JavaScript

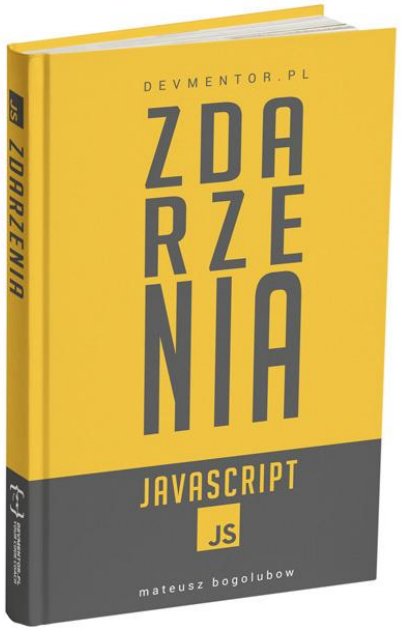
ZDARZENIA

Zdarzenia (ang. events), a w zasadzie ich obsługa to niezbędny element każdej strony internetowej.

Czy nie chciałbyś umieć reagować na działania użytkownika? Pokazać element, gdy zostanie kliknięty przycisk powyżej lub powiększyć obraz, gdy użytkownik najedzie na niego kursorem?

Umiejętność reagowania na akcję użytkownika jest niezbędna do zbudowania nowoczesnej strony internetowej.

Dzięki temu materiałowi będziesz wiedział jak tego dokonać!



Zadania do wykonania

Pod poniższym adresem znajdziesz zadania,
które obejmują materiał prezentowany w niniejszym ebook-u.

https://github.com/devmentor-pl/practice-js-events

01. Wprowadzenie

Zdarzenia (ang. events) to czynności, które są wykonywane na naszej stronie internetowej.

Każda taka czynność jest rejestrowana przez przeglądarkę. Następnie za pomocą JS możemy nasłuchiwać tych zdarzeń i na nie reagować.

Kliknięcie w przycisk, najechanie myszką na obrazek, zmiana danych w polu formularza czy po prostu wciśnięcie klawisza na klawiaturze jest event-em!

```
const btnEl = document.querySelector('.btn');
if(btnEl) {
    // jeśli został wyszukany

    btnEl.addEventListener('click', function() {
        // wywowaj tą funkcję jeśli
        // wyszukany przycisk
        // zostanie kliknięty

        console.log('button was clicked');
        // wyświetl napis w konsoli
    });
}
```

To funkcja określa jaka czynność ma zostać wykonana.

W poprzednim przykładzie użyliśmy funkcji anonimowej tj. bez nazwy.

Natomiast nic nie stoi na przeszkodzie, aby wykorzystać funkcję nazwaną lub wyrażenie funkcyjne jak wydać to na przykładzie obok.

```
const handleClick = function() {
    console.log('button was clicked');
}
// utworzyłem wyrażenie funkcyjne
// tj. funkcja została przypisane
// do zadeklarowanej zmiennej

const btnEl = document.querySelector('.btn');
if(btnEl) {
    // jeśli został wyszukany element...

    btnEl.addEventListener('click', handleClick);
    // nasłuchuj kliknięcia na ten element
    // tz. wywołaj funkcję handleClick jeśli
    // wyszukany przycisk zostanie kliknięty
}
```

#01 Nasłuchiwanie zdarzeń

01. Wprowadzenie

Aby nasłuchiwać zdarzenia należy wskazać na jakim elemencie ma mieć miejsce event.

Jeśli nie jest to obiekt document czy window to musimy wyszukać dany element, a potem wywołać na nim metodę .addEventListener().

W najprostszej formie metoda ta przyjmuje dwa parametry tj. typ zdarzenia oraz funkcję, która ma być wywołana.

```
const btnEl = document.querySelector('.btn');
if(btnEl) {
    // jeśli został wyszukany element...

    btnEl.addEventListener('click', function() {
        console.log('button was clicked');
    });
    // nasłuchuj kliknięcia na ten element
    // tz. wywołaj funkcję anonimową (bez nazwy)
    // jeśli wyszukany przycisk zostanie kliknięty
}
```

Mimo, że poprzedni kod działa to zdecydowanie lepszym rozwiązaniem jest utworzenie funkcji w taki sposób, aby można było się po nazwie do niej odwołać.

To pozwoli nam mieć większą kontrolę nad kodem - będziemy mogli w każdej chwili usunąć nasłuchiwanie.

Dodatkowo kod jest bardziej czytelny i mamy możliwość wykorzystać go wielokrotnie.

```
const handleClick = function() {
    console.log('button was clicked');
}
// utworzyłem wyrażenie funkcyjne

const btnEl = document.querySelector('.btn');
if(btnEl) {
    // jeśli został wyszukany element...

    btnEl.addEventListener('click', handleClick);
    // wywołaj funkcję handleClick jeśli
    // wyszukany przycisk zostanie kliknięty
    // zauważ, że tutaj funkcji nie wywołuje
    // przekazuje jedynie jej nazwę
    // dopiero w odpowiednim momencie
    // funkcja ta zostanie wywołana przez JS
}
```

Ilość dostępnych nazw zdarzeń jest bardzo duża. Każdy odpowiada za inną akcję.

Jeśli chcesz poznać wszystkie event-y to zapraszam na stronę MDN.

Na początek wystarczy nam znajomość kilku z nich.

W razie potrzeby zawsze możesz przeszukać listę dostępnych zdarzeń i sprawdzić czy nie ma takiego, który pomoże zrealizować Twoje zadanie.

- **click** - kliknięcie
- **mouseenter** - najechanie kursorem
- **mouseleave** - zjechanie kursorem
- **mousemove** - poruszanie się kursorem
- **change** - zmiana wartości dla pola formularza
- **keyup** - wciśnięcie klawisza na klawiaturze (dokładniej puszczenie)
- **submit** - wysłanie formularze
- **DOMContentLoaded** - załadowanie całego drzewa DOM

Zaczniemy od DOMContentLoaded, który często jest wykorzystywany do zabezpieczenia się przed wyszukiwaniem elementów, które jeszcze nie zostały załadowane do DOM-u.

Jeśli nasz plik JS podepniemy w sekcji <head> to przeglądarka najpierw będzie interpretować kod JS (więc rozpocznie wyszukiwanie), dopiero potem będzie ładować elementy do drzewa DOM.

W takim wypadku każde wyszukiwanie elementu będzie zwracać null.

```
const contentEl = document.querySelector('.content')
// Jeśli plik JS jest podpięty do sekcje <head/>
// to zmienna [contentEl] będzie zawierać [null].

const init = function() {
    const contentEl = document.querySelector('.content');
    // Jeśli wyszukanie nastąpi po załadowaniu DOM
    // to element zostanie odnaleziony prawidłowo.
    // Wykorzystując DOMContentLoaded miejsce
    // podpięcia pliku JS nie ma już znaczenia!
}

document.addEventListener('DOMContentLoaded', init);
// nasłuchiwanie na obiekcie document lub window
```

Na tym samym elemencie możemy stworzyć nasłuchiwanie dla kilku różnych typów eventów.

```html
<section>
    <button class="btn">click me</button>
    <p>move cursor on button</p>
</section>
```

```js
const btnEl = document.querySelector('.btn');
const textEl = document.querySelector('.text');

const onEnter = function() {
    textEl.textContent = 'enter...';
}

const onLeave = function() {
    textEl.textContent = 'leave...';
}

if(btnEl && textEl) {
    btnEl.addEventListener('mouseenter', onEnter);
    btnEl.addEventListener('mouseleave', onLeave);
}
```

Jeśli chcemy pobierać z pola formularza dane (jeszcze przed wysłaniem formularza!) możemy to zrobić za pomocą event-u keyup.

```html
<section>
    <input class="email" />
</section>
```

```js
const emailEl = document.querySelector('.email');

const getValue = function() {
    console.log(emailEl.value);
    // wyświetl aktualną zawartość pola
}

emailEl && emailEl.addEventListener('keyup', getValue);
// operator logiczny && pozwoli wykonać
// operację po prawej stronie
// tylko wtedy, gdy lewa strona będzie prawdziwa
```

Możemy także, uruchomić wskazaną funkcję dopiero, gdy pole formularza (w tym przypadku <select>) zmieni swoją zawartość.

```
<select name="user">
    <option value="1">Jan Kowalski</option>
    <option value="2">Bartek Adamczyk</option>
    <option value="3">Wojtek Polakowski</option>
</select>
```

```
const userEl = document.querySelector('[name="user"]');
// wyszukaj element, który posiada
// atrybut [name] ustawiony na [user]
const showCurrentUserId = function() {
    console.log( userEl.value );
    // pobieramy zawartość atrybutu [value]
    // ustawionego dla wybranego <option/>
}
if(userEl) {
    userEl.addEventListener(
        'change',
        showCurrentUserId,
    );
    // wywołaj metodę przypisaną do [showCurrentUserId]
    // gdy wybrany element z listy
    // będzie inny niż poprzedni
}
```

#02 Usuwanie nasłuchiwania

01. Wprowadzenie

Jeśli potrzebuje wykonać ograniczoną ilość razy daną akcję związaną ze zdarzeniem to musisz usunąć nasłuchiwanie.

Można to zrobić za pomocą metody `.removeEventListener()`, do której musimy przekazać nazwę event-u do usunięcia oraz funkcję.

```
const counterEl = document.querySelector('.counter');
const incrementCounter = function() {
    let value = parseInt(counterEl.innerText);
    // pobieram tekst z przycisku
    // oraz zamieniam go na liczbę
    counterEl.innerText = ++value;
    // przypisuje nową wartość
    // powiększając ją o 1
    if( value > 3 ) {
        counterEl.removeEventListener(
            'click', incrementCounter
        );
        // usuń nasłuchiwanie, gdy
        // [value] przekroczy wartość 3
    }
}
counterEl.addEventListener('click', incrementCounter);
```

Dlatego już na samym początku zasugerowałem, aby nie używać funkcji anonimowych.

Choć jest to rozwiązanie wygodne w użyciu (w szczególności, gdy używamy tzw. funkcji strzałkowej - o niej więcej będziemy mówić przy ES6) to niesie ze sobą dość spore ograniczenia.

```
const btnEl = document.querySelector('button');

if(btnEl) {
    btnEl.addEventListener('click', function() {
        btnEl.innerText -= 1;

        // nie mam jak usunąć tego nasłuchiwania
        // ponieważ nie mam nazwy funkcji
        // którą muszę przekazać przy
        // .removeEventListener('click', [nazwa fn])
    });
}
```

02. Propagacja zdarzeń

Propagacja to rozchodzenie się zdarzeń na kolejne elementy.

Po kliknięciu w `<button>` zdarzenie również zostanie wywołane na `<section>`

```
<section>
    <button>click me</button>
</section>
```

```
const btnEl = document.querySelector('button');
const sectionEl = document.querySelector('section');
if(btnEl && sectionEl) {
    btnEl.addEventListener('click', function() {
        console.log('button was clicked');
    });

    sectionEl.addEventListener('click', function() {
        console.log('div was clicked too!');
        // zauważ, że klikając w <button />
        // równocześnie klikasz też w <div/>
        // który zawiera w sobie ten przycisk
    });
}
```

Propagację można by przyrównać np. do chodzenia po dywanie.

Jeśli masz dywan w swoim pokoju i po nim chodzisz to oznacza, że chodzisz również po podłodze? Prawda?

Każdy nasz krok oddziałuje przez dywan na podłogę (i na odwrót). Podobnie się dzieje przy propagacji!

Klikając w przycisk jednocześnie klikamy w inne elementy, które znajdują się pod nim (są jego przodkami).

```
const btnEl = document.querySelector('button');
const sectionEl = document.querySelector('section');
if(btnEl && sectionEl) {
    btnEl.addEventListener('click', function() {
        console.log('button was clicked');
    });

    sectionEl.addEventListener('click', function() {
        console.log('div was clicked too!');
        // zauważ, że klikając w <button />
        // równocześnie klikasz też w <div/>
        // który zawiera w sobie ten przycisk
    });
}
```

Zastanówmy się jeszcze w jaki sposób moglibyśmy wywołać na obu elementach tą samą funkcję, która doda do kliknięte go elementu klasę, będąca nazwą znacznika.

Z pomocą przyjdzie nam tutaj specjalna zmienna this, której wartość zależy od kontekstu tj. miejsca wywołania funkcji.

```
const btnEl = document.querySelector('button');
const sectionEl = document.querySelector('section');

const addClassToElement = function() {
    const className = this.tagName.toLowerCase();
    // pobieram nazwę kliknięte go tagu
    // oraz zamieniam wielkie litery na małe
    this.classList.add(className);
    console.log(className + ' was clicked');
}

if(btnEl && sectionEl) {
    btnEl.addEventListener('click', addClassToElement);
    // this wskazuje na [btnEl]
    sectionEl.addEventListener('click', addClassToElement);
    // this wskazuje na [sectionEl]
}
```

Sama zmienna this nie jest ściśle powiązana z efektem propagacji.

Często this jest wykorzystywany przy rozprzestrzenianiu się zdarzeń jednak ma też inne zastosowanie.

Dzięki tej zmiennej możemy się odwołać do konkretnego elementu, na którym zostało wywołane zdarzenie (i tylko na tym!), mimo że nasłuchujemy wiele elementów tą samą funkcją.

```
<section>
   <button>1</button><button>2</button><button>3</button>
</section>
```

```
const btnsList = document.querySelectorAll('button');
const changeText = function() {
    this.innerText = 'clicked';
    // zmień tekst na kliknięty elemencie
}
btnsList.forEach(function(btnEl) {
    // do każdego przycisku
    // który jest dostępny pod zmienną [btnEl]
    btnEl.addEventListener('click', changeText);
    // dopisz nasłuchiwanie na event [click]
    // [this] będzie wskazywał na [btnEl]
});
```

Wracając do samej propagacji, która jak już wiesz jest rozprzestrzenianiem się zdarzenia po elementach będących w relacji rodzic - dziecko.

Propagacja posiada 2 fazy. Pierwsza polega na przemieszczaniu się od korzenia do elementu, na którym został wywołany event.

Druga faza to powrót tj. zaczynamy od elementu, na którym wywołane zostało zdarzenie i kończąc na korzeniu.

```
const btnEl = document.querySelector('button');
const sectionEl = document.querySelector('section');
if(btnEl && sectionEl) {
    btnEl.addEventListener('click', function() {
        console.log('button was clicked');
    });

    sectionEl.addEventListener('click', function() {
        console.log('div was clicked too!');
        // zauważ, że klikając w <button />
        // równocześnie klikasz też w <div/>
        // który zawiera w sobie ten przycisk
    });
}
```

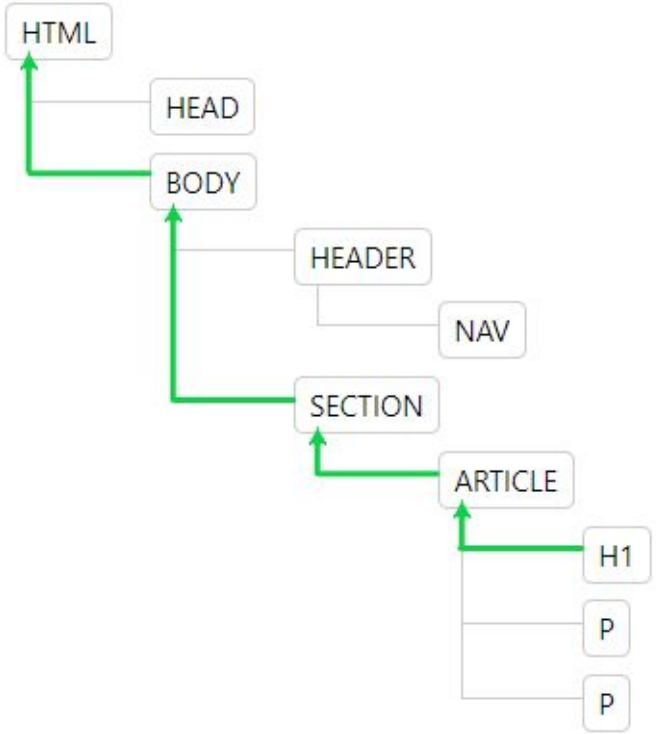
#01 Faza bubbling

02. Propagacja zdarzeń

Faza **babbling** polega na rozprzestrzenianiu się (propagacji) zdarzeń w górę.

Najpopularniejsze event-y korzystają z tej fazy do uruchamiania callback-ów (funkcji przekazanych jako drugi parametr).

Gdy użytkownik kliknie na element `<h1>` to zaczynamy propagację od liścia tj. `<h1>`, a kończymy na korzeniu tj. `<html>` - poprzez pozostałe elementy.



zródło: HTML Tree Generator

Ten faza jest wykorzystywana domyślnie podczas tworzenia nasłuchiwania.

Tak naprawdę .addEventListener() posiada 3 parametry - ostatni decyduje o użytej fazie i przyjmuje true lub false.

```
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem
const showTagName = function() {
    console.log( this.tagName );
    // pobieram nazwę tagu,
    // na którym zostało wywołane zdarzenie
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName, false);
    // ostatni parametr jest ustawiany domyślnie na false
    // dlatego nie musimy go pisać dla [click]
    // co oznacza, że propagacja typu [bubbling]
    // ma zostać wykorzystana
});
```

Przedstawiony obok przykład wyświetli w konsoli nazwę tagów w kolejności:

- *H1*
- *ARTICLE*
- *SECTION*

Będzie to miało miejsce po kliknięciu w element `<h1>`.

Jeśli kliknięcie byłoby wykonane na elemencie `<article>` (lub jego innych dzieciach) to nazwę tagu `<h1>` nie zobaczymy w konsoli.

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem
const showTagName = function() {
    console.log( this.tagName );
    // pobieram nazwę tagu,
    // na którym zostało wywołane zdarzenie
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName, false);
    // ostatni parametr jest ustawiany domyślnie na false
    // dlatego nie musimy go pisać dla [click]
    // co oznacza, że propagacja typu [bubbling]
    // ma zostać wykorzystana
});
```

#02 Faza capturing

02. Propagacja zdarzeń

Faza **capturing** jest odwrotnym rozwiązaniem do *bubbling*.

Rozpoczyna się od korzenia, a kończy na elemencie wywołującym zdarzenie.

Jeśli chcemy jej użyć to wystarczy wartość ostatniego parametru ustawić na true.

Teraz kolejność w konsoli będzie taka:

- *SECTION*
- *ARTICLE*
- *H1*

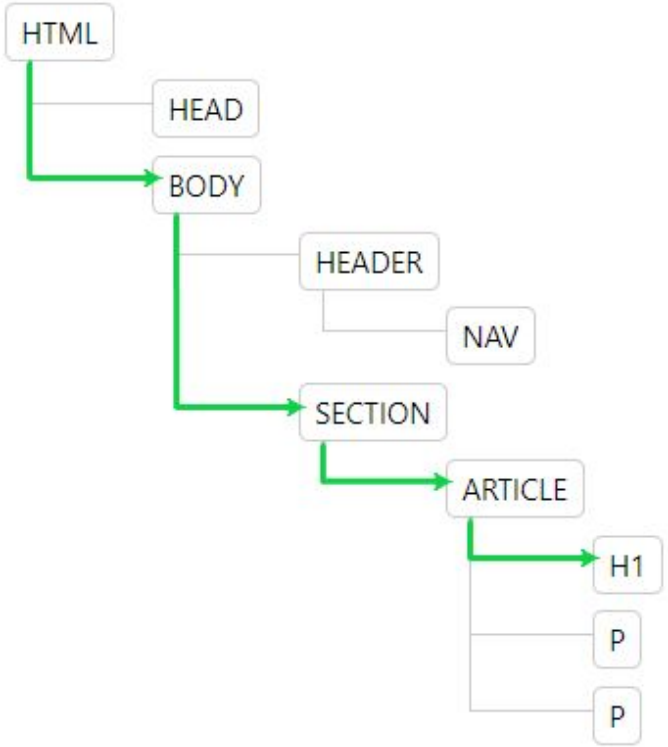
```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem

const showTagName = function() {
    console.log( this.tagName );
    // pobieram nazwę tagu,
    // na którym zostało wywołane zdarzenie
}

itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName, true);
    // ostatni parametr jest ustawiony na true
    // co oznacza, że będzie używany [capturing]
    // propagacja odbywa się z góry na dół
});
```

Tak naprawdę obie fazy występują jednocześnie (najpierw *capturing*, potem *bubbling*), a my decydujemy, której chcemy użyć.

Jak widać na obrazku obok faza *capturing* pozwala nam wywoływać *callback-i* najpierw na przodkach, potem dopiero na docelowym elemencie.



zródło: HTML Tree Generator

W przypadku, gdy korzystamy z fazy *capturing* to podczas usuwania nasłuchiwania musimy również pamiętać o trzecim parametrze ustawionym na `true`.

Tak jak w przedstawionym przykładzie obok. Gdzie za drugim razem już na `<article>` nie zostanie wywołany *callback* ponieważ usuwamy nasłuchiwanie.

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem
const showTagName = function(e) {
    console.log( this.tagName );
    if(this.tagName === 'ARTICLE') {
        this.removeEventListener(
            'click', showTagName, true
        );
        // usuwam nasłuchiwanie na <article/>
        // ponieważ korzystam z [capturing]
        // to 3 parametr ustawiony jest na [true]
    }
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName, true);
});
```

W zależności od rodzaju elementu, na którym podpinamy nasłuchiwanie, faza *bubbling* może nie być wykorzystywana.

Powinniśmy sprawdzić to w dokumentacji, co możemy uczynić np. na stronie MDN z listą eventów.

Po wyborze konkretnego event-u znajdziemy informacje czy *Bubbles* jest ustawione (*Yes*) czy nie (*No*).

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem

const showTagName = function() {
    console.log(this.tagName);
    // wyświetlam informacje o tagu
}

itemsList.forEach( function(item) {
    item.addEventListener('mouseenter', showTagName);
    // mimo, że nie użyłem fazy capturing
    // to kolejność wyświetlanych tagów w konsoli
    // jest zgodna z fazą capturing
    // tj. section, article, h1
});
```

03. Obiekt Event

Obiekt *Event*, który jest przekazywany jako pierwszy parametr *callback-u* pozwala nam zdobyć dodatkowe informacje na temat wywołanego zdarzenia.

```html
<section>
    <button>btn 1</button>
    <button>btn 2</button>
</section>
```

```js
const btnsList = document.querySelectorAll('button');
const showInformation = function(e) {
    // nazwa parametru jest dowolna
    // najczęściej jest to [e] lub [event]
    console.log(e.type, this.innerText);
    // mogę dzięki właściwości [type]
    // pobrać informacje o typie event-u
}
btnsList.forEach( function(item) {
    item.addEventListener('mouseenter', showInformation);
    item.addEventListener('click', showInformation);
    // nasłuchuje na tym samym elemencie
    // kliknięcie i najechanie kursorem myszy
});
```

Obiekt ten pozwala pobrać bardzo szczegółowe informacje na temat czasu kliknięcie, pozycji kursora czy wykorzystujemy fazę *bubbling*, a nawet jaki element zapoczątkował propagację.

```
<section>
    <button>btn 1</button>
    <button>btn 2</button>
</section>
```

```
const btnsList = document.querySelectorAll('button');
const showObject = function(e) {
    console.log(e);
    // wyświetlam obiekt [e] w konsoli,
    // aby sprawdzić jakie właściwości posiada

}
btnsList.forEach( function(item) {
    item.addEventListener('click', showObject);
    // nasłuchuje event [click] na tym elemencie
});
```

Te informacje możemy wykorzystać wew. *callback-u*. Musimy tylko trzymać się nazewnictwa wew. funkcji.

Jeśli nazwaliśmy parametr jako `e` to już potem musimy trzymać się tej nazwy.

```
<section>
    <button>btn 1</button>
    <button>btn 2</button>
    <p>Kliknij lub najedź na button</p>
</section>
```

```
const btnsList = document.querySelectorAll('button');
const pElement = document.querySelector('p');
const renderInfo = function(e) {
    if(pElement) {
        let text = parseInt(e.timeStamp / 1000);
        // ilość sekund od uruchomienia strony
        text += ': '+ this.tagName;
        // nazwa nasłuchiwanego tagu
        text += ' -> '+ e.type;
        // typ uruchomionego zdarzenia
        pElement.innerText = text;
    }
}
btnsList.forEach( function(item) {
    item.addEventListener('click', renderInfo);
    item.addEventListener('mouseenter', renderInfo);
});
```

#01 .preventDefault()

03. Obiekt Event

Niektóre elementy HTML posiadają domyślną akcję powiązaną z danym typem event-u.

Dla linku (<a>) domyślną akcją po kliknięciu jest przekierowanie użytkownika do adresu zdefiniowanego w atrybucie href.

Czasami chcemy zablokować domyślną akcję i umożliwia nam to .preventDefault()

```
const linksList = document.querySelectorAll('a');
// pobieram wszystkie <a/> na stronie

const getHref = function(e) {
    e.preventDefault();
    // zatrzymuję domyślną akcję

    const href = this.getAttribute('href');
    // pobieram adres, który jest zapisany
    // w atrybucie href w kliknięty <a/>

    console.log(href);
}

linksList.forEach( function(item) {
    item.addEventListener('click', getHref);
});
```

Zatrzymanie domyślnej akcji może być warunkowe. Nie musimy go uruchamiać na samym końcu czy początku naszego *callback-a*.

```
<ul>
    <li><a href="https://devmentor.pl">DM</a></li>
    <li><a href="https://google.pl">G</a></li>
</ul>
```

```
const linksList = document.querySelectorAll('a');
const confirmRedirect = function(e) {
    const newUrl = this.getAttribute('href');
    const userDecision = confirm(
        'Are you sure? -> ' + newUrl
    ); // pobieram potwierdzenie

    if(!userDecision) {
        // jeśli użytkownik nie potwierdził
        e.preventDefault();
        // nie przekierowuje go na
        // nowy adres url
    }
}
linksList.forEach( function(item) {
    item.addEventListener('click', confirmRedirect);
});
```

Często zatrzymanie domyślnej akcji formularza jest niezbędne, aby móc sprawdzić poprawność przesyłanych danych - ten temat będziemy omawiać później.

```
<form action="https://devmentor.pl">
    <section>
        <label>Age: <input name="age"  /></label>
    </section>
    <section><input type="submit" /></section>
</form>
```

```
const formEl = document.querySelector('form');
const handleSubmit = function(e) {
    e.preventDefault();
    // jeśli nie użyję zatrzymania domyślnej akcji
    // to dane formularza zostaną wysłane do
    // adresu zdefiniowanego w atrybucie [action]
    // jeśli formularz nie posiada tego atrybutu
    // to strona zostanie przeładowana

    console.log('submit!');
}

formEl && formEl.addEventListener('submit',
handleSubmit);
```

#02 .stopPropagation()

03. Obiekt Event

Wiemy już czym jest propagacja, ale czy wiemy, że możemy nad nią zapanować?

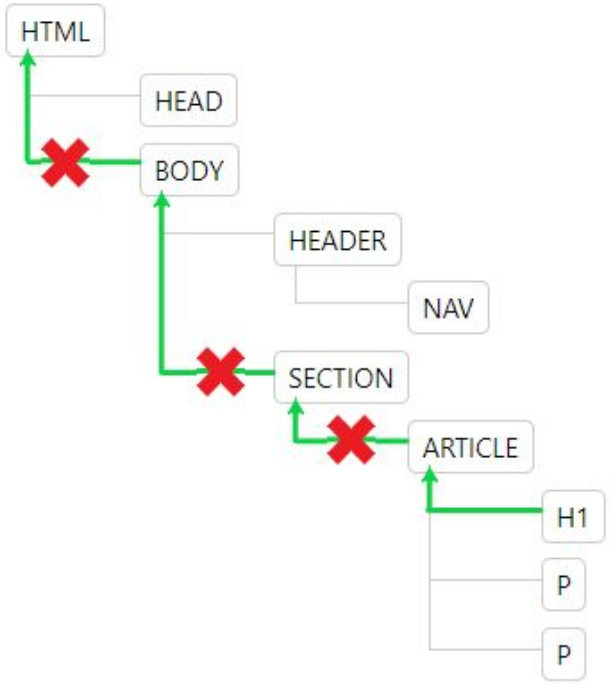
Wystarczy w odpowiednim callback-u wywołać metodę `.stopPropagation()`.

```
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem
const showTagName = function(e) {
    console.log( this.tagName );
    if(this.tagName === 'ARTICLE') {
        e.stopPropagation();
        // zatrzymuję propagację na <article/>
        // dla elementów wyżej w drzewie DOM
        // tj. przodków  dla <article/>
        // nie są uruchamiane callback-i
    }
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName);
});
```

Dzięki zatrzymaniu propagacji na elemencie `<article>` po kliknięciu w `<h1>` w konsoli zobaczymy jedynie elementy:

- *H1*
- *ARTICLE*



zródło: HTML Tree Generator

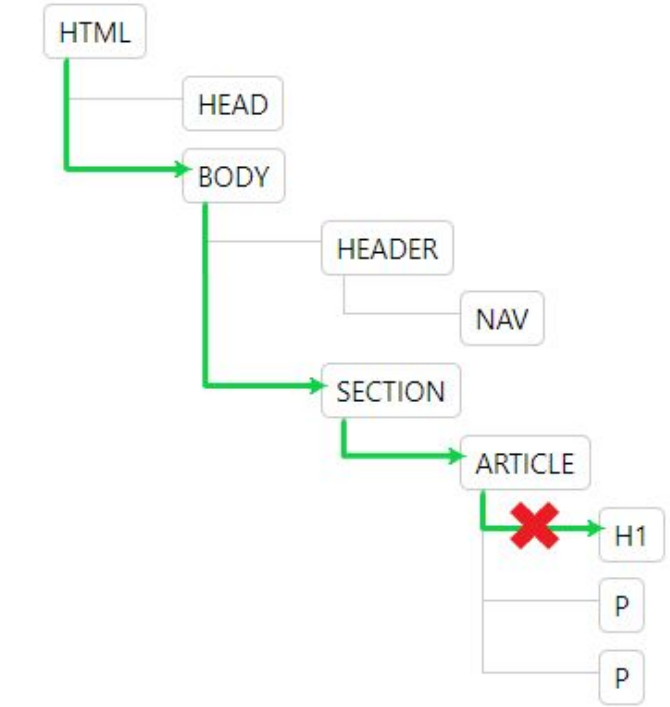
Jeżeli wykorzystamy fazę *capturing* to metoda `.stopPropagation()` zatrzyma propagację w dół.

```
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
); // wyszukuje wszystkie elementy razem
const showTagName = function(e) {
    console.log( this.tagName );
    if(this.tagName === 'ARTICLE') {
        e.stopPropagation();
        // zatrzymuję propagację na <article/>
        // dla elementów poniżej w drzewie DOM
        // tj. dzieci  dla <article/>
        // nie są uruchamiane callback-i
    }
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName, true);
});
```

Dzięki zatrzymaniu propagacji na elemencie `<article>` po kliknięciu w `<h1>` w konsoli zobaczymy jedynie elementy:

- *SECTION*
- *ARTICLE*



zródło: HTML Tree Generator

#03 .stopImmediatePropagation()

03. Obiekt Event

`.stopImmediatePropagation()` jest rozszerzeniem wcześniej poznanej metody `.stopPropagation()`.

Nie tylko zatrzymuje propagację, ale również nie wywołuje pozostałych event-ów tego samego typu.

```
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const showTagName = function(e) {
    console.log( this.tagName );
    if(this.tagName === 'ARTICLE') {
        e.stopImmedietPropagation();
    }
}
const showText = function() {
    console.log('click on article!');
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName);
    if(item.tagName === 'ARTICLE') {
        item.addEventListener('click', showText);
    }
});
```

W zaprezentowany przykładzie dodaję dodatkowy event typu click dla elementu `<article>`.

Gdyby zakomentować linię z wywołaniem metody `.stopImmediatePropagation()`, to w konsoli zobaczymy:

- *H1*
- *ARTICLE*
- *click on article!*
- *SECTION*

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const showTagName = function(e) {
    console.log( this.tagName );
    if(this.tagName === 'ARTICLE') {
        //e.stopImmediatePropagation();
    }
}
const showText = function() {
    console.log('click on article!');
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName);
    if(item.tagName === 'ARTICLE') {
        item.addEventListener('click', showText);
    }
});
```

Gdy wrócimy do pierwotnej wersji. Nasza lista wpisów w konsoli będzie następująca:

- *H1*
- *ARTICLE*

Należy pamiętać, że kolejność wywoływania *callback-ów* tego samego typu na tym samym elemencie zależy od kolejności ich podpięcia do nasłuchiwania.

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const showTagName = function(e) {
    console.log( this.tagName );
    if(this.tagName === 'ARTICLE') {
        e.stopImmediatePropagation();
    }
}
const showText = function() {
    console.log('click on article!');
}
itemsList.forEach( function(item) {
    item.addEventListener('click', showTagName);
    if(item.tagName === 'ARTICLE') {
        item.addEventListener('click', showText);
    }
});
```

#04 .target vs .currentTarget

03. Obiekt Event

Obie właściwości z nazwy rozdziału pozwalają się odwołać do elementów, na których miało miejsce zdarzenie.

.target to zawsze element DOM, na którym zostało wywołane zdarzenie.

```
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const showInfo = function(e) {
    let text = e.target.tagName +': ';
    // pobieram nazwę tagu elementu,
    // na którym zostało wywołane zdarzenie
    // w tym przypadku na jaki element kliknięto
    text += this.tagName;

    console.log( text );
}

itemsList.forEach( function(item) {
    item.addEventListener('click', showInfo);
});
```

Natomiast `.currentTarget` to element, na którym został wywołany *callback*.

Co oznacza, że `this === .currentTarget`.

```
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const showInfo = function(e) {
    let text = e.currentTarget.tagName +': ';
    // pobieram nazwę tagu elementu,
    // na którym został uruchomiony callback
    text += e.currentTarget === this ? 'Y' : 'N';
    // sprawdzam czy zmienne wskazują
    // na ten sam element w drzewie DOM

    console.log( text );
}

itemsList.forEach( function(item) {
    item.addEventListener('click', showInfo);
});
```

Dzięki porównaniu `.target` oraz `.currentTarget` można sprawdzić czy *callback* jest wywoływany przez propagację czy bezpośrednio.

```html
<section>
    <article>
        <h1>Lorem ipsum dolor sit.</h1>
    </article>
</section>
```

```js
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const showInfo = function(e) {
    if(e.target === e.currentTarget) {
        // wykonaj instrukcje tylko wtedy,
        // gdy callback jest uruchomiony
        // bezpośrednio tj. nie przez propagację

        const text = e.target.tagName;
        console.log(text + ' was clicked');
    }
}

itemsList.forEach( function(item) {
    item.addEventListener('click', showInfo);
});
```

04. Rozszerzenie

#01 Wywoływanie zdarzeń

04. Rozszerzenie

Aby wywołać even z poziomu kodu wystarczy wywołać metodą na elemencie o nazwie eventu np. `.click()`.

Dzięki właściwości `.isTrusted` możemy sprawdzić czy został uruchomiony przez użytkownika czy z poziomu kodu JS.

```html
<section>
    <button>click me</button>
</section>
```

```js
const btnElement = document.querySelector('button');
const handleClick = function(e) {
    console.log('button was clicked');
    console.log(e.isTrusted);
    // zwraca [true] jeśli uruchomione przez użytkownika
    // zwraca [false] jeśli z poziomu kodu JS
}

btnElement.addEventListener('click', handleClick);
btnElement.click();
```

Przedstawione wcześniej rozwiązanie nie pozwala nam definiować dodatkowych opcji dlatego zdecydowanie bardziej polecam użycie .dispatchEvent().

To metoda, która wywołuje przekazany przez parametr *event* wraz z ustawieniami, o których dowiesz się na następnym slajdzie.

Do utworzenia zdarzenia możemy użyć obiektu Event, co prezentuje przykład obok.

```
const btnElement = document.querySelector('button');
const handleClick = function(e) {
    console.log('button was clicked');
    console.log(e.isTrusted);
    // zwraca [true] jeśli uruchomione przez użytkownika
    // zwraca [false] jeśli z poziomu kodu JS
}

btnElement.addEventListener('click', handleClick);

const eventClick = new Event('click');
// tworzę event typu [click]

btnElement.dispatchEvent(eventClick);
// wywołuję event na elemencie [btnElement]
```

Obiekt Event może przyjmować jako drugi parametr dodatkowe ustawienia

- *bubbles*
- *cancelable*

To dzięki nim możemy określać zachowanie naszego zdarzenia.

```
const btnElement = document.querySelector('button');
const handleClick = function(e) {
    console.log('button was clicked');
    console.log(e.isTrusted);
    // pobieram przekazane dane przy wywołaniu
}
btnElement.addEventListener('click', handleClick);
const eventClick = new Event('click', {
    'bubbles': true,
    // czy wykorzystujemy fazę bubbling przy propagacji
    'cancelable': true,
    // czy można zatrzymać event
    // za pomocą .preventDefault()
});
btnElement.dispatchEvent(eventClick);
```

Powinniśmy mieć również świadomość, że każdy event przynależy do grupy zdarzeń, które mają swoje szczególne cechy, np:

- *Event*
- *MouseEvent*
- *KeyboardEvent*
- *DragEvent*
- ...

Dlatego lepszym rozwiązaniem będzie określenie naszego event-u z przykładu jako MouseEvent.

```
const btnElement = document.querySelector('button');
const handleClick = function(e) {
    console.log('button was clicked');
    console.log(e.isTrusted);
    // pobieram przekazane dane przy wywołaniu
}
btnElement.addEventListener('click', handleClick);
const eventClick = new MouseEvent('click', {
    'bubbles': true,
    // czy wykorzystujemy fazę bubbling przy propagacji
    'cancelable': true,
    // czy można zatrzymać event
    // za pomocą .preventDefault()
});
btnElement.dispatchEvent(eventClick);
```

#02 Tworzenie własnych zdarzeń

04. Rozszerzenie

Własne event-y tworzymy na tych samych zasadach co do tej pory tylko używamy własnej nazwy np. render.

Przy wywołaniu wykorzystujemy obiekt CustomEvent.

```
<section>
    <button>fire event</button>
    <p>loading...</p>
</section>
```

```
const btnElement = document.querySelector('button');
const pElement = document.querySelector('p');
const handleRender = function(e) {
    this.innerText = e.detail;
    // wstaw tekst przekazany przy wywołaniu
}
const handleClick = function() {
    const renderEvent = new CustomEvent(
        'render', { detail: 'new content!'}
    );
    // tworzę event o nazwie [render]
    // oraz przekazuję dodatkowe dane w [detail]
    pElement.dispatchEvent(renderEvent);
    // wywołuje utworzony event na elemencie [pElement]
}
btnElement.addEventListener('click', handleClick);
pElement.addEventListener('render', handleRender);
// podpinam nasłuchiwanie eventu [render]
```

Obiekt `CustomEvent` podobnie jak obiekt `Event` przyjmuje w drugim parametrze dodatkowe ustawienia.

Tym razem mogą być nimi:

- **bubbles** - czy używamy *bubbling* do propagacji
- **cancelable** - czy `.preventDefault()` zadziała
- **detail** - wartość, która jest dostępna w *callback-u* pod właściwością `.detail`

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const handleOutput = function(e) {
    console.log(this.tagName, e.type);
}
itemsList.forEach( function(item) {
    item.addEventListener('output', handleOutput);
    if(item.tagName === 'H1') {
        const outputEvent = new CustomEvent(
            'output', {
                detail: item.tagName,
                bubbles: false,
            }
        );
        item.dispatchEvent(outputEvent);
    }
});
```

W przykładzie obok nasłuchujemy zdarzenia o nazwie `output`, na wszystkich wyszukanych elementach.

Następnie tylko na elemencie `<h1>` uruchamiamy ten element.

Ponieważ nie wykorzystujemy fazy *bubbling*, a w nasłuchiwaniu nie wskazaliśmy fazy *capturing* to *callback* zostanie uruchomiony tylko na elemencie `<h1>`.

Przy tym zdarzeniu nie będzie występować propagacja.

```
const itemsList = document.querySelectorAll(
    'section, article, h1'
);
const handleOutput = function(e) {
    console.log(this.tagName, e.type);
}
itemsList.forEach( function(item) {
    item.addEventListener('output', handleOutput);
    if(item.tagName === 'H1') {
        const outputEvent = new CustomEvent(
            'output', {
                detail: item.tagName,
                bubbles: false,
            }
        );
        item.dispatchEvent(outputEvent);
    }
});
```

Zadania do wykonania

Pod poniższym adresem znajdziesz zadania,
które obejmują materiał prezentowany w niniejszym ebook-u.

https://github.com/devmentor-pl/practice-js-events