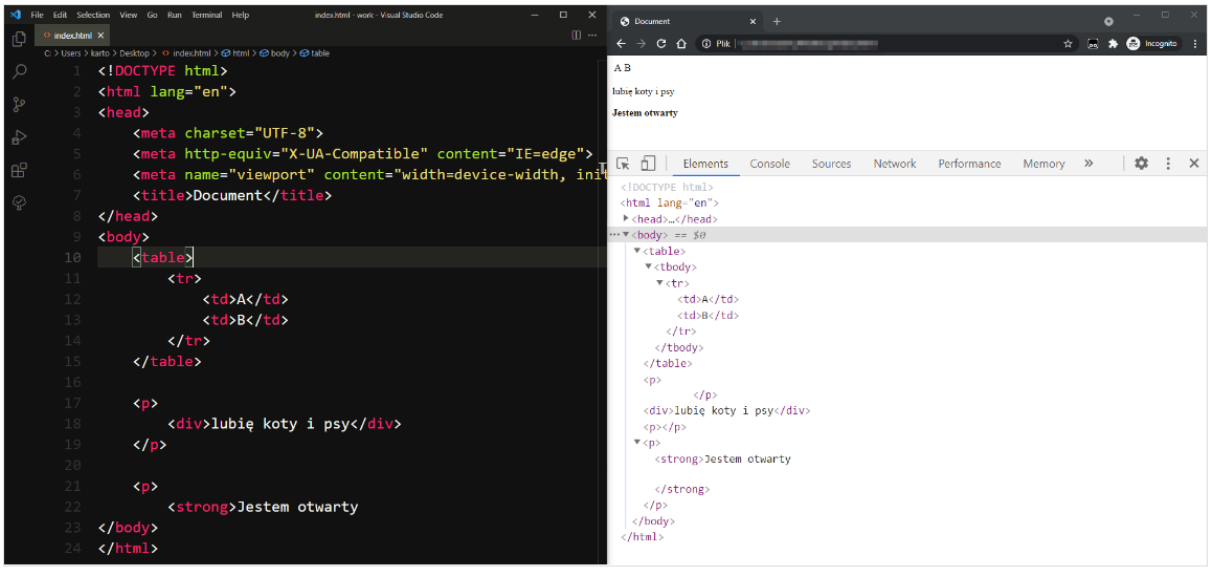
Document Object Model (DOM)

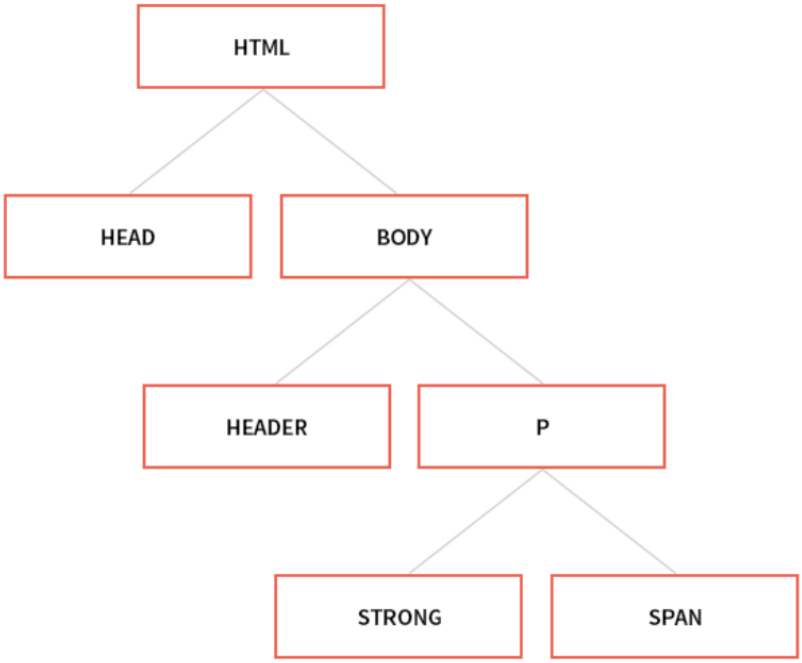
Do odzwierciedlenia struktury elementów na stronie Javascript korzysta z DOM czyli Document Object Model. Model ten opisuje jak zaprezentować tekstowe dokumenty HTML (te, które piszesz sobie jako tekst w edytorze) w postaci modelu obiektowego w pamięci komputera.

Przeglądarka czyta twój plik* jako tekst, a następnie odpowiednio go przetwarza, zamieniając twój zapis na odpowiednie drzewko elementów. Nie oznacza to, że uzyskane drzewo będzie identyczne z tym co zapisałeś w pliku HTML. Spróbuj dla przykładu stworzyć nie do końca prawidłowy HTML. Przeglądarka przeczyta go, przetworzy, a przy okazji zamieni na prawidłowy.



Jak widzisz, przy przetwarzaniu powyższego dokumentu, kilka rzeczy musiało być naprawione. Prawidłowe tabele powinny mieć tbody, wewnątrz elementu p nie może być div, a część elementów musiała zostać zamknięta.

Każdy dokument HTML składa się z różnych elementów. Na górze takiego drzewa znajduje się document (który z kolei znajduje się w oknie przeglądarki - window), a w nim odpowiednie elementy takie jak html, body, paragrafy itp, do których możemy się odwoływać poprzez odpowiednie właściwości i metody.



Będąc w debuggerze przeglądarki, wpisz w konsoli window i naciśnij enter. Jak wiemy, dla Javascriptu odpalanego w przeglądarce jest to główny obiekt.
Gdy rozwiniesz jego właściwości, zobaczysz, że jedną z nich jest **document** (gdy najedziesz na tą właściwość, podświetli się cała strona).
Obiekt ten to interfejs - obiekt, który reprezentuje wyświetlaną w przeglądarce daną stronę, a równocześnie zawierający wiele funkcji i właściwości, dzięki którym możemy dynamicznie taką stronę modyfikować.

Zanim pobierzemy jakąś wartość – małe przypomnienie

Żeby móc odwoływać się do elementów na stronie, powinniśmy mieć pewność, że są one już dostępne dla skryptu, czyli powinny być już wczytane przez przeglądarkę.

W klasycznym podejściu strona czyta HTML od góry do dołu. Jeżeli natrafi na skrypt, wczyta go i uruchomi. Jeżeli w takim skrypcie odwołujemy się do elementów poniżej, to mamy błąd, ponieważ skrypt wykonał się zanim wczytały się elementy.

Można to rozwiązać na minimum trzy sposoby.

- Po pierwsze możemy nasz skrypt umieścić na samym końcu body (tuż przed </body>).
- Po drugie możemy do niego dodać atrybut defer.
- Możemy też użyć zdarzenia DOMContentLoaded, które odpalane jest w momencie wczytaniu całego drzewa DOM.

W każdym razie pamiętaj, że jeżeli przy pierwszych próbach pobrania elementu ze strony dostajesz null, przyczyną wcale nie musi być źle napisany kod, co skrypty znajdujące się na górze kodu strony.

Pobieranie elementów

W dzisiejszych czasach najwygodniej jest pobierać elementy za pomocą metod **querySelector(selectorCss)** i **querySelectorAll(selectorCss)**.

Obie te metody jako parametr wymagają podania w formie tekstu selektora CSS, który normalnie w CSS wskazywał by na dane elementy.

Metody te możemy odpalić zarówno dla całego dokumentu, jak i dla każdego elementu z osobna. W tym drugim przypadku szukamy elementów wewnątrz danego elementu.

```
//w całym dokumencie
const buttons = document.querySelectorAll(".button");

//w .module
const module = document.querySelector(".module");
const buttons = module.querySelectorAll(".button");
```

Metoda querySelector()

zwraca pierwszy pasujący element, lub null, gdy nic nie znajdzie.

```
//pobieramy pierwszy element .btn-primary
const btn = document.querySelector(".btn-primary");

//pobieramy pierwsze li listy ul
const li = document.querySelector("ul li");

//pobieram element, który ma id module
const module = document.querySelector("#module");
```

Metoda querySelectorAll(selector)

ma bardzo podobne działanie, z tą różnicą, że zwraca kolekcję elementów lub pustą kolekcję gdy nic nie znajdzie.

```
const buttons = document.querySelectorAll(".button");

for (const btn of buttons) {
    console.log(btn);
}
```

Inne metody

Elementy na stronie możemy też wyłapywać za pomocą poniższych metod. Nie mają one jednak takich możliwości jak powyższe.

getElementById("id")	pobiera jeden element o danym id
getElementsByTagName("tag-name")	pobiera elementy o danym znaczniku
getElementsByClassName("class-name")	pobiera elementy o danej klasie
getElementsByName("name")	pobiera elementy o danym atrybucie name (w sumie mało użyteczne)

Pętle po kolekcjach

Jeżeli pobieramy pojedynczy element za pomocą querySelector() lub getElementById(), od razu możemy zacząć na nim działać ustawiając mu tekst, style, czy inne właściwości, ale też pobierając w nim inne elementy.

```
const element = document.querySelector("div");
element.style.color = "red"; //ustawiam kolor tekstu na czerwony

const p = element.querySelector("p"); //pobieram w nim paragraf
p.innerText = "Przykładowy tekst";
```

Jeżeli jednak używamy metod do pobrania wielu elementów (np. querySelectorAll) w rezultacie dostajemy kolekcję elementów (nawet jeżeli elementów jest jeden lub wcale). Dla nas oznacza to tyle, że praktycznie zawsze będziemy musieli tutaj działać tak jak na tablicy z obiektami:

```
const elements = document.querySelectorAll(".module");

elements.style.color = "red"; //błąd - bo to kolekcja

elements[0].style.color = "red"; //ok bo pierwszy element w kolekcji

//ok, bo robimy pętlę
for (const el of elements) {
    el.style.color = "red";
}
```

Metod typowych dla tablic takich jak (filter, some, map) nie będziemy mogli tutaj użyć, ponieważ kolekcje przypominają tablice, ale nimi nie są.

```
const elements = document.querySelectorAll(".module");

elements.map(el => el.style.color = "red"); //błąd - map jest dla tablic
```

Możemy to obejść, konwertując taką kolekcję na typową tablicę za pomocą spread syntax lub Array.from():

```
const buttons = document.querySelectorAll("button");

[...buttons].map(el => el.style.color = "red");
```

Wyjątkiem jest tutaj forEach, która została w nowych przeglądarkach (już nie takich nowych) dodana także dla kolekcji zwracanych przez querySelectorAll:

```
const elements = document.querySelectorAll(".module");

elements.forEach(el => {
    el.style.color = "blue"
});
```

Gotowe kolekcje

Nie każdy element na którym będziemy pracować musimy pobierać za pomocą powyższych metod.

Wiele elementów mamy już podstawione pod stosowne zmienne jako właściwości obiektu document:

```
document.body //element body

document.all //kolekcja ze wszystkimi elementami na stronie
document.forms //kolekcja z formularzami na stronie
document.images //kolekcja z grafikami img na stronie
document.links //kolekcja z linkami na stronie
document.anchors //kolekcja z linkami będącymi kotwicami
```

Ale nie są to jedyne kolekcje gotowe do użycia. Gdy dla przykładu pobierzemy ze strony jakiś formularz i wypiszemy go w konsoli, okaże się że, obiekt taki też ma swoje gotowe kolekcje np. elements, która zawiera wszystkie elementy formularza. Podobnie element select, który zawiera właściwość options, która zawiera kolekcję elementów options danego selekta.

Czy musisz je znać? Nie. Zawsze możesz pobrać dane elementy korzystając np. z querySelector, przy czym czasami warto sobie skrócić swój kod korzystając z gotowców - szczególnie gdy prowadzisz zajęcia, a kursanci zaczynają się nudzić...

Żywe kolekcje

W większości przypadków najlepszym wyborem do pobierania elementów będą metody querySelector/querySelectorAll. Dzięki temu, że jako parametr podajemy selektor CSS, mamy tutaj o wiele większe możliwości niż w przypadku starszych metod. Ale nie tylko większe możliwości oddzielają tą metodę od ich starszych sióstr.

Poniżej stworzyłem diva z kilkoma elementami span. Po kliknięciu na przycisk pobieram te spany na dwa sposoby: za pomocą querySelectorAll() i getElementsByTagName(), po czym wypisuję je w konsoli.

```html
<div class="test">
    <span>1</span>
    <span>2</span>
    <span>3</span>
    <span>4</span>
    <span>5</span>
</div>
```

```js
const parent = document.querySelector(".test");
const spanGroupA = parent.querySelectorAll("span");
const spanGroupB = parent.getElementsByTagName("span");

console.log("querySelectorAll: ", spanGroupA);
console.log("getElementsByTagName: ", spanGroupB);
```

Jak zauważysz niby zwracane są podobne elementy, ale pod postacią innych struktur. I tak querySelectorAll() zwraca **NodeList**,
natomiast getElementsByTagName() zwraca **HTMLCollection**.

Ten drugi rodzaj zwracany jest też przez inne metody takiej jak getElementsBy*COŚ* ale także przez właściwości, które poznamy przy poruszaniu się po drzewie.

HTMLCollection różnią się od NodeList tym, że nie pozwalają używać na sobie forEach() (chyba, że skonwertujemy je na tablicę), ale przede wszystkim tym, że zawierają tak zwane żywe kolekcje, które na bieżąco odzwierciedlają stan html.

Przykładowo pobierasz spany w danym elemencie za pomocą querySelectorAll i getElementsByTagName. Jeżeli w przyszłości liczba tych spanów się zmieni (dojdą nowe lub zostaną usunięte), kolekcja pobrana za pomocą getElementsByTagName automatycznie się zaktualizuje, natomiast querySelectorAll będzie odzwierciedlać stan z momentu pobrania.

Odpowiednie testy możesz zobaczyć na tej stronie.