**JavaScript zaawansowany**

**DOM - zarządzanie**

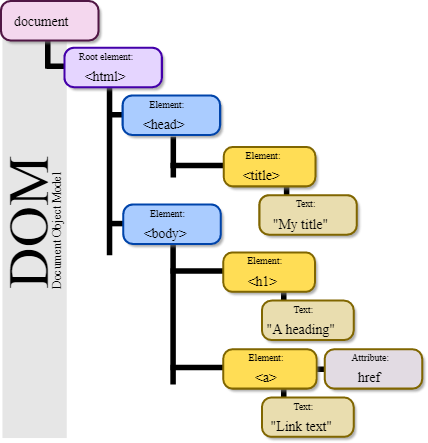
**Wstęp**

W szkoleniu tym poznasz bardzo praktyczne zagadnienia, z których będziesz nieustannie korzystał podczas tworzenia kolejnych skryptów   
w JS. Poniżej skupimy się bowiem na zarządzaniu **DOM**, czyli **Document Object Model**. Jednym słowem, poznamy sposoby na zmianę wyglądu strony “on fly” - w czasie rzeczywistym.

**Czym jest DOM?**

**DOM** jest on wykorzystywany przez JavaScript, aby możliwe było odzwierciedlenie struktury elementów na stronie. Model ten opisuje jak zaprezentować tekstowe dokumenty HTML (mowa o tych, które piszesz jako tekst w edytorze) w postaci modelu obiektowego w pamięci komputera.

Każdy dokument HTML związany jest z indywidualnie dostosowanym drzewem elementów i elementami określonymi. 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 pola i metody.



Źródło: wikipedia

Podczas renderowania strony, kod HTML czytany jest od góry do dołu. Dlatego żeby móc poprawnie odwoływać się do elementów na stronie, powinniśmy mieć pewność, że są one już dostępne dla skryptu JS. Wykonanie takiego skryptu przed wygenerowaniem elementów HTML,   
na których to miałby on manipulować, skutkowałoby brakiem jakichkolwiek zmian.

Wspominam Ci o tym, ponieważ w szkoleniu tym możesz spotkać się   
z rozwiązaniem, w którym kod JS będziemy umieszczać tuż przed znacznikiem **</body>** lub wraz z atrybutem **defer**. Możemy też użyć zdarzenia DOMContentLoaded lub window.onLoad które jest wczytane   
w momencie wczytaniu całego drzewa DOM.

**Pobieranie elementów**

Aby móc przystąpić do manipulowania elementami na stronie, musimy najpierw uzyskać do nich dostęp. Dowolne elementy ze strony HTML najwygodniej jest pobierać za pomocą metody **querySelector(selectorCss)** oraz **querySelectorAll(selectorCss)**.

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

Metoda querySelector() zwraca pierwszy pasujący do selektora element, lub null, gdy nic nie znajdzie.

|  |
| --- |
| //pobieramy pierwszy element .student-info const btn = document.querySelector(".student-info");  //pobieramy pierwsze li listy ul const li = document.querySelector("ul li");  //pobieram element, który ma id mentors const module = document.querySelector("#mentors"); |

Druga metoda querySelectorAll(selector) ma działanie podobne do querySelector(), z tą różnicą, że zwraca kolekcję elementów dopasowanych do określonego selektora. Gdy żaden z elementów nie zostanie znaleziony, zwracana jest pusta kolekcja.

|  |
| --- |
| const buttons = document.querySelectorAll(".students");  for (const student of students) {  console.log(student); } |

Metody te możemy wywoływać nie tylko na obiekcie document, ale również na każdym pobranym elemencie z osobna. Taka możliwość jest naprawdę przydatna, gdy chcemy pobrać zagnieżdżone elementy.

|  |
| --- |
| // w całym dokumencie const buttons = document.querySelectorAll(".students");  // w .students const students = document.querySelector(".students"); const buttons = students.querySelectorAll(".skills"); |

Elementy na stronie możemy również pobierać, wykorzystując poniższe metody:

* **getElementById("id")** pobiera jeden element o danym id,
* **getElementsByClassName("class-name")** pobiera elementy o danej klasie
* **getElementsByTagName("tag-name")** pobiera elementy o danym znaczniku
* **getElementsByName("name")** pobiera elementy o danym atrybucie name (niespecjalnie użyteczne)

Jednak wykorzystanie każdego z nich może zostać zastąpione przez querySelectora, dlatego zalecałbym nie wykorzystywać wyżej przedstawionych sposobów.

**Modyfikacja pobranego elementu**

Po pobraniu elementu, możemy od razu przystąpić do modyfikowania jego zawartości i wyglądu. Możemy ustawiać mu tekst, zmieniać kolor czcionki, tła itd.

Poniżej przykład modyfikacji dowolnego diva oraz selektora p:

|  |
| --- |
| const element = document.querySelector("div"); element.style.color = "red";   const p = element.querySelector("p");  p.innerText = "Przykładowy tekst"; |

Pierwszy snippet kodu skutkuje zmianą koloru tekstu znajdującego się   
w pobranym divie, natomiast drugi fragment kodu spowoduje zastąpienie tekstu umieszczonego w paragrafie na “Przykładowy tekst”. Pamiętaj, że querySelector() powoduje pobranie pierwszego od góry elementu (który spełnia określony warunek).

Jeżeli jednak używamy metody do pobrania wielu elementów (np. **querySelectorAll**), to otrzymamy kolejkę elementów. Implikuje to tyle,   
że chcąc zmodyfikować pobrane elementy, będziemy musieli odwoływać się do nich za pomocą indeksów lub pętli.

|  |
| --- |
| **UWAGA!**  Kolekcja to nie tablica! Jest to obiekt przypominający pod kątem funkcjonalności tablicę, jednak nie możemy wywoływać na nim choćby takich metod jak: **filter**, **map**, **reduce**.  Jeżeli jednak będzie nam zależeć na zamianie takiej kolekcji na tablicę, to możemy wykorzystać ją przekonwertować za pomocą [spread syntax](https://kursjs.pl/kurs/es6/spread-rest.php) lub [Array.from()](https://kursjs.pl/kurs/super-podstawy/tablice.php#Array.from). |

Przykład operowania na elementach kolekcji:

|  |
| --- |
| const students = document.querySelectorAll(".students");  students.style.color = "red"; // błąd - bo to kolekcja  students[0].style.color = "red"; // zmieniamy kolor pierwszego elementu   // poniżej zamieniamy kolor każdego elementu o divie "students"  for (const student of students) {  student.style.color = "blue"; }  // lub możemy wykonać to samo, wykorzystując .forEach()  const students = document.querySelectorAll(".module");  students.forEach(student => {  student.style.color = "blue" }); |

Poniżej zaprezentuję Ci rozwiązanie, o którym już wspomniałem, czyli przekształcenie kolekcji w listę, a następnie wywołanie, na każdym elemencie, metody map():

|  |
| --- |
| const students = document.querySelectorAll(".students");  [...students].map(student => student.style.color = "blue"); |

**Gotowe kolekcje**

Aby odwoływać się do niektórych elementów z DOM, nie musimy wcale wykorzystywać querySelector(). Wiele elementów mamy już podstawione pod stosowne zmienne i możemy się do nich bez problemu odwoływać.   
Są to głównie grupy, w których umieszczone są wszystkie elementy, obrazy, formularze, linki - znajdujące się na stronie.

Poniżej kilka przykładowych grup, do których możemy się odwoływać, chcąc zarządzać określonymi elementami na stronie.

|  |
| --- |
| document.domain // domena dokumentu  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 |

**Selektor :scope**

Rozważ poniższy fragment kodu HTML:

|  |
| --- |
| <**div** class="students">  <**div**>  <**div** class="student">George</**div**>  </**div**> </**div**> |

Pobierzmy element .students.

|  |
| --- |
| const students = document.querySelector('.students'); |

Teraz chcielibyśmy wyszukać element w jego wnętrzu. Czyli:

|  |
| --- |
| const students = document.querySelector('.students'); const student = students.querySelector('div > div > div'); |

Jaki wynik będzie przechowywała stała student?

W teorii, stała ta nie powinna wskazywać na żadną wartość, ponieważ   
wewnątrz div students nie przechowujemy czterech selektorów <div>. Okazuje się jednak, że będzie ona reprezentowała diva o klasie student.

Dzieje się tak, ponieważ element, na którym wywołaliśmy metodę querySelector również jest divem. Dlatego też zapis ‘div > div > div’ rozpoczyna przeszukiwanie od klasy .students, później przechodzi do selektora <div> (tego bez klasy) i na końcu trafia na .student. Aby więc jasno określić, że przeszukujemy elementów należących tylko   
i wyłącznie do danego selektora, nie wliczając do poszukiwań jego samego, wykorzystamy :scope.

Bądź jednak świadomy tego, iż :scope nie działa na każdej z przeglądarek. Alternatywą dla omawianego selektora może być nadanie odpowiednich klas lub id-ków dla danych elementów i uzyskiwanie dostępu za ich pośrednictwem.

Poniższy querySelector() odwołuje się do nieistniejącego trzeciego diva wewnątrz klasy .students.

|  |
| --- |
| const students = document.querySelector('.students'); const notExistingElem = students.querySelector(:scope div div div'); // tutaj już odwołujemy się do nieistniejącego elementu |

**Przechodzenie po drzewie DOM**

Powyżej poznałeś jeden z popularniejszych sposobów na odwoływanie się do określonych elementów w DOM-ie. Jednak istnieje również szereg innych metod, których znajomość może się przydać, operując na elementach strony. Są to:

|  |  |
| --- | --- |
| **Rodzic**:   * parentNode * parentElement   **Dziecko:**   * childNodes * children * firstChild * firstElementChild * lastChild | **Elementy spokrewnione:**   * nextSibling * nextElementSibling * previousSibling * previousElementSibling |

Aby dobrze zrozumieć i przede wszystkim zapamiętać działanie powyższych metod, przetestujemy ich działanie w praktyce, na żywym kodzie HTML.

**Szablon HTML**

|  |
| --- |
| <**div** class="container">  <**div** id="main">  <**h2**>Add Student</**h2**>  <**form**>  <**input** type="text">  <**input** type="submit">  </**form**>  <**h2**>Students</**h2**>  <**ul** id="students">  <**li**>Student 1</**li**>  <**li**>Student 2</**li**>  <**li**>Student 3</**li**>  </**ul**>  </**div**> </**div**> |

1. **ParentNode**

Umożliwia pobranie rodzica danego elementu.

**Kod**

|  |
| --- |
| let studentsList = document.querySelector('#students'); console.log(studentsList.parentNode); |

**Rezultat**

Elementem parentNode dla studentsList jest div o id=”main”.



1. **ParentElement**

ParentElement ma takie samo działanie jak parentNode. Jedyną różnicą jest fakt, iż właśnie omawiana metoda zwraca null, gdy próbujemy odwołać się do rodzica, który nie istnieje.

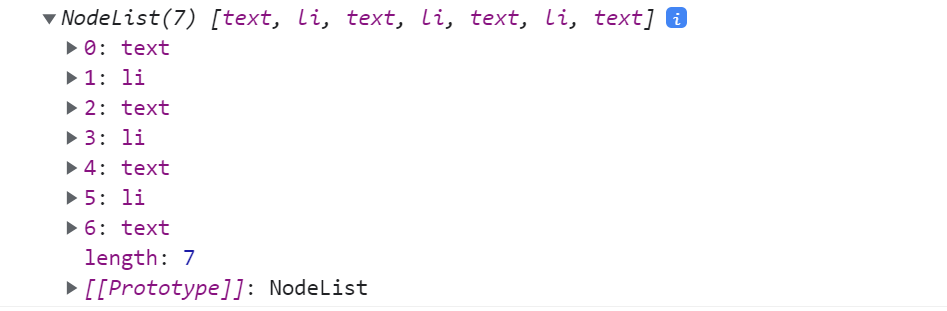
|  |
| --- |
| // zwraca document node document.documentElement.parentNode;   // zwraca null document.documentElement.parentElement; |

1. **ChildNodes**

Metoda ta umożliwia pobranie wszystkich elementów (dzieci) zagnieżdżonych w danym selektorze.

|  |
| --- |
| let studentsList = document.querySelector('#students'); console.log(studentsList.childNodes); |

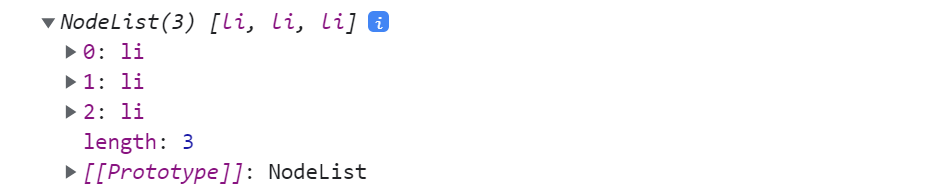
**Rezultat**



Zauważ, że na pobranej liście NodeList znajdują się również elementy   
o nazwie text. Reprezentują one przerwy między kolejnymi selektorami <li>. Poważnie, nie żartuję! Spójrz, jak zachowa się childNodes(), gdy usuniemy odstępy między kolejnymi znacznikami.

|  |
| --- |
| <**div** class="container">  <**div** id="main">  <**h2**>Add Student</**h2**>  <**form**>  <**input** type="text">  <**input** type="submit">  </**form**>  <**h2**>Students</**h2**>  <**ul** id="students"><**li**>Student 1</**li**><**li**>Student 2</**li**><**li**>Student 3</**li**></**ul**>  </**div**> </**div**> |

**Rezultat:**



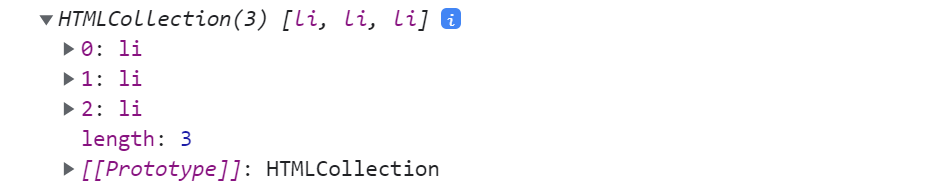
Przyznaj, że takie zachowanie omawianej funkcjonalności (czyli pobieranie również odstępów między dziećmi) może być sporym utrapieniem   
w zarządzaniu drzewem DOM. Dlatego też zalecam korzystać   
z alternatywnego rozwiązania - metody .children().

1. **Children**

Wróćmy do kodu HTML, na którym działamy od początku tej sekcji (ten, który zawierał przerwy między kolejnymi elementami listy porządkowej). Sprawdźmy, jak zachowa się atrybut children.

|  |
| --- |
| let studentsList = document.querySelector('#students'); console.log(studentsList.children); |

**Rezultat**



Jak widzisz, od razu pobiera on interesujące nas elementy, bez konieczności usuwania białych znaków między <li>. O wiele wygodniejszy   
i efektywniejszy sposób od używania childNodes.

1. **FirstChild oraz FirstElementChild**

Powiedzmy sobie wprost - osobiście uważam, że firstChild jest, tak jak childNodes, niezbyt przydatnym rozwiązaniem. Jest on równoważny zapisowi studentsList.childNodes[0]. ChildNodes pobiera odstępy między selektorami, tak więc użycie firstChild może często kończyć się zwróceniem ujrzanego już przez Ciebie elementu text.

**Kod**

|  |
| --- |
| let studentsList = document.querySelector('#students'); console.log(studentsList.firstChild); |

**Rezultat**



Dlatego też w większości przypadków bardziej sprawdzi się wykorzystanie firstElementChild, który to jest odpowiednikiem zapisu: studentsList.children[0].

1. **LastChild oraz lastElementChild**

Wymienione właściwości działają podobnie do firstChild oraz firstElementChild, z tą różnicą, że pobierają nie pierwsze, ale ostatnie dziecko należące do danego selektora. Znowu - sytuacja tutaj przedstawiona jest podobna do już omówionej - lastChild może zwrócić nam pusty tekst (będący po prostu odstępem między selektorami),   
a wykorzystując lastElementChild jesteśmy pewni, że nie otrzymamy żadnego niepożądanego elementu.

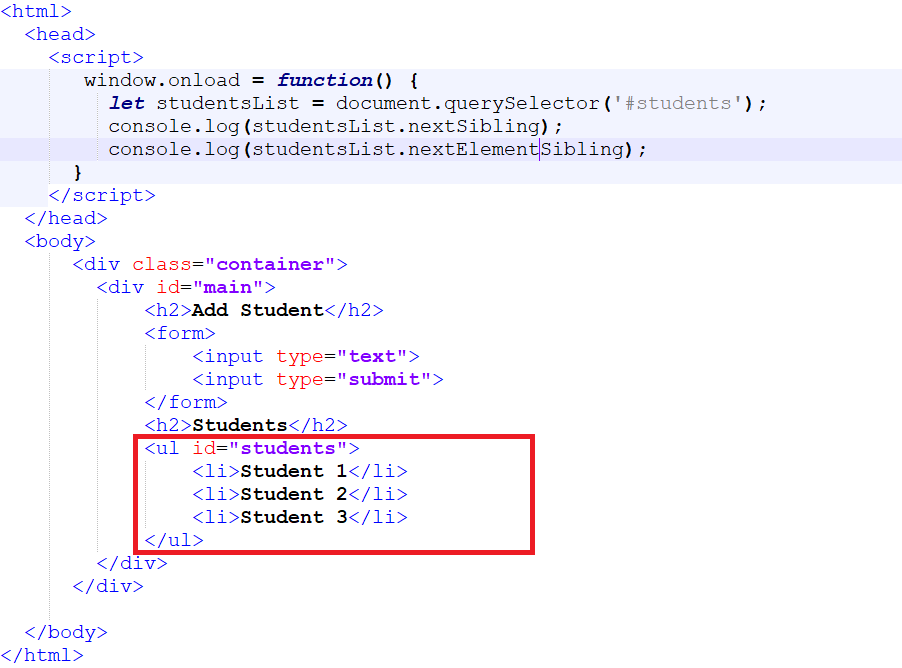
1. **NextSibling** oraz **nextElementSibling**

Atrybuty te umożliwiają pobranie sąsiadujących elementów w stosunku   
do wybranego. Tutaj, jak już się zapewne domyślasz, zachowanie jest identyczne do wcześniej omówionych selektorów. NextSibling pobiera również elementy reprezentujące białe znaki między selektorami, natomiast nextElementSibling - nie. Przez sibling rozumiemy element sąsiadujący   
i znajdujący się na tym samym poziomie co ten, do którego się odwołujemy.

|  |
| --- |
| let studentsList = document.querySelector('#students'); console.log(studentsList.nextSibling);  console.log(studentsList.nextElementSibling); |

**Rezultat**

Zauważ, że nextElementSibling zwrócił typ null. Wynika to z tego, iż po selektorze <ul> nie występuje żaden inny element HTML-a (jedynie zamykamy wcześniej otwartego diva “main”).



<ul> jest ostatnim selektorem w drzewie HTML

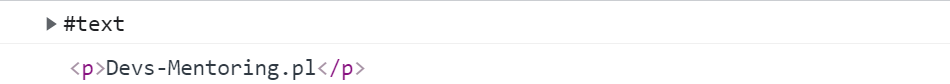
Jeżeli natomiast dodamy poniżej listy nieuporządkowanej nowy selektor, np. <p></p>. Wówczas będziemy mogli się do niego odwołać dzięki wykorzystaniu **nextElementSibling**.



Nowo dodany paragraf

|  |
| --- |
| let studentsList = document.querySelector('#students'); console.log(studentsList.nextSibling);  console.log(studentsList.nextElementSibling); |

**Rezultat**



1. **PreviousSibling** oraz **previousElementSibling**

Po zapoznaniu się z powyższą sekcją, podejrzewam, że te metody nie wymagają szczegółowego objaśnienia. Sytuacja jest niemal identyczna do nextSibling oraz nextElementSibling, z tą różnicą, że za pomocą omawianych metod odwołujemy się do elementów poprzedzających określony selektor.

**Dynamiczne tworzenie elementów**

Powyżej poznałeś sposoby na pobieranie i modyfikowanie istniejących selektorów. Nieodłącznym aspektem związanym z zarządzaniem drzewem DOM będzie również tworzenie nowych elementów w strukturze strony.   
Do tego celu posłuży nam metoda .createElement. Oprócz niej będziemy też niejednokrotnie wykorzystywali metody pomocnicze, np. createElement, createTextNode.

Nie ma jednego szablonu, dlatego najlepiej będzie przeanalizować wykorzystanie wspomnianych rozwiązań w praktyce. Poniżej przykład kodu, w którym dynamicznie zarządzamy numeryczną listą elementów.

|  |
| --- |
| <**html**>  <**head**>  <**script**>  window.onload = function() {  const students = document.querySelector('#mentor-Adam');  let newStudent = document.createElement("li"); newStudent.appendChild(document.createTextNode("Kacper"));  students.appendChild(newStudent);  }  </**script**>  </**head**>  <**body**>  <**ol** id="mentor-Adam">  <**li**>Piotr</**li**>  <**li**>Rafał</**li**>  <**li**>Justyna</**li**>  </**ol**>  </**body**> </**html**> |

Flow przeprowadzonych operacji w JS:

* Pobierana jest lista elementów (ol), do której dodamy nową selektor (jej id to **#mentor-Adam**).
* Tworzymy nowy element <li> - document.createElement(“li”)
* Do tak utworzonego obiektu, dodajemy node tekstowy - “Kacper”
* Na końcu umieszczamy utworzony selektor <li> wewnątrz listy porządkowej <ol> (przechowywanej pod zmienną students)

W powyższym przykładzie użyliśmy pewnych charakterystycznych metod, które umożliwiają nam sprawne zarządzanie drzewem DOM. Są to:

* **createElement** - metoda ta tworzy w dokumencie HTML nowy element określony przez *tagName* wysłany jako argument. W naszym przypadku był to selektor “li”.
* **createTextNode** - tworzy nowy tekst, jako argument przyjmuje treść tworzonego tekstu
* **appendChild** - dodaje nowy element (dziecko) do końca listy danego selektora (rodzica).

**AddEventListener**

Na zakończenie, zapoznamy się z pojęciem addEventListenera, który umożliwia ustawianie listenera na dany element w kodzie i dzięki temu wykonaniu określonych działań po wykonaniu pewnej czynności.

Przykładem może być sytuacja, gdzie potrzebujemy wykrywać kliknięcia użytkownika na dany obiekt HTML i wykonywać określone kroki, jak np. wywołanie odpowiedniej funkcji.

JavaScript dostarcza nam event handler, który umożliwia realizację wyżej przedstawionych założeń. Jest nim metoda addEventListener(). Może być ona “podpięta” pod określony element HTML, dla którego chcesz monitorować danye eventy.

**Syntax**

Poniżej syntax wspomnianej metody addEventListener():

target.addEventListener(event, function, useCapture: optional)

**-** **target:** element HTML, do którego chcemy dodać handler

**- event:** string, który określa nazwę eventu, np. **load** lub **click**. Pozostałe eventy, które możesz wykorzystać w kodzie znajdziesz tutaj: [HTML DOM Event Object](https://www.w3schools.com/jsref/dom_obj_event.asp)

**- function:** tutaj określasz funkcję, która ma być wywoływana, gdy event jest wykryty. To właśnie w takiej funkcji będziesz określał, jakie dynamiczne czynności mają się wykonywać

**- useCapture:** opcjonalna wartość typu Boolean, która określa, czy event powinien być wykonywany w formie “capture” czy “bubbling phase”. Domyślnie ustawiany jest na false.

|  |
| --- |
| **Bubbling phase** to sytuacja, gdzie event wywoływany na danym elemencie HTML, powoduje najpierw wywołanie na nim danej funkcji, a następnie na jego rodzicach i innych przodkach.  **Capture mode** jest natomiast sytuacją odwrotną, najpierw wywoływane są eventy z elementów HTML będących rodzicami klikniętej pozycji, |

an optional Boolean value (true or false) that specifies whether the event should be executed in the capturing or bubbling phase. In the case of nested HTML elements (such as an img within a div) with attached event handlers, this value determines which event gets executed first. By default, it's set to false which means that the innermost HTML event handler is executed first (bubbling phase).

**Prosty przykład wykorzystanie addEventListenera**

Aby lepiej zapamiętać poznane dotąd zagadnienia, przejdźmy do praktycznego wykorzystanie omawianej metody. Zaprezentuję Ci, w jaki sposób ustawić najprostszy listener na przycisk w kodzie.

Naszym celem jest napisanie skryptu, który będzie wyświetlał wiadomość przez console.log po kliknięciu danego przycisku.

|  |
| --- |
| <!DOCTYPE html> <**html**> <**body**>  <**button** id="test\_btn">Click here!</**button**>   <**h1** id="test\_text"></**h1**>  <**script**> document.getElementById("test\_btn").addEventListener("click", function(){  document.getElementById("test\_text").innerText = "Devs-Mentoring.pl"; } );  </**script**> </**body**> </**html**> |

Zgodnie z syntaxem, który przedstawiliśmy powyżej:

* **target** to element HTML o ID test\_btn
* **function** to funkcja anonimowa, która odpowiada za dynamiczną zmianę zawartości nagłówka test\_text
* flaga **useCapture** jest ustawiana domyślnie na false

**Przesyłanie eventu jako argumentu**

Nierzadko będziemy potrzebowali sprawdzić więcej informacji o uruchomionym event-cie w ramach metody addEventListener, jak np. który element został kliknięty.

W takim wypadku wystarczy odpowiednio sparametryzować funkcję anonimową (lub funkcję strzałkową).

|  |
| --- |
| button.addEventListener('click', (e) => {  console.log(e.target.id) }); |

Jak widzisz, umieściliśmy na listę parametrów zmienną, która reprezentuje obiekt eventu. Dzięki temu możemy z takiego obiektu odpowiednio korzystać w obrębie wywoływanej, po nastąpieniu eventu, funkcji. My akurat wyświetlamy w niej ID wciśnięto buttona.

**Usuwanie event handlera**

Jeżeli z jakiegoś powodu, nagle nie chcesz już w programie nasłuchiwać eventów na danym elemencie HTML, możesz wykorzystać metodę target.removeEventListener(event, function, capture);

Argumenty takiego wywołania metody muszą być identyczne do tych, które podałeś przy wykorzystaniu metody addEventListener.