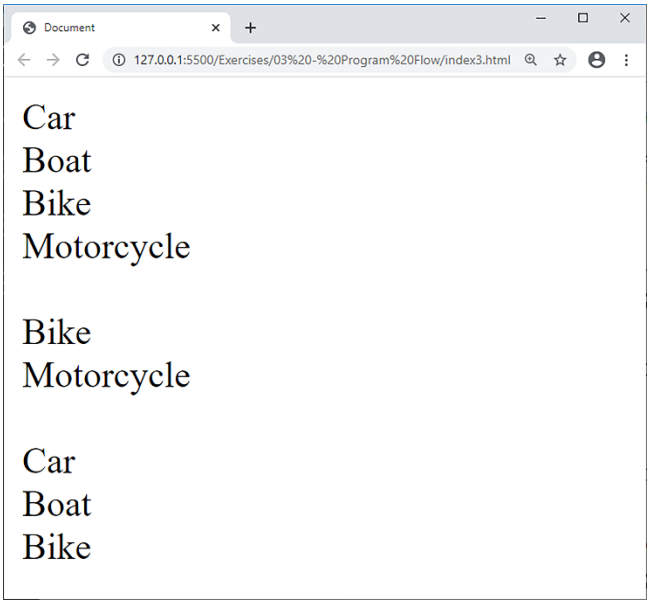
```
let cards = ['Ace of spades', 'King of spades', 'Queen of spades'];

let counter = 0;
while(counter < cards.length)
{
    console.log(cards[counter]);
    counter++;
}
```

Exercise 3-4: Create an array named **vehicles** and add four vehicles: *Car*, *Boat*, *Bike* and *Motorcycle*.

1. Create a while-loop that displays all values in the *vehicles* array.

2. Create a while-loop that skips the first two elements in the array and displays the rest in the console.
3. Create a while-loop that displays all except the last element of the array in the console.
4. Repeat steps 1-3 and display the content on the web page. Use the **+=** operator to concatenate the output string; you can store it in a variable named **output** or concatenate directly to the **innerHTML** property of the **exercise-area** .

```
Loop 1                                  index.js:6
Car                                     index.js:9
Boat                                    index.js:9
Bike                                    index.js:9
Motorcycle                              index.js:9

                                        index.js:12
Loop 2
Bike                                    index.js:15
Motorcycle                              index.js:15

                                        index.js:18
Loop 3
Car                                     index.js:21
Boat                                    index.js:21
Bike                                    index.js:21
```

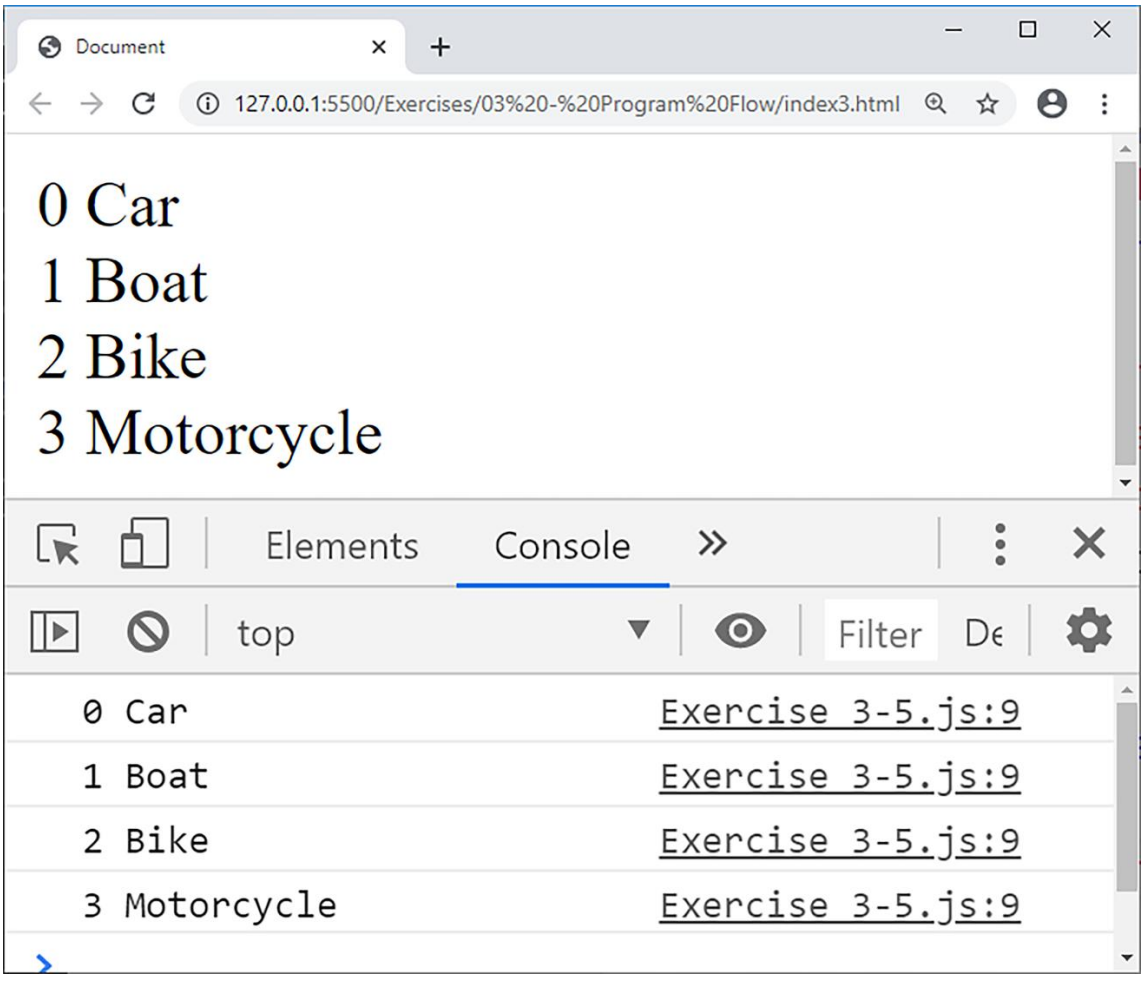


Array.forEach

The **forEach** function is available directly on array variables and iterates over the entire array. It requires a function to execute for each element in the array; The function takes two parameters, one for the current value and one for the current index and makes them available in the function block.

```
Array.forEach(function(value, index){
    // Execute code for every value in the array.
    // You can use the value parameter to get the
    // current value and the index to get the current index.
});

let cards = ['Ace of spades', 'King of spades', 'Queen of spades'];

cards.forEach(function(value, index){
    console.log(`${index} ${value}`);
});
```

Exercise 3-5: Create an array called *vehicles* and add few vehicles to it, then use the **forEach** function to iterate over the array and display all the values in the console and on the web page.



04—Functions

A function is a block of code that you can execute as many times as needed.

The Traditional Way

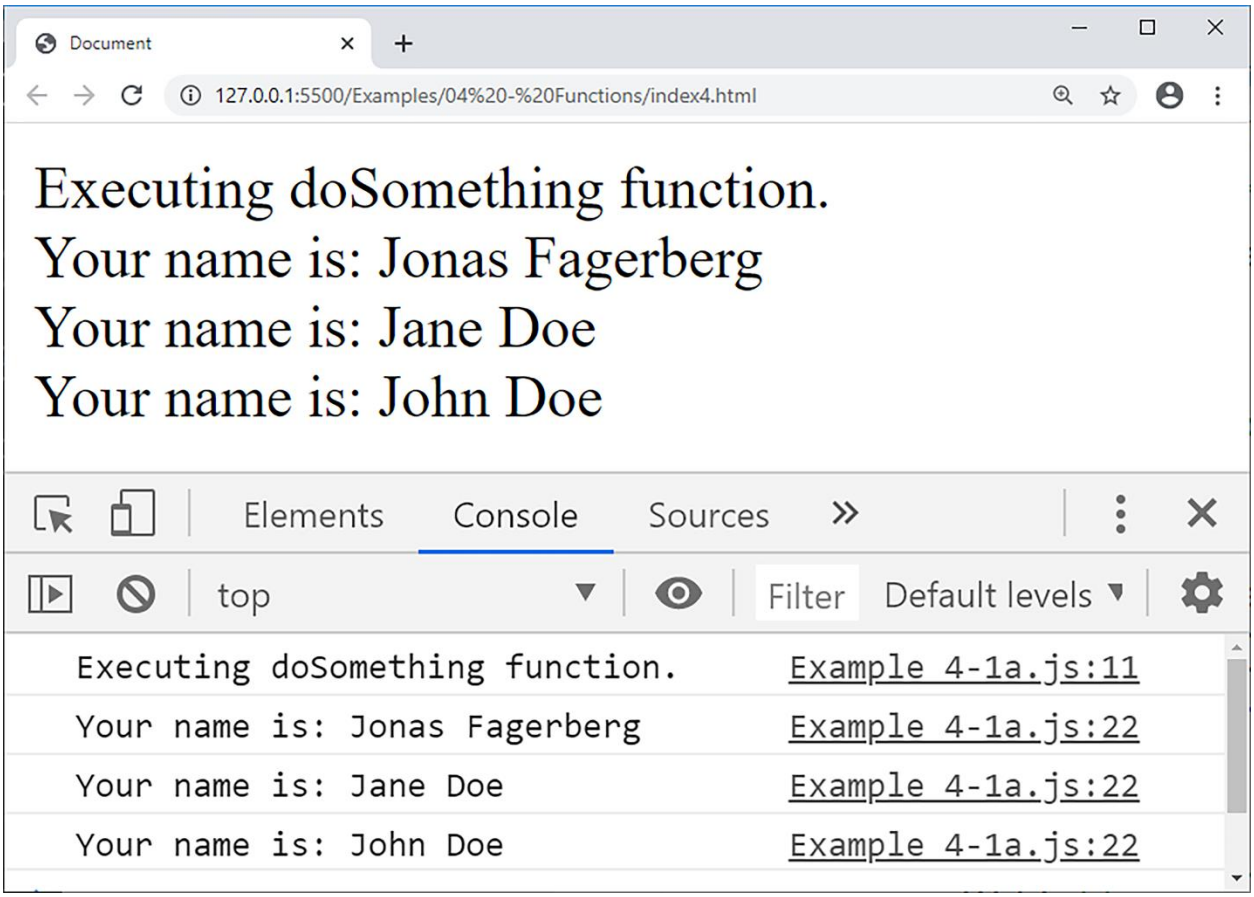
The traditional way of creating a function is to use the function keyword followed by parenthesis (which could contain parameters as a comma-separated list) and then opening and closing curly-braces to specify the beginning and end of the code you want to execute when calling the function; the section inside the curly-braces—the function *body*—and the **function** keyword, the parenthesis, and parameters known as the *function definition*.

You call the function by specifying the function name followed by parenthesis and end with a semicolon. If the function takes parameters, you specify them within the parenthesis as a comma-separated list.

Keep in mind that it is essential to pass in the right number of parameters in the correct order. A common bug is to omit values or pass them into the function in the wrong order.

Example 4-1a: Creating and calling a traditional function

```
const div = document.getElementById("example-area");
div.innerHTML = "";

// The doSomething function
function doSomething()
{
    const output = "Executing doSomething function.";
    console.log(output);
    div.innerHTML = `${output}</br>`;
}

doSomething(); //Calling the doSomething function

// The displayName function
function displayName(fName, lName)
{
    const output = `Your name is: ${fName} ${lName}`;
    div.innerHTML += `${output}</br>`;
    console.log(output);
}

displayName('Jonas', 'Fagerberg'); //Calling the function
displayName('Jane', 'Doe'); //Calling the function again
displayName('John', 'Doe'); //Calling the function again
```



The Modern Way (Fat Arrow)

A more modern and concise approach to creating *arrow* functions (sometimes referred to as *fat arrow* functions) is to use the *arrow* (**=>**) syntax, which usually needs fewer code lines and makes the code more readable. It might take a while to understand how the *arrow* functions are declared, but once you start using them, you don't want to go back to the traditional way.

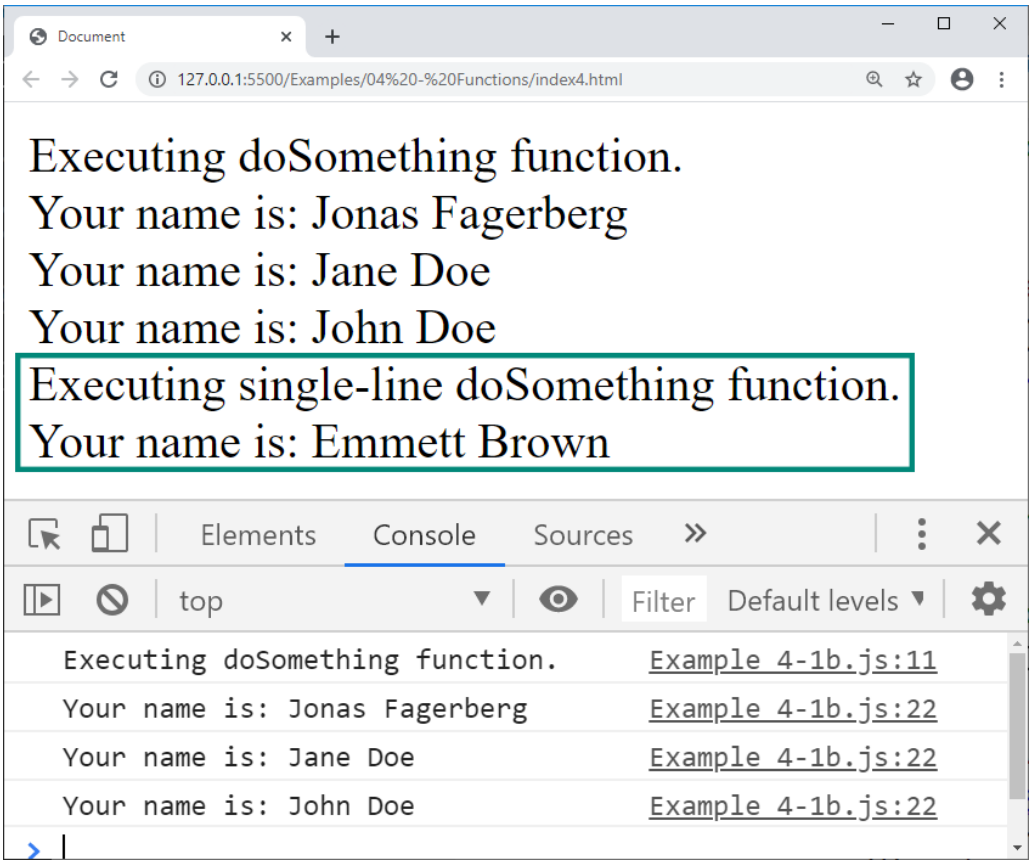
An arrow function can be condensed to one line of code if the function executes only one line of code.

Let's implement the previous example using *arrow* functions with and without function bodies.

Example 4-1b: Creating and calling an arrow function

```
const div = document.getElementById("example-area");
div.innerHTML = "";

// The doSomething function
doSomething = () =>
{
    const output = "Executing doSomething function.";
    console.log(output);
    div.innerHTML = `${output}</br>`;
}
```

```
doSomething(); //Calling the doSomething function

// The displayName function
displayName = (fName, lName) =>
{
    const output = `Your name is: ${fName} ${lName}`;
    div.innerHTML += `${output}</br>`;
    console.log(output);
}

displayName('Jonas', 'Fagerberg'); //Calling the function
displayName('Jane', 'Doe'); //Calling the function again
displayName('John', 'Doe'); //Calling the function again

doSomething2 = () => div.innerHTML +=
    `Executing single-line doSomething function.</br>`;

doSomething2(); //Calling the single line doSomething2 function

displayName2 = (fName, lName) => div.innerHTML += `Your name is: ${fName} ${lName
}</br>`;

displayName2('Emmett', 'Brown'); //Calling the single line displayName2 function
```



Function return values

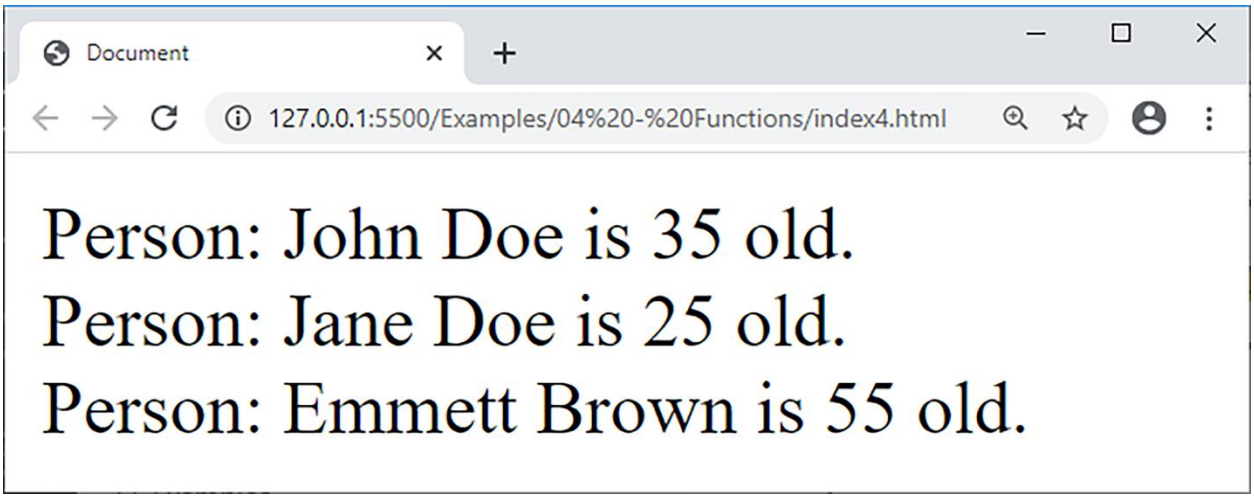
To pass out values from a traditional function, you use the **return** keyword followed by the value to return. You can manipulate values that you send into the function before returning the result.

To return a value from an *arrow* function with a function body, you use the **return** keyword the same as with traditional function. If you create a single line *arrow* function, however, you omit the return keyword.

```
personalInfo = (fName, lName, age) => `Person: ${fName} ${lName} is ${age} old.`;
```

Example 4-2: Returning personal data from a function.

```
const div = document.getElementById("example-area");

function person(fName, lName, age)
{
    let personalData = `Person: ${fName} ${lName} is ${age} old.`;
    return personalData;
}

div.innerHTML = person('John', 'Doe', 35);

let pers = person('Jane', 'Doe', 25);
div.innerHTML += `</br>${pers}`;

// Arrow function that returns a value
personalInfo = (fName, lName, age) => `Person: ${fName} ${lName} is ${age} old.`;

// Calling the arrow function
div.innerHTML += `</br>${personalInfo('Emmett', 'Brown', 55)}`;
```



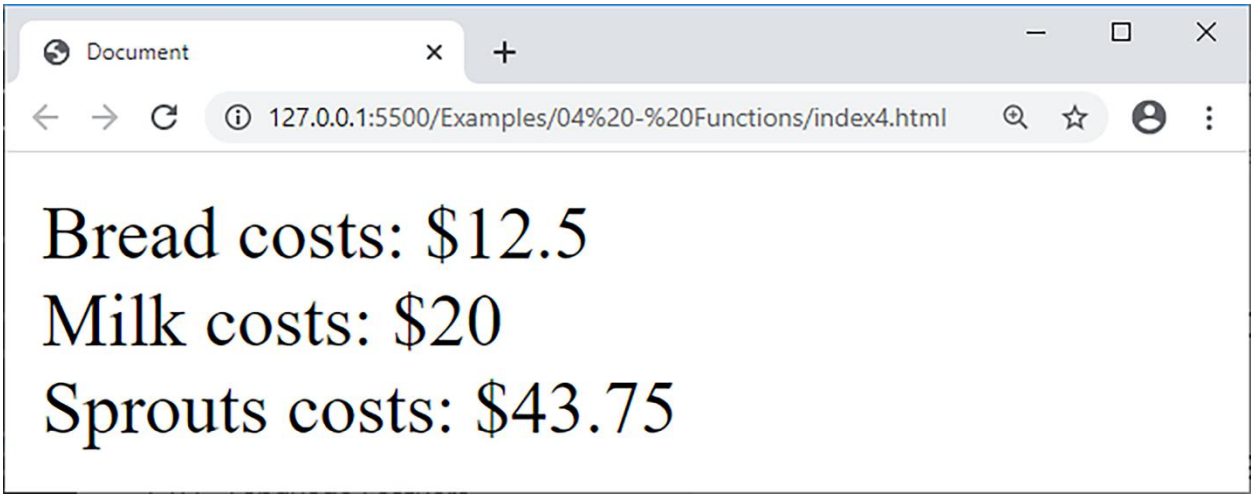
Calling a function within a function

It is quite common to call functions within other functions to keep the code more readable and to have functions with a single purpose; this is best practice. You should avoid massive functions with endless lines of code that do a myriad of things; instead, break out each functionality to a separate function that has one specific and targeted purpose.

Example 4-3: Calling a function within a function.

Here you have two user-defined functions, where the **product** function is calling the **calculatePrice** function to get the total cost of the product; it then uses that information to assemble a string that it returns to the calling function, which in this case is the **log** function that displays the information.

```
const div = document.getElementById("example-area");
div.innerHTML = "";

function calculatePrice(amount, vat)
{
    let total = amount + amount * vat;
    return total;
}

function product(product, amount, vat)
{
    let total = calculatePrice(amount, vat); // Call the other function
    let message = `${product} costs: $${total}`;
    return message;
}

// Calling the product function
div.innerHTML = `${product('Bread', 10, 0.25)}</br>`;

// Arrow function
calculatePriceSingleLine = (amount, vat) => amount + amount * vat;

// Calling the calculatePriceSingleLine single line arrow function
// inside the productCallingSingleLine function
productCallingSingleLine = (product, amount, vat) =>
{
    let total = calculatePriceSingleLine(amount, vat); // Call the other function
    let message = `${product} costs: $${total}`;
    return message;
}

// Calling the productCallingSingleLine arrow function
div.innerHTML += `${productCallingSingleLine('Milk', 16, 0.25)}</br>`;

// Calling a single line arrow function from another single line arrow function
productSingleLine = (product, amount, vat) =>
    `${product} costs: $${ calculatePriceSingleLine (amount, vat)}`;

// Calling the single line productSingleLine arrow function
div.innerHTML += `${productSingleLine('Sprouts', 35, 0.25)}`;
```

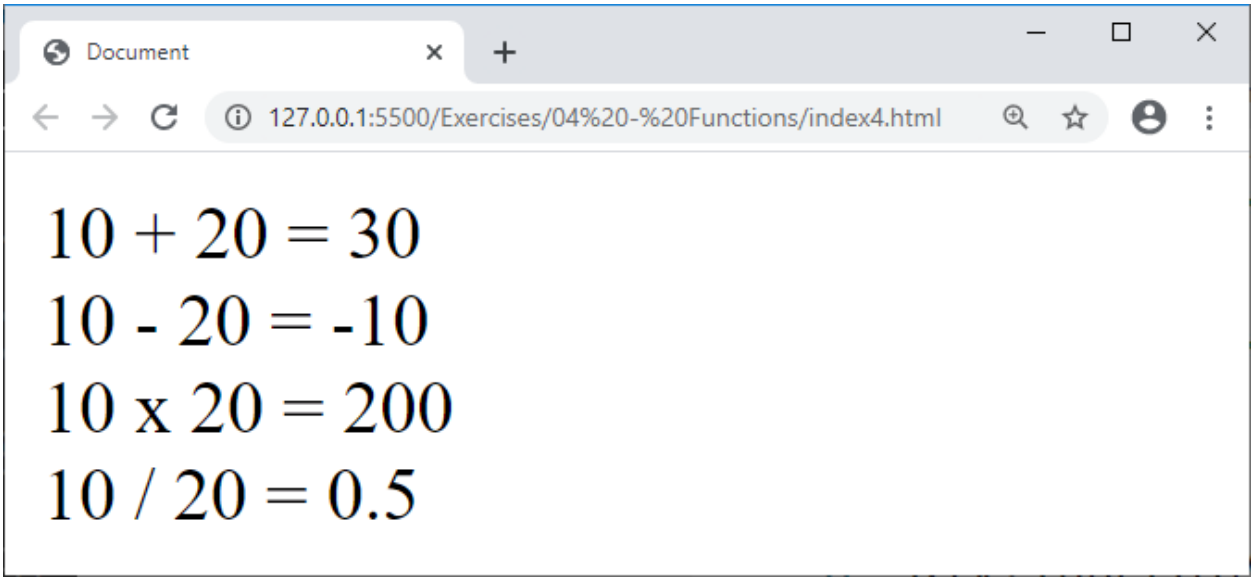


Exercise 4.1 Calling functions

Create four functions named **add**, **subtract**, **multiply** and **divide** that that takes two parameters, each named **a** and **b**. Add the values in the **add** function and return the result, subtract the values in the **subtract** function and return the result, multiply the values in the **multiply** function and return the result, divide the values in the **divide** function and return the result.

Create a new function named **Calculate** that takes two parameters named **a** and **b** and calls the previous functions with the parameter values. Display the result from the four function calls on the web page.

Call the **Calculate** function with the values *10* and *20*.



Scope

Variables created inside a function block is only available within that block. That means that if you try to use that variable outside the function, you get the error: *variable is undefined*.

Example 4-4: Scope

If you look at the code below, you can see that the function block contains a variable named message. If you try to display that variable's value in the console or use it in any way outside the function block where it is declared, you get an *undefined* error because the variable doesn't exist outside the **myMessage** function.

```
function myMessage()
{
    let message = "Message from function";
    return message;
}

myMessage();

console.log(message);
```

```
▶Uncaught ReferenceError: message is not defined  index.js:12
    at index.js:12
```

Now let's add a variable with the same name outside the method without removing the one in the function; this creates a new variable with a scope that is reachable outside the function, and you won't get the error anymore. Instead, the Console displays the value of the new variable.

```
message = "Message outside the function";

function myMessage()
{
    let message = "Message from function";
    return message;
}

myMessage();

console.log(message);
```

```
Message outside the function                    index.js:15
```

05—Objects

An object is a way to group information that belongs together. Let's say that you want to store a deck of cards. You could create two arrays, one for the value of the card, and one for the suit. A better solution is to use objects, where each object in the array contains all the data for a single card.

Let's look at how to create an object. You declare a variable with the **let** keyword and a name, like before, but you assign it curly-braces (**{}**) to instantiate it as an empty object.

```
let person = {}; // Empty object
```

If you want to create an object and assign values in the declaration, you add name-value pairs and separate the key and value with a colon, and each property with a comma.

To read values from an object, you use the variable name followed by a dot and the name of the property name.

```
let person = {
    firstName: "John",
    lastName : "Doe",
    age : 45,
    isAwake: true
};

console.log(person);
console.log(person.firstName);
console.log(person.lastName);
console.log(person.age);
console.log(person.isAwake)
```

```
                                                    index.js:14
▶ {firstName: "John", lastName: "Doe", age: 45, isAwake: true}
John                                                index.js:15
Doe                                                 index.js:16
45                                                  index.js:17
true                                                index.js:18
```

Exercise 5-1: Creating an object

1. Create an object named **card** that has three properties: suit, valueText and value. valueText should hold the card's value in text (Queen, Jack, …) and value should hold the numerical value for the card (12, 11, …).
2. Choose a card and use its values to assign to the properties of the **card** object.
3. Display the card information in the console.

```
Clubs                                               index.js:10
Queen                                               index.js:11
12                                                  index.js:12
```

Passing an Object to a Function

You can pass objects to a function the same way you pass any parameters. You can then change the object's values inside the function.

Example 5-2: Passing an object to a function

Let's add a function that changes the name of the person from our previous example. As you can see, the **person** object you pass into the function along with the first name and last name is assigned those parameter values. Then the **person** object's properties are accessed and changed. When calling the function farther down the code, send in the **person** object as the first parameter followed by the first- and last name.

```
function changeName(person, fName, lName)
{
    person.firstName = fName;
    person.lastName = lName;
}

let person = {
    firstName: "John",
    lastName : "Doe",
    age : 45,
    isAwake: true
};

console.log('Before change:', person.firstName, person.lastName, person.age,
person.isAwake);

changeName(person, "Bart", "Simpson"); // Calling the function with the object


console.log('After change:', person.firstName, person.lastName, person.age,
person.isAwake);
```

```
Before change: John Doe 45 true                    index.js:18
After change: Bart Simpson 45 true                 index.js:21
```

Exercise 5-2: Changing the card's suit by calling a function

1. Use the **card** object from the previous exercise and add a function named **changeSuit** that takes a **card** object and a new suit for the card as parameters.
2. Assign the new suit from the parameter to the passed in **card** object's **suit** property.
3. Display the card's information in the console before and after calling the **changeSuit** function you just created.

```
Before change                    index.js:15
Clubs Queen 12                   index.js:16
After change                     index.js:20
Hearts Queen 12                  index.js:21
```

Object Arrays

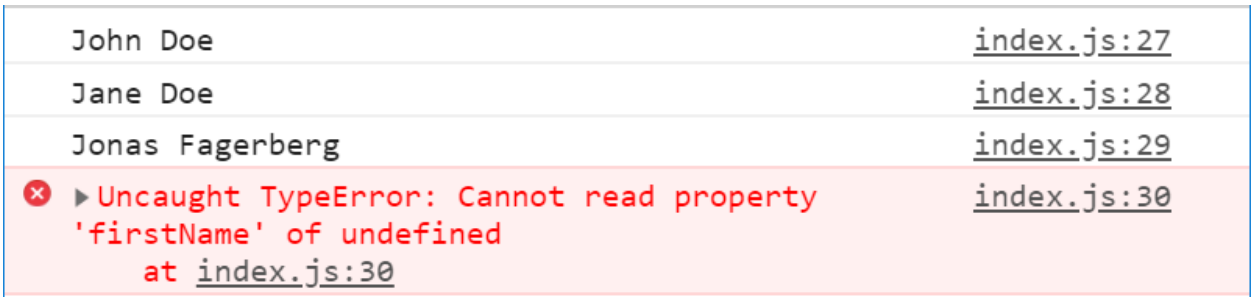
Storing objects in separate variables can be useful, but storing them in arrays can make it easier to work with the data. Let's say we want to store multiple persons. We could create one object variable for each person, but that would get out of hand very quickly and is generally not something you want to do. Instead, you want to create an array that holds the object; this makes it easy to work with using loops.

You add objects to an array the same way you do with regular values, such as strings and numbers.

Example 5-3: An object array

You can add objects to the array when it is declared and also by using the **push()** method. Note that you get an error if you try to access a value from a non-existing object in the array, the value isn't **undefined**.

```
let persons = [
    {
        firstName: "John",
        lastName : "Doe",
        age : 45,
        isAwake: true
    },
    {
        firstName: "Jane",
        lastName : "Doe",
        age : 35,
        isAwake: false
    }
];

persons.push({
    firstName: "Jonas",
    lastName : "Fagerberg",
    age : 45,
    isAwake: true
});

console.log(persons[0].firstName, persons[0].lastName);
console.log(persons[1].firstName, persons[1].lastName);
console.log(persons[2].firstName, persons[2].lastName);
console.log(persons[3].firstName, persons[3].lastName); // Generates an error
```



Exercise 5-3: Object arrays

1. Create an empty array named cards.
2. Add a function named **AddCard** that takes the suit and value as parameters.
3. Create a new object in the function and assign the passed-in parameters to the properties of the object. **Tip:** If the properties and parameters have the same values, you can skip the property name in the object creation.
4. Call the **AddCard** function to add a few cards.
5. Add another function called **DisplayCards** that displays all the cards stored in the array in the console.
6. Call the **DisplayCards** function to display all the cards in the console.

Built-in Objects

There are many built-in objects in JavaScript. Here, we only cover a few of them: Math, Date, String, and Number. You can find more built-in objects by visiting developer.mozilla.org and search for *Standard Built-in Objects*.

Math

The **Math** object has functions that can truncate numerical values, generate random numbers, and many more functions that you can read about on Mozilla.

To generate a random number between zero and one, you call the **random()** function in the **Math** class. In an upcoming BlackJack application exercise you need to shuffle a deck of cards, to achieve this you could multiply the result from the **random()** function with 52, which returns a decimal number. To convert the result to an integer, you pass it into the **truncate()** method on the **Math** object.

Example 5-4a: The **Math** object

```
let result = Math.random() * 52;
let truncated = Math.trunc(result);
console.log(result, truncated);

// Or you can do it on one line
```

```
let truncatedResult = Math.trunc(Math.random() * 52);
console.log(truncatedResult);
```

```
9.664486426578954 9                                    index.js:7
24                                                     index.js:11
```

Date

The **Date** class can be useful when working with dates in different formats. You can use an object of the **Date** class directly to get a long-format date string, or you can use one of its functions, such as **toDateString**.

Example 5-4b: The **Date** class

```
let date = new Date();

console.log(date);
console.log(date.toDateString());
```

```
Tue Aug 14 2018 15:22:18 GMT+0200 (Central European Summer Time)  index.js:7
Tue Aug 14 2018                                                   index.js:8
```

String Manipulation

You can use string manipulation when comparing two strings to make sure that the text case isn't an issue. If you, for instance, compare "Hello" with "hello" they don't evaluate as the same string when compared, but if you use either of the **toLowerCase()** or **toUpperCase()** functions, they match.

Example 5-4c: String Manipulation

```
let string1 = "This is a STRING",
    string2 = "This is a string";
let lower = string1.toLowerCase();
let upper = string1.toUpperCase();

console.log(string1);
console.log(lower);
console.log(upper);
console.log(string1 === string2);
console.log(string1.toLowerCase() === string2.toLowerCase());
```

```
This is a STRING                    index.js:11
this is a string                    index.js:12
THIS IS A STRING                    index.js:13
false                               index.js:14
true                                index.js:15
```

Number

The **Number** class has many functions, but here we will focus on the **isNaN()** function that can help determine if a value is a number or not. If you divide zero by zero, we have said that it will not generate an error, but, instead return the value **NaN** (Not a Number). To check if a value is a number you can simply send in the value to the **isNaN** function to find out.

Example 5-4d: Not a number

Dividing zero by zero

```
let result = 0/0;

console.log('Is NaN: ' + Number.isNaN(result));

if(!Number.isNaN(result))
    console.log(result);
else
    console.log('Error: Not a number');
```

```
NaN: true                           index.js:8
Error: Not a number                 index.js:13
```

06—JavaScript Selectors (Working with web pages)

Now, the fun begins. In this chapter, you create a simple web page and interact with elements on the page through what's called the DOM (Document Object Model). The DOM is an object representation of the web page's content, such as HTML5 tags and metadata about the page.

As a JavaScript developer, you will most likely hear other developers talk about the DOM when talking about a web page because as JS developers, we are more interested in the page in object form. You can use the DOM to interact with, remove elements from and add elements to the page.

In the browser's F12 Developer Tools, you can inspect the DOM by opening the Inspector tab. When you hover over the elements in the DOM tree, they get highlighted in the browser. You can also do it the other way around by selecting an element in the browser to see the corresponding element in the DOM tree. To do this, you click the inspect button to the far left in the F12 Developer Tools window; it has the hover-description: *Pick an element from the page* or *Select an element on the page to inspect it*.

To write HTML5, you must first add a *.html* page to the project, if there isn't already one that you can use. To add a new HTML document, click on the **New File** button in the desired section of the **Explorer** window, which you can open with the **Explorer** button to the left of the code editor. The most commonly used name for the main page of a site is *index.html*.

In Visual Studio Code, if there is a code snippet that you can use to add the barebones of a page to an HTML file. Write an exclamation mark (**!**) and press the **Tab** key once on the keyboard.

If you are used to JQuery, here's a site that helps you convert JQuery to vanilla JavaScript with examples.

https://tobiasahlin.com/blog/move-from-jquery-to-vanilla-javascript/

Button Events

Consider the following HTML which has a paragraph <p> element and a <button> element.

```
<body>
    <p id="example-area"></p>
    <button id="ok-button">OK</button>

    <script src="index.js"></script>
</body>
```

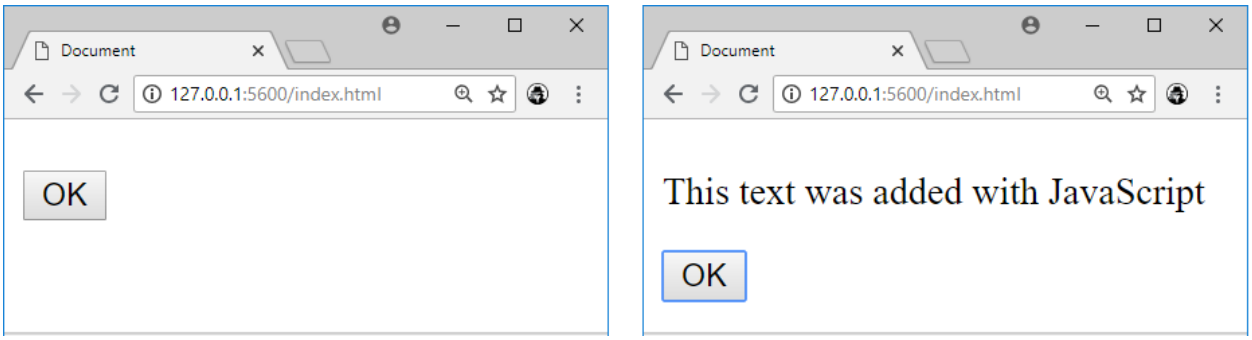
Example 6-1: Hook up click event with JavaScript

A button event is triggered when a user clicks the button. To add an event listener (function), you use the **getElementById** function as you did with the paragraph and pass in the **id** of the button, which in this case is **ok-button**.

You then use the element object (stored in the variable where you saved the button element) to create the event with the **addEventListener** function. That function takes two parameters: the first is a string specifying the type of event you want to listen to, in this case, *click*. The second is a callback function that executes when the user clicks the button; write the code you want to execute in the function block.

Place the code where the paragraph text is updated inside the function block. Save the page and click the button to add the text to the paragraph.

```
let paragraph = document.getElementById("example-area");
let button = document.getElementById('ok-button');

button.addEventListener('click', function(){
    paragraph.innerText = `This text was added with JavaScript`;
});
```



Example 6-2: Hook up click event with the **onclick** HTML attribute

In some scenarios, it's only possible to use HTML attributes to call a method when an event is triggered; one such scenario is if you add the HTML dynamically. To call a function from HTML, you use the appropriate event attribute—in this case, **onclick**—and specify the name of the function to call inside the event's quotation marks in the HTML, then you pass in any parameters within the function parenthesis.

We continue where we left off in the previous example by adding another button that calls a function named **displayPerson** when clicked. Note that you can add the parameter values through variables if the HTML is generated dynamically with a backtick string.

New HTML:

```
<button id="change-text-button" onclick="displayPerson('John', 'Doe', 34)">
Change Text</button>
```

New JavaScript:

```
let paragraph = document.getElementById("example-area");

displayPerson = (firstName, lastName, age) => paragraph.innerText =
    `${firstName} ${lastName} ${age}`;
```

Example 6-3: Adding HTML dynamically with JavaScript

When creating a dynamic website, you need to add HTML based on user choices or program flow; in these scenarios, you can create the HTML with a backtick string and add it to the desired HTML container, such as a <li> to a <ul> list.

Consider the following HTML. elements—containing a <button> that calls a function displaying the personal information in the <p>—are added dynamically to the <ul> list.

```
<p id="example-area"></p>
<ul id="example-ul">
    <!-- The dynamic HTML goes here -->
</ul>
```

The JavaScript begins by targeting the two containers and storing them in variables.

```
let paragraph = document.getElementById("example-area");
let ul = document.getElementById("example-ul");
```

The **addPerson** function adds the <li> elements to the <ul> element; note that the backtick string contains the HTML for the <li> element and that the **displayPerson** function parameter values are fetched from the **addPerson** function parameters.

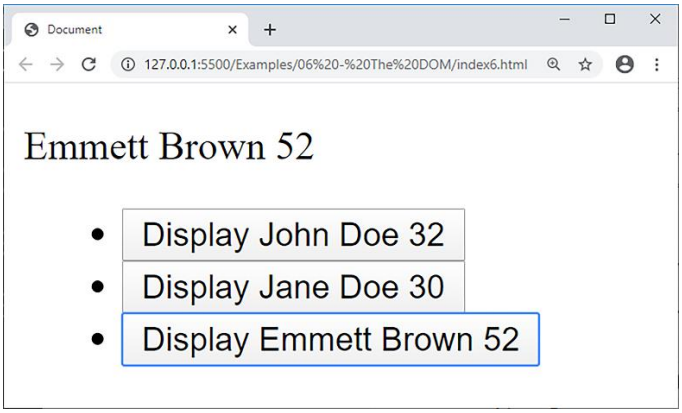
```
addPerson = (firstName, lastName, age) => ul.innerHTML +=
`<li><button id="change-text-button" onclick="displayPerson('${firstName}',
'${lastName}',${age})">Display ${firstName} ${lastName} ${age}</button></li>`;
```

The **displayPerson** function called from the buttons in the <li> element update the HTML of the <p> element.

```
displayPerson = (firstName, lastName, age) => paragraph.innerHTML = `${firstName}
 ${lastName} ${age}`;
```

Then the **AddPerson** function is called to dynamically add <li> elements to the <ul> list.

```
addPerson('John', 'Doe', 32);
addPerson('Jane', 'Doe', 30);
addPerson('Emmett', 'Brown', 52);
```



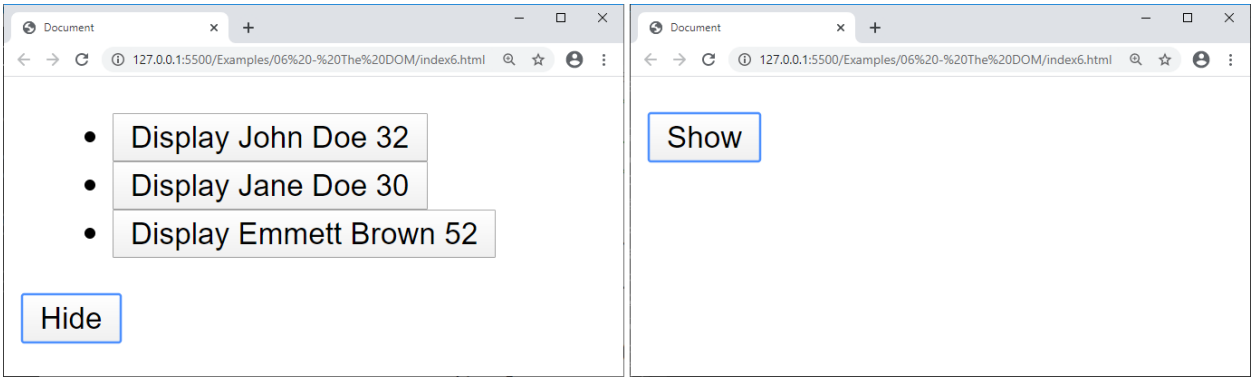
Hiding and Showing Elements to the Page

You can use the **style.display** property of an element to remove or add it to the page; this can be very useful to show only relevant information on the page or to change the page content dynamically when the user interacts with it.

Example 6-4: Hiding/showing elements

Let's add another button with the id **toggle-button** that shows/hides the <ul> from the previous example.

```
<button id="toggle-button">Hide</button>
```



The **style** property of an HTML element makes it possible to apply any CSS styling you want to it.

Target the button and store it in a variable named **toggleButton**, for which you hook up the **click** event. Inside the event's function block, you add an if-statement that checks if the **div.style.display** property is equal to *none*; if it is, then assign the value *block* to the **display** property to show the <ul> element, else hide it by assigning *none* to the **display** property.

Inside the if/else-blocks, toggle the text of the button between *Hide* and *Show* depending on the current state of the <ul> element (if its hidden or visible).

```
let toggleButton = document.getElementById('toggle-button');

toggleButton.addEventListener('click', function(){
    if(ul.style.display === 'none'){
        ul.style.display = 'block';
        toggleButton.innerHTML = "Hide";
    }
    else{
        ul.style.display = 'none';
        toggleButton.innerHTML = "Show";
    }
});
```

The QuerySelector and QuerySelectorAll Functions

In this section, you learn how to target elements on your page, which are nodes in the DOM. Once you have selected one or more elements, you can start to manipulate them with JavaScript. You have already learned how to target with id, and now you'll learn how to target elements using element types, CSS selectors, and attributes.

To select one element, you call the **document.querySelector** function and pass in the selector. To select multiple elements, you call the **document.querySelectorAll** function and pass in all selectors; note that you need to iterate over the returned HTML elements to manipulate them.

Note that selectors are case sensitive.

querySelector: https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector
querySelectorAll: https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll

Select element by tag name

You can select elements in the DOM by their element names. A tag can be a , <p>, , and so on. To select elements this way, you place the element name inside quotation marks.

```
// Select first match
document.querySelector('p')
document.querySelector('div')
document.querySelector('span')
document.querySelector('button')

// Select all matches (requires iteration)
document.querySelectorAll('p')
document.querySelectorAll('div')
document.querySelectorAll('span')
document.querySelectorAll('button')
```

Example 6-5: Selecting multiple elements

You can also select multiple elements at once. The following example would select all paragraphs, divs, and spans on the page (in the DOM). Note that you must loop through the selected elements to manipulate them. Use the **querySelectorAll** function to select multiple elements that you can iterate over.

```
myFunction = () => {
    const elements = document.querySelectorAll("p, button");
    elements.forEach(e => e.style.backgroundColor = "red");
}
```

Selecting Descendants

You can easily target the descendants of an element by separating the parent from the child with a space. Let's say you want to select all the list items (<li>) in an unordered list (<ul>).

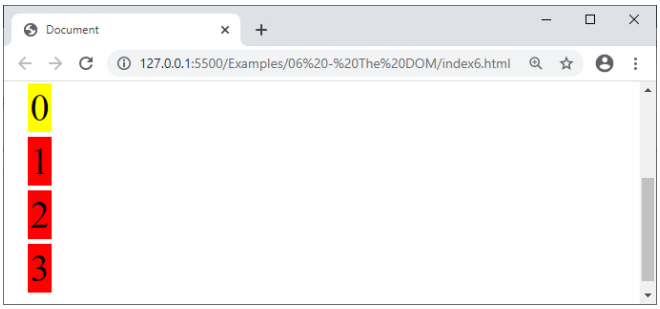
```
document.querySelector('ul li') // Select first match
document.querySelectorAll('ul li') // Select all matches
```

Example 6-6: Selecting descendants

In this example, the background color is changed to red for all <tr> elements that are descendants to the table with the id **example-table**, and the background color is changed to yellow for the first <tr> in the same table.

```
<table id="example-table">
    <tbody>
        <tr><td>0</td></tr>
        <tr><td>1</td></tr>
        <tr><td>2</td></tr>
        <tr><td>3</td></tr>
    </tbody>
</table>
```

```
[...document.querySelectorAll("#example-table tr")]
    .forEach(e => e.style.backgroundColor = "red"); // Select all matches

document.querySelector('#example-table tr')
    .style.backgroundColor = "yellow"; // Select first match
```



Select element by id

When you select an element by its id, only one element should be selected because the id should always be unique in a page. To specify an id when calling one of the **querySelector** and **querySelectorAll** functions, you prefix it with the hashtag (#). Selecting by id is the most efficient way to select an element in the DOM.

```
<p id="example-area"></p>
```

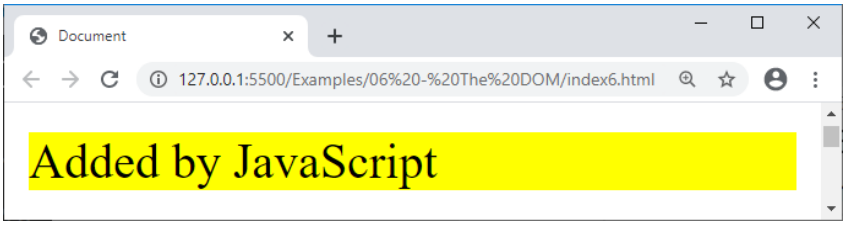
```
document.getElementById('example-area')
document.querySelector("#example-area")
document.querySelectorAll("#example-ul")
```

Example 6-7: Selecting item by id

In this example, the background color is changed to yellow for the <p> elements with the id **example-area**, and its text is changed to *Added by JavaScript*.

```
<p id="example-area"></p>
```

```
document.getElementById('example-area').style.backgroundColor = "yellow";
document.querySelector("#example-area").innerHTML = "Added by JavaScript";
```



Select element by class

When you select elements by CSS selector (class), an array of node objects are returned from the DOM. The class name must be prefixed with a period (.).

You can also select multiple classes in the same selector.

```
document.querySelectorAll('.text1, .text2') // Select all matches
```

You can also combine the class with the element's name to narrow the selection.

If you place the element name before the class name without space in between, the class must be on that element to be selected.

```
document.querySelector('p.text1') // Select first match
document.querySelectorAll('p.text1') // Select all matches
```

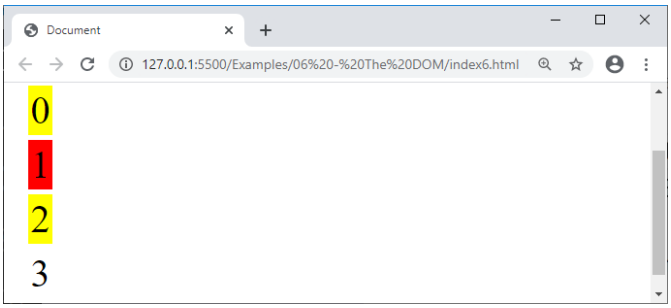
If you specify the element name before the class name with space in between, the class must be on that element or one of its descendants to be selected.

```
document.querySelector('p .text1') // Select first match
document.querySelectorAll('p .text1') // Select all matches
```

Example 6-8: Selecting item by class

In this example, the background color is changed to yellow for the <tr> elements that has the even CSS class. Note that only the first <tr> with the **odd** class is changed because the **querySelector** function is called.

```
<table id="example-table">
    <tbody>
        <tr class="even"><td>0</td></tr>
        <tr class="odd"><td>1</td></tr>
        <tr class="even"><td>2</td></tr>
        <tr class="odd"><td>3</td></tr>
    </tbody>
</table>

const elements = document.querySelectorAll('.even'); // Select all matches
elements.forEach(e => e.style.backgroundColor = "yellow");

document.querySelector('.odd').style.backgroundColor = "red" //Select first match
```



Select element by attribute

When you select elements that have a specific attribute applied or have an attribute with a specific value, the result is an array with the affected elements. You check for an attribute by placing its name inside square brackets after the element name.

When you look at the syntax, you can read the square brackets as *where the **element** has*.

The following code will select all <li> elements with an attribute named *my-element*.

```
<li my-element>Element 1</li>
```

```
document.querySelector('li[my-element]') // Select first match
document.querySelectorAll('li[my-element]') // Select all matches
```

The following code will select all <li> elements with an attribute named *my-element*. That has the value **true**.

```
<li my-element="true">Element 1</li>
```

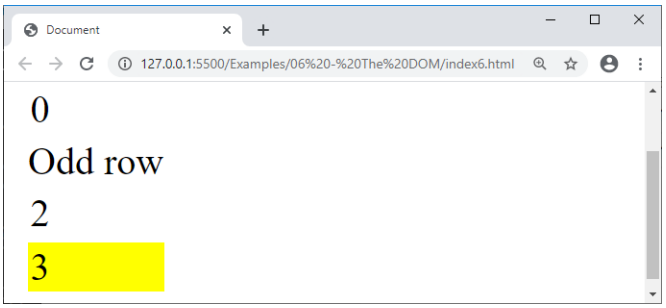
```
document.querySelector('li[my-element=true]') // Select first match
document.querySelectorAll('li[my-element=true]') // Select all matches
```

Example 6-9: Selecting item by attribute

In this example, the text of the <tr> elements with the **is-odd** attribute is changed to *Odd row*, and the background color is changed yellow for the <tr> element with **is-odd="true"**.

```
<table id="example-table">
    <tbody>
        <tr class="even"><td>0</td></tr>
        <tr class="odd" is-odd><td>1</td></tr>
        <tr class="even"><td>2</td></tr>
        <tr class="odd" is-odd="true"><td>3</td></tr>
    </tbody>
</table>
```

```
document.querySelector('tr[is-odd]').innerHTML = "Odd row";
document.querySelector('tr[is-odd="true"]').style.backgroundColor = "yellow";
```



Selecting input elements

When selecting input elements, you use the **input** selector for selecting all input-type elements, which include input, select, textarea, button, image, radio, and more.

```
document.querySelector('input[type="radio"]') // Select first match
document.querySelectorAll('input[type="radio"]') // Select all matches
```

Iterating Through Nodes

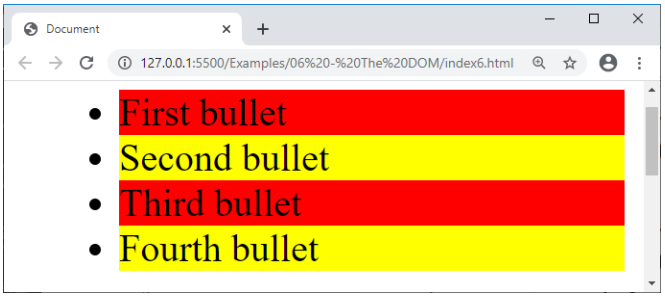
Use the **spread** operator to get to each element in the returned node list before iterating with the **forEach** to manipulate the element.

The index parameter contains the index of the current element in the list of elements selected.

Example 6-10: Iterating over <li> elements

In this example, when the **querySelectorAll** function is called the first time, all <li> elements are colored red. When **querySelectorAll** function is called the second time, even <tr> elements are colored red and odd yellow. You can add them one at a time to see the difference.

```
<ul id="example-ul">
    <li>First bullet</li>
    <li>Second bullet</li>
    <li>Third bullet</li>
    <li>Fourth bullet</li>
</ul>

[...document.querySelectorAll("#example-ul li")]
    .forEach(e => e.style.backgroundColor = "red");

[...document.querySelectorAll("#example-ul li")]
    .forEach((e, index) => (index % 2) == 0
        ? e.style.backgroundColor = "red"
        : e.style.backgroundColor = "yellow");
```



Selecting odd or even rows or elements

To select odd or even rows, you can filter the on the element's index using modulo (%). To select elements with an even index, you filter on *(index % 2) == 0*, for odd indexes you use the negated version of that expression *(index % 2) != 0*. Keep in mind that indexes are zero-based.

The following code selects all even rows in all the tables on the page.

```
<table>
    <tbody>
        <tr><td>0</td></tr>
        <tr><td>1</td></tr>
        <tr><td>2</td></tr>
        <tr><td>3</td></tr>
    </tbody>
</table>
```

Use the **spread** operator to get to each element in the returned node list, the filter function to get elements with an even index, and **forEach** to manipulate the element.

```
[...document.querySelectorAll('tr')]
    .filter((elem, index) => (index % 2) == 0)
    .forEach(e => e.style.backgroundColor = "yellow");
```

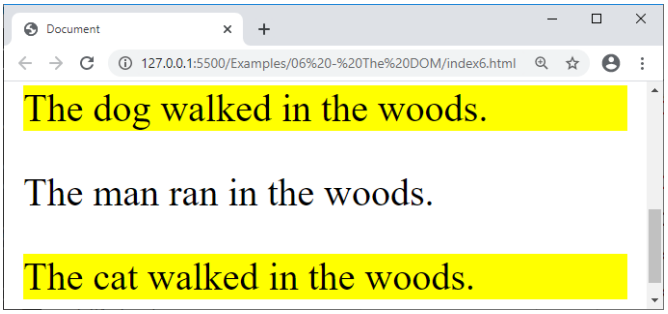
Selecting elements that contain a specific value

You can use the **includes** function to find elements that contain a certain text.

Example 6-11: Selecting elements that contains a phrase

The following code searches for paragraphs that contain the text *walked in*.

```
<p>The dog walked in the woods.</p>
<p>The man ran in the woods.</p>
<p>The cat walked in the woods.</p>

document.querySelectorAll('p').forEach(e => {
    if(e.textContent.includes('walked in'))
        e.style.backgroundColor = "yellow"
});
```



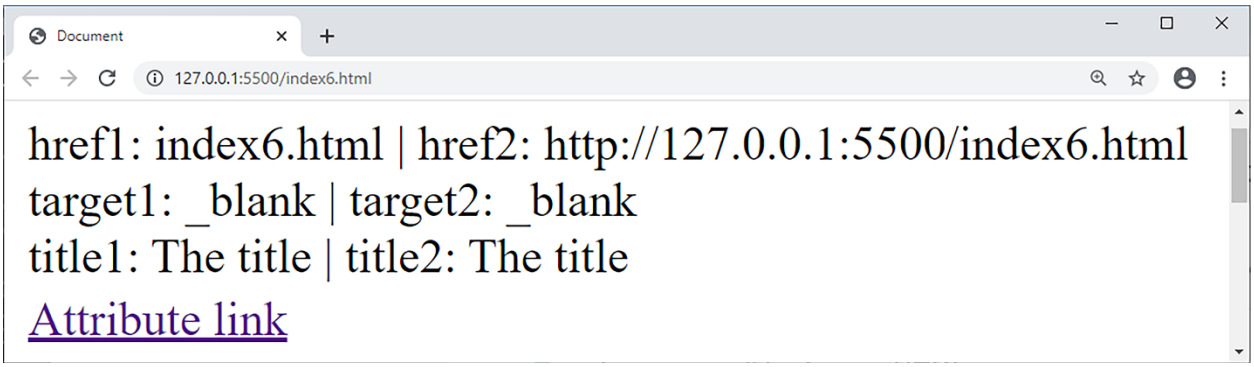
Reading Properties and Attributes

To get to the properties and attributes of a raw DOM object, you can use the dot syntax on the element, for example, **element.*propertyName***, or call the **getAttribute** function with the name of the property or attribute. Note that they might return different values.

Example 6-12: Reading properties and attributes

In this example, three properties and attributes are read from an anchor tag using the **getAttribute** function and the attribute name.

```
<p id="example-area"></p>
<a id="my-anchor" href="index6.html" target="_blank" title="The title">
Attribute link</a>

const href1 = document.getElementById("my-anchor").getAttribute("href");
const href2 = document.getElementById("my-anchor").href;
const target1 = document.getElementById("my-anchor").target;
const target2 = document.getElementById("my-anchor").getAttribute("target");
const title1 = document.getElementById("my-anchor").title;
const title2 = document.getElementById("my-anchor").getAttribute("title");
document.getElementById("example-area").innerHTML =
    `href1: ${href1} | href2: ${href2}</br>
     target1: ${target1} | target2: ${target2}</br>
     title1: ${title1} | title2: ${title2}`;
```



Writing to Properties and Attributes

In the previous section, you learned how to read from properties and attributes, in this section you'll learn how to change property and attribute values. Note that the attribute will be added if missing.

```
document.getElementById("elementId").setAttribute(attribute, value);
```

In a loop, you can use the current element and index.

```
document.querySelectorAll('p').forEach((element, index) => {
    element.title = `The index is: ${index}`;
});
```

```
<p title="The index is: 0"></p>
<p title="The index is: 1"></p>
```

Example 6-13: Changing the **title** property of all <tr> elements

In this example, an attribute named **title** with the text *Index:* followed by its index will be added to all <tr> elements in the table.

```
<table id="example-table">
    <tbody>
        <tr class="even"><td>0</td></tr>
        <tr class="odd" is-odd><td>1</td></tr>
        <tr class="even"><td>2</td></tr>
        <tr class="odd" is-odd="true"><td>3</td></tr>
    </tbody>
</table>

// Adding the attributes
document.querySelectorAll('#example-table tr').forEach((element, index) => {
    element.title = `Index: ${index}`;
});

// Reading the attributes
const paragraph = document.getElementById("example-area");

document.querySelectorAll('#example-table tr').forEach(element => {
    paragraph.innerHTML += `${element.title}</br>`;
});
```

Has an Attribute

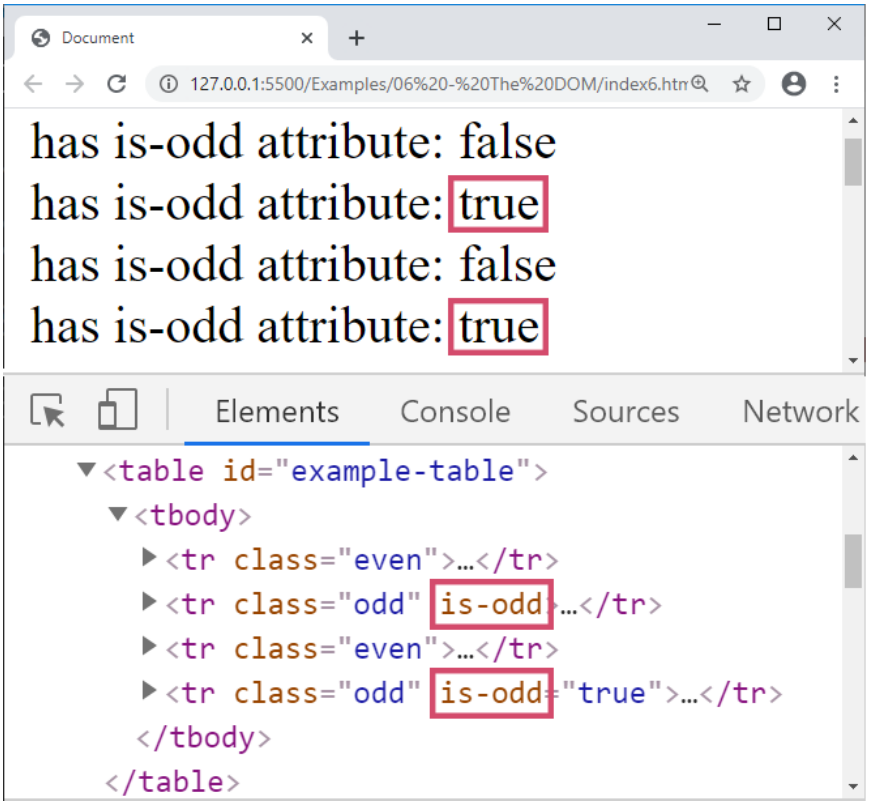
To find out if an element has an attribute, you can call the **hasAttribute** function on the element.

element.hasAttribute(*attributename*)

Example 6-14: Check if elements has an attribute

In this exercise the <tr> elements in the table are checked for the **is-odd** attribute.

```
<table id="example-table">
    <tbody>
        <tr class="even"><td>0</td></tr>
        <tr class="odd" is-odd><td>1</td></tr>
        <tr class="even"><td>2</td></tr>
        <tr class="odd" is-odd="true"><td>3</td></tr>
    </tbody>
</table>

// Find out if elements have an attribute
const paragraph = document.getElementById("example-area");

document.querySelectorAll('#example-table tr').forEach(element => {
    paragraph.innerHTML +=
        `<tr> has is-odd attribute: ${element.hasAttribute("is-odd")}</br>`;
});
```



Adding and Removing Elements

To append an element at the end of another element, you call the **appendChild()** function, and to insert an element into the target element you call the **insertBefore()** function and pass in the new element and the element the new element should be placed before. You can use the **firstChild** property to fetch the first child element, or you can fetch the already existing element as you have learned earlier.

targetNode.appendChild(*nodeToAppend*)

targetNode.insertBefore(*nodeToInsert, nodeToInsertBefore*)

To remove an element, you call the **removeChild()** function with the element you want to remove.

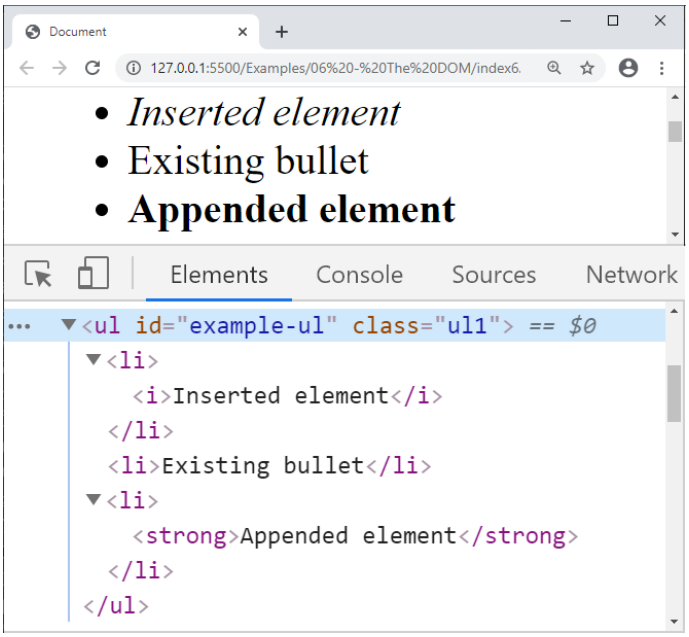
targetNode.removeChild(*nodeToRemove*)

Example 6-15: Append and Insert <li> in

In this example, an <li> element will be appended to the end of the <ul>, and another inserted at the beginning of the <ul>.

```
<ul id="example-ul" class="ul1">
    <li>Existing bullet</li>
</ul>

const ul = document.getElementById('example-ul');

// Append <li> at the end of the <ul>
const li1 = document.createElement("li"); // Create a <li> node
```

```
li1.innerHTML = "<strong>Appended element</strong>"
ul.appendChild(li1); // Append <li> to the <ul>

// Append <li> at the beginning of the <ul>
const li2 = document.createElement("li");
li2.innerHTML = "<i>Inserted element</i>"
const firstLi = ul.firstChild;
ul.insertBefore(li2, firstLi); // Insert <li> to the <ul>
```



Modifying CSS Styles

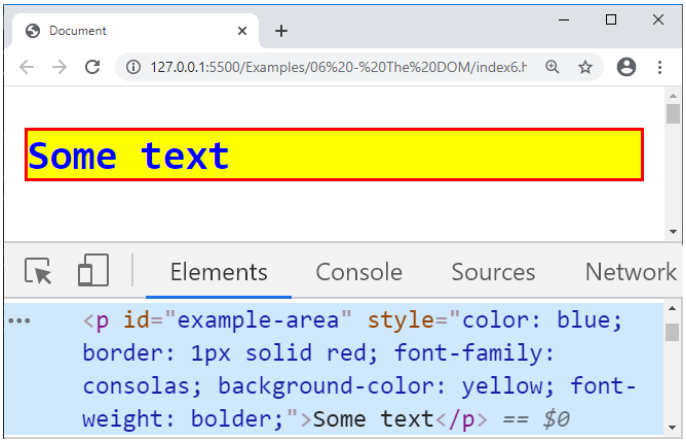
To style elements with CSS, you can either set one specific style with the **style** property or assign multiple styles at the same time, with either the **cssText** property or the **setAttribute** function. Note that the styles are applied to the element itself, making it harder to override them with CSS.

```
element.style.property = value;

element.style.cssText = 'color: red; border: 1px solid black;';

element.setAttribute("style", "color:red; border:1px solid blue;");
```

Example 6-16: Styling With CSS

In this example, the element's HTML and text are styled with the **style** attribute.

```
const p = document.getElementById('example-area');
p.innerHTML = 'Some text';
p.style.cssText = 'color: blue; border:1px solid red; font-family:consolas';
p.style.backgroundColor = 'yellow';
p.style.fontWeight = 'bolder';
```



Modifying Classes

You can style elements with CSS classes or use the classes in application logic. To manipulate the classes on an element, you use the **classList.add()**, **classList.remove()**, **classList.contains()** or **classList.toggle()** functions.

Add class

To add a class, you use the **classList.add()** function and pass in the name of the classes to add as a spread array. If there are already classes on that element, they remain, and the function adds the new class to the element's list of classes.

```
element.classList.add(...['class1', 'class2', 'class3']);
```

Remove Class

To remove an already existing class, you call the **classList.remove()** function and pass in the classes to remove as a spread array.

```
element.classList.remove(...['class1', 'class2', 'class3']);
```

Has Class

To look if an element has a class, you call the **classList.contains** function with the name of the class.

```
element.classList.contains('class1');
```

Toggle Class

To be able to toggle a class on and off (adding and removing) is a handy feature in dynamic front-end applications, you can achieve this by calling the **classList.toggle** function and pass in the class to toggle. You could, of course, use the **classList.add** and **classList.remove** functions, but it would involve more code.

```
element.classList.toggle('class1');
```

Example 6-17: Adding, removing, toggling and checking for Classes

In this example, CSS class selectors are added, removed, and toggled using the previously described functions. The CSS classes are in the *index6.css* file. Adding, removing, and toggling classes are done by clicking the links.

HTML

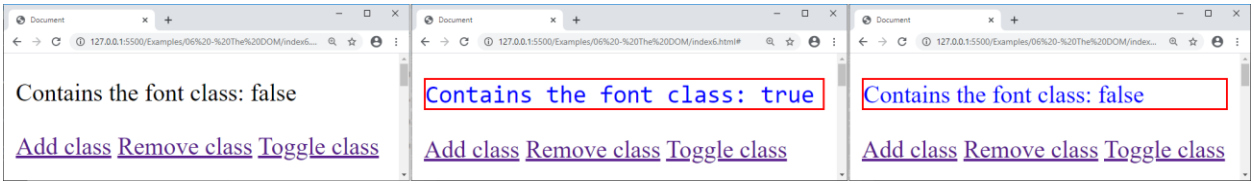
```
<p id="example-area"></p>
<a id="add-classes" href="#">Add class</a>
<a id="remove-class" href="#">Remove class</a>
<a id="toggle-class" href="#">Toggle class</a>
```

CSS in the index6.css file

```
.font{
    font-family:'consolas';
}

.border-warning{
    border:1px solid red;
}

.text-color{
    color: blue;
}
```

JavaScript

```
const p = document.getElementById('example-area');
const addClasses = document.getElementById('add-classes');
const removeClass = document.getElementById('remove-class');
const toggleClass = document.getElementById('toggle-class');

// Add CSS class selectors
p.classList.add(...['border-warning', 'text-color', 'font']);

// Check if a CSS class selector has been applied to an element
hasClass = () => p.innerHTML =
    `Contains the font class: ${p.classList.contains('font')}`;
hasClass();

addClasses.addEventListener('click', event => {
    // Add CSS class selectors
    p.classList.add(...['border-warning', 'text-color', 'font']);
    hasClass();
});

removeClass.addEventListener('click', event => {
    // Remove a CSS class selector
    p.classList.remove(...['font']);
    hasClass();
});

toggleClass.addEventListener('click', event => {
    // Toggle a CSS class selector
    p.classList.toggle('font');
    hasClass();
});
```



Traversing the DOM

Sometimes it's necessary to traverse the DOM tree to reach the next element or to get to the parent or children of an element, to achieve this, you can call the **next()**, **children()** and **parent()** functions.

HTML used in the examples

```
<ul id="first-ul">
    <li>Element A</li>
    <li>Element B
        <ul id="second-ul">
            <li>Element C</li>
            <li>Element D
                <ul id="third-ul">
                    <li>Element E</li>
                    <li id="second-element">Element F</li>
                    <li>Element G</li>
                </ul>
            </li>
        </ul>
    </li>
</ul>
```

The Children Property

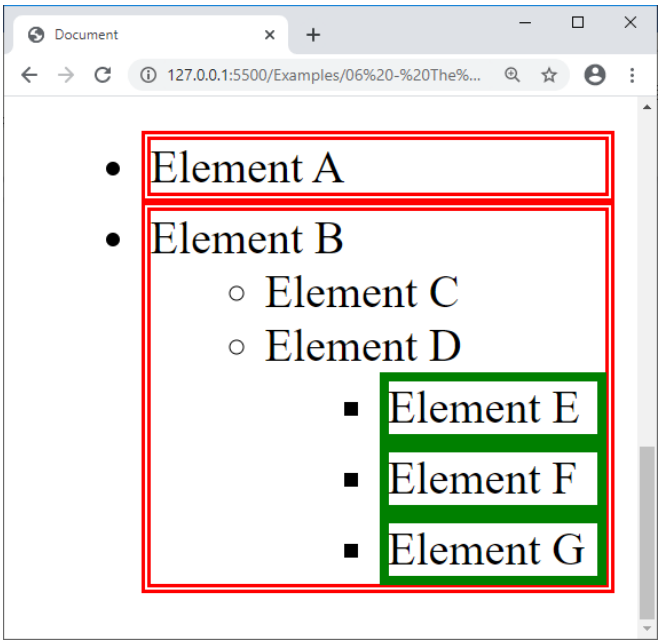
To reach the immediate children (directly inside) of an element, you can call the **children** property. Note that the **spread** operator must be used to get to the array containing the child elements.

element.children

```
const ul = [...document.querySelector("#the-ul").children]
```

Example 6-18: Finding the children of an element

```
// Red border around Element A and B
const fisrtUlChildren = [...document.querySelector("#first-ul").children]

fisrtUlChildren.forEach(element => {
    element.style.border = "3px double red";
});

// Green border around Element E, F and G
const thirdUlChildren = [...document.querySelector("#third-ul").children]

thirdUlChildren.forEach(element => {
    element.style.border = "3px solid green";
});
```



The ParentNode and ParentElement Properties

To reach the parent (immediate ascendant) of an element, you can call the **parentNode** or **parentElement** property.

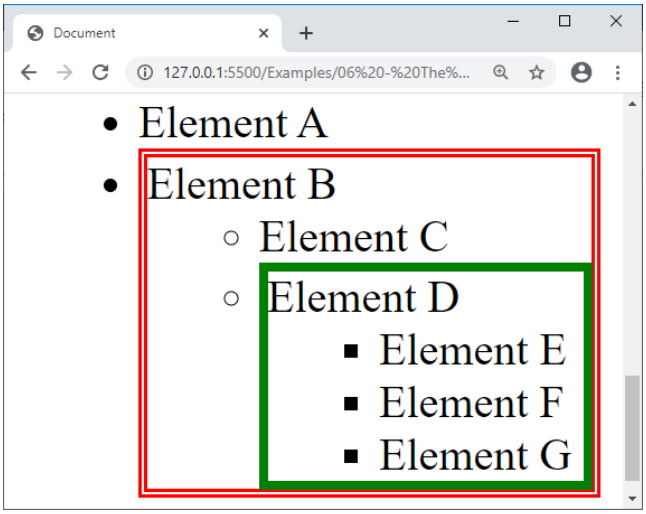
The difference between **parentElement** and **parentNode**, is that **parentElement** returns **null** if the parent node isn't an element node.

```
element.parentNode;
```

```
element.parentElement;
```

Example 6-19: Finding an element's parent

```
// Red border around Element B
const secondUlParent = document.getElementById("second-ul").parentNode;
secondUlParent.style.border = "3px double red";

// Green border around Element D
const thirdUlParent = document.getElementById("third-ul").parentElement;
thirdUlParent.style.border = "3px solid green";
```



The NextElementSibling and NextSibling Properties

To locate the next sibling to the selected element, you can use one of the **nextElementSibling** and **nextSibling** properties.

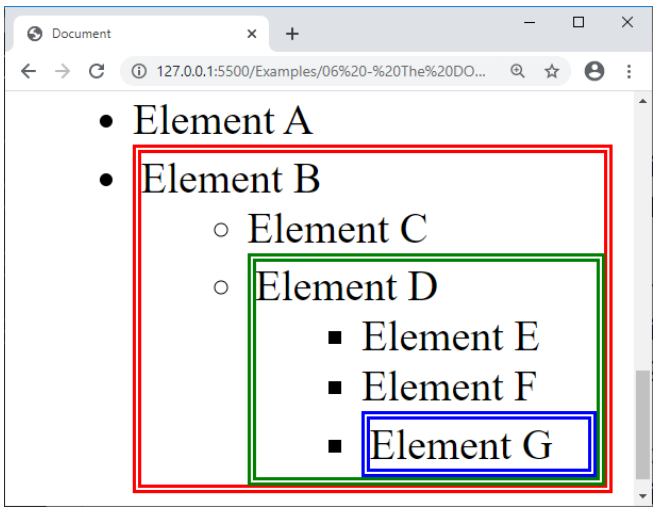
The difference between the **nextElementSibling** and **nextSibling** properties, is that **nextSibling** returns the next sibling node as an element node, a text node or a comment node, while **nextElementSibling** returns the next sibling node as an element node (ignores text and comment nodes).

```
element.nextSibling;
```

```
element.nextElementSibling;
```

Example 6-20: Finding the next sibling

```
// Blue border around Element G
const sibling1 = document.getElementById("second-element").nextElementSibling;
sibling1.style.border = "3px double blue";

// Green border around Element D
const sibling2 = document.getElementById("second-ul").firstElementChild
    .nextElementSibling;
sibling2.style.border = "3px double green";

// Red border around Element B
const sibling3 = document.getElementById("first-ul").firstElementChild
    .nextElementSibling;
sibling3.style.border = "3px double red";
```

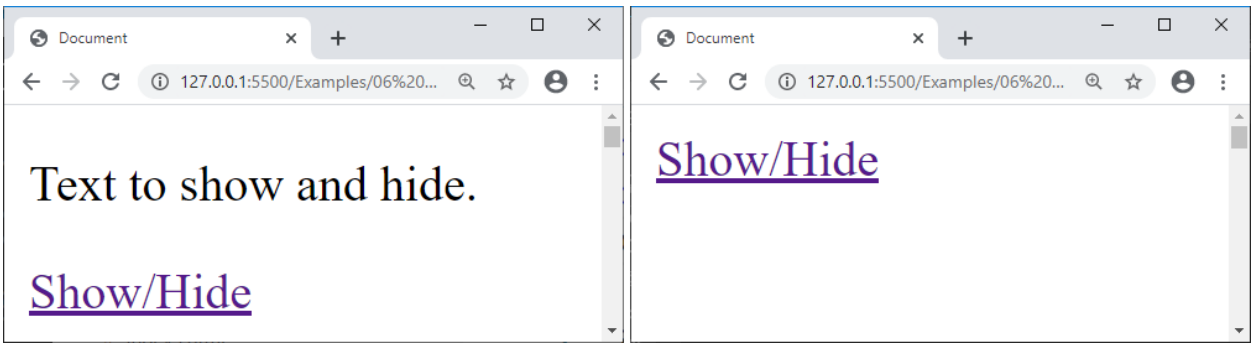


Showing/Hiding Elements

To hide an element, you assign **none** to the **style.display** property; To display a hidden element, you can, for instance, assign **block** or **inline** to the **style.display** property depending on how the element should be displayed.

Example 6-21: Hiding and showing an element

```
<p id="example-area"></p>
<a id="show-hide" href="#">Show/Hide</a>

const p = document.getElementById('example-area');
const showHide = document.getElementById('show-hide');
p.innerHTML = 'Text to show and hide.';

showHide.addEventListener('click', event => {
    if (p.style.display === "none") {
        p.style.display = "block";
    } else {
        p.style.display = "none";
    }
});
```



07—Events

Events are beneficial for user interaction, such as **mousedown**, **mouseup**, **change**, and **click**. You set up all these events the same way; they have an event handler function and an optional event parameter that has data relating to the event, such as the x- and y-position of the mouse for the mouse events. Here we'll focus on **click** and **change**. Each event has a corresponding HTML attribute that is used in dynamic scenarios—that are prefixed with **on**, such as **onclick** and **onchange**—these event attributes are not best practices according to Mozilla, instead, event bubbling should be used.

You can find detailed information about the events on the Mozilla site:
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events

Event bubbling sends the event request to the parent, grandparent, and so on until reaching a matching event. To achieve bubbling, you register the event with the container (element) that contains the element that the user clicks.

Imagine that you are creating a to-do application using a <ul> where the user adds new tasks to the list as <li> elements; to handle the click events for marking the task as complete or clicking a button in the <li> element, could be added with the **onclick** HTML attribute, but that's not best practices because you mix HTML and JavaScript; instead, you add the click event with JavaScript to the <ul> element and let the click event bubble up from the <li> and its child elements to the <ul> container.

Note that you, in this scenario, need to check the **event** object sent to the click event function to use the element the user clicked on. You might need to traverse the DOM to the parent or grandparent element to read an attribute or CSS class selector. Let's say that the <li> element has an icon and text inside a (see HTML below) and the <li> has an attribute, maybe an id, then you need to find the parent or grandparent of the <li> to get to that attribute. You can read the attribute with the **getAttribute** function.

If the user clicks the description or the icon, you need to traverse the DOM to the element's grandparent to be able to read the **cid** attribute, and if the clicking the span, you need to find the parent to read the sane attribute using the **event** object.

```
<ul>
    <li cid="1"><span><i>icon</i><span>Description</span></span></li>
    <li cid="2"><span><i>icon</i><span>Description</span></span></li>
</ul>

id1 = event.target.getAttribute("cid");
id2 = event.target.parentElement.getAttribute("cid");
id3 = event.target.parentElement.parentElement.getAttribute("cid");
```

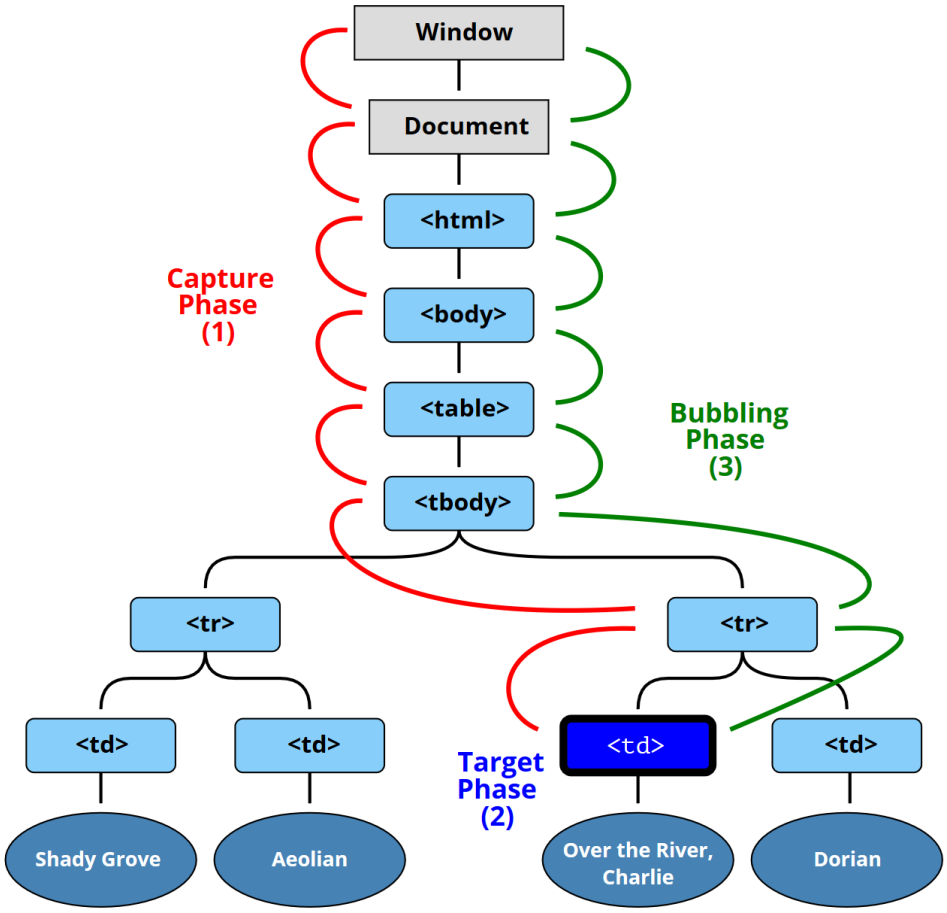


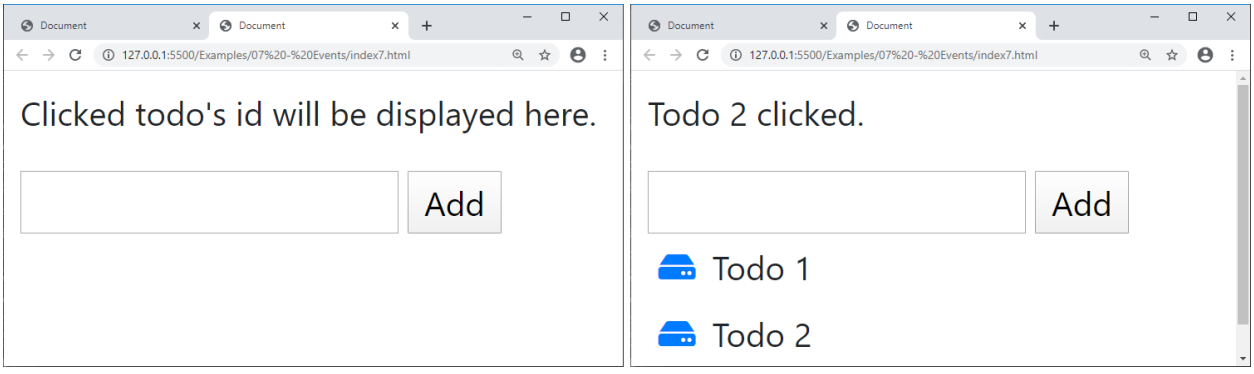
Image from: https://javascript.info/bubbling-and-capturing

Click

The **onclick** event is triggered when the user clicks an element, and it doesn't need to be a button or an anchor tag; this event works with most elements.

Example 7-1: The **onclick** Event

In this example, to-do items are added to the <ul> when the button is clicked, reading the text from the textbox. When clicking a to-do, display its id (from the **todoId** attribute) in the <p> element. We use Bootstrap 4 and font awesome icons in this example.



```
<body class="ml-2 mt-2">
    <p id="example-area">Clicked todo's id will be displayed here.</p>
    <input id="todo-text" type="text" />
    <button id="add-todo">Add</button>
    <ul id="todo-ul" style="list-style-type: none;padding-inline-start:5px;">
    </ul>

    <script src="Example 7-1.js"></script>
</body>

// Caching HTML elements
const p = document.getElementById('example-area');
const addTodo = document.getElementById('add-todo');
const todoText = document.getElementById('todo-text');
const todoUl = document.getElementById('todo-ul');

// Button click event
addTodo.addEventListener('click', event => {
    // Create the to-do <li> element
    const li = document.createElement("li");
    li.setAttribute("todoId", todoUl.children.length + 1);
    li.innerHTML =
        `<span class="d-flex nav-link-inline py-1">
            <i class="text-primary mr-2 mt-1 fas fa-hdd"></i>
            <span>${todoText.value}</span>
        </span>`;

    // Add the to-do <li> element to the <ul>
    todoUl.appendChild(li);
});

// Todo item click event
todoClick = function(event) {
    let id = 0;
    // Locate the todoId attribute
    if(event.target.localName == 'li')
        id = event.target.getAttribute("todoId");
    else if(event.target.parentElement.localName == 'li')
        id = event.target.parentElement.getAttribute("todoId");
    else if(event.target.parentElement.parentElement.localName == 'li')
        id = event.target.parentElement.parentElement.getAttribute("todoId");

    // Display the id in the paragraph
    if(id > 0) p.innerHTML = `Todo ${id} clicked.`
}

// Register the bubbling click event for the to-do <li> elements
todoUl.onclick = todoClick;
```

Change

The **onchange** event is beneficial when working with drop-down elements; it's triggered when the user selects an item in the drop-down list.

Example 7-2: Drop-down values and text

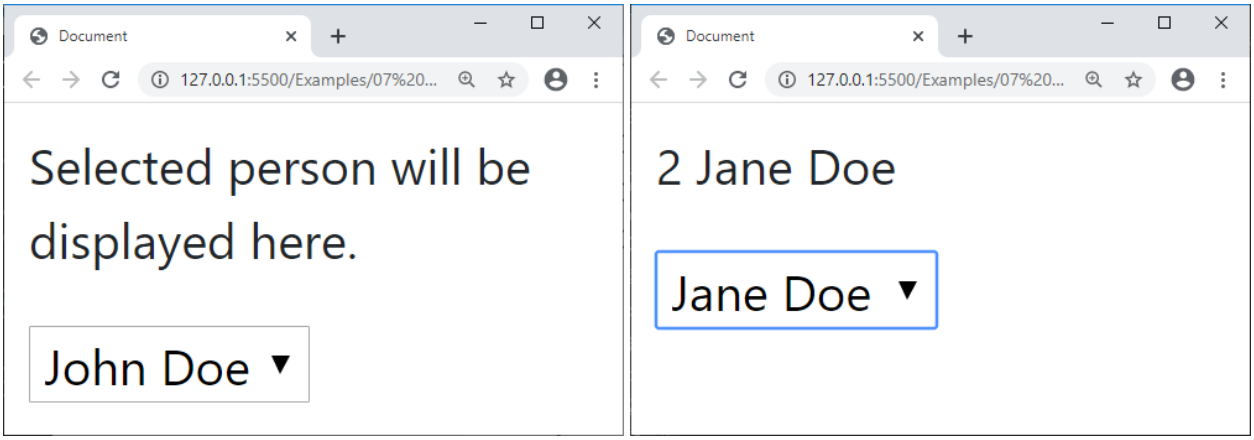
You can hook up the **onchange** event in any of the previously described ways; I'll show you two ways here.

HTML

```
<p id="example-area">Selected person will be displayed here.</p>
<select id="names">
    <option value="1">John Doe</option>
    <option value="2">Jane Doe</option>
</select>
```

JavaScript

```
// Caching HTML elements
const p = document.getElementById('example-area');
const names = document.getElementById('names');

// Hooking up event the first way
names.onchange = (event) => {
    const value = parseInt(event.target.value);
    const name = event.target.selectedOptions[0].text;
    p.innerHTML = `${value} ${name}`;
}

 names.addEventListener('change', event => {
    const value = parseInt(event.target.value);
    const name = event.target.selectedOptions[0].text;
    p.innerHTML = `${value} ${name}`;
});
```



08—Classes

What is a class, and why do we need them? A class is a blueprint that defines what is available in the objects that are created (instantiated) from the class. Each object shares the same functions, so you don't need to implement them for each object, on top of that, each object encapsulates its data to avoid

broadcasting it—you can even manipulate the data for each object through properties before storing or reading it. Name classes with Pascal casing, that is, the first letter in each word of the name begins with a capital letter, even the first letter.

You create an instance as a variable using the class as its data type followed by a pair of constructor parentheses and the **new** keyword to create the instance. Constructor functions are discussed later in this chapter.

```
let person1 = new Person();
```

Functions

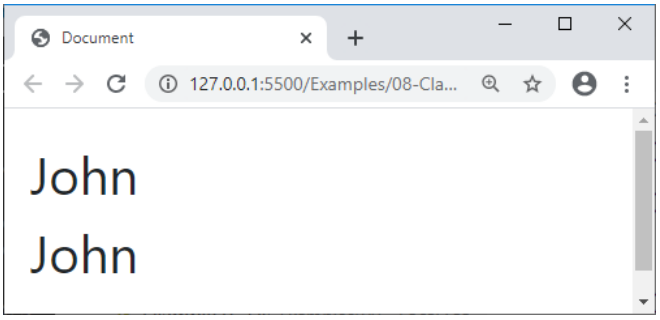
Functions work the same way as the functions you have created before; they process some data, executes code, and might return a result. The difference is in how you create them. A function within a class doesn't use the **function** keyword and only needs its name and a possible parameter list stated, followed by its function body where the function's executable code goes.

```
class TheClassName{
    functionName1(){
        // The function's executable code goes here.
    }

    functionName2(a, b){
        // The function's executable code goes here.
    }
}
```

Example 8-1: Creating a class with a function

Now, imagine that you want to create an object that contains a person's name; you could certainly do that without classes, but classes make it much easier and more concise. A **Person** class would be a reusable blueprint for creating as many **person** objects as needed; to separate concerns, you could create one JavaScript file per class that you link to in the HTML file. You decide to use a function named **getName** to return the name from the object, any objects you create from that class shares that function.

```
class Person{
    getName(){
        return "John";
    }
}

let person1 = new Person();
let person2 = new Person();

p.innerHTML = `${person1.getName()}</br>${person2.getName()}`;
```



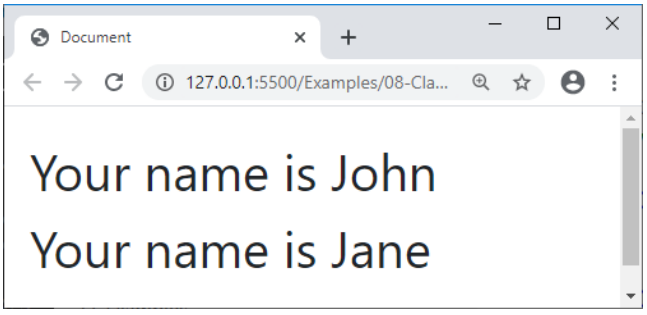
The result shouldn't surprise if you studied the code closely; it reflects what we have been discussing about shared class functions when creating objects from the same class. Indeed, the two person objects (person1 and person2) share the same function, and therefore, in this case, return the same data because it's an immutable piece of string data that returns from it.

Logic would, therefore, dictate that it would be nice to be able to send data into the function and do something with it, and then return the result. Sending in data is, of course, possible and useful in many situations, such as performing a calculation where one or more values pass into the function and it returns a result. Let's see how that would work with our **Person** class and the objects.

Example 8-2: Creating a class with a parameterized function

The only changes from the previous code are that we now pass in a parameter called **name** to the function, manipulate the returned value with a backtick string, and pass in a name to the **getName** function for each person object.

```
const p = document.getElementById('example-area');

class Person{
    getName(name){
        return `Your name is ${name}`;
    }
}

let person1 = new Person();
let person2 = new Person();

p.innerHTML = `${person1.getName('John')}</br>${person2.getName('Jane')}`;
```



Indeed, this returns different results for the two person objects when calling the **getName** function, but it's not optimal—the function never stores the name as encapsulated data that is part of each object separately and never shared between any of the objects.

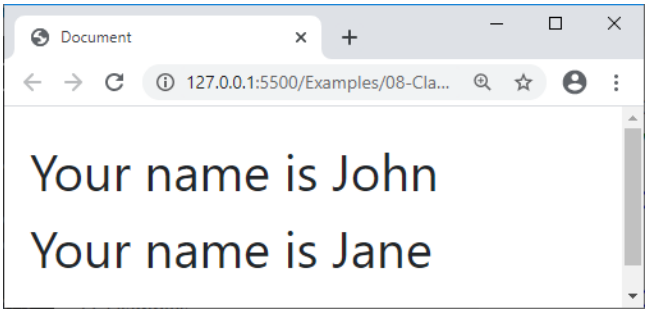
Constructor

The next step would be to change the class definition to allow data to pass in when creating the instance; you do this by adding a special constructor function that executes when creating a new instance of the class. This unique function can have parameters like any other function. You add a constructor by adding a function with that name.

The data passed into the constructor can then be stored in a field, a hidden variable that is part of the object and isn't shared with other objects—it's encapsulated. To create "hidden" fields in a class that store unique data for each object created from it, you use the **this** keyword, which points to (refers to) the object itself. You cannot create a regular variable inside the class and use it to encapsulate data in objects; you must prefix the variables with **this** and a dot; a common practice is to prefix the "hidden" variable's name with an underscore. Because the name is passed into and stored by the constructor, the **getName** function no longer need the **name** parameter because it is accessible through the **this._name** variable.

Example 8-3: Creating a class with a constructor

```
const p = document.getElementById('example-area');

class Person{
    constructor(name){
        this._name = name
    }

    getName(){
        return `Your name is ${this._name}`;
    }
}

let person1 = new Person('John');
let person2 = new Person('Jane');

p.innerHTML = `${person1.getName()}</br>${person2.getName()}`;
```



Properties

A property is a way to manipulate a value before it is stored or fetched from a backing "hidden" variable. In some cases, a property concatenates values from multiple backing variables with a backtick string; for instance, a person's full name comprised by values from the first name and last name variable.

Properties consist of two special functions that are declared with the **get** and **set** keywords. If you only want to be able to read with the property, you omit the **set** property function. Omitting the **set** function is a half-truth because all variables are accessible from the prototype object and despite an omitted **set** property.

Note that if we chose to have a constructor that assigns an initial value to the backing variable, it can now call the **set** property instead of using the backing variable directly.

You assign a value to the property by setting it equal to the value you want to assign.

Example 8-4: Creating a class with a property

```
const p = document.getElementById('example-area');

class Person{
    constructor(name){
        // Calls the set property
        this.name = name
    }

    get name() { return `Your name is ${this._name}`; }
    set name(value) { this._name = value; }
}

let person1 = new Person('John');
let person2 = new Person('Jane');

p.innerHTML = `${person1.name}</br>${person2.name}`;

// Assigning a new name to person1
person1.name = "Carl";
p.innerHTML += `</br><span style='color:red;'>${person1.name}</span>`;
```

Inheritance

With inheritance, a class can extend (inherit) all functions and state (data) from the superclass (the class it inherits from). So far, we have worked with a single class name **Person**, but what if we decide to create

an **Employee** class? Instead of reinventing the wheel, and implement the same properties and functions that we already have declared in the **Person** class, we could have the **Employee** class inherit the **Person** class and thus get access to the already existing functionality and state of that class.

To determine if it's appropriate to use inheritance, think of it as being of the inherited type, an employee *is a* person. Another way of viewing it is that the **Employee** class becomes more *specialized* than the class it inherits.

To inherit a class, you use the **extends** keyword followed by the class you want to inherit after the inheriting class name.

```
class Employee extends Person{
    constructor(name, group){
        // Passes the name to the Person class' constructor
        super(name);
    }
}
```

The **Employee** class can then implement functions, properties, and variables that aren't part of the **Person** class.

*You should avoid deep inheritance chains where many classes are involved; for instance, the **Person** class could be the ultimate superclass that is inherited by the **Employee** class, which in turn **Manager** class inherits, which then, in turn, the **Executives** class inherits. An inheritance chain like this can be hard to understand and follow.*

The Super Function

If the class that is inherited has a constructor with other parameters than the once from the inheriting class, the inheriting (the more specialized) class must call that constructor using the **super** function and pass any parameters required by the constructor as parameters to the **super** function.

The **super** function is not restricted to constructors; any overridden function—with the same name in the superclass (the inherited class)—can call it, which you learn later in this chapter.

You can also use the **super** keyword to invoke any function in the inherited class by using the dot notation. The following code would call a function named **foo** in the inherited class.

```
super.foo();
```

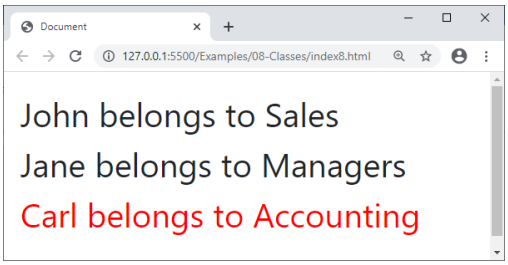
You should never try to call functions in the inherited class with the **this** keyword because that relates to the inheriting class.

Example 8-5: Inheriting a class

Note that we pass in the name to the **Person** class with the **super** function, which stores it in the **_name** variable. Also note that we create instances of the **Employee** class, not the **Person** class, because the **Employee** class inherits it; because of the inheritance, we can still access the **name** property in the **Person** class (although not from the constructor).

Remove the text *Your name is* from the **name** property and return the only the name.

```
const p = document.getElementById('example-area');

class Person{
    constructor(name){
        this.name = name
    }

    get name() { return this._name; }
    set name(value) { this._name = value; }
}

class Employee extends Person{
    constructor(name, group){
        super(name); // Calls the constructor in the Person class.
        this.group = group;
    }

    get group() { return this._group; }
    set group(value) { this._group = value; }

    get employee(){ return `${this.name} belongs to ${this.group}`;}
}

let employee1 = new Employee('John', 'Sales');
let employee2 = new Employee('Jane', 'Managers');

p.innerHTML = `${employee1.employee}</br>${employee2.employee}`;

employee1.name = "Carl"; // This property belongs to the Person class.
employee1.group = "Accounting"; // This property belongs to the Employee class.
p.innerHTML += `</br><span style='color:red;'>${employee1.employee}</span>`;
```



Overriding Functions

When overriding a function in the base class, you tell the runtime compiler that the new function should be used instead of the one in the inherited class. But what if you wanted to reuse the logic in the already existing function from the inherited class' function? You could do that by calling the **super** function inside the overriding function in the inherited class.

Example 8-6: Overriding a function

Let's try overriding a function; this means that we need a function in both classes with the same name. In this scenario, we want to be able to call a function named **doWork** on either a **Person** object or an **Employee** object, and the logic shouldn't care which object it is executing the code on; the result will,

however, depend on which object calls the function. To achieve this, we create a function named **makeEveryoneWork** that takes a *rest* parameter named **people**, which receives the people that should do some work. The logic inside this function should be agnostic to what type of person object it calls the **doWork** function on. In other words, we want this function, which isn't in either of the **Person** of the **Employee** class, to receive instances of these classes and call their **doWork** function seamlessly.

So, we continue with the code from the previous example and add a function named **doWork** to both the **Person** of the **Employee** classes, but the returned value is different. The function in the **Person** class returns "Free" because such persons are doing pro-bono work, which is free. The function in the **Employee** class returns "Paid" because such persons are on the company payroll.

If we call the **doWork** function on a **Person** object, it returns *free*, whereas the same function on an **Employee** object returns *paid*.

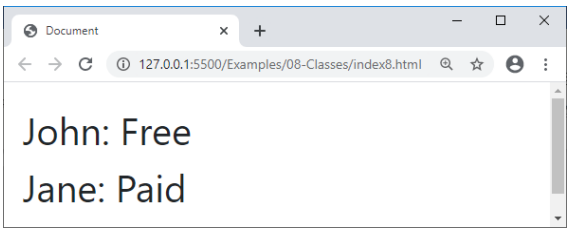
```
// This function is in the Person class.
doWork(){
    return "Free";
}

// This function is in the Employee class.
doWork(){
    return "Paid";
}
```

We replace all code below the **Person** and **Employee** classes with a function named **makeEveryoneWork** that calls the **doWork** function on the objects we pass into it through its **people** *rest* parameter.

```
makeEveryoneWork = (...people) => {

    people.forEach(person => {
        p.innerHTML += `${person.name}: ${person.doWork()}</br>`;
    });
}
```

Next, we create one object from each class and call the **makeEveryoneWork** function, passing in the two objects to it.

```
let person = new Person('John');
let employee = new Employee('Jane', 'Managers');

makeEveryoneWork(person, employee);
```



As you can see, we call the **doWork** method in the **Person** class when executing the **person** object, and it executes the **doWork** method in the **Employee** class when processing the **employee** object.

09—Local Storage

If you need to store data in the user's browser, Local Storage is the way to go. It can store more information than a cookie on the user's local machine. Note that this doesn't replace a back-end database; it's is a way to store large amounts of data such as user preferences and other settings locally in the browser without affecting the website performance. At least 5MB of storage is available, and the browser never sends the data to the server.

The browser stores the data per domain and protocol, and all pages from that origin can share the data.

You use either the **window.localStorage** to store data without an expiration date or **session.localStorage** for data that expires when the browser's session ends (the browser tab is closed).

You can store single values, arrays of data, JSON objects, or even HTML elements with their contents.

Saving Data

You save data to a property in Local Storage by calling the **setItem** method on either the **localStorage** object or the **sesionStorage** object. The first parameter is the name of the property, and the second is the value to store. Note that the value can be a long string, such as HTML elements.

```
localStorage.setItem("lastname", "Smith");
```

You can also use the property name directly to assign the value.

```
localStorage.lastname = "Smith";
```

Retrieving Data

To retrieve a value, you call the **getItem** method on either the **localStorage** object or the **sesionStorage** object; the only parameter needed to retrieve data is the name of the property containing it.

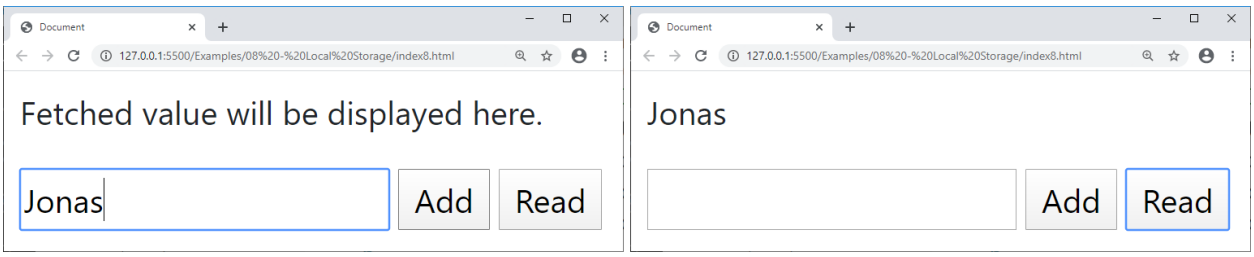
```
document.getElementById("result").innerHTML = localStorage.getItem("lastname");
```

You can also use the property name with dot-notation to fetch the value.

```
document.getElementById("result").innerHTML = localStorage.lastname;
```

Example 08-1: Store data in Local Storage

This example shows how to store and retrieve a name from Local Storage when the user clicks a button.



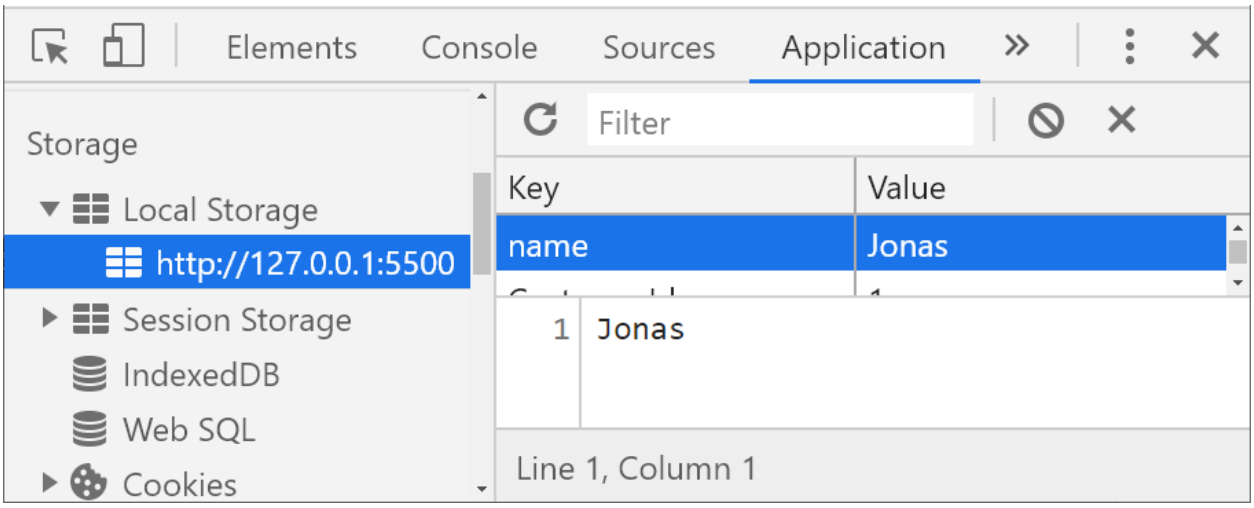
HTML

```
<p id="example-area">Fetched value will be displayed here.</p>
<input id="todo-text" type="text" />
<button id="save">Add</button>
```

```
<button id="read">Read</button>
```

JavaScript

```
// Caching HTML elements
const p = document.getElementById('example-area');
const todoText = document.getElementById('todo-text');
const onSave = document.getElementById('save');
const onRead = document.getElementById('read');

// Hooking up event first way
onSave.onclick = () => {
    //localStorage.setItem("name", todoText.value);
    localStorage.name = todoText.value;
    todoText.value = '';
}

onRead.onclick = () => {
    //p.innerHTML = localStorage.getItem("name");
    p.innerHTML = localStorage.name;
}
```



Example 08-2: Store JSON data in Local Storage

This example shows how to store and retrieve an JSON array from Local Storage when the user clicks a button.

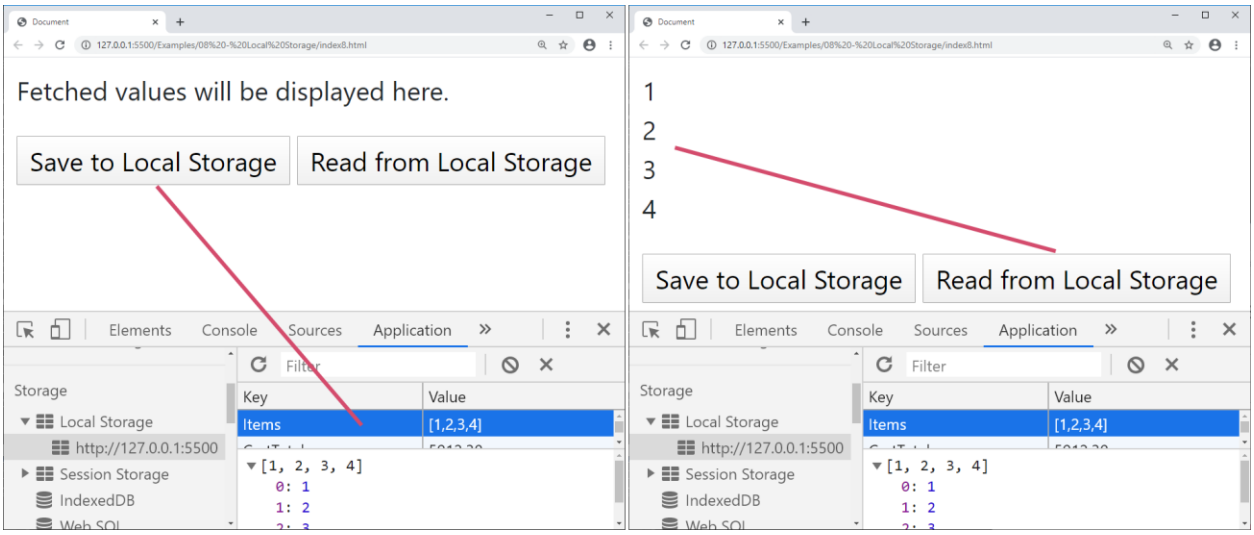
HTML

```
<p id="example-area">Fetched values will be displayed here.</p>
<button id="save">Save to Local Storage</button>
<button id="read">Read from Local Storage</button>
```

JavaScript

```
// Caching HTML elements
const p = document.getElementById('example-area');
```

```
const todoText = document.getElementById('todo-text');
const onSave = document.getElementById('save');
const onRead = document.getElementById('read');

// Hooking up the Save onclick event
onSave.onclick = () => {
    const items = [1, 2, 3, 4];
    // Convert the data in the array into JSON
    localStorage.Items = JSON.stringify(items);
}

// Hooking up the Read onclick event
onRead.onclick = () => {
    // Convert the data in Local Storage into JSON
    let items = JSON.parse(localStorage.Items);
    p.innerHTML = '';
    items.forEach(item => {
        p.innerHTML += `${item}</br>`;
    });
}
```



10—Asynchronous API Calls

To make your page more dynamic, you can use asynchronous API calls to a server—without affecting the web site's performance by loading the data in parallel with other processes already executing on the site. When you make an asynchronous call, the application continues to work (without locking and freezing the browser) and awaits the result from the request; when the response arrives, a function triggers where you can manipulate the data and display it to the user. You can use asynchronous calls to load data for a specific part of the browser without reloading the page.

You can perform asynchronous calls in several ways with JavaScript; the most modern approach is to use the **async-await** pattern, which uses promises in the background. A promise is what it sounds like, a promise of a result sometime in the future (we don't know how far off in the future though, it can be a millisecond, a minute, or longer! Depending on how long it takes the server to respond to our request—

if it does.) The execution runs separately from the main application thread, which means that the browser continues to work while it is waiting for the response.

When the code reaches the **await** keyword, the execution of that method halts until receiving a response to our request (API call) from the server, then the execution continues in that function. Important to know is that calling other functions that after the call to the asynchronous function doesn't halt them; they continue as soon as we send the asynchronous API call at the **await** keyword to the server, which means that you need to implement any code that relies on the answer from the API call below the **await** keyword in the same function, or a function called below the **await** keyword in the same function. There are other ways of solving this, but we won't be going into them in this beginner course.

In this course, you work with the **async-await** pattern because it's the most modern approach and the easiest to learn. When implementing one of the obligatory exercises, you get very familiar with this concept.

To make the asynchronous API calls, you use a library from Axios that implements the asynchronous functionality using promises, which works with the **async-await** pattern. Scroll down on their GitHub page until you find the CDN link and copy it, then paste that link above any other JavaScript links in the body of the HTML page. https://github.com/axios/axios

If you are familiar with JQuery, this is JavaScript's way of implementing AJAX calls.

Async-await

To call an API asynchronously with the **async-await** pattern, you must decorate the function that makes the call with the **async** keyword, and the call to the Axios library with the **await** keyword. I recommend implementing error handling with try/catch blocks inside the function, where the try-block encompasses the code that you want to check for errors, and the catch-block handles any errors that occur.

The Axios library contains functions for the necessary HTTP verbs, such as **get**, **put**, **post**, and **delete**. **Get** fetches data incorporating the necessary ids into the URL or as URL parameters. **Put** updates a resource, and therefore might need data sent in the request body, the same goes for the **post** method that adds a new resource. **Delete** removes a resource and therefore needs the id of the resource to remove.

In the code below, the **getCustomers** function is decorated with the **async** keyword to enable awaiting an asynchronous call. The call to the API is made by calling the **get** method in the Axios library and passing in the URL to the API endpoint. The call immediately awaits a result on the next line of code and stores the returned data in a constant named **customer**.

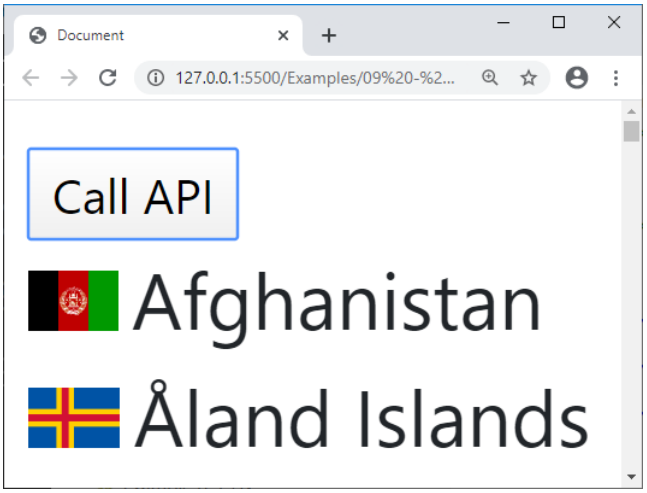
```
getCustomers = async () => {
    try {
        // Call the API to fetch data
        const customer = axios.get(UrlToAPI);
        // Await the result from the server and store
        // the result in a variable named customers
        const { data: customers } = await customer;

        // Iterate over the returned result
        customers.forEach(c => {
        });
```

```
    }
    catch (err) { console.log("getCustomers: ERROR", err); }
}
```

Example 9-1: Calling an API asynchronously

This example shows how to call an API with the Axios library and how to handle a successful as well as a faulty call. When the user clicks the button, the **get** method in the Axios library makes an asynchronous call to the desired URL; Axios return data as JSON by default.

If the call is successful, the loop executes, displaying the returned data in the browser—in this case, all countries and their flags. If the call returns an error, the catch-block executes, displaying an error message in the <p> element and the console.



HTML

```
<body class="ml-2 mt-2">
    <p id="error"></p>
    <button id="get-countries">Call API</button>
    <div id="countries"></div>

    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <script src="Example 9-1.js"></script>
</body>
```

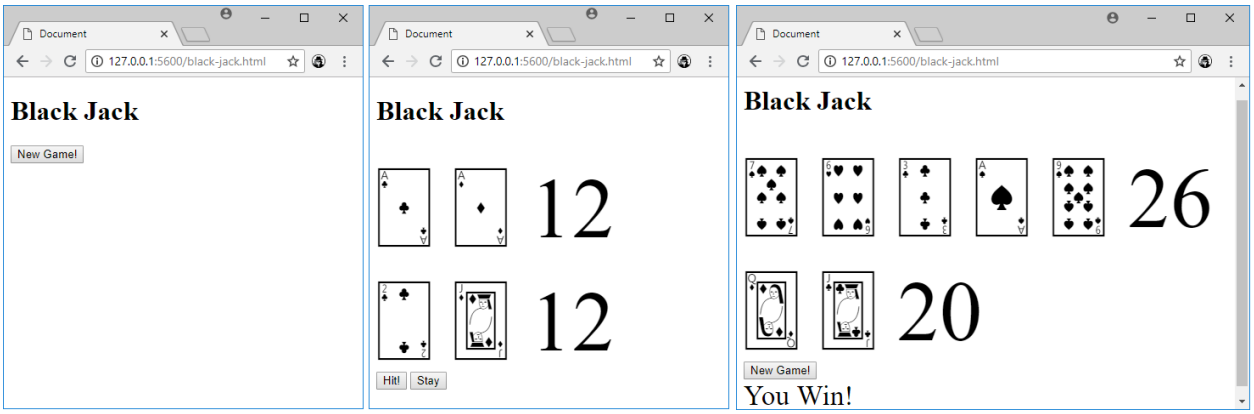
JavaScript

```
const p = document.getElementById('errors');
const div = document.getElementById('countries');
const get = document.getElementById('get-countries');

get.onclick = async () => {
    try {
        // Call the API to fetch data
        const result = axios.get('https://restcountries.eu/rest/v2/all');
        // Await the result from the server and store
        // the result in a variable named customers
```

```
        const { data: countries } = await result;

        // Iterate over the returned result
        countries.forEach(country => {
            div.innerHTML +=
                `<img id='banner' height='20px;'
                     style='margin-top:-11px;' src='${country.flag}'/>
                 <span style='font-size:26px;'>${country.name}</span><br/>`;
        });
    }
    catch (err) {
        p.innerHTML = `Couldn't reach the API.`
        console.log("getCustomers: ERROR", err);
    }
}
```

11—Obligatory Assignment—Blackjack

Now it's time for the fun stuff—creating an actual application. You build a Blackjack application from scratch with JavaScript and a little HTML and styling, below is an image of what the finished application looks like in the browser. ***Read through the entire assignment before you start coding!***



You implement the code in stages when building the application.

There are three buttons: **New Game!**, **Hit!** and **Stay**. The **New Game!** button starts a new game and hides when clicked. Clicking this button shows the other two hidden buttons. When the player clicks the **Hit!** button, the player receives a new card. For each new card the player gets, the game calculates the totals for both the player and dealer and evaluates if the player is bust. If the player is bust, the hand ends, and the **New Game!** button becomes visible along with a text proclaiming the winner. Also, hide the **Hit!** and **Stay** buttons. If the player clicks the **Stay** button, execute the same scenario described for the **Hit!** button.

The rules of Blackjack

The goal is to get as close to 21 as possible without getting bust (go over 21). Display the dealer's first two cards along with yours. Then prompt the player to hit (take another card) or stay (the dealer begins drawing his cards). If the dealer's cards have a higher value than the player, the bank wins. If the player's and dealer's cards have the same value, it's a draw. If the player's cards have a higher value than the dealer's, you win. The dealer must draw on 17 and stay on 18; this means that the dealer isn't allowed to draw any more cards if his cards have a higher value than 17. If you get an ace and a (10, Jack, Queen, or King), you have a Blackjack, which means that you automatically win the game. The same goes for the dealer, who also can win by drawing five cards under 21. 10, J, Q and K all have the value 10, the Ace 1 or 11, and all other cards have the value they display. Note that these are slightly modified and simplified rules to narrow the scope of the assignment.

- Goal: Get as close to 21 as possible without going bust.
- If you get Blackjack (an Ace and 10, J, Q, or K), you win.
- If the dealer gets a Blackjack, he wins.
- You cannot get a draw if both have Blackjack because the dealer hasn't technically seen his second card before you see your two cards; this means that you always win if you have a Blackjack.
- You can choose to stay at any point and not draw any more cards. Then the dealer starts drawing cards until:
    - He gets a higher value than you, but less than or equal to 21 (the dealer wins)

    - He gets bust (gets a value over 21); you win.
    - You get the same value (a draw); it's a tie.
    - He gets five cards under or equal to 21; the dealer wins.
    - He must draw on 17 and stay on 18; this means that if his hand has the value 18 or over, he is not allowed to draw another card, even if your hand has a higher value.
- The 'face cards' 10, J, Q, and K all have the value 10, and Ace has the value 1 or 11. All other cards have the value they display.

01—Setting Up the Project

The first thing to do is to create a folder for the project and add HTML and JavaScript files to the project in Visual Studio Code. In this assignment, you should use the knowledge from previous chapters. Hence, there is very little code in the assignment's description.

1. Create a folder on your hard drive for the project.
2. Open Visual Studio Code and Select **File-Open Folder** and open the folder you created in the previous step; this assures that all files you add end up in that project folder.
3. Click the **Extensions** button in the left sidebar menu.
4. Search for *Live Server*, which is a server that updates the browser's content in real-time when you save the files in your project.
5. Click the **Install** button for Live Server to install it unless already installed.
6. Click the **Explorer** button in the left sidebar menu to get back to your project files.
7. Click the **New File** button and add a file called *black-jack.html*.
8. Repeat step 7 for a file named *black-jack.js*.
9. Open the *black-jack.html* file and add the barebones of the HTML page by typing an exclamation mark (**!**) and hitting **Tab** once on the keyboard.
10. Change the text in the <title> tag to *Blackjack*.
11. Add a <script> tag at the end of the <body> element for the JavaScript file you added to the project.
12. Add a <h1> tag with the text *Blackjack* to the beginning of the <body> element; this is the title of the application.
13. Add element below the <h1> element that will hold the dealer's and player's hands (cards) and value. You add the content using JavaScript later. To be able to reach the element from JS, you need to add an id, let name it *output-area*. To make the cards readable, you need to bump up the font size to about 75pt (you can use an inline style).
14. Add a <button> element with the text *New Game!* And the id *new-game-button*.
15. Add a <button> element with the text *Hit!* And the id *hit-button*.
16. Add a <button> element with the text *Stay* and the id *stay-button*.
17. Add a element with the id *winner-area* and set the font size to about 25pt.

02—Creating a Collection of Cards

The first step to a Blackjack game is to have a deck of cards. You use an online source to get the cards as characters and add them to an array along with their values. Each card in the array is an object with two properties: **card** and **value**. Because these values never change, you can add them directly in the array's square brackets when it is declared.

You can find the card characters here: https://unicode-table.com/en/search/?q=player+cards

1. Open the JavaScript file.
2. Create an array called **cards** for the card objects that have two properties: **card** (which holds the card's character from the site above) and **value** (which holds the card's value 1-10. Ace has the value 1).
3. Display all the cards in the array in the console to verify that the array is correctly declared.
4. Right-click on the *.html* file in the file list, then select **Open With Live Server** to start the application in a browser.
5. Open the F12 Developer Tools and verify that all 52 cards are in the array.
6. Remove the code that displays the array in the console.
7. Add a loop that iterates over the array and adds the card character to a string variable for each card.
8. Display the content of the string in the *output-area* . Verify that the browser displays all 52 cards.

03—Shuffling the deck

Playing with an unshuffled deck wouldn't be very much fun, you, therefore, have to create a function that shuffles the cards in the array you created in the previous step and stores the shuffled cards in a new array called **deck**. Use the **Math** class to shuffle the cards and the **slice()** array function to make a temporary copy of the **cards** array that you can work with when shuffling the cards.

You can use the spread operator (**...**) to get the value from the array that contains the card to splice (remove) from the temporary cards array with the **splice()** function.

1. Create an empty array called **deck** immediately below the cards array you added earlier. This array holds the shuffled cars.
2. Add a function called **shuffleDeck** that has no parameters.
3. Copy the **cards** array into a new array called **tmpDeck** inside the function. Later, you use this array to splice out the cards one at a time based on a random number.
4. Add a while loop on the number of elements left in the **tmpDeck** array for as long as there are elements left in the array.
5. Inside the loop, find the index for a random card among the remaining cards in the **tmpDeck** array and store the index in a variable called **pos**.
6. Splice out the card from the **tmpArray** at the index stored in the **pos** variable. Store the card in a variable called **card**.
7. Add the **card** to the **deck** array using the spread operator (if you recall, the splice function returns an array of the spliced values).
8. Now that the **shuffleDeck** function is complete make a call to it below the function.
9. Change the loop that displays the cards to iterate over the **deck** array instead of the **cards** array to display the shuffled cards.
10. Save the changes and verify that the browser displays the shuffled cards.

04—Draw a Card

When playing Blackjack, the dealer needs to be able to draw a card from the deck. Let's add a function that draws a card and draw two cards and displays them in the *output-area* . Also, delete the loop that displays the whole deck of cards in the browser, you no longer need it.

1. Delete the loop that displays the whole deck of cards in the browser.
2. Add a function named **drawCard** that has no parameters.
3. Return the first card from the **deck** array using the array's **shift()** method.
4. Draw two cards by calling the **drawCard** function and display them in the *output-area* .
5. Save the application and verify that the browser displays the two cards from the **deck** array.

05—Show Hands

Now that you can shuffle the deck and draw cards from it, it is time to add functionality that draws the first two cards for the player and the dealer and display them in the browser.

1. Add two empty arrays called **dealer** and **player** below the **deck** array.
2. Cache the *output-area* in a variable called **outputArea** below the two arrays you just added. Caching an element can improve performance because the browser doesn't have to look up the element in the DOM every time.
3. Remove the code at the end of the JS file that displays data in the browser.
4. Add a new function called **showHands** that has no parameters at the end of the JS file.
5. Inside the function, loop over the **player** array and append each card's character (from the card's **card** property) to a variable called **playerCards**.
6. Repeat the previous step for the **dealer** array and store the result in a variable called **dealerCards**.
7. Concatenate the **dealerCards** string with the **playerCards** string with a new-line character in between and display the result using the **outputArea** variable you created earlier.
8. Create another function called **dealInitialCards** that has no parameters below the **showHands** function.
9. Inside the function, call the **drawCard** function twice for the player and add the cards to the **player** array.
10. Repeat the previous step for the **dealer** array.
11. Call the **ShowHands** method you created earlier at the end of the **dealInitialCards** function.
12. Call the **dealInitialCards** function below the function. Save the code and verify that the browser displays two cards for the dealer and two for the player.

06—Refactoring the Show Hands function

Now, refactor a few repetitive steps from the **ShowHands** method to make it able to show any hand.

1. Rename the **ShowHands** function **ShowHand** and add two parameters: **hand** and **score**. You use the score parameter later when you calculate the value of each hand.
2. Replace the two variables **playerCards** and **dealerCards** with one named **cards**.
3. Delete the loop iterating over the **dealer** array.
4. Change the loop iterating over the **player** array to iterate over the **hand** parameter and store the cards in the **cards** variable.
5. Concatenate the **cards** string with a space character and the value in the **score** parameter and add a new-line character at the end of the string. Display the result by concatenating the value with the already existing value in the **outputArea** element.
6. Change the **ShowHands** function call to call the modified **ShowHand** function and pass in the **dealer** array and **null** for the **score** parameter.
7. Add another call to the **ShowHand** function and pass in the **player** array and **null**.

8. To clear the game table before the hands are displayed, add a new function named **ClearTable** that has no parameters. If you don't do this, the next hand becomes part of the current hand.
9. Assign an empty string to the **outputArea** element.
10. Call the **ClearTable** function at the top of the **dealInitialCards** function.
11. Save all changes and verify that the browser displays the dealer and payer hands as before with the value **null** beside the cards.

07—Calculate Hand Value

Now, it's time to calculate the value and replace the **null** value passed into the **showHand** function with the actual hand value.

1. Add a function named **calculateHand** that has a parameter named **cards** that receive the cards from the player or dealer.
2. Add a variable named **score** and initialize it to zero.
3. Use the **find** function on the **cards** array passed into the function to find out if the player has an ace on the hand; this is important to know to determine if the hand is a Blackjack. Use the value property on the cards as a comparison value and check if it one (1); if the result from the **find** function is **undefined,** the hand has no ace.
4. Use the **forEach** array function to calculate the hand's score and store the result in the **score** variable you added earlier.
5. Add an if statement that checks if the score plus ten (10) is less than 21; if it is, then add 10 to the score. You add 10 because the ace can count as 1 or 11, but you don't want to add 10 if it makes the hand bust.
6. Return the hand's score form the function.
7. Create a variable named **playerScore** below the **outputArea** variable at the top of the file. Add a call to the **calculateHand** above the **ShowHand** function calls in the **dealInitialCards** function and pass in the **player** array; store the returned value in the **playerScore** variable.
8. Add a call to the **calculateHand** below the previous call to the function and pass in the **dealer** array. Store the returned value in a variable named **dealerScore** that you add below the **playerScore** variable at the top of the file. Pass in the values from the **playerScore** and **dealerScore** variables to respective **ShowHand** call; this displays the player's and dealer's score in the browser.
9. Save the file and verify that the browser displays the score beside the dealer's and player's cards and that the score is correct.

08—Start a New Game

Now, it's time to hook up a **click** event to the **New Game!** button and add code to start a new game.

1. Add caching variables for the buttons, as you did with the *output-area* .
2. Delete the code that calls the **dealInitialCards** and **shuffleDeck** functions; you'll add calls to them inside the next function you create.
3. Add a function named **startNewGame** that takes no parameters.
4. Hide the *start-new-game* button and show the *hit-button* and *stay-button* buttons at the top of the function.
5. Clear the **deck**, **player** and **dealer** arrays inside the function.
6. Call the **shuffleDeck** and **dealInitialCards** functions at the end of the function.

7. Add a **click** event listener for the *new-game-button* <button> variable.
8. Call the **startNewGame** function from the event function.
9. Save the file and click the **New Game!** Button to deal the first cards. The other buttons won't work because they have no **click** event handlers yet.

09—Draw a New Card

Now, it's time to hook up a **click** event to the **Hit!** button and add code to draw another card. The **dealInitialCard** needs refactoring to break out the code that calculates the scores and displays the hand into a separate function named **ShowHands**. You do this to avoid code duplication.

1. Add a new function named **ShowHands** that takes no parameters.
2. Cut out the code that calculates the score and the calls to the **ShowHand** function from the **dealInitialCards** function (it should be the last four code lines in the function).
3. Add a call to the **ShowHand** function at the end of the **dealInitialCards** function.
4. Cut out the **clearTable** function call from the **dealInitialCards** function and add it above the first **ShowHand** function call in the **ShowHands** function.
5. Replace the code that assigns the **cards** and **score** data to the **outputArea** with a return statement that returns the same data from the **ShowHand** function.
6. Append the result from the calls to the **ShowHand** function calls to the **outputArea** variable.
7. Add a new function named **dealAnotherCard** that has a parameter named **hand** that receives either the player's or the dealer's hand.
8. Call the **drawCard** function and add the returned card to the array in the **hand** parameter.
9. Add a **click** event listener for the *hit-button* button.
10. Call the **dealAnotherCard** function and then the **ShowHands** function from the **click** event.
11. Save the file and click on the **Start Game!** button in the browser.
12. Click the **Hit!** button to deal another card to the player. Note that you can deal cards until the deck runs out of cards. You fix this later when you implement the comparison between the player and dealer hands.

10—Determine the Winner

Now, it's time to find out who's the winner and display the result in the *winner-area* below the buttons in the browser. To find out the winner, you first determine if the player or the dealer has Blackjack or are bust. To do this, you add a function named **hasBlackJack** that has two parameters: **hand** and **score** and checks if the hand has two cards with the value 21. Then you add a second function named **isBust** that has a parameter named score that checks if the score is higher than 21.

1. Add a function named **hasBlackJack** that has two parameters: **hand** and **score**.
2. Inside the function, return true if the hand has precisely two cards, and the score is 21.
3. Add a function named **isBust** that has one parameter named **score**.
4. Return true from the **isBust** function if the score is higher than 21.
5. Add a function named **determineWinner** that has one parameter named **stayed** that is assigned **true** if the player clicks the **Stay** button.
6. Add three constants named **dealerWins**, **playerWins,** and **draw** with the values *Dealer Wins!*, *You win!*, and *Draw* respectively. These are the possible outcomes that are displayed depending on who wins or if the scores are the same.

7. Add an if-statement that checks if the player is bust and returns the value from the **dealerWins** constant if the expression is true. Use the score that you stored in the **playerScore** variable earlier.
8. Add an else if-statement that checks if the dealer is bust and returns the value from the **playerWins** constant if the expression is true. Use the score that you stored in the **dealerScore** variable earlier.
9. Add an else if-statement that checks if the dealer has drawn exactly five cards with a score of 21 or less; return the value in the **dealerWins** constant if the expression is **true**.
10. Add an else if-statement that checks if the dealer and player have the same score; return the value in the **draw** constant if the expression is true.
11. Add an else if-statement that checks if the player's score is higher than the dealer's score; return the value in the **playerWins** constant if the expression is true.
12. Add an else if-statement that checks if the dealer's score is higher than the player's score; return the value in the **dealerWins** constant if the expression is true.
13. Add an else-block where you check if the player or dealer has Blackjack:
    a. Create a variable named **dealerBJ** that you assign the result of a call to the **HasBlackJack** with the **dealer** and **dealerScore** variables.
    b. Create a variable named **playerBJ** that you assign the result of a call to the **HasBlackJack** with the **player** and **playerScore** variables.
    c. Add an if-statement that checks if the **playerBJ** and **dealerBJ** variables are both **true**; return the value from the **draw** constant if the expression is true.
    d. Add an if-statement that checks if the **playerBJ** variable is **true**; return the value from the **playerWins** constant if the expression is true.
    e. Add an if-statement that checks if the **dealerBJ** variable is **true**; return the value from the **dealerWins** constant if the expression is true.
14. Return an empty string below the else-block to avoid returning **undefined**.
15. Create a new function named **showGameButtons** that has no parameters.
16. Move the three code lines that hide and show buttons from the **startNewGame** function to the function you just created.
17. Add a call to the **showGameButtons** function at the top of the **startNewGame** function.
18. Copy the **showGameButtons** function and rename it **hideGameButtons**.
19. Change *inline* to *none* for the **hitButton** and **stayButton** variables, and *none* to *inline* for the **newGameButton**.
20. Cache the *winner-area* in a variable called **winnerArea** below the **outputArea** at the top of the file. Caching an element can improve performance because the browser doesn't have to look up the element in the DOM every time.
21. Add a parameter named **stayed** with the default value **false** to the **showHands** function. This parameter is assigned **true** when the player clicks the **Stay** button.
22. Add a variable named **winner** that stores the result from a call to the **determineWinner** function at the end of the **ShowHands** function. Pass in the **stayed** parameter to the **determineWinner** function.
23. Assign the value in the **winner** variable to the **winnerArea** variable.

24. Add an if-statement at the end of the **ShowHands** function that checks that the value of the **winner** variable isn't an empty string and calls the **hideGameButton** function if the expression is true.
25. Call the **hideGameButtons** function below the **dealerScore** variable at the top of the file to hide the **Hit!** and **Stay** buttons when the game starts.
26. Save the file and click the **Start Game!** button. Click the **Hit!** button until you are bust and display a message stating that the dealer won below the buttons.

11—Stay

Now, it's time to add a click event handler that makes it possible for the player to stay (stop drawing cards) and let the dealer draw his cards.

1. Add a **click** event listener for the **stayButton** variable below the other event handler at the top of the page.
2. Call the **hideGameButtons** function inside the event function to hide the **Hit!** and **Stay** buttons and display the **New Game!** button.
3. Add a while loop that iterates while the dealer score is less than the player score, and the player's score is less than or equal to 21, and the dealer's score is less than or equal to 21.
4. Add a new card to the **dealer** array and then call the **ShowHands** function with the value **true** as its parameter value inside the while loop.
5. Save the file and start the game. Click the **Hit!** button until you have a hand you like, and then click the **Stay** button to let the dealer draw cards. Play a few hands to see that the game works.

12—Obligatory Assignment—TODO

In this assignment, you create a TODO application where the user can store a list of TODO items locally in the browser's local storage and retrieve those items and display them when the user returns to the application.

The list below contains the HTML markup for the web page to save you some time.

01—Creating the Project

1. Create a new folder named **todo** for your application.
2. Add the three files: **todo.html**, **todo.js**, and **todo.css**.
3. Reference the JavaScript and CSS files in the HTML file.
4. Add the Bootstrap CDN to style the HTML easier.
5. Add the following HTML in the <body> element.

```
<h1 style="text-align: center;">TODO</h1>
<form id="todo-form">
    <div class="form-group row">
        <input class="form-control col-sm-8 offset-sm-1" type="text"
         id="todo-list-item" placeholder="Enter Todo">

        <button type="submit" class="btn btn-primary col-sm-2">
            Add Todo</button>
    </div>
</form>

<ul id="todo-items" class="col-sm-10 offset-sm-1">
        <!-- placeholder for Todo items -->
</ul>
```

02—Target the HTML Elements

In this part of the exercise, you add cache the main HTML element that the user will interact with.

1. Add a contant named **todoItems** that targets the <ul> element with the id **todo-items** where the to-do <li> element swill be displayed.
2. Add a contant named **todoForm** that targets the <form> element with the id **todo-form** where the textbox and button for adding new to-do items are housed.
3. Add a contant named **todoItem** that targets the <input> element with the id **todo-list-item** where the new to-do item's description is entered.

03—Submitting a Form

In this part of the exercise, you add JavaScript code to submit the form you created in the previous exercise. When the form is submitted, an alert is displayed to verify that it was submitted.

1. Use the **todoForm** element constant to add the **onsubmit** event to it with a *fat-arrow* function. This event will execute when the button in the form is clicked to submit the form's content.
2. To handle the submit process manually, you must add an **event** parameter to the **submit** function and call the **preventDefault** method on the **event** parameter at the beginning of the event function.
3. Log the message "Form Submitted" and the **event** parameter to the browser's console below the previous method call inside the **submit** function.
4. Save all files.

5. Open the application in the browser and click the button
6. Open the **Console** tab in the F12 tools.
7. Examine the event object.

04—Reading from the Textbox

In this part of the exercise, you use jQuery to read the value from the textbox and log the value to the Browser's **Console** tab.

1. Use the **todoItem** constant to read the text from the textbox and store it in a variable called **item**.
2. Comment out or delete the code logging the text to the console.
3. Use the **console.log** to display the text in the **item** variable.
4. Verify that the text is displayed.

05—Appending the Text to the TODO List

In this part of the exercise, you append the text from the variable you created earlier as an <li> to the <ul> that contains all TODO items.

1. Comment out or delete the code logging the text to the console.
2. Inside the if-block, use the **todoItems** constant to append an <li> with the text from the **item** variable to the <ul>.
1. After the <li> has been appended to the <ul> clear the textbox to prepare for another TODO item.
3. Remove or comment out the **console.log** call.
4. Save all files and refresh the browser.
5. Write some text in the textbox and click the button; the list below the textbox displays the text. Add a few more items to the list.

06—Adding HTML to the List Item

In this part of the exercise, you change the TODO list item to contain a checkbox followed by the item text and a link with "x" as the link text; this link removes the item from the list. When you add the item to the list, you also clear the textbox. You also add a CSS file to style the TODO item; Also, remove the list style decoration (the list item's dot) and display the link to the far right of the item row. There should also be a horizontal line separating each item on the TODO list.

2. Change the HTML code in the Append method to be a list item containing a checkbox followed by the item text and a link with the link text "x" and a class named *remove*. Add a horizontal line to the end of the list item after the link.
3. Use CSS to remove the list item's list-style decoration (the list item's dot).
4. Use CSS to add a 10px right and left margin to the checkbox to push it in and create space between it and the text.
5. Use CSS to Push the "x" link to the far right, give it a 10px right margin and change the mouse cursor to a pointer (the hand with a pointing finger).
6. Save all files and try the application in the browser.

07—Prevent Saving an Item Without Text

In this part of the exercise, you add conditional logic that ensures that the code won't add the item if the textbox is empty.

1. Add an if-block around the code that adds the item to the list and clears the textbox. The if-statement should check that the **item** variable isn't empty.

2. Save all files and try the application in the browser.

08—Checking if the Checkbox is Checked

In this part of the exercise, you use add a **change** event that checks the state of all the checkboxes dynamically. Checked items should have a strikethrough added with CSS.

1. Add a CSS selector named *completed* in the **todo.css** file; also, add the **line-through** text-decoration to it.
2. Use a *fat-arrow* function to add the **change** event to the **todoItems** constant to leverage the bubbling effect when changing the state of an HTML element inside the <li> elements you add to the <ul>.
3. Add an **event** parameter to the function.
4. Log the text "Checkbox Checked" and the **event** parameter from inside the function body.
5. Save all files and switch to the application in the browser.
6. Add a few TODO items to the list and try checking a few checkboxes. The text and **event** object should appear in the **Console** tab. Expand one of the **event** objects and inspect it.
7. Comment out or delete the logging code.
8. Target the checkbox's parent element (the surrounding list element) by using the **parentElement** property on the **event.target** object, then chain on the property and function that toggles the *completed* CSS selector (class).
9. Save all files and switch to the application in the browser.
10. Add a few TODO items to the list and try checking a few checkboxes. A completed (checked) item gets a strikethrough.

09—Dynamically Removing a TODO Item

In this part of the exercise, you add a **click** event that contains code to remove the item when you click its "x" link.

1. Use a *fat-arrow* function on the **todoItems** constant to add the **click** event that works with dynamically added buttons in the added TODO links.
2. Add an **event** parameter to the function.
3. Log the text "TODO Removed" and the **event** parameter from inside the function body.
4. Save all files and switch to the application in the browser.
5. Add a few TODO items to the list and click their links. The text should appear in the **Console** tab. Inspect the **event** object.
6. Comment out or delete the logging code.
7. Add a variable named **hasRemoveClass** and assign **true** or **false** depending on if the clicked element has the removed class in its class list.
8. Add an if-block that is executed if the **hasRemoveClass** variable contains the value **true**.
9. Inside the if-block, add a variable named **li** that targets the link's parent element (the surrounding list element) by using the **parentElement** property on the **event.target** object.
10. Use the **todoItems** constant to remove the element stored in the **li** variable.
11. Save all files and switch to the application in the browser.
12. Add a few TODO items to the list and click the "x" link to remove them.

10—Store and Retrieve Data in the Browser's Local Storage

In this part of the exercise, you store and retrieve the TODO items in the browser's Local Storage memory to persist the data even if the browser is closed and re-opened. JavaScript has a built-in object to handle the Local Storage named **localStorage**. The **GetItem** function fetches a stored item from the

storage, and the **SetItem** stores an item in the storage, or you can use the property name directly in the **localStorage** object. Store the data in a local storage memory area called **TodoItems**.

Because you want the items to be read from the storage when the browser refreshes, you need to place that code at the beginning of the JavaScript file (above all other functions but below the constants that target the HTML elements). The TODO list saves to the browser's Local Storage when the user adds a new item.

The Local Storage memory in the browser can be accessed by:

- Chrome: Open the **Application** tab in the F12 tools and expand the **Local Storage** node.
- Firefox: Open the **Storage** tab in the F12 tools and expand the **Local Storage** node.
- Edge: Open the **Debug** tab in the F12 tools and expand the **Local Storage** node.

1. Add an if-block at the beginning of the JavaScript file below the constants that target the HTML elements that checks that the **TodoItem** property in the Local Storage exists and isn't empty.
2. Inside the if-block, add the code to fetch and display the TODO items. Use the **localStorage** object's **GetItem** function or the property name **TodoItems** to fetch the items from the local storage memory and display them as HTML content using the **todoItems** constant to display them in the <ul>.
3. Locate the if-block in the **submit** function and write code that saves all TODO items as HTML to the **TodoItems** local storage memory.
4. Save all files and switch to the application in the browser.
5. Add a few TODO items and refresh the browser. The items are fetched from the Local Storage and are displayed.
6. Open the **TodoItems** Local Storage in the browser and verify that it contains the items.

11—Remove data from the Browser's Local Storage

In this part of the exercise, you remove data from the browser's Local Storage memory when the remove (x) link is clicked. You save the TODO list using the **localStorage** object's **SetItem** function.

1. Switch to the application in the browser.
2. Add a few TODO items to the Local Storage and refresh the browser if you haven't already.
3. Click the **remove** link (x) for an item and refresh the browser. The item remains in the list because it only was removed from the browser page, not from Local Storage.
4. Locate the event function that toggles the **remove** CSS class and write code to save the TODO items below the code where that removes the list item.
5. Save all files and switch to the application in the browser.
6. Click the **remove** link (x) for an item and refresh the browser. The item has been removed from Local Storage and is therefore not displayed on the TODO list.

12—Store Checked Items in the Browser's Local Storage

In this part of the exercise, you store whether items are checked to the browser's Local Storage memory. You can do this by adding/removing the **checked** attribute on the checkbox when its state changes and the **change** event triggers.

1. Switch to the application in the browser.
2. Check and uncheck a few TODO items and refresh the browser. The checkboxes are unchecked after the refresh because the code didn't save the **checked** attribute with the element to the browser's Local Storage.

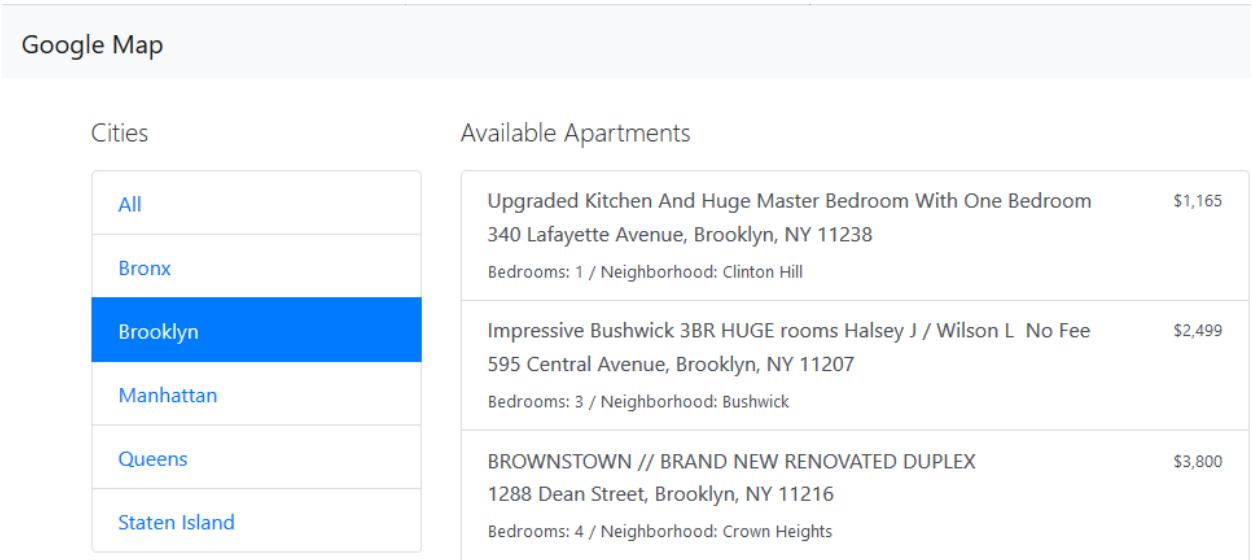
3. Locate the **change** event function and add an if-statement inside the already existing if-block that adds the **checked** attribute if the checkbox is checked and removes it otherwise.
4. Save the TODO list to the Local Storage at the end of the outer if-block below all other code.
5. Save all files and switch to the application in the browser.
6. Check a few checkboxes and refresh the browser. The checkboxes remain ticked after the refresh.
7. Uncheck a few checkboxes and refresh the browser. The checkboxes remain unticked after the refresh.

13—Obligatory Assignment—Google Maps

In this assignment, you call the **myjson** web service to fetch data about fictitious apartments based on the selected city and display them in a list. Use Axios to make the asynchronous call to the web service when the user clicks one of the listed cities and Bootstrap 4's **List Group** component to display the cities and apartments. The cities are hard coded in the HTML and not fetched with an Axios call. The apartment web service URL *https://api.myjson.com/bins/2sadq?pretty=1* returns all apartments, so you'll need to filter the result on a specific city when a user clicks it.

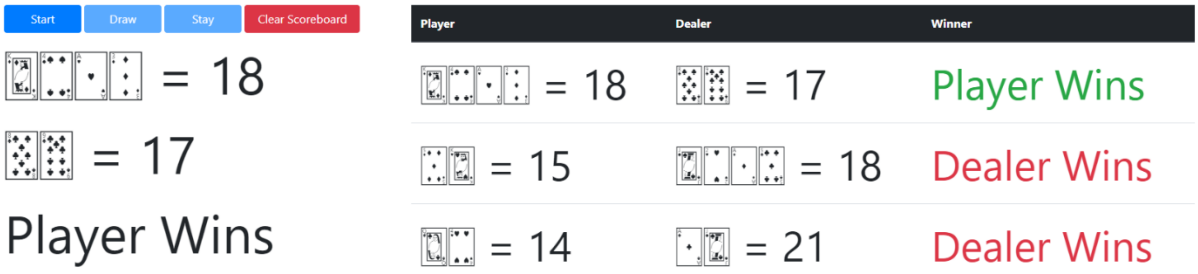
When a user clicks an apartment, display Google Maps in a separate browser window or tab. Use the address for the selected apartment to display the location on the map. Google Maps' web service URL is: *'http://maps.google.com?q=' + address*

An error message should be displayed if the Axios call fails.



14—Obligatory Assignment (Advanced)—Blackjack with Classes

In this assignment, you will build a Blackjack application using JavaScript classes. The UI should have buttons to start a new game, draw a card from the card deck, stay when the player doesn't want any more cards. There should also be a button for clearing the scoreboard that is stored in Local Storage.



01—Building the UI

1. Add a new folder for the application.
2. Add a HTML file named **blackjack.html** to the folder.
3. Add the HTML below to the file.
4. Add a JS file named **blackjack.js** to the folder.
5. Add <script> tag for the JS file in the HTML file.
6. Add a CDN link to the Bootstrap CSS library (not the JQuery library).
7. Add a folder named **Classes** to the application folder.
8. Save all files.

```
<div class="row mt-3">
    <div class="col-3 offset-1">
        <button id="start-button" class="btn btn-primary col-sm-2">Start
        </button>
        <button id="draw-button" disabled class="btn btn-primary
            col-sm-2">Draw</button>
        <button id="stay-button" disabled class="btn btn-primary
            col-sm-2">Stay</button>
        <button id="clear-scoreboard-button" class="btn btn-danger
            col-sm-3">Clear Scoreboard</button>

        <div id="player-content" style="font-size: 72px;"></div>
        <div id="dealer-content" style="font-size: 72px;"></div>
        <div id="result-content" style="font-size: 72px;"></div>
        <hr />
        <div id="deck-content" style="font-size: 72px;"
             class="col-sm-10"></div>
    </div>
    <div class="col-5">
        <table class="table">
            <thead class="thead-dark">
                <th scope="col">Player</th>
                <th scope="col">Dealer</th>
                <th scope="col">Winner</th>
            </thead>
            <tbody id="scoreboard-content" style="font-size: 57px;"></tbody>
        </table>
    </div>
</div>
```

02—Caching HTML Elements

In this exercise, you will add constants that caches the buttons and other HTML elements that are needed to run the application.

1. Open the JavaScript file.

2. Add a constant named **startButton** that caches the button with id **start-button**.
3. Add a constant named **drawButton** that caches the button with id **draw-button**.
4. Add a constant named **stayButton** that caches the button with id **stay-button**.
5. Add a constant named **clearButton** that caches the button with id **clear-scoreboard-button**.
6. Add a constant named **playerContent** that caches the with id **player-content**.
7. Add a constant named **dealerContent** that caches the with id **dealer-content**.
8. Add a constant named **resultContent** that caches the with id **result-content**.
9. Add a constant named **deckContent** that caches the with id **deck-content**.
10. Add a constant named **scoreboardContent** that caches the <tbody> with id **scoreboard-content**.
11. Save all files.

03—Adding Button Events

In this exercise, you will use *fat-arrow* functions to add **click** events to the buttons, and a function named **clearTable** that clears all content elements. Also, add a unction named **disableButtons** that enables/disables the buttons based on the values passed into the function.

1. In the JavaScript file, add a *fat-arrow* parameter-less function named **clearTable**.
2. Inside the function, clear the HTML of all content elements.
3. Use a *fat-arrow* function to add a function named **disableButtons** with three parameters named **start**, **draw**, **stay** that have the default values **false**, **true**, **true**.
4. Inside the function, add three if/else-blocks that checks each of the three parameters and adds/removes the **disabled** attribute from respective button.
5. Use a *fat-arrow* function to add a click event for the **start** button.
6. Inside the event, write "Start button clicked" to the console window.
7. Use a *fat-arrow* function to add a **click** event for the **draw** button.
8. Inside the event, write "Draw button clicked" to the console window.
9. Use a *fat-arrow* function to add a **click** event for the **stay** button.
10. Inside the event, write "Stay button clicked" to the console window.
11. Use a *fat-arrow* function to add a **click** event for the **clear** button.
12. Inside the event, write "Clear button clicked" to the console window.
13. Inside the **startButton click** event, call the **clearTable** function to clear the table.
14. Inside the **startButton click** event, call the **disableButtons** function with the parameter values to **true**, **false**, **false** to remove the **disable** attribute from the **stay** and **draw** buttons.
15. Inside the **stayButton click** event, call the **disableButtons** function without any parameter values to add the **disable** attribute to the **draw** and **stay** buttons and remove it from the **start** button.
16. Save all files and start the application.
17. Click the buttons and make sure that their respective messages are displayed in the browser's Console window and that the **stay** and **draw** buttons are disabled when the **stay** button is clicked.

04—Adding the Table Class

In this exercise, you will add a class named **Table** that defines the functionality for the Blackjack table where the game is played. All functionality needed for the UI is in this class; you therefore only need to create an instance of this class in the **blackjack.js** file to play the game.

You task right now is to create the skeleton for this class that you will add functionality to as you implement the solution.

1. Add a class named **Table** to a JavaScript file with the same name in the **Classes** folder.
2. Add a constructor that initializes three fields named **_player**, **_dealer**, and **_deck** to **null**; these fields are assigned objects later for a **player**, a **dealer**, and the **deck** of cards used when playing the game.
3. Also initialize three fields named **_dealerWins**, **_playerWins**, and **_draw** with the values *Dealer Wins*, *Player Wins*, and *Draw* respectively.
4. Add a function named **newGame** that logs *New game function* to the Console window.
5. Add a <script> tag for the **table.js** file to the HTML file.
6. Add a constant named **table** with the other constants in the **blackjack.js** file and assign an instance of the **Table** class to it.
7. At the end of the **startButton click** event, call the **newGame** function on the **table** constant.
8. Start the application and click the **Start** button.
9. Open the browser's Console window and make sure that the text *New game function* is displayed from the call to the **newGame** function on the **table** object.

05—Adding the Card Class

In this exercise, you will add a class named **Card** that represent a blueprint for a single playing card; it must be able to store the card value and symbol. These values are stored in fields named **_value** and **_symbol** that are exposed outside the class with **get** properties.

The car must also be able to tell the onlooker if it is an ace because that is essential when calculating the total for the player's and dealer's hands.

1. Add a class named **Card** to a JavaScript file with the same name in the **Classes** folder.
2. Add a constructor that has two parameters named **value** and **symbol** that will receive the card's value and symbol.
3. Inside the constructor, initialize two fields named **_value** and **_symbol** and assign the respective parameter to them.
4. Add two **get** properties named **value** and **symbol** that returns the values from the **_value** and **_symbol** fields respectively.
5. Add a **get** property named **isAce** that returns **true** if the card's value is equal to 1 (it is an ace.)
6. Add a <script> tag for the **card.js** file above the **table.js** <script> tag in the HTML file.
7. Save all files.

06—Adding the Deck Class

In this exercise, you will add a class named **Deck** that represents a blueprint for a deck of cards; it must be able to store the 52 initial cards in an array field. The card array is exposed with a **get** property named **cards**.

You will add functionality to this class in the upcoming exercises.

1. Add a class named **Deck** to a JavaScript file with the same name in the **Classes** folder.
2. Add a parameters-less constructor that initializes an array field named **_cards** with an empty array.
3. Add a **get** property named **cards** that returns the **_cards** array.
4. Add a <script> tag for the **deck.js** file between the **card.js** and **table.js** <script> tags in the HTML file.
5. Save all files.

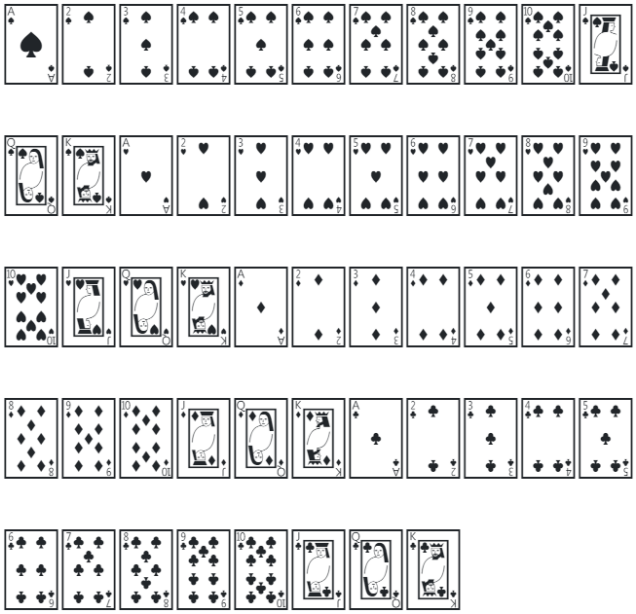
07—Adding the newDeck Function

In this exercise, you will add a function named **newDeck** to the **Card** class; the purpose of this function is to generate 52 cards using Unicode playing cards characters and a value from 1 to 10 depending on the card. You can find the Unicode characters here https://altcodeunicode.com/alt-codes-playing-card-symbols.

The cards between 2-9 has the same card value, and all other cards have the value 10, except for the ace which has the value 1 (or 11).

Add one new card for each suit (hearts, spades, diamonds, and clubs) and value combination using two nested loops, where one iterates over the four suits, and the other 0-13; add the card to the **_cards** array in the **Deck** class.

Add another function named **displayRemainingCards** that creates a concatenated string with the card symbols from all the cards using the **cards** property on the **_deck** instance. Call this function from the **startButton click** event. The result should be an unsorted deck of 52 cards.



1. Add a parameter-less function named **newDeck** to the **Deck** class.
2. Inside the function, clear the **_cards** array.
3. Using the Unicode decimal values for the cards and loops, create the 52 cards in the deck.
    a. Note that there is an extra Knight card for each suit that you need to skip (it represents another symbol for a Jack).
    b. Note that there is a gap of 2 symbols between each suit that you need to handle.
4. Open the **Table** class and locate the **newGame** function.
5. Inside the function, create an instance of the **Deck** class that you assign to the **_deck** field.
6. Below the **Deck** instance, call the **newDeck** function on the **_deck** field.
7. Below the **newGame** function, add a new parameter-less function named **displayRemainingCards**.
8. Inside the function, that returns a concatenated string with the card symbols from all the cards in the deck using the **cards** property on the **_deck** instance.
9. Open the **blackjack.js** file.

10. Add a *fat-arrow* function named **displayOutput** that takes two parameters named **htmlContainer** and **output**; the former will receive the HTML element that the content in the **output** parameter will be displayed.
11. Inside the function, assign the value in the **output** parameter to the **innerHTML** property of the **htmlContainer** element.
12. Locate the **startButton click** event.
13. Below the call to the **table.newDeck** function, call the **displayOutput** function and pass in the **deckContent** constant (that points to the with the id **deck-content**) and the returned cards from a call to the **table.displayRemainingCards** function.
14. Save all files.
15. Start the application and click the **Start** button to see the cards.

08—Adding the shuffleCards Function

In this exercise, you will add a function named **shuffleCards** to the **Deck** class that will randomize the cards in the **_cards** array.

Use the **slice** function to copy the cards in the **_cards** array to a variable named **tmpCards** and clear the **_cards** array; you do this to be able to cut out one card at a random position in the **tmpCards** array using the **splice** function and push it into the now empty **_cards** array. When the process has been performed for all cards in the **tmpCards** array, the **_cards** array will contain a shuffled deck of cards.

1. Open the **Deck** class and add a parameter-less function named **shuffleCards**.
2. Inside the function, copy the **_cards** array into a temporary array named **tmpCards** using the **slice** function.
3. Empty the **_cards** array so that it can hold the randomized (shuffled) cards.
4. Add a while loop that iterates while there are cards remaining in the **tmpCards** array.
5. Inside the while loop, calculate a random number between 1 and the number of remaining cards in the **tmpCards** array and store it in a variable named **pos**.
6. Use the **splice** function to cut out the card at the position in the **tmpCards** array corresponding to the random value in the **pos** variable; store the card in a variable named card.
7. Add the randomly picked card to the **_cards** array. Note that the **splice** function returns an array, so you need to convert it into a card.
8. Open the **Table** class and locate the **newGame** function.
9. Below the **_deck.newDeck** function call, call the **shuffleCards** function on the **_deck** object.
10. Save all files and start the application.
11. Click the **Start** button to display the shuffled cards.

09—Adding the drawCard Function

In this exercise, you will add a function named **drawCard** to the **Deck** class that removes the first card in the **_cards** array and returns it (you can use the **shift** function).

1. Open the **Deck** class and add a parameter-less function named **drawCard**.
2. Inside the function, use the **shift** function to return the first card in the **_cards** array.
3. Save all files and start the application.

10—Adding the Result Class

In this exercise, you will add a class named **Result** that represents a blueprint for the result of a player's hand; it is used inside the **Hand** class that you will add next.

1. Add a class named **Result** to a JavaScript file with the same name in the **Classes** folder.
2. Add a parameters-less constructor that:
    a. Initializes a field named **_score** with the value 0.
    b. Initializes an array field named **_cards** with an empty array.
    c. Initializes a field named **_hasAce** with the value **false**.
    d. Initializes a field named **_cardAdded** with the value **false**.
3. Add a **get** property named **cards** that returns the **_cards** array.
4. Add a **get** property named **score** that returns the value from the **_score** field.
5. Add a **get** property named **isBust** that returns **true** if the value in the **_score** field is greater than 21.
6. Add a **get** property named **isBlackjack** that returns **true** if the value in the **_score** field is equal to 21 and there are exactly 2 cards in the **_cards** array.
7. Add a **get** property named **hasAce** that returns the value from the **_hasAce** field.
8. Add **get** and **set** properties named **cardAdded** that returns and assigns the value for the **_cardAdded** field.
9. Add a function named **caculateScore** that has a parameter named **cards** which receives the cards to calculate the result for.
10. Inside the function:
    a. Assign the cards from the **cards** parameter to the **_cards** array.
    b. Assign 0 to the **_score** field.
    c. Iterate over all the cards in the **_cards** array. For each card, add its value to the **_score** field and assign **true** to the **_hasAce** field if the card is an ace.
    d. Below the loop, add an if-statement that checks if the result has an ace and the score + 10 is less than; this means that the hand has an ace that should be counted as 11 instead of its default value. Add 10 to the **_score** field if the if-expression evaluates to **true**.
11. Add a <script> tag for the **result.js** file immediately below the **card.js** <script> tag in the HTML file.
12. Save all files.

11—Adding the Hand Class

In this exercise, you will add a class named **Hand** that represents a blueprint for a player's hand; it must have an array named **_cards** for the player's cards, have a field named **_stay** (and its corresponding **get** and **set** property) that keeps track of if the player is satisfied and doesn't want any more cards, and a field named **_result** (and its corresponding **get** property) that reports back the result of the player's hand; it should be an instance of the **Result** class.

It should also have a function named **addCard** that adds a card to the **_cards** array from the deck that is passed into it as a parameter if the **stay** property is **false**.

1. Add a class named **Hand** to a JavaScript file with the same name in the **Classes** folder.
2. Add a parameters-less constructor that initializes:
    a. An array field named **_cards** with an empty array.
    b. A field named **_stay** with the value false.
    c. A field named **_result** with an instance of the **Result** class.

3. Add a **get** property named **stay** that returns **true** if the **_stay** field is **true**, or the **isBust** property in the **_result** object is **true**, or the **isBlackjack** property in the **_result** object is **true**.
4. Add a **set** property named **stay** that assigns the value passed to it to the **_stay** field.
5. Add a **get** property named **result** that returns the **Result** object in the **_result** field.
6. Add a function named **addCard** that has a parameter named **cards**, which receives the card deck.
7. Inside the function:
    a. Assign **false** to the **cardAdded** property of the **_result** object to specify that the card hasn't been added yet.
    b. Add an if-block that checks that the **stay** property in the **_result** object isn't **false**, meaning that the card can be added to the player's hand.
    c. Inside the if-block:
        i. Call the **drawCard** method on the **deck** parameter passed into the function and store it in a variable named **card**.
        ii. Add the card to the **_cards** array.
        iii. Calculate the score for the hand by calling the **calculateScore** function on the **_result** object. Pass in the **_cards** array to the function.
        iv. Assign **true** to the **cardAdded** property of the **_result** object to specify that the card has been added.
8. Open the **Table** class and locate the **newGame** function.
9. Inside the function immediately below the code creating an instance of the **Deck** class, assign an instance of the **Hand** class to the **_player** field.
10. Add a <script> tag for the **hand.js** file immediately below the **result.js** <script> tag in the HTML file.
11. Save all files.

12—Adding the drawPlayerCard and getPlayerOutput Function

In this exercise, you will add a function named **drawPlayerCard** to the **Table** class that adds a card to the player's **_cards** array. Call the function twice at the end of the **newGame** function to assign the player's two initial cards. Also, add a function named **getPlayerOutput** that will return the player's card symbols and score as a string; call the function from the **blackjack.js** file.

1. Open the **Table** class and add a parameter-less function named **drawPlayerCard**.
2. Inside the function, call the **addCard** on the **_player** object and pass in the **_deck** array as its parameter.
3. Call the **drawPlayerCard** function twice at the end of the **newGame** function to assign the player's two initial cards.
4. Inside the **Table** class add a parameter-less function named **getPlayerOutput**.
5. Inside the function:
    a. Create a string with all the player's card symbols by iterating over the cards.
    b. Append the score to the string to display the value of the cards.
6. Open the **blackjack.js** file and locate the **startButton click** event.
7. Inside the event below the call to the **displayOutput** function, add another call to the call to the **displayOutput** function and pass in the **playerContent** constant—that points to the with the id **player-content**—and a call to the **getPlayerOutput** on the **table** object.
8. Save all files and start the application.
9. Click the **Start** button. The player's cards should be displayed.

13—Draw a new Player Card

In this exercise, you will call the **drawPlayerCard** and **displayOutput** functions from the **drawButton click** event in the **blackjack.js** file to give the player a new card each time the **Draw** button is clicked.

1. Open the **blackjack.js** file and locate the **drawButton click** event.
2. Inside the event, call the **drawPlayerCard** function on the **table** object to give the player another card.
3. Copy the two **displayOutput** function calls from the **startButton click** event and paste them in at the end of the **drawButton click** event.
4. Save all files and start the application.
5. Click the **Start** button. The player's initial two cards should be displayed.
6. Click the **Draw** button a couple of times; the player should receive one new card for each click of the button and the card should be removed from the displayed shuffled deck. Note that the score is calculated as the new cards are added to the player.

14—Adding the DealerHand Class

In this exercise, you will add a class named **DealerHand** that represents a blueprint for the dealer's hand. It should inherit (extend) the Hand class because it is a specialized hand that should contain all the same functionality; it should however override the **stay get** and **set** property functions because the rules for when the dealer must stop taking cards is different from a player hand.

Is should stop taking cards if:

- The **stay** property in the inherited class is **true**.
- The dealer score is greater than 16.
- The dealer score is less than or equal to 21 and the dealer has five of more cards.

1. Add a class named **DealerHand** to a JavaScript file with the same name in the **Classes** folder.
2. Use the **extend** keyword to inherit the **Hand** class.
3. Add a constructor that calls the inherited class' constructor.
4. Override the **stay get** property function and implement the restrictions described by the three bullets above.
5. Override the **stay set** property function and store the passed-in value in the inherited class'**_stay** field.
6. Add a <script> tag for the **dealerHand.js** file immediately below the **hand.js** <script> tag in the HTML file.
7. Save all files.

15—Draw and Display the Dealer Cards

In this exercise, you will add functions to the **Table** class that draws the dealer's cards and displays them in the browser.

1. Open the **Table** class and locate the **newGame** function.
2. Inside the function below the creation of the **Hand** instance, create an instance of the **DealerHand** class and assign it to the **_dealer** field.
3. Add a parameter-less function named **drawDealerCard** that draws a card from the deck and assigns it to the dealer.
4. Call the **drawDealerCard** function twice at the end of the **newGame** function to assign the dealer's two initial cards.

5. Add a parameter-less function named **drawDealerCards** that draws new cards for the dealer while the dealer's **stay** property is **false**.
6. At the end of the **stayButton click** event, call the **drawDealerCards** on the **table** object to draw the remaining dealer cards when the **Stay** button is clicked.
7. Add a parameter-less function named **getDealerOutput**.
8. Inside the function:
    a. Create a string with all the dealer's card symbols by iterating over the cards.
    b. Append the score to the string to display the value of the cards.
9. Open the **blackjack.js** file and locate the **startButton click** event.
10. At the end of the event, add another call to the call to the **displayOutput** function and pass in the **dealerContent** constant—that points to the with the id **dealer-content**—and a call to the **getDealerOutput** on the **table** object.
11. Copy the **displayOutput** function call and paste it in at the end of the **stayButton click** event.
12. Save all files and start the application.
13. Click the **Start** button. The player and dealer cards should be displayed.
14. Click the **Draw** button to draw new cards until you are satisfied with the player's hand.
15. Click the **Stay** button when you are satisfied with the player's hand. The dealer's remaining cards should be drawn from the deck until the **stay** property of the **DealerHand** instance returns **true**.
16. Save all files.

16—Displaying the Winner

In this exercise, you will add a function named **findWinner** with a parameter named **playerStayed** that determines if the player has stayed (with the default value **false**) to the **Table** class; it contains the logic for determining the winner. You will also add functions to the **blackjack.js** file to display the winner.

The rules in the order they should be implemented:

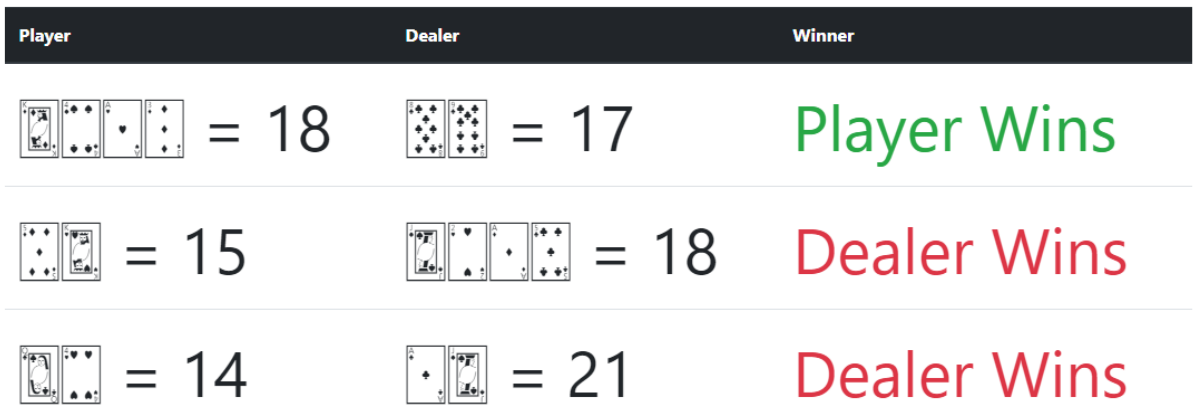
- The dealer wins if the player is bust.
- The player wins if the dealer is bust.
- The player wins if he has Blackjack.
- The dealer wins if he has Blackjack.
- The dealer wins if the player has stayed and the dealer's score is greater than the player's score.
- The player wins if the player has stayed and the player's score is greater than the dealer's score.
- It's a draw if the player has stayed and the player's and dealers score are equal.
- Return an empty string if none of the conditions above apply.

1. Open the **Table** class and a new function named **findWinner** with a parameter named **playerStayed** that determines if the player has stayed (with the default value **false**).
2. Inside the function:
    a. Store the player's **result** object in a variable named **pr**.
    b. Store the dealer's **result** object in a variable named **dr**.
    c. Assign the value in the **playerStayed** variable to the player's **stay** property if the value is **true**.
    d. Return value of the **_dealerWins** constant if the player is bust.
    e. Return value of the **_playerWins** constant if the dealer is bust.

   f. Return value of the **_playerWins** constant if the player has Blackjack.
   g. Return value of the **_dealerWins** constant if the dealer has Blackjack.
   h. Return value of the **_dealerWins** constant if the player has stayed and the dealer's score is greater than the player's score.
   i. Return value of the **_playerWins** constant if the player has stayed and the player's score is greater than the dealer's score.
   j. Return value of the **_draw** constant if the player has stayed and the player's score is equal to the dealer's score.
   k. Return an empty string if none of the conditions have been executed.
3. Open the **blackjack.js** file and a new function named **displayWinner** with a parameter named **playerStayed** that determines if the player has stayed (with the default value **false**).
4. Inside the function:
   a. Call the **findWinner** function on the **table** object and pass in the **playerStayed** parameter. Store the result in a variable named **winner**.
   b. Call the **displayOutput** function and pass in the **resultContent** constant—which points to the with the id **result-content**—as its first parameter, and the **winner** variable as its second parameter.
   c. If the **winner** variable contains a result other than an empty string, call the **disableButtons** function without any parameter values—to use its default parameter values—to reset the buttons for a new game.
   d. Return the result in the **winner** variable.
5. Call the **displayWinner** function without a parameter value—to use its default **false** parameter value, which means that the player hasn't stayed yet— at the end of the **startButton** and **drawButton click** events.
6. Call the **displayWinner** function with **true** as its parameter value—which means that the player has stayed—at the end of the **stayButton click** event.
7. Save all files and start the application.
8. Click the **Start** button. The player and dealer cards should be displayed.
9. Click the **Draw** button to draw new cards until you are satisfied with the player's hand.
10. Click the **Stay** button when you are satisfied with the player's hand. The dealer's remaining cards should be drawn from the deck until the **stay** property of the **DealerHand** instance returns **true**.

17—Updating the Scoreboard

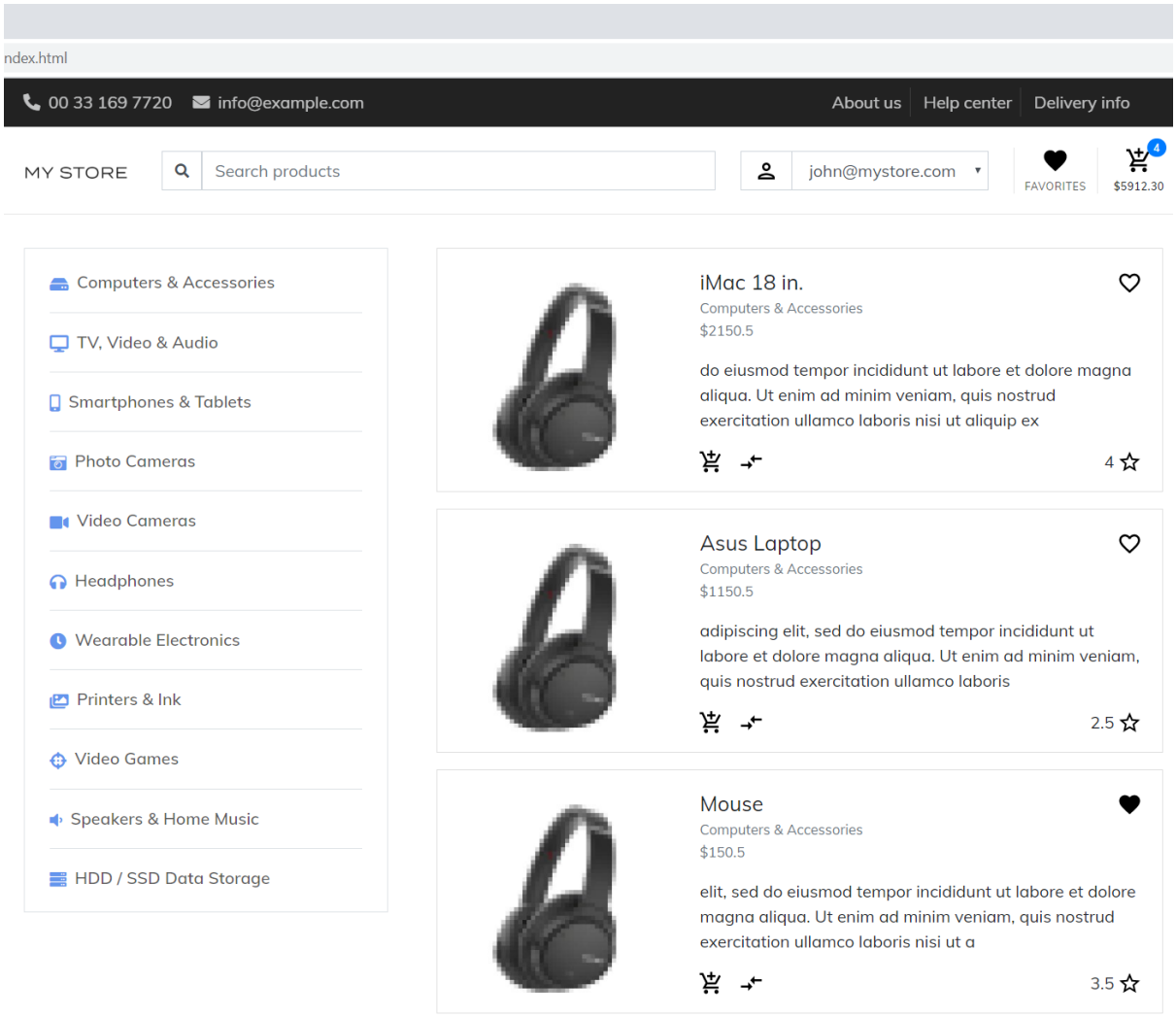
In this exercise, you will add a class named **Scoreboard** that represents a blueprint for the scoreboard on the right-hand side of the page.



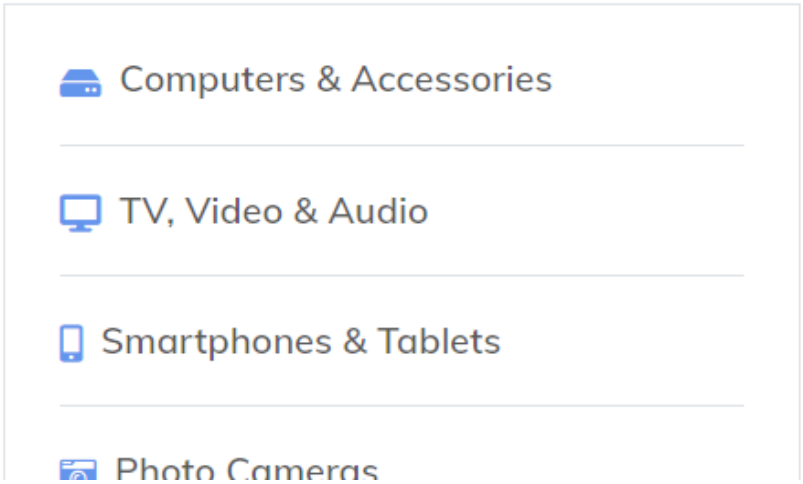
1. Add a class named **Scoreboard** to a JavaScript file with the same name in the **Classes** folder.
2. Add a constructor that has a parameter named **scoreboard** that receives the <tbody> element where the played hands and winners are displayed.
3. Store the **scoreboard** parameter in a field named **_scoreboard**.
4. If a Local Storage property named **scoreboard** exist and its value isn't **undefined**, then display its content in the <tbody>.
5. Add a function named **addScore** that adds a new score to the scoreboard; it needs to receive the **player**, **dealer**, and **winner** information.
6. Add a variable named **color** that is assigned the Bootstrap class **text-danger** if the dealer has won, **text-success** if the player has won, and **text-warning** if it's a draw.
7. Add a variable named **td** that represent the cells of the new row in the <tbody> of the scoreboard table.
    a. The first cell should display the player's cards and score (see image above).
    b. The second cell should display the dealer's cards and score (see image above).
    c. The third cell should display the winner (see image above).
8. Add a variable named **scores** and assign it the HTML from the scoreboard.
9. Add the **td** and **scores** variable values to the scoreboard using a backtick string.
10. Store the HTML from the scoreboard in the Local Storage **scoreboard** property to retain the information even if the browser is closed.
11. Add a function parameter-less named **clearScoreboard**.
    a. Remove the Local Storage **scoreboard** property.
    b. Clear the HTML in the scoreboard's <tbody>.
12. Store the <tbody> element's HTML in a Local Storage property named **scoreboard**.
13. Add a <script> tag for the **scoreboard.js** file immediately above the **blackjack.js** <script> tag in the HTML file.
14. Open the **blackjack.js** file and locate the code that assigns the **Table** class instance to the **table** variable and add a variable named **scoreboard** below it.
    a. Assign an instance of the **Scoreboard** class to the **scoreboard** variable.
    b. Pass in the **scoreboardContent** constant—that points to the <tbody> of the scoreboard <table> element—to the constructor.
15. Locate the if-block that checks if the **winner** variable contains a value inside the **displayWinner** function.
    a. Call the **addScore** function on the **scoreboard** object.
    b. Pass in the player's output, the dealer's output, and the value from the **winner** variable.
16. Inside the **clearButton** click event, clear the scoreboard.
17. Save all files and start the application.
18. Click the **Start** button. The player and dealer cards should be displayed.
19. Click the **Draw** button to draw new cards until you are satisfied with the player's hand.
20. Click the **Stay** button when you are satisfied with the player's hand. The dealer's remaining cards should be drawn from the deck until the **stay** property of the **DealerHand** instance returns **true**.
21. The hands and the winner should be displayed in the table on the right-hand side.
22. Click the **Clear Scoreboard** button to remove all played hands from the scoreboard table.

15—Obligatory Assignment (Advanced)—Web Shop

In this assignment, you add asynchronous calls to an in-memory API to fetch the necessaey data to display product data for a fictitious web shop. All HTML, CSS, and images are provided; your task is ti add the necessary JavaScript to make calls to the API and add HTML dynamically using the response from the API calls.



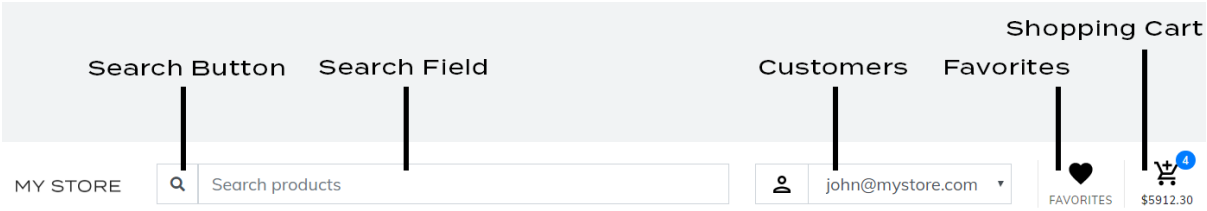
Categories



Product

Product with discount

The toolbar

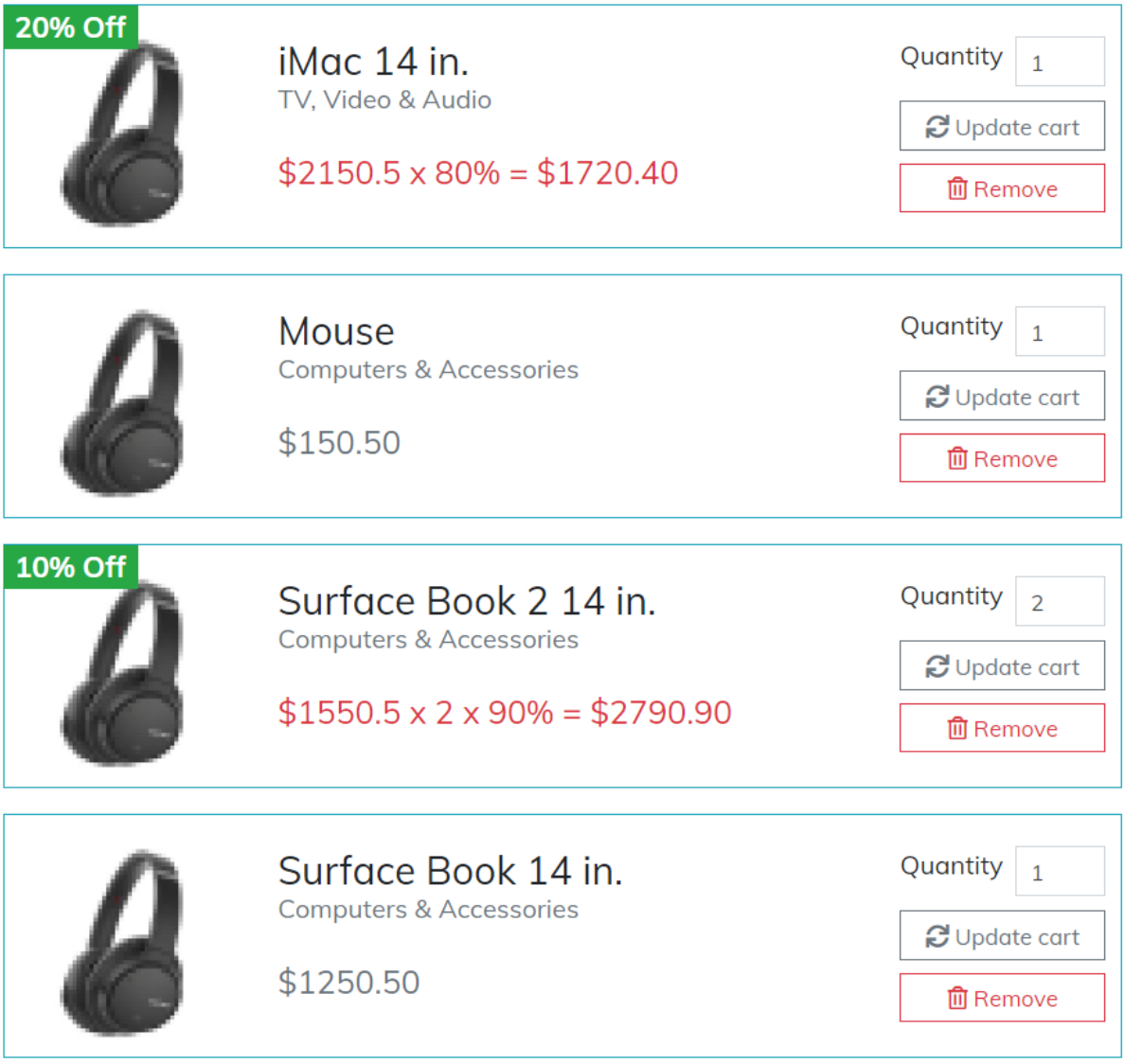


When a user enters a search term in the search field and either clicks the magnifier button or presses enter on the keyboard, the products matching the search criteria should be displayed in the products section.

When a new user is selected in the drop-down the shopping cart information for that user should be displayed by the shopping cart icon.

If the Favorites button is clicked, the favorites for the selected user should be displayed in the products section. All favorites should have a black heart.

If the shopping card button is clicked, the items for the selected user should be displayed in the products section as a "shopping cart product". Note that there are four different "shopping cart products" and a total below the last item.



API Endpoints (URLs)

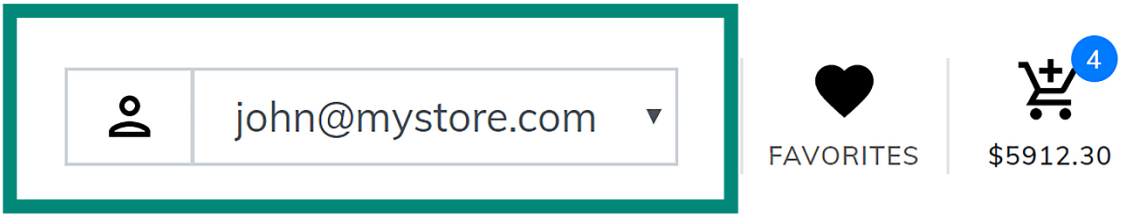
The value in square brackets is the HTTP verb (Axios function) to be used with that URL.

To make it easier for you, the base URL is provided in the **baseUrl** constant in the **constants.js** file; you can append the necessary endpoint (destination) and its parameters.

- Get all favorites for a customer:
  [get] *https://localhost:5001/api/favorites?customerId=1*
- Get all product categories:
  [get] *https://localhost:5001/api/categories*
- Get all products for a given category and customer:
  [get] *https://localhost:5001/api/categories/1/products?customerId=1*
- Get all customers:
  [get] *https://localhost:5001/api/customers*
- Search for products:
  [get] *https://localhost:5001/api/products/abc?customerId=1*
- Get shopping cart info for the button:
  [get] *https://localhost:5001/api/BasicShoppingCart?customerId=1*
- Get full shopping cart info:
  [get] *https://localhost:5001/api/ShoppingCartProducts?customerId=1*
- Remove a product from the customer's shopping cart:
  [delete] *https://localhost:5001/api/ShoppingCartProducts?customerId=1&productId=2*
- Add a favorite to the customer's favorites when the heart is clicked
  [put] *https://localhost:5001/api/favorites?productId=2&customerId=1&returnData=3&categoryId=2&searchTerm=abc*
  Does not need any data in the body because all data is provided by the parameters.
- Update a product in the shopping cart when the **Update** button is clicked:
  [put] *https://localhost:5001/api/ShoppingCartProducts*
  Needs a shopping cart product in the body, which can be fetched from the **CartItems** property in Local Storage with the **productId** passed into the function.
- Add a product to the shopping cart:
  [post] *https://localhost:5001/api/ShoppingCartProducts*
  Needs the following data in the body: *customerId: parseInt(customerId), productId: parseInt(productId), count: 1*.

01—Fetch the Customers

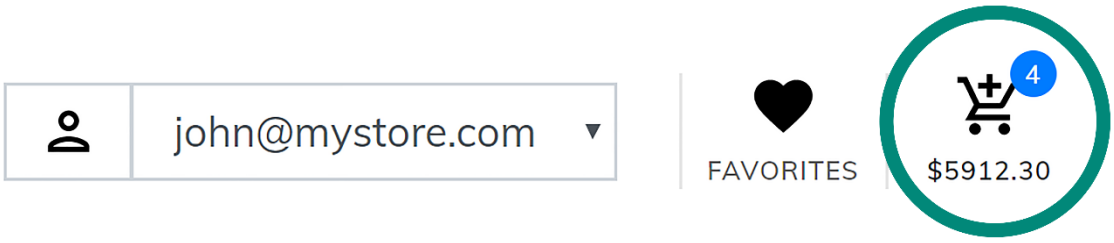
To display the customers in the drop-down, you call the in-memory API to fetch them. Then you need to use the returned JSON data to create dynamic HTML that is displayed in the drop-down. You find the current hard-coded HTML at the bottom of the **nav-middle.html** file; replace the <option> elements with elements that you create dynamically.



1. Only execute the code you add inside the **getCustomers** function's try-block in the **login.js** file if the customers haven't already been loaded (check the **customersLoaded** variable).
2. Fetch the drop-down HTML element.
3. If the element exists, set the **customersLoaded** variable to **true** to specify that the customers have been fetched.
4. Call the API endpoint to fetch all customers and store the returned data in a variable. Note that the function is asynchronous. *https://localhost:5001/api/customers*; use the **baseUrl** constant to avoid hard coding the base URL with every API call.
5. Iterate over all the returned customers and add them as options to the drop-down.
6. Store the first customer's id in a Local Storage property named **CustomerId** for easy reuse; you do this because the first customer in the drop-down is automatically selected.
7. Call the **basicCart** function that displays the customer's shopping cart information on the shopping cart button. You will implement that function next.
8. Start the application by right-clicking on the **index.html** file and select **Open with Live Server**.
9. The email addresses for the customers john and jane should be displayed in the drop-down.

02—Fetch the Selected Customer's Basic Shopping Cart Information

To display the customer's shopping cart information, you called the **basicCart** function from the **getCustomers** function in the previous exercise. Now, it's time to call the in-memory API to fetch the necessary data for the shopping cart button. Use the returned JSON data to update any element with the class **cart-total** with the total amount returned; note that it is a class because the total can be displayed in several places at the same time, not only on the button. Also, display the number of products in the cart within the cart's blue circle.

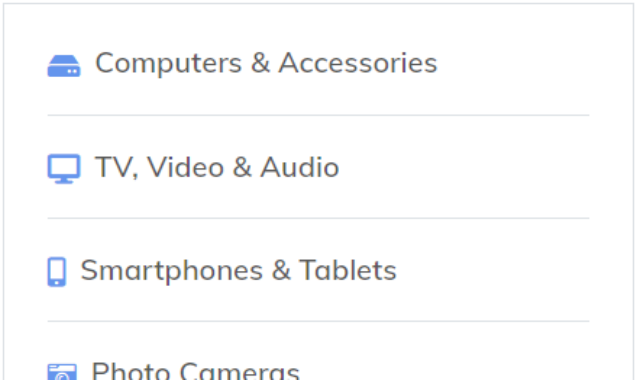


1. Let's begin by implementing the **displayBasicCart** function in the **shoppingCart.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Store the **cart-count** element and the elements with the **cart-total** class in two variables.
3. If the HTML elements exists, then update the cart count with the value from the **count** property in the **cart** object passed into the function.
4. Use a loop to update all the elements that has the **cart-total** class with the value from the **total** property in the **cart** object rounded to two decimals.
5. Store the value from the **total** property in the **cart** object rounded to two decimals in a Local Storage property named **CartTotal** for easy access.

6. Assign **true** to the **basicCartLoaded** variable to indicate that it has been successfully displayed; you will use this value in the **basicCart** function you will implement later in this exercise.
7. Open the storage.js file and implement the **localStorageKeyExist** function. It should return **true** if the Local Storage property with the name passed in through the **key** parameter exists.
8. Go back to the **shoppingCart.js** file.
9. Inside the try-block of the **basicCart** function, call the API to fetch the customer's basic shopping cart information asynchronously only if the **basicCartLoaded** variable is **false** (the information hasn't been fetched yet) and there is a customer id in the **CustomerId** Local Storage property (use the **localStorageKeyExist** function you just implemented). *https://localhost:5001/api/BasicShoppingCart?customerId=1*
10. Call the **displayBasicCart** function and pass it the returned data from the API.
11. Start the application.
12. The total and number of products should be displayed on the shopping cart button.

03—Fetch the Product Categories

To display the product categories, you call the in-memory API to fetch them. Then you use the returned JSON data to create dynamic HTML that is displayed in the **categories** . You find the current hard-coded HTML in the **home-categories.html** file; replace the <li> elements with elements that you create dynamically.



1. Let's begin by implementing the asynchronous **loadCategories** function in the **loadCategories.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Only execute the code inside the try-block if the <li> elements haven't already been loaded (the **categoryUlLoaded** variable is **false**).
3. Fetch the **categories** and cache it in a variable.
4. Only execute the rest of the code if the <ul> element could be found.
5. Assign **true** to the **categoryUlLoaded** variable to indicate that the <li> elements have been fetched.
6. Call the API to fetch the categories. *https://localhost:5001/api/categories*.
7. Create one <li> element for each category and make sure that the last element doesn't have bottom border. Also add an attribute for the category id to each <li> element.
8. To activate the **click** event to the dynamically added <li> elements, you need to add the **ulOnClick** event function to the <ul>, not the individual <li> elements. You will implement the **ulOnClick** event function in a later exercise.
9. Start the application.
10. The product categories should be displayed.

04a—Create a Product Card

To display a product, you need to create a product card with dynamic HTML in the **createProductCard** function in the **loadCategoryProduct.js** file. Call this function for each product you fetch with the in-memory API in the **loadCategoryProduct** function, which you implement in the next exercise.

1. Let's begin by implementing the **createProductCard** function in the **loadCategoryProduct.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Add a variable named **heart** that holds the name of the heart image to use for the product card by checking the **isFavorite** property on the **product** object that is passed into the function; display a black heart if the property is **true**, otherwise display the heart outline.
3. Use the **product** object's **imageUrl** property to display the product image.
4. Use the **product** object's **name** property where the product title is displayed or used.
5. Use the **product** object's **percentOff** property to display the product's discount if any; only display the green square if the product has a discount.
6. Use the **product** object's **category** property to display the product's category.
7. Use the **product** object's **price** property to display the product's price.
8. Use the **product** object's **description** property to display the product's description.
9. Use the **product** object's **id** property where the product's id is used.
10. Use the **product** object's **heart** variable to display the correct heart image for the product.
11. Use the **product** object's **stars** property to display the product's review score.
12. Return the HTML from the function.

04b—Display the Products for the Selected Category

To display the products for a selected category, you call the in-memory API in the **loadCategory-Product** function to fetch them. Then you use the returned JSON data to create dynamic HTML that is displayed in the **products** . You find the current hard-coded HTML in the **home-products.html** file; replace the product elements with elements that you create dynamically.

In the next exercise, you will call the **loadCategoryProduct** function from the **onUlClick click** event that is triggered when a category is clicked, and the products for the first category when the category list has been loaded.

1. Let's implement the **loadCategoryProduct** function if the **loadCategoryProduct.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Only implement the code inside the try-block in the **categoryId** parameter that is passed into the function is greater than 0; a product always belongs to a category.
3. Fetch the **products** and cache it in a variable.

4. Only implement the rest of the code if the **products** element is found and the **CustomerId** Local Storage property exist; the customer id is mandatory for the API endpoint. You can call the **localStorageKeyExist** function you created earlier to check if it exists.
5. Assign the **CATEGORY** value from the **action** object in the **constants.js** file to the **lastAction** variable to keep track of the last action the user took (in this case chose a category in the **categories** list.)
6. Store the id of the selected category in the **selectedCategoryId** variable to keep track of the clicked category.
7. Call the API to fetch the categories.
   https://localhost:5001/api/categories/1/products?customerId=1
8. Inside the loop that iterates over the returned products, call the **createProductCard** function to create a product card for each product.
9. Save all files.

04c—Display the Products when a Category is Clicked

To display the products when a category is selected, you implement the **ulOnClick click** event in the **loadCategories.js** file. You also want to load the products for the first category when the **categories** list has been loaded.

To call the API for the products, the category id must be obtained from the <li> element. Note, however, that, because the user can click on one of the <li> element's children or grandchildren, you might have to traverse up the DOM tree to find the category id. Use the **parentElement** to locate the <li> element, which has the attribute containing the id.

1. Let's implement the **ulOnClick click** event function in the **loadCategories.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Use if/else if-statements to find the category id by checking the element itself, and its parent and grandparent elements for the category id.
3. Call the **loadCategoryProduct** function with the category id if one was found.
4. Go to the end of the **loadCategories** function.
5. Below the **onUlClick** event assignment, store the category id from the first category in a variable.
6. Below the variable, call the **loadCategoryProduct** function with the id in the variable you just created.
7. Save all files and start the application.
8. The products for the first category *Computers & Accessories* should be displayed.
9. Click the next category to display the products for it (only the two first categories have products.)
10. Click the first category to display its products again.

05—Search for Products

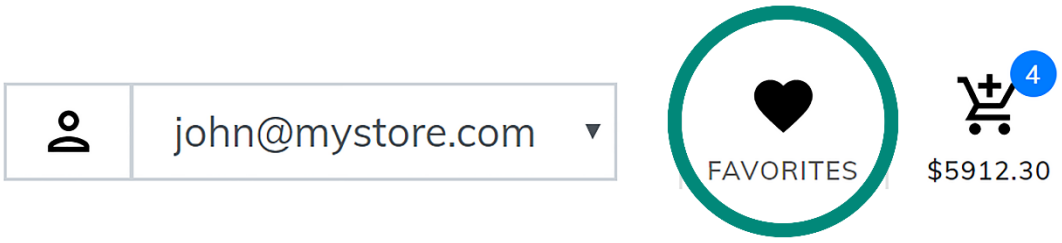
To display the products during a search, you call the in-memory API in the **search** event function in the **search.js** file to fetch them. Then you use the returned JSON data to create dynamic HTML that is displayed in the **products** . You use the function **createProductCard** you added earlier to display each product in the search response; replace the product elements with elements that you create dynamically.

The search should be performed if the magnifier button to the left of the textbox is clicked or if enter is pressed while the textbox is in focus.

1. Let's implement the **search** function in the **search.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Only implement the code inside the try-block if the event is a mouse event or if it's triggered by the **Enter** key on the keyboard.
3. Turn off the default behavior for the event.
4. Fetch the **products** and cache it in a variable.
5. Fetch the **product-search** <input> and cache it in a variable.
6. Only implement the rest of the code if the **products** and **product-search** elements are found and the **CustomerId** Local Storage property exist; the customer id is mandatory for the API endpoint. You can call the **localStorageKeyExist** function you created earlier to check if it exists.
7. Assign the **SEARCH** value from the **action** object in the **constants.js** file to the **lastAction** variable to keep track of the last action the user took (in this case searched for products.)
8. Fetch the value from the <input> element and store it in the **searchTerm** variable in the **constants.js** file.
9. Call the API to fetch the products.
   https://localhost:5001/api/products/abc?customerId=1
10. Inside the loop that iterates over the returned products, call the **createProductCard** function to create a product card for each product.
11. Save all files.
12. Start the application.
13. Enter *imac* in the textbox and press enter; only iMac products should be displayed.
14. Enter *surface* in the textbox and click the magnifier button; only Surface Book products should be displayed.

06a—Favorites

To display the customer's favorite products, you call the in-memory API in the **getFavorites** event function in the **favorites.js** file to fetch them. Then you use the returned JSON data to create dynamic HTML that is displayed in the **products** . You use the function **createProductCard** you added earlier to display each product in the search response; replace the product elements with elements that you create dynamically.



1. Let's implement the **getFavorites click** event function in the **favorites.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Only implement the code inside the try-block if the event is a mouse event.
3. Turn off the default behavior for the event.
4. Fetch the **products** and cache it in a variable.

5. Only implement the rest of the code if the **products** element is found and the **CustomerId** Local Storage property exist; the customer id is mandatory for the API endpoint. You can call the **localStorageKeyExist** function you created earlier to check if it exists.
6. Assign the **FAVORITES** value from the **action** object in the **constants.js** file to the **lastAction** variable to keep track of the last action the user took (in this viewing the customer's favorites.)
7. Call the API to fetch the products.
   https://localhost:5001/api/favorites?customerId=1
8. Inside the loop that iterates over the returned products, call the **createProductCard** function to create a product card for each product.
9. Save all files and start the application.
10. Click the **Favorites** button; The current customer's favorite products should be displayed.

06b—Add and Remove a Favorite

To add or remove one of the customer's favorite products you click a product's heart, then you call the in-memory API in the **favorite** event function in the **favorites.js** file to update it; the API call returns the updated product list. Then you use the returned JSON data to create dynamic HTML that is displayed in the **products** . You use the function **createProductCard** you added earlier to display each product in the search response; replace the product elements with elements that you create dynamically.

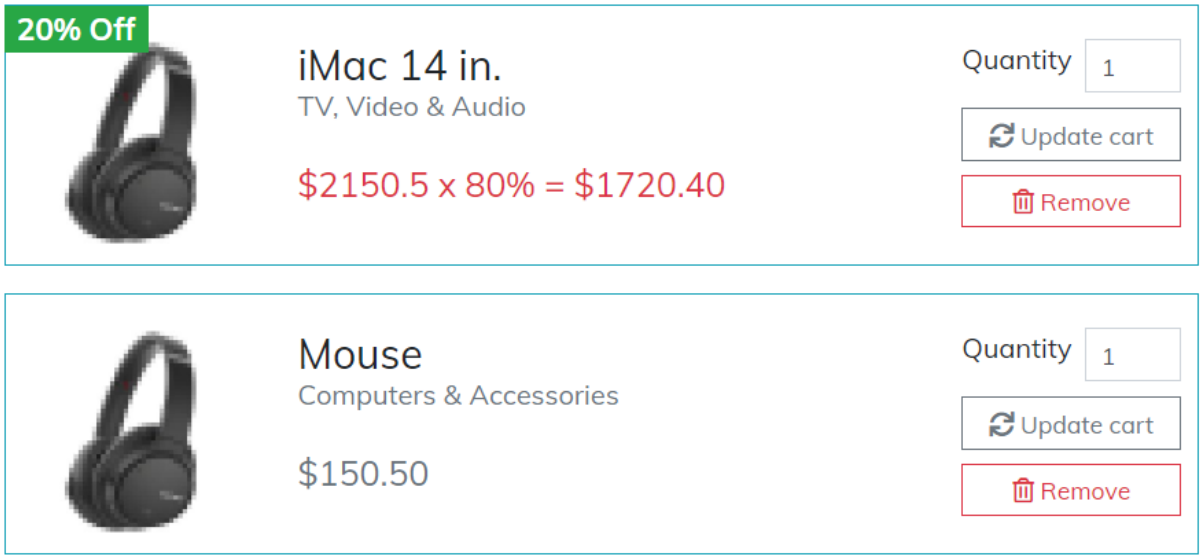
1. Let's implement the **favorite click** event function in the **favorites.js** file that updates the involved HTML elements; write the code inside the try-block.
2. Only implement the code inside the try-block if the event is a mouse event and the **productId** parameter's value is greater than 0; to update the correct product, you need its product id.
3. Turn off the default behavior for the event.
4. Fetch the **products** and cache it in a variable.
5. Only implement the rest of the code if the **products** element is found and the **CustomerId** Local Storage property exist; the customer id is mandatory for the API endpoint. You can call the **localStorageKeyExist** function you created earlier to check if it exists.
6. Call the API to fetch the products. Note that the value of the **lastAction** variable is sent as a parameter (**returnData**) to the API endpoint to ensure that the correct products are returned; if products are displayed after clicking a category, after a search, or if the **Favorites** button has been clicked, then the API need to know that to return the correct products.
   a. Note that the **put** verb should be used when calling the API because you are updating the resource.

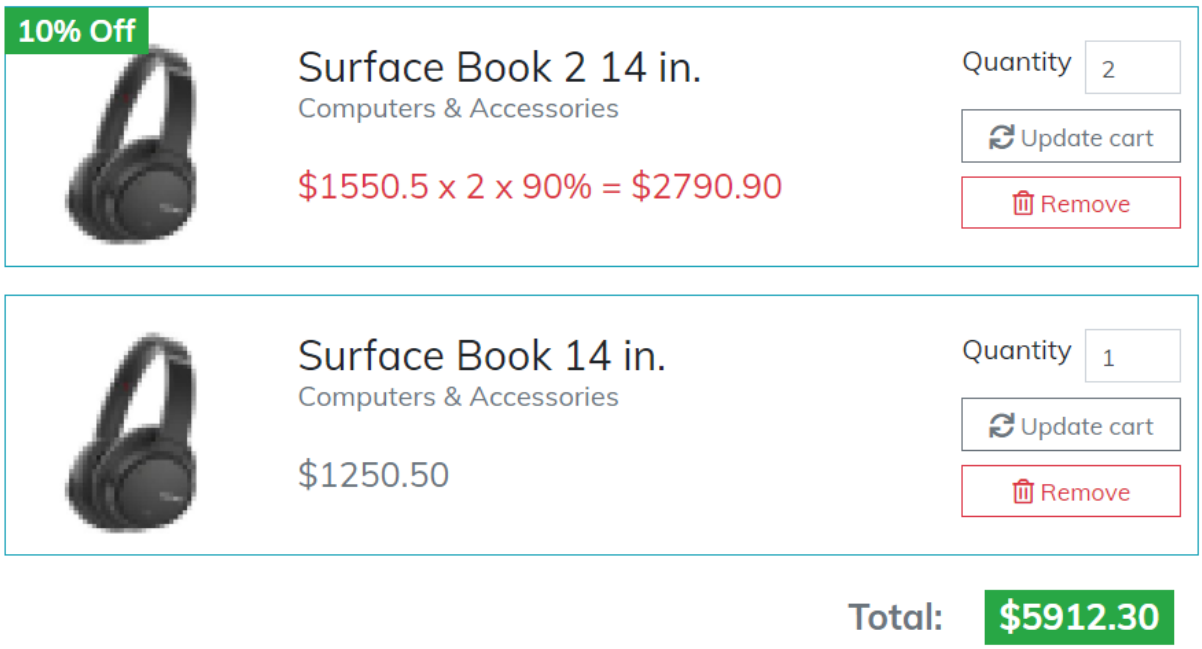
   b. The *productId* value is stored in the **productId** parameter.
   c. The *customerId* value is stored in the **CustomerId** Local Storage property.
   d. The *returnData* value is stored in the **lastAction** variable.
   e. The *categoryId* value is stored in the **selectedCategoryId** variable.
   f. The *searchTerm* value is stored in the **searchTerm** variable.

   https://localhost:5001/api/favorites?productId=2&customerId=1&returnData=3&categoryId=2&searchTerm=abc
7. Inside the loop that iterates over the returned products, call the **createProductCard** function to create a product card for each product.
8. Save all files and start the application.
9. Click the **Heart** button for a couple of products and make sure that the heart changes.
10. Click the **Favorites** button; the added favorite products should be displayed for the current customer.
11. Click a black heart to remove it from the list of favorites; the remaining favorites should be displayed.
12. Do a search and click the **Heart** button for a product; the search result should still be displayed with the favorite status changed (the heart for the product should have been toggled.)
13. Click one of the two first categories to display its products.
14. Click the **Heart** button for one of the products and make sure that the heart changes.
15. Click the **Favorites** button; the added favorite should be displayed for the current customer.

07a—Show the Shopping Cart

To display the shopping cart, you need to create each cart item with dynamic HTML in the **createCartItem** function in the **shoppingCart.js** file as you did for the products cards earlier; note that there are four calculations you should take into account when displaying the total on the cards (see image.) The **displayCartItems** function creates all the cart items by calling the **createCartItem** function and add the total below the last cart item. The **loadCart** function calls the in-memory API to fetch the shopping cart items and then calls the **displayCartItems** function with the returned JSON data.





1. You implement the **createCartItem** function in the **shoppingCart.js** file as you did with the product cards in the **createProductCard** function, but with different HTML. You find the static HTML for the card items in the **products** in the **home-products.js** file (if you haven't already deleted it.)
2. Only implement the code inside the try-block of the **displayCartItems** function if the **product** exists and the **CustomerId** and **CartItems** properties exists in the Local Storage.
3. Fetch the cart items from the **CartItems** Local Storage property and parse it to JSON.
4. Iterate over the cart items and call the **createCartItem** function for each cart item.
5. Add HTML that displays the cart total below the last cart item to the **products** .
6. Only implement the code inside the try-block of the **loadCart** function if the **CustomerId** property exists in the Local Storage.
7. Call the API to fetch the shopping cart items.
   https://localhost:5001/api/ShoppingCartProducts?customerId=1
8. Store the fetched cart products (in the **products** property of the returned data) in a Local Storage property named **CartItems**.
9. Parse the returned cart total (in the **total** property of the returned data) to a float value and format it to two decimals and store it in a Local Storage property named **CartTotal**.
10. Call the **displayCartItems** function to generate the HTML for the cart items and display them.
11. Save all files and start the application.
12. Click the **Cart** button to display the customer's shopping cart.

07b—Add a Product to the Shopping Cart

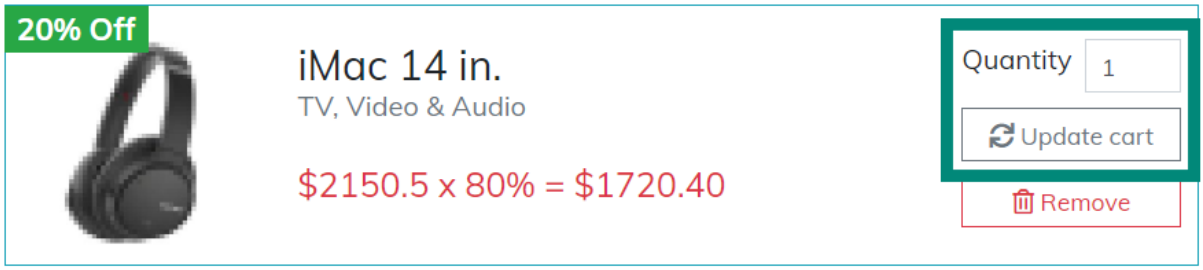
To add a product to the customer's shopping cart, you need to implement the **addToCart** function that calls the in-memory API to add the product to the customer's shopping cart and then calls the **displayBasicCart** function with the returned JSON data. Note that the **post** verb is used to call the API endpoint.

1. Only implement the code inside the try-block of the **addToCart** function if the **CustomerId** property exists in the Local Storage.
2. Call the API to add the product to the customer's shopping cart; the API returns the data for a basic shopping cart that is used to update the shopping cart button.
    a. Needs the following data in the body: *customerId: parseInt(customerId), productId: parseInt(productId), count: 1*.

    https://localhost:5001/api/ShoppingCartProducts
3. Call the **displayBasicCart** function to update the shopping cart button.
4. Save all files and start the application.
5. Click the **Cart** button at the bottom of a product to add it to the customer's shopping cart.
6. Click the **Cart** button in the toolbar to display the customer's shopping cart; the product you added should be displayed with any other shopping cart items.

07c—Update a Product's Quantity in the Shopping Cart

To add a product to the customer's shopping cart, you need to implement the **updateQuantity change** event function and the **updateCart** function that calls the in-memory API to add the product to the customer's shopping cart and then calls the **displayBasicCart** function with the returned JSON data. Note that the **put** verb is used to call the API endpoint.

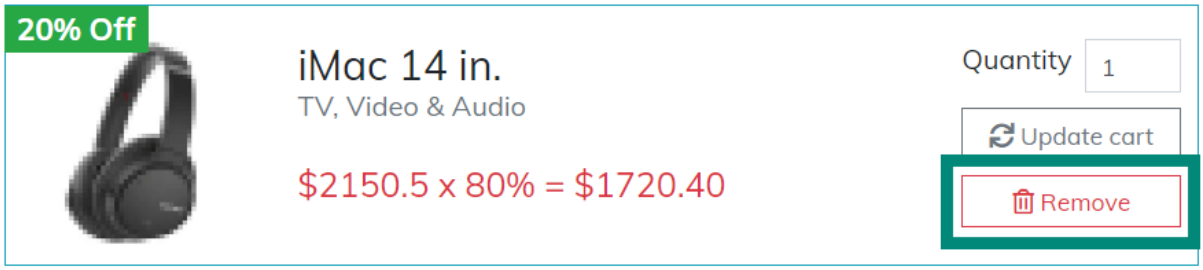


1. Only implement the code inside the try-block of the **updateQuantity change** event function if the **productId** parameter is greater than 0 and the **CartItems** and **CustomerId** properties exists in the Local Storage.
2. Update the correct product's quantity among the shopping cart products in the **CartItems** Local Storage property.
3. Only implement the code inside the try-block of the **updateCart** function if the **productId** parameter is greater than 0 and the **CartItems** and **CustomerId** properties exists in the Local Storage.
4. Fetch the correct product from the **CartItem** Local Storage property using the **productId** parameter value and save it in a variable.

5. Only implement the call to the API if a product matching the product id was found among the **CartItems** in the Local Storage.
6. Call the API with the fetched cart product as body data. Note that the **put** verb should be used.
   https://localhost:5001/api/ShoppingCartProducts
7. Assign **false** to the **basicCartLoaded** variable to make it possible to update the shopping cart button's data.
8. Call the **basicCart** function to update the shopping cart button's data.
9. Save all files and start the application.
10. Click the **Cart** button in the toolbar to display the customer's shopping cart.
11. Change the quantity for one of the cart products and click the **Update cart** button on the product card. The quantity should have been persisted if you return to the shopping cart, and the shopping cart button's data should have been updated.

07d—Remove a Product from the Shopping Cart

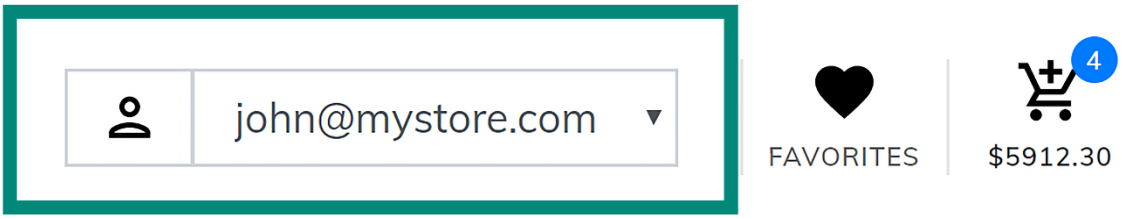
To remove a product to the customer's shopping cart, you need to implement the **deleteProduct-FromCart** function that calls the in-memory API to remove the product from the customer's shopping cart and then removes the product from the shopping cart in Local Storage and calls the **displayCart-Items** function to display the updated shopping cart. Note that the **delete** verb is used to call the API endpoint.



1. Only implement the code inside the try-block of the **deleteProductFromCart** function if the **productId** parameter is greater than 0 and the **CartItems** and **CustomerId** properties exists in the Local Storage.
2. Fetch the products from the **CartItem** Local Storage property and save the result in a variable.
3. Fetch the index of the product matching the **productId** parameter value and the customer id from the **CartItem** Local Storage property; save the result in a variable.
4. Call the API with the product id and customer id to delete the product from the customer's cart. Note that the **delete** verb should be used.
   https://localhost:5001/api/ShoppingCartProducts?customerId=1&productId=2
5. Call the **displayBasicCart** function with the data returned from the API.
6. Remove the deleted product from the cart items in the Local Storage using the index you fetched earlier.
7. Call the **displayCartItems** function to update the displayed products in shopping cart.
8. Save all files and start the application.
9. Click the **Cart** button in the toolbar to display the customer's shopping cart.
10. Click the **Remove** button on a product card; the product should be removed from the shopping cart, and the shopping cart button's data should be updated.

08—Choose Customer

When choosing another customer in the drop-down, the **onSelectedCustomer change** event function in the **login.js** file should update the shopping cart button's data by calling the **basicCart** function and load products for the first product category by calling the **loadCategoryProducts** function.



1. Assign **false** to the **basicCartLoaded** variable to make it possible to update the shopping cart button's data.
2. Update the **CustomerId** Local Storage property with the value from the selected customer in the drop-down.
3. Call the **basicCart** function to update the shopping cart button's data.
4. Call the **loadCategoryProducts** function with the category id stored in the **firstCategoryId** variable in the constants.js file to update the product list.
5. Save all files and start the application.
6. Select another customer from the drop-down; the shopping cart button's data should be updated and the products for the first category should be displayed.

Extra Exercises

https://www.w3resource.com/javascript-exercises/

https://www.w3schools.com/js/js_exercises.asp

https://rmurphey.com/blog/2016/02/13/exercises-for-js-beginners

https://github.com/TheOdinProject/javascript-exercises

https://coderbyte.com/challenges

https://practity.com/582-2/

https://www.learn-js.org/

https://www.teaching-materials.org/javascript/exercises/functions

https://www.sitepoint.com/5-typical-javascript-interview-exercises/