4. Document Object Model

Obsah

1 Document Object Model (DOM) Introduction

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page. (1)

A Web page is a document. This document can be either displayed in the browser window or as the HTML source. But it is the same document in both cases. The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript. (1)

DOM is structured as a tree of objects, so let’s see an illustration example of this structure. (2)

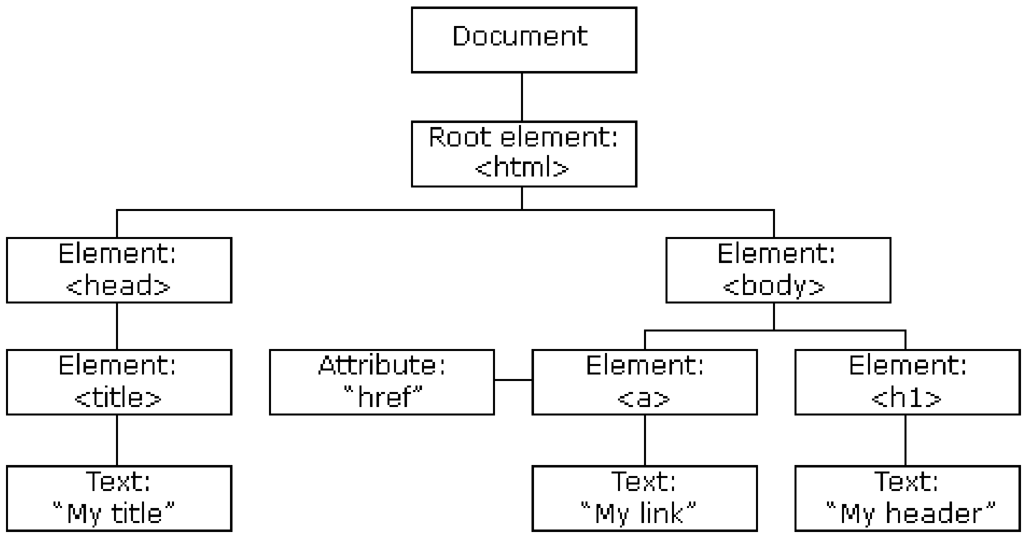
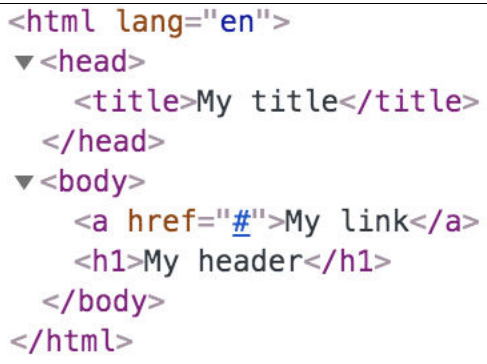


Figure 1 - The HTML DOM Tree of Objects illustration. Source: (2)

And let’s now see, how the real DOM of the previous illustration looks like in the browser.



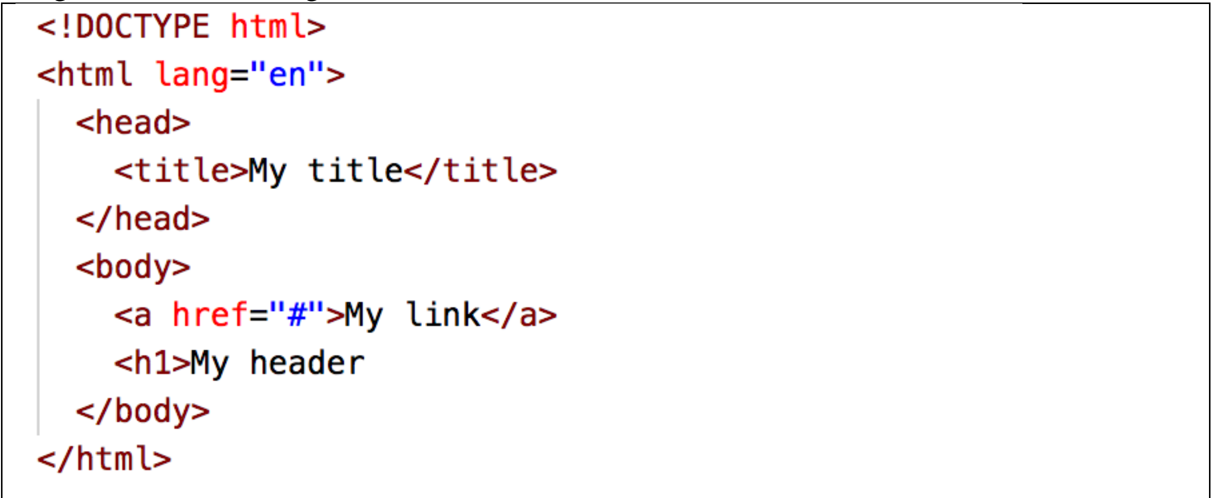
```
<html lang="en">
 ▼<head>
    <title>My title</title>
  </head>
 ▼<body>
    <a href="#">My link</a>
    <h1>My header</h1>
  </body>
</html>
```

Figure 2 - Real HTML DOM from browser. Source: my own image

1.1 The DOM is NOT the same thing as your source HTML

This is weird to hear, because in the example above, it looks exactly like our HTML document, right? Yes, it does, as I said earlier, the DOM just allows us to view our document in a different way, but it still needs to represent the same document.

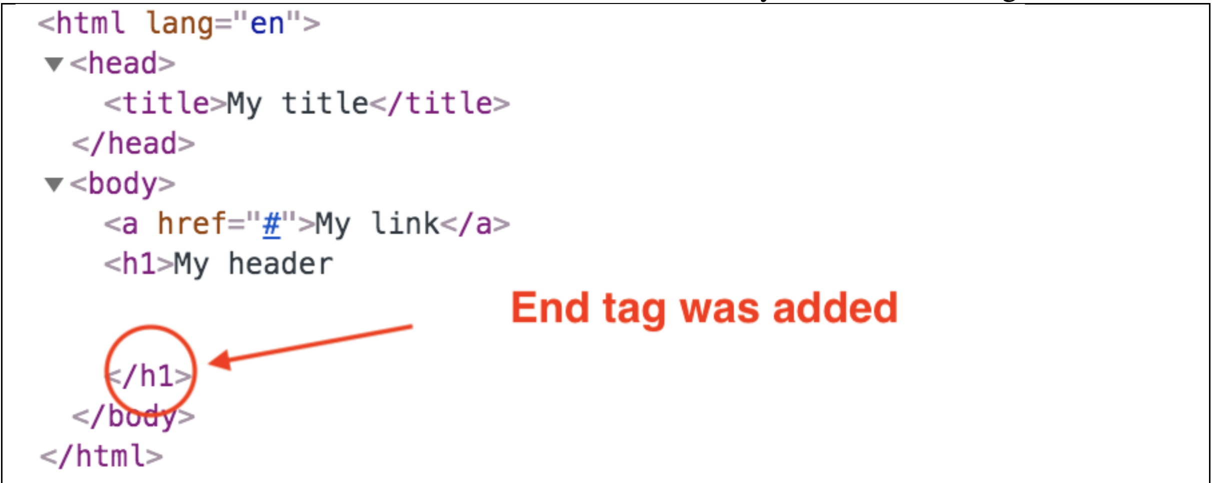
So how is it different? Well, it differs from your HTML document, when you write an invalid HTML code. Let's look at the following example, of HTML document, where we forgot to end our **h1** tag as shown below. (3)



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="#">My link</a>
    <h1>My header
  </body>
</html>
```

Figure 3 - HTML document with invalid h1 tag. Source: my own image

How the DOM looks for this HTML document? As you can see, end tag was added.



```
<html lang="en">
▼<head>
    <title>My title</title>
  </head>
▼<body>
    <a href="#">My link</a>
    <h1>My header
    </h1>
  </body>
</html>
```

Figure 4 - the DOM example for invalid HTML document. Source: my own image

There is another situation, when the DOM differs from the HTML document, can you guess that situation? Well, it's when you manipulate with it via JavaScript. (3)

1.2 JavaScript vs the DOM

Let's rather say again, that the DOM is NOT some JavaScript feature. The DOM is just an interface that is able to display our HTML documents in a certain way, so scripting languages like JavaScript can manipulate with it.

You can imagine the DOM as a sheet of paper and JavaScript as a pen with a gum eraser on it. Paper gives you a way to show something and pen with an eraser allows you to actually write/draw something on that piece of paper and eventually, remove something as well.

1.3 What can JavaScript do with the DOM?

With the DOM, JavaScript gets all the power it needs to create dynamic HTML: (2)

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

1.4 Simple example of DOM manipulation

The goal is to have a button, which changes the heading text on click. So, what do we have to do, to achieve that?

It includes two steps

- Select elements that we want to work with in our JavaScript code
- Add event listener (say what should happen and when)

1.4.1 Select elements

First, we need to select elements, that we want to work with in our JavaScript code. In this case, it's that button element, and the heading element.

To do so, we can use **document.getElementById()** method, which returns the element that has the ID attribute with the specified value. (4)

1.4.2 Add an event listener

Now we can add an event listener. We can do so via method **addEventListener()** which is a method that sets up a function that will be called whenever the specified event is delivered to the target (button in our case). (5) Let's have a look at the syntax of **addEventListener()**.

```
element.addEventListener(event, function, useCapture);
```

As you can see in the syntax, **addEventListener()** method accepts three parameters: (6)

- The first parameter is the type of the event (like "**click**" or "**mousedown**" or any other HTML DOM Event.)
- The second parameter is the function we want to call when the event occurs (in our case, it's the **setNewHeadingText()** function).
- The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

And that's it! Now the heading changes after button click to "Hello students!". Just to clarify, we used **innerHTML** property to change the text inside the heading. The **innerHTML** property sets or returns the HTML content (inner HTML) of an element. (7)

```
const btn = document.getElementById("btn"); //select button element and
assign it to a variable
const heading = document.getElementById("heading"); //select heading
element and assign it to a variable

function setNewHeadingText() {
   heading.innerHTML = "Hello students!"; //changes text inside the h1
element
}
```

```
btn.addEventListener("click", setNewHeadingText); //add listener to our
button element, that listens to click event and calls given function
```

1.4.3 Side note – event listener function improvement

Our code works fine, but there is a way to improve it a bit. In the code example above, our logic of what should happen on a button click, refers to an external named function **setNewHeadingText()**.

I wrote it that way, because it's easier to read for beginners in my opinion, BUT this is mostly not the way developers do it, especially for such a short amount of function logic. Mostly, anonymous functions are used.

```
btn.addEventListener(
        "click",
        () => (heading.innerHTML = "Hello students!")
); //add listener to our button element, that listens to click event and
calls given function
```

Why are anonymous functions used? Well, for two reasons. First, in many cases, functions executed after some event, are not reusable, so it's pointless to create named function, when you call it only in one place anyway. It's simply easier to read, when you just create an anonymous function, which is tied to that event listener.

Second, functions in most cases need parameters. Let's come back to our previous code example where we have a named function and say, that we want to be able to pass our new heading into function via function parameter, instead of having hard coded value inside function as it is right now.

So, what to do? The following can come up to your mind, but it doesn't work as expected. It actually changes the heading immediately after we open our file, can you guess why? Well, it's, because we need to pass a reference to the function, so it runs after the event happens). But because we've used rounded brackets, that function is called immediately after we've opened our file.

```
btn.addEventListener("click", setNewHeadingText("Hello students!"));
```

Okay, what now? Well, as it turns out, we need anonymous function for this anyway! This way, an anonymous function is declared, and the only thing it does when it's called is, that it calls our **setNewHeadingText** function. So why bother to create a named function, since you have to use an arrow function anyway. It's just easier and better to declare only that anonymous function and write all the code inside that function.

```
btn.addEventListener("click", () => setNewHeadingText("Hello
students!"));
```

2 Game - Guess the color

There are many methods and properties, which the DOM provides (see whole HTML DOM section on https://www.w3schools.com/jsref/dom_obj_attributes.asp), but it's not important to go through all of them and memorize them. Because of that we won't be going through all of them, as that would be very boring, and you can always simply look for what you need on the internet.

What is important though, is for you to understand how to work with the DOM. So, what I will do in the rest of this class is, that I will create a very simple game, which covers the majority of basic tasks that you normally need to do, while working with the DOM.

2.1 Instructions

Create a game called “Guess the color” which has very simple rules. At the beginning an RGB code of some color is presented. Below that code, four options are presented. If the player picks the wrong option, then that option disappears. If the player picks the right one, then a win modal appears, and the player is presented with an option to play again.

2.2 Final result

2.2.1 Initial load

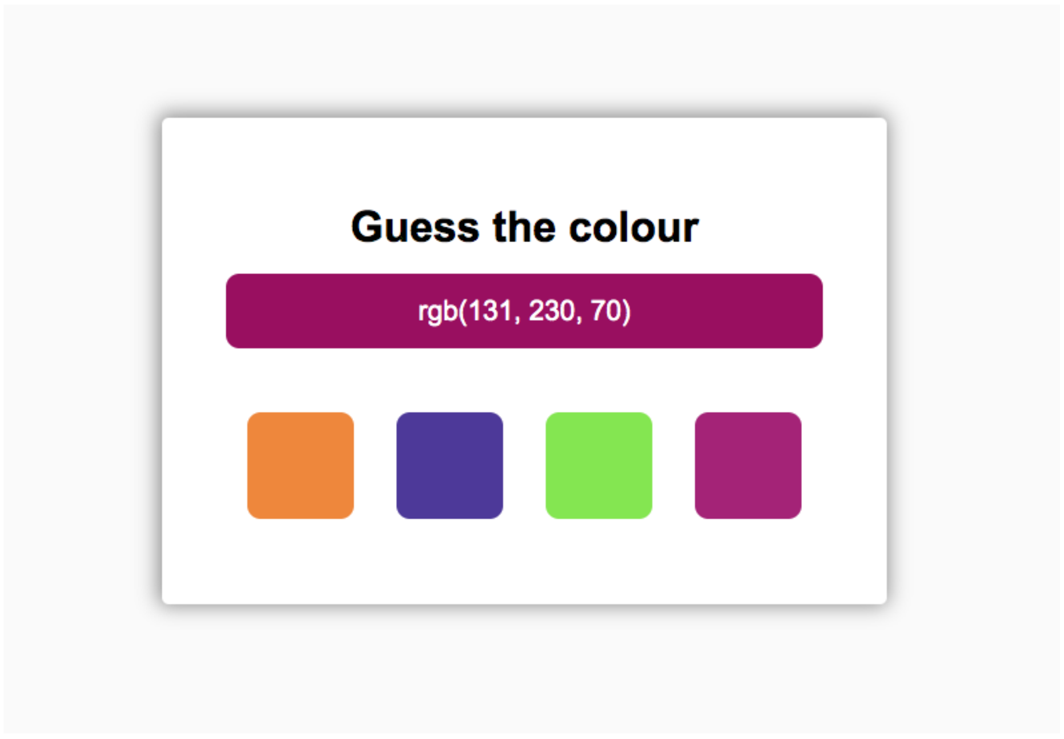


Figure 5 - Guess the color - initial load. Source: my own image

2.2.2 Win modal

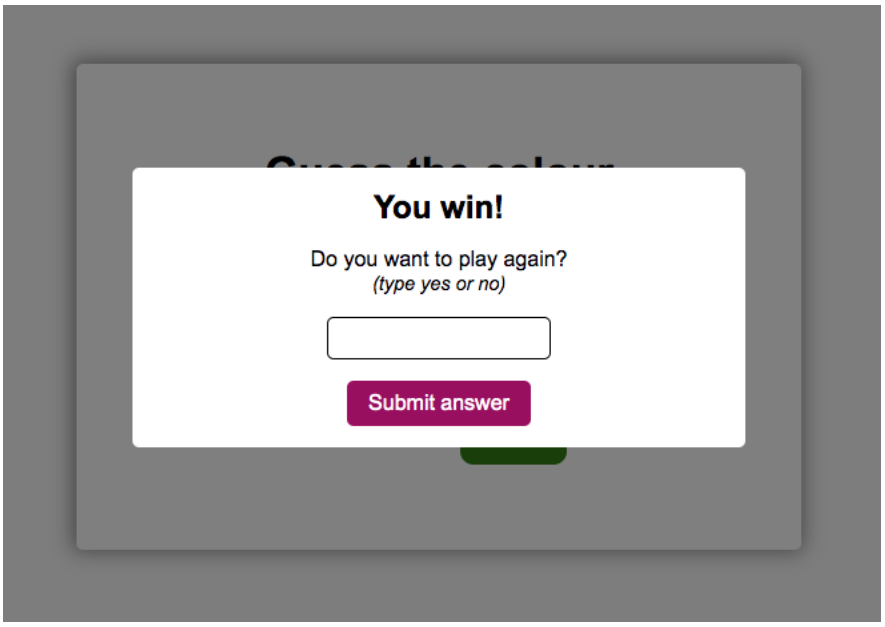


Figure 6 - Guess the color. Win modal. Source: my own image

2.3 First steps

The static HTML page with styles is already given to us, we just need to give it some life, so it reacts to user actions and displays content accordingly.

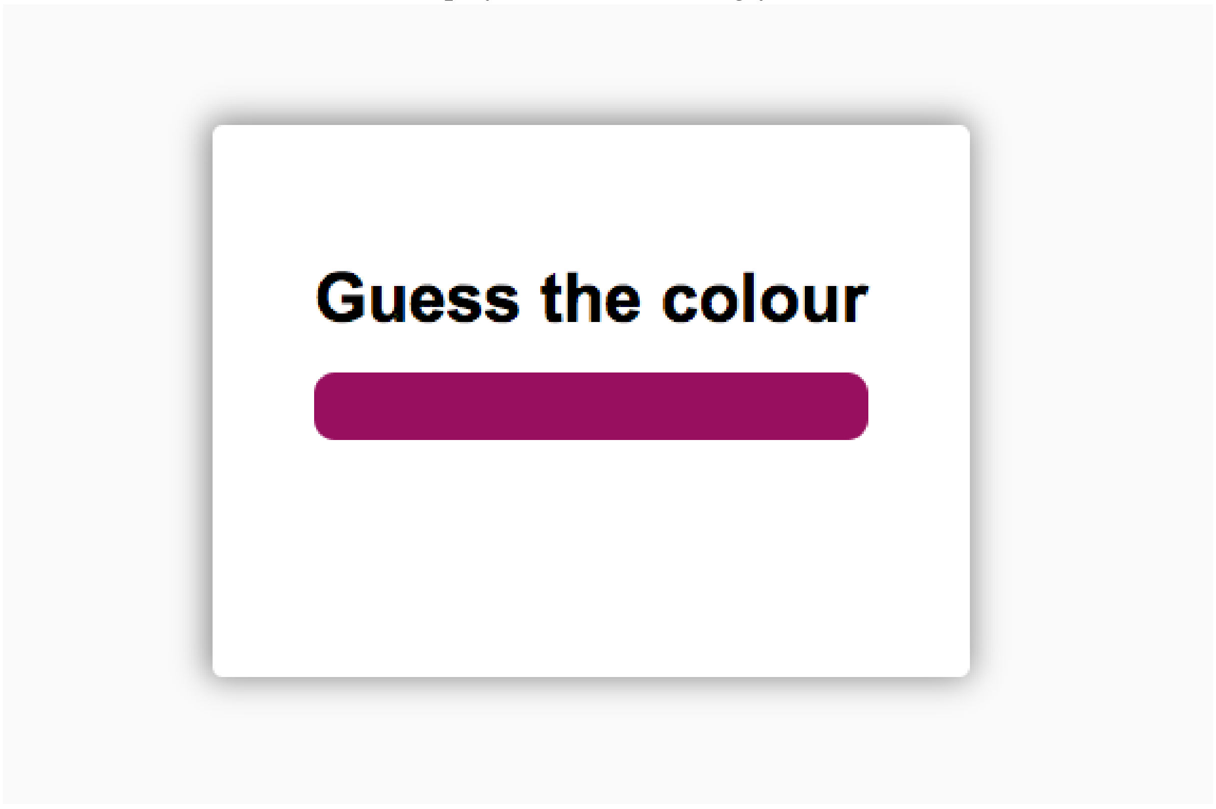
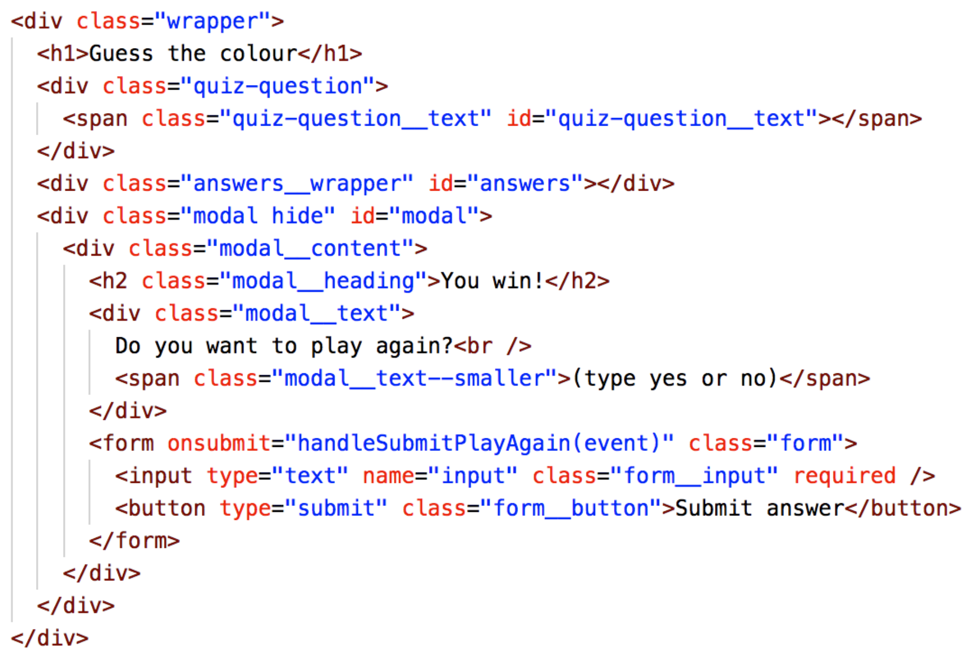


Figure 7 - static HTML page. Source: my own image

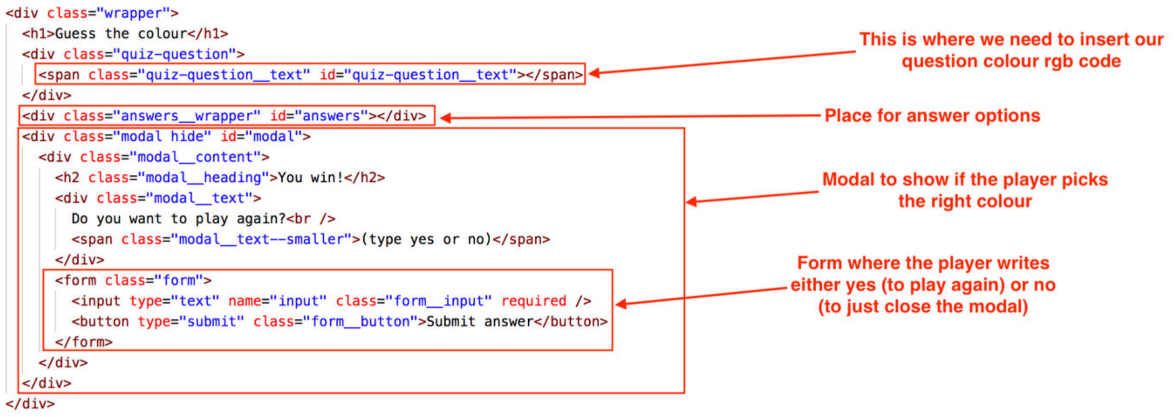
Let’s check out the HTML code to see what we need to do next.



```html
<div class="wrapper">
  <h1>Guess the colour</h1>
  <div class="quiz-question">
    <span class="quiz-question__text" id="quiz-question__text"></span>
  </div>
  <div class="answers__wrapper" id="answers"></div>
  <div class="modal hide" id="modal">
    <div class="modal__content">
      <h2 class="modal__heading">You win!</h2>
      <div class="modal__text">
        Do you want to play again?<br />
        <span class="modal__text--smaller">(type yes or no)</span>
      </div>
      <form onsubmit="handleSubmitPlayAgain(event)" class="form">
        <input type="text" name="input" class="form__input" required />
        <button type="submit" class="form__button">Submit answer</button>
      </form>
    </div>
  </div>
</div>
```

Figure 8 - HTML code of the game. Source: my own image

Okay, so there are many pieces, which we need to pick up in our JS code and do something with it. Let’s highlight those pieces of code.



```html
<div class="wrapper">
  <h1>Guess the colour</h1>
  <div class="quiz-question">
    <span class="quiz-question__text" id="quiz-question__text"></span>
  </div>
  <div class="answers__wrapper" id="answers"></div>
  <div class="modal hide" id="modal">
    <div class="modal__content">
      <h2 class="modal__heading">You win!</h2>
      <div class="modal__text">
        Do you want to play again?<br />
        <span class="modal__text--smaller">(type yes or no)</span>
      </div>
      <form class="form">
        <input type="text" name="input" class="form__input" required />
        <button type="submit" class="form__button">Submit answer</button>
      </form>
    </div>
  </div>
</div>
```

This is where we need to insert our question colour rgb code
Place for answer options
Modal to show if the player picks the right colour
Form where the player writes either yes (to play again) or no (to just close the modal)

Figure 9 - HTML code with highlighted pieces of code and explanation of what needs to be done with it. Source: my own image

2.4 Create question and answer options

What do we need to do? Well, let’s think about it for a second. There is an rgb representation of some color, and then there are four color samples (answer options) from which one of them is a visual representation of that rgb color code (the correct answer).

There are very likely many options to solve this problem, but I chose the following one. First, I want to create an array of four randomly generated rgb color codes and then I want to pick one of those codes and set that as our question color.

2.4.1 Generate random number from 0 to 255

RGB color code consists of three numbers in range between 0 to 255, so the first thing we need to do, is to generate a random number within that range.

The following code may seem little complicated, but it actually isn't. The problem is, that built in function for generating a random number called **Math.random()** generates only a random floating number between 0 (inclusive) and 1 (exclusive), so we need to tweak it a bit, so it generates a number from 0 to 255. (8)

What we are doing is, that we generate a number between 0 and 1, multiply it by 256, and them then rounding via **Math.floor()** method.

Math.floor() method returns the largest integer less than or equal to a given number. (9)

Why is it multiplied by 256? Well, it's because, **Math.random()** range end is exclusive.

```
const getRandomRgbNumber = () => Math.floor(Math.random() * 256);
```

2.4.2 Generate a random rgb color

Now that we are able to generate random numbers within the range of 0 to 255, we can generate random rgb color. First, how does it look in CSS, when you want to define a color via rgb? Something like this: **rgb(142,43,16);** This is exactly, what we need to achieve.

In the following code, a random number is generated for each color (red, green, blue) and then these random numbers are put into one string, which represents a CSS way of defining a color via rgb.

```
const generateColor = () => {
   const red = getRandomRgbNumber();
   const green = getRandomRgbNumber();
   const blue = getRandomRgbNumber();
   return `rgb(${red}, ${green}, ${blue})`;
};
```

2.4.3 Create an array of four randomly generated rgb colors

We got the ability to generate an rgb representation of color, so we can finally create our array of rgb colors. We need to create a variable where this array is going to be stored (in the code example below, it's the **answerColors** variable). It needs to be a global variable, because variety of functions will need to access it. We also need to create a function, that will fill in that array. As you can see in the example, we have a **for loop**, where we say, that we want to do 4 iterations and within each iteration, new rgb color is generated and then pushed into our global array **answerColors**.

```
const answerColors = [];
const createAnswerColorsArr = () => {
   for (let i = 0; i < 4; i++) {
      answerColors.push(generateColor());
   }
};
```

2.4.4 Render answer options to the DOM

Now it gets interesting. We have our array of rgb colours, but now we need to actually show it to our player. We need to render four clickable squares and assign them those colors from our **answerColors** array.

First, we need to get access to the element, where want to render our new elements. As you can see at **Figure 9 - HTML code with highlighted pieces of code and explanation of what needs to be done with it**, we need to access the element with an id of **answers**. Let's do that.

```
const answers = document.getElementById("answers");
```

What next? Well, now the magic can happen. As it turns out, JavaScript has a method called **insertAdjacentHTML()**. The **insertAdjacentHTML()** method inserts a text as HTML, into a specified position. (10)

Well that's exactly what we need!

In the following code example, a **for loop** is created for array **answerColors**. Within each iteration of that for loop, we access previously selected element **answers** to which we attach new HTML button via **insertAdjacentHTML()**.

insertAdjacentHTML method needs 2 parameters. First one specifies the position relative to the element (**answers** in our example). There are four options: (11)

- **'beforebegin'**: Before the element itself.
- **'afterbegin'**: Just inside the element, before its first child.
- **'beforeend'**: Just inside the element, after its last child.
- **'afterend'**: After the element itself.

Beforeend is used, because after each iteration, new button is rendered to the DOM, and we want that button to be inside **answer** and after its last child. E.g. there are 2 buttons already rendered into the DOM. The third button is then positioned after those 2 already existing buttons. If we would use **afterbegin** it would be positioned before those 2 already existing buttons.

The second parameter of **insertAdjacentHTML** is a string representation of an element, that we want to render to the DOM. There is probably nothing surprising except for one thing, **onclick**.

I previously used **addEventListener()** method to attach an event listener to element (see **1.4.2 add event listener)** as it turns out, there is another way to do this via **onclick**. Some programmers prefer to keep HTML code clean and simple, so it only shows stuff, so in most cases they use **addEventListener()** method. But while you definitely can do this via **addEventlistener()**, in some cases like this one, it’s just much easier to do it via **onclick**, therefore, that’s what I chose. What I want to say with this is, that there are cases, where this decision between **addEventListener()** and **onclick** is just a matter of preference, therefore there is no right or wrong.

As you can see **this** is passed as a parameter. Just ignore it now for now, I will get to it once I will be explaining **handleAnswerOptionClick** function.

```
const renderAnswerOptions = () => {
   for (let i = 0; i < answerColors.length; i++) {
      answers.insertAdjacentHTML(
        "beforeend",
        `<button class="answers__option" style="background-color:
${answerColors[i]}" onclick="handleAnswerOptionClick(this)"
id="${i}"></button>`
      );
   }
};
```

2.5 Render the rgb code of the color the player needs to guess

Now that we have our answer options rendered. We need to set, which one of them should the player guess and render rgb code of that color in the DOM.

2.5.1 Select the element we need to work with

We need to find out the element, where our rgb code should be rendered, so let’s come back to **Figure 9** again, and according to that, we need to grab element with a class **quiz-question__text"** and id of the same name. There are many ways for selecting an element from the DOM, so while we could definitely select it by using **getElementById()**, let’s use a different approach and use **querySelector()** to select it.

The **querySelector()** method returns the **first** element that matches a specified *CSS* selector(s) in the document. That word “**first**” in previous sentence is important, because it means, that if you have more occurrences of element with specified CSS selector, it will select only the first one. (12)

Also notice, that the definition talks about "specified CSS selector(s)", so you can either use ID or class. Let's show both of them.

First select it by ID. The only thing you need to do, is use "**#**" in front of its name, so the JS knows that you want to select that element by its ID.

```
const quizQuestionText = document.querySelector("#quiz-question  text");
```

Now by its class. It's basically the same, you just need to use "**.**" in front of its name instead of "**#**". In our case, names of ID and class are the same, so we just need to switch from "**#**" to "**.**". It doesn't really matter which one we use in this scenario, so I just keep it selected by its class.

```
const quizQuestionText = document.querySelector(".quiz-question  text");
```

2.5.2 Create a function that renders quiz color rgb code - beginning

Okay, so let's now create a function that picks the quiz color from the **answerColors** array and renders that rgb value into the DOM.

Now it gets interesting. How to pick that quiz color? We could hard code the array position from which the quiz color would be taken. But that's not a good solution, as that would mean, that the correct answer button would always be at the same position. The better solution would be, to randomly pick some value from **answerColors** array. In other words, we want to randomly generate an index number between 0 and 3 and set a value with that index number as our quiz color.

2.5.3 DRY principle and our getRandomRgbNumber() function

But wait, we already have a function, that focuses on generating random numbers called **getRandomRgbNumber()**, right? The problem with that one is, that it has hard coded range from which it generates random numbers. That fact makes that function very hard to reuse.

What are our options now? The first one is, to create another piece of logic, that just has our desired range OR rewrite our already existing function, so it is possible to reuse it.

This is a very common dilemma for programmers. That's why DRY principle exists. DRY stands for Don't Repeat Yourself, it is a software development principle aiming to reduce code duplication which can lead to a poor refactoring and a poor maintenance. In other words, if you have to copy and paste the same exact lines of code which provide the same functionality, then chances are that your code doesn't follow the DRY at all. (13)

Why you should follow DRY principle? Well, let's come back to our random numbers generating and imagine, that we would write our random generating logic at each place we need it, and we would suddenly decide, that we want to exclude numbers at the start and end of that range. We would have to look for each place where our logic is written and change it.

That sounds terrible, right? It's just better to create one reusable piece of code, that does one thing – generates random numbers from given range. That way, if we would need to change that logic, we need to do it only in that one place.

2.5.4 Rewriting our function for generating random numbers

In the following code example, you can see our rewritten function for generating random numbers. Code should be self-explanatory, so the name of that function needed to be changed as well.

So, our new function **gerRandomInt** gets two parameters – minimum value and maximum value. We've already discussed what **Math.floor** does, but as a reminder, it rounds the number to the largest integer less than or equal to a given number. The rest of that formula just takes care that randomly generated number will be within the specified range.

```
const getRandomInt = (min, max) => {
   return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

2.5.5 Adjusting previous code to the new function

Now that have our reusable function for generating random numbers within given range, we need to adjust our previously written code to these changes. So our **generateColor** function now looks as following

```
const generateColor = () => {
   const red = getRandomInt(0, 255);
   const green = getRandomInt(0, 255);
   const blue = getRandomInt(0, 255);
   return `rgb(${red}, ${green}, ${blue});`;
};
```

2.5.6 Create a function that renders quiz color rgb code – finish

After our little refactoring, we can come back to our quiz color rgb code rendering. First the **quizColor** value is randomly picked from **answerColors** array via its index. Then the value of **quizColor** is inserted into our previously selected element via **.innerHTML** property.

```
const renderQuizColor = () => {
   quizColor = answerColors[getRandomInt(0, 3)];
   quizQuestionText.innerHTML = quizColor;
};
```

2.6 Calling our functions when the file opens

We have all the logic that game needs at the start. Now we can call those functions as following.

```
createAnswerColorsArr();
renderAnswerOptions();
renderQuizColor();
```

2.7 Handle answer option click

So, we took care of our quiz color render, and answer options render. Now we need to allow the player to actually guess the color. The player needs to be able to click on one of the options that he thinks is correct. If it is correct, he should be presented with the win modal, if not, that than wrong option should disappear.

If we go back to section **2.4.4 Render answer options to the DOM** than we can see, that the method below, is the method we are calling and passing given element as a parameter into it. That's why we can easily access that clicked elements properties. Because we set id's of buttons to indexes of values they contain from **answersColors** array, we can use that in our advantage, access those values via index and check whether the given value is equal to quizColor value. If it is, let's call function **handleModalChange** to show win modal (we will implement that in a second). If it is not, let's hide that wrong answer option. As it turns out, you can easily access inline styles of given element and modify them, which is what we're doing here.

```
const handleAnswerOptionClick = (btn) => {
   if (answerColors[btn.id] === quizColor) handleModalChange(true);
   else btn.style.opacity = 0;
};
```

2.8 Modal

Now the player can click on an answer that he thinks is correct, but even if he is right, he has no way to find out, because our win modal is not showing! Let's go ahead and fix that.

First, we need to select HTML element again as following.

```
const modal = document.querySelector(".modal");
```

Now that we can work with it, let’s add **handleModalChange** function. This function accepts boolean parameter **show**. If **show** is true, the modal is presented to the player, if not, the modal disappears.

In JavaScript, you can add or remove css classes of elements as you wish. That’s what we’ve done here. Our modal shows and disappears because we’re adding/removing class **hide** from the modal element.

```
const handleModalChange = (show) =>
     show ? modal.classList.remove("hide") : modal.classList.add("hide");
```

In order to understand how it works, let’s inspect our css. Class **modal** is absolutely positioned and takes up the whole screen. Let’s focus on its display property. Here we have flex, so the modal is displayed as we want.

```
.modal {
   position: absolute;
   bottom: 0;
   left: 0;
   width: 100%;
   height: 100%;
   background-color: rgba(0, 0, 0, 0.5);
   display: flex;
   align-items: center;
   justify-content: center;
}
```

What actually makes that modal to be presented or hidden is class **hide**. That class makes our modal to disappear via setting **display** property to **none**.

```
.hide {
   display: none;
}
```

2.9 Modal form to play again

Normally, I would go for a clickable button that allows the player to play again. But for learning purposes, let’s do it via form input, because forms are heavily used in web apps, so it’s important for you to know how to work with them.

You could use **addEventListener** to call our logic that needs to happen on submit, but it turns out that form element allows you to use attribute **onsubmit**.

If you go back to Figure 8 and Figure 9, you can see, that we use **onsubmit** attribute to call function **handleSubmitPlayAgain**, and pass a parameter **event** to it. Parameter **event** presents **SubmitEvent** in this case, which contains all we need.

Let’s check out our **handleSubmitPlayAgain** function. First odd thing that you can notice, is **event.preventDefault()**. Why is it there and what is it doing?

What is it doing? The **preventDefault()** method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur. (14)

Why is it there? Well, normally, when you submit a form, it evokes page refresh. In many cases that’s exactly what you want, but not in our case. Because in our case, page refresh would remove all those DOM changes, we’ve done via JavaScript so far. Thanks to **event.preventDefault()** page refresh is prevented from happening, but also form submit in general is prevented from happening. What does that mean for us? It means, that normal form submit behaviour doesn’t happen at all. Normally, form submit attempts to try to send the form data to the server. That’s why the page is refreshed, because in normal web apps, you send the data to the server, and server redirects you to some other page where the server serves you the data you requested via for example the form.

We’re not sending any data to the server, so it’ not that important for us, but just keep that on mind for your future work.

So, what is happening next? We can also access input fields values via the event parameter. The only thing we need, is to name our inputs, which we’ve done. Our input has name **answerInput**.

Now we need to check, what the player wrote into that input. We can access that via **.value** property that the input has. If the player writes “**yes**”, the functions needed to create a new game are called, modal is closed, and input value is set to an empty string. If the player writes “**no**” just the input is set to an empty string and the modal is closed.

```
const handleSubmitPlayAgain = (event) => {
   event.preventDefault();
   const input = event.target.answerInput;
   if (input.value === "yes") {
      answers.innerHTML = "";
      createAnswerColorsArr();
      renderAnswerOptions();
      renderQuizColor();
      input.value = "";
      handleModalChange(false);
   } else if (input.value === "no") {
      input.value = "";
      handleModalChange(false);
   }
};
```

2.10 Bug fix – clear answerOptions array

Our game works at first, but when we want to play again, we have 8 options instead of desired 4. How so? Let’s look at the code, where we are creating our array. As you can see, we’re only pushing item into an array, but we never clear that array, so when the new game starts, 4 new items are pushed into an already existing array of 4 other items.

```
const createAnswerColorsArr = () => {
   for (let i = 0; i < 4; i++) {
   answerColors.push(generateColor());
   }
};
```

Let’s fix that by creating another function to clear our array. Of course, that you could do this clear out inside **createAnswerColorsArr** but I prefer to have small functions that are doing only one thing.

```
const clearAnswerColorsArr = () => {
   answerColors = [];
};
```

We have another error, because **answerColors** array is a constant, so we can’t reassign new empty array to it, let’s change that.

```
let answerColors = [];
```

Now adjust our form submit function so the array is first set to an empty one and then new items are pushed into it.

```
const handleSubmitPlayAgain = (event) => {
   event.preventDefault();
   const input = event.target.answerInput;
   if (input.value === "yes") {
      answers.innerHTML = "";
      clearAnswerColorsArr(); // added clear function
      createAnswerColorsArr();
      renderAnswerOptions();
      renderQuizColor();
      input.value = "";
```

```
        handleModalChange(false);
    } else if (input.value === "no") {
        input.value = "";
        handleModalChange(false);
    }
};
```

2.11 Bonus – show modal with bounce animation

You can achieve some interesting UI effects with combination of JS and CSS, to make your website just look cooler and improve UX.

At this moment, our game works, but it's just weird to have our modal show up immediately out of nowhere. Let's add an animation, so our Modal slides from the top, and bounces a little at the bottom a little before it settles down.

Many times, these effects are much easier to create that you would've thought.

In our case, all we need to do is two things - create a following css animation and add it to our modal via JavaScript

```
@keyframes bounce {
  0% {
    bottom: 100%;
  }
  25% {
    bottom: 0;
  }
  50% {
    bottom: 100px;
  }
}
```

Now just add it to our modal via JavaScript. In the following code, our **handleModalChange** is improved, to add animation to the modal via inline styles when the modal should be visible and then remove that animation, when the modal should be hidden.

```
const handleModalChange = (show) => {
    if (show) {
        modal.classList.remove("hide");
        modal.style.animation = "bounce 1s";
    } else {
        modal.classList.add("hide");
        modal.style.animation = null;
    }
};
```

You can play with it even more, for example add some animation for modal closing as well. My point is, once you learn JavaScript, you can create some interesting effects, that you would not be able to create without it.

3 Citovaná literatura

1. **© 2005-2020 Mozilla and individual contributors.** Introduction to the DOM. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.
2. **Copyright 1999-2020 by Refsnes Data.** JavaScript HTML DOM. *w3schools.com.* [Online] https://www.w3schools.com/js/js_htmldom.asp.
3. **Aderinokun, Ire.** What, exactly, is the DOM? *bitsofcode.* [Online] 26. November 2018. https://bitsofco.de/what-exactly-is-the-dom/.
4. **Copyright 1999-2020 by Refsnes Data.** HTML DOM getElementById() Method. *w3schools.com.* [Online] https://www.w3schools.com/jsref/met_document_getelementbyid.asp.
5. **© 2005-2020 Mozilla and individual contributors.** EventTarget.addEventListener(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener.
6. **Copyright 1999-2020 by Refsnes Data.** JavaScript HTML DOM EventListener. *w3schools.com.* [Online] https://www.w3schools.com/js/js_htmldom_eventlistener.asp.
7. —. HTML DOM innerHTML Property. *w3schools.com.* [Online] https://www.w3schools.com/jsref/prop_html_innerhtml.asp.
8. **© 2005-2020 Mozilla and individual contributors.** Math.random(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random.
9. —. MDN web docs. *Math.floor().* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/floor.
10. **Copyright 1999-2020 by Refsnes Data.** HTML DOM insertAdjacentHTML() Method. *w3schools.com.* [Online] https://www.w3schools.com/jsref/met_node_insertadjacenthtml.asp.
11. **© 2005-2020 Mozilla and individual contributors.** Element.insertAdjacentHTML(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML.
12. **Copyright 1999-2020 by Refsnes Data.** HTML DOM querySelector() Method . *w3schools.com.* [Online] https://www.w3schools.com/jsref/met_document_queryselector.asp.
13. **lalimijoro.** The DRY principle and why you should use it. *Medium.* [Online] 15. July 2018. https://medium.com/@Ialimijoro/the-dry-principle-and-why-you-should-use-it-f02435ae9449.
14. **Copyright 1999-2020 by Refsnes Data.** preventDefault() Event Method. *w3schools.com.* [Online] https://www.w3schools.com/jsref/event_preventdefault.asp.