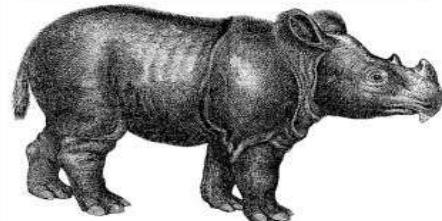


Kompleksowe orientowane
programowania w języku JavaScript

Oprawa A4 / 1 DOM

Wprowadzenie

JavaScript



HELIOS
O'REILLY®

Shelley Powers

Wykład 4 i 5

Wprowadzenie do Java Script

języka wielu paradymatów

dr inż. Grzegorz Rogus

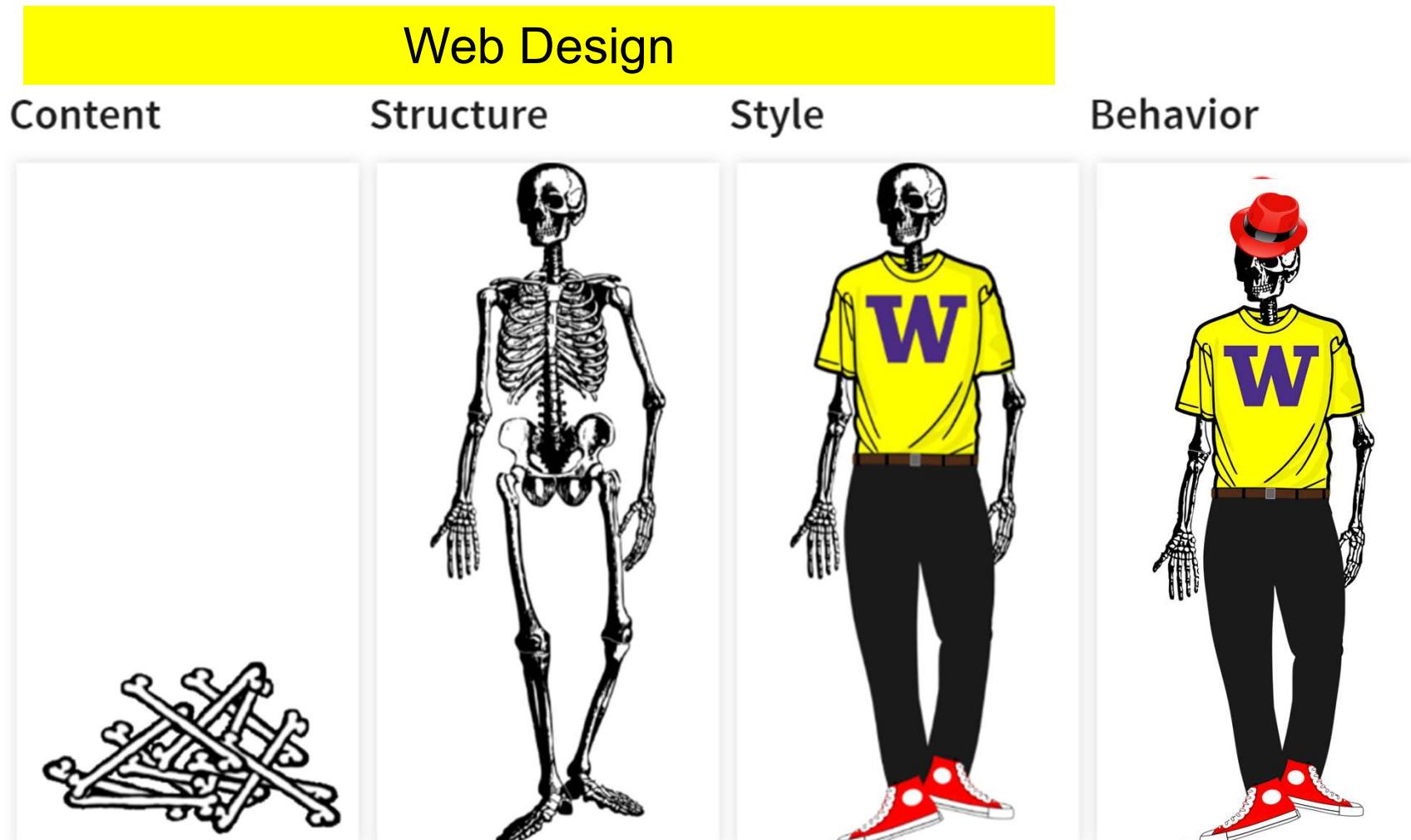


JavaScript

<https://javascript.info/>

```
1 <script>
2   if(age > 19){
3     alert("Adult");
4   } else{
5     alert("Teenager");
6   }
7 </script>
```

front-end development triad



Words and images

HTML

CSS

Javascript & Server
programs

Web Application

DEFINICJA JAVASCRIPT

(oficjalna nazwa ECMA-262, ECMAScript 6 – czerwiec 2015r.)

Dotyczasowa definicja

Skryptowy język programowania, którego celem jest dodanie dynamiki do możliwości HTML'a.

Umożliwia:

- manipulację wyglądem i położeniem elementów HTML;
- zmiany zawartości elementów HTML (innerHTML);
- pobieranie danych z formularzy i sprawdzanie ich poprawności;
- asynchroniczne ładowanie danych na stronę (Ajax);

Interaktywny klej pomiędzy HTML a CSS

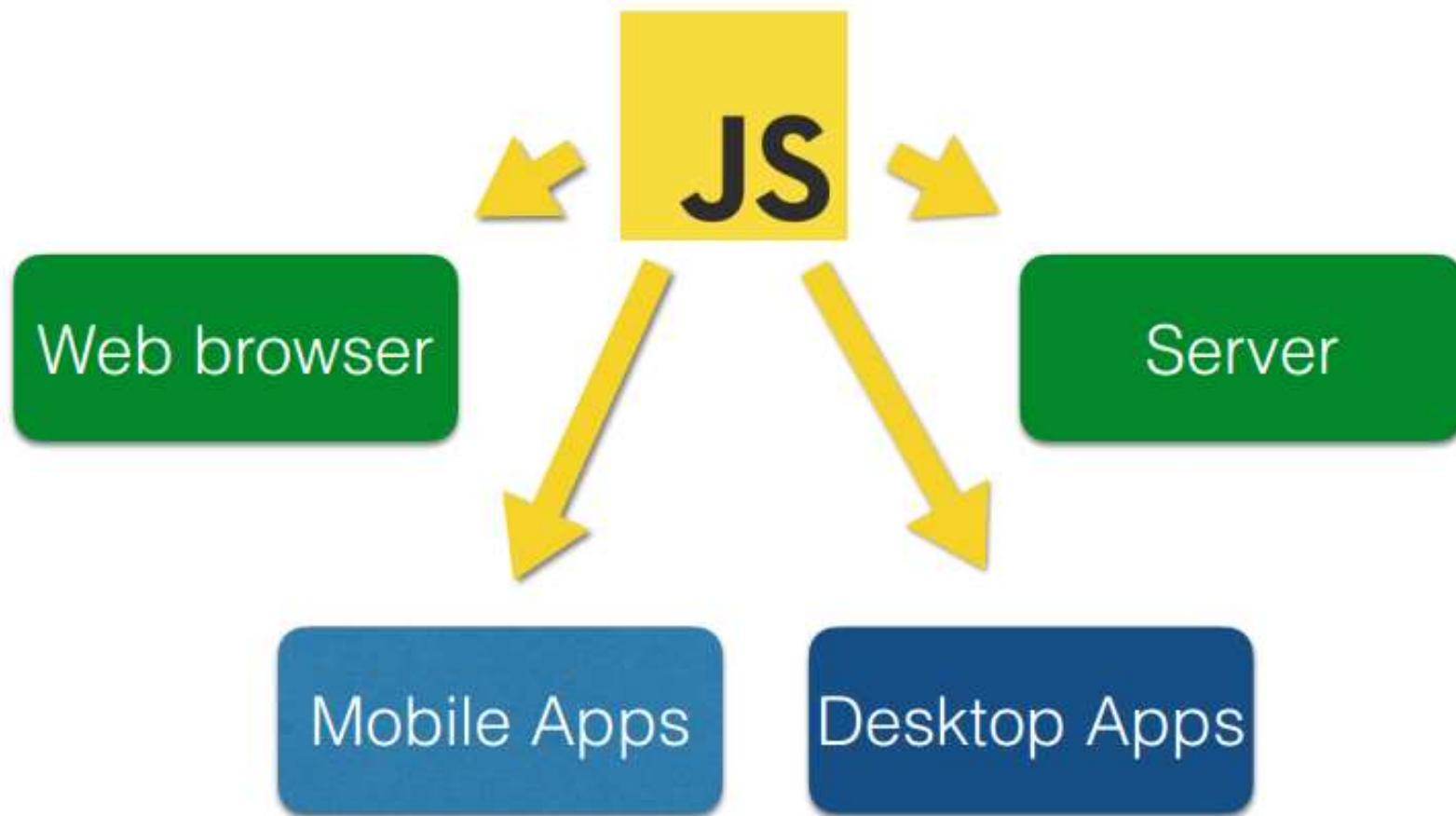


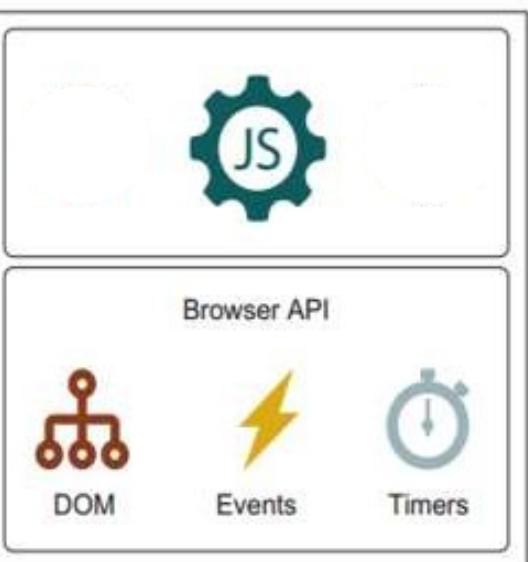
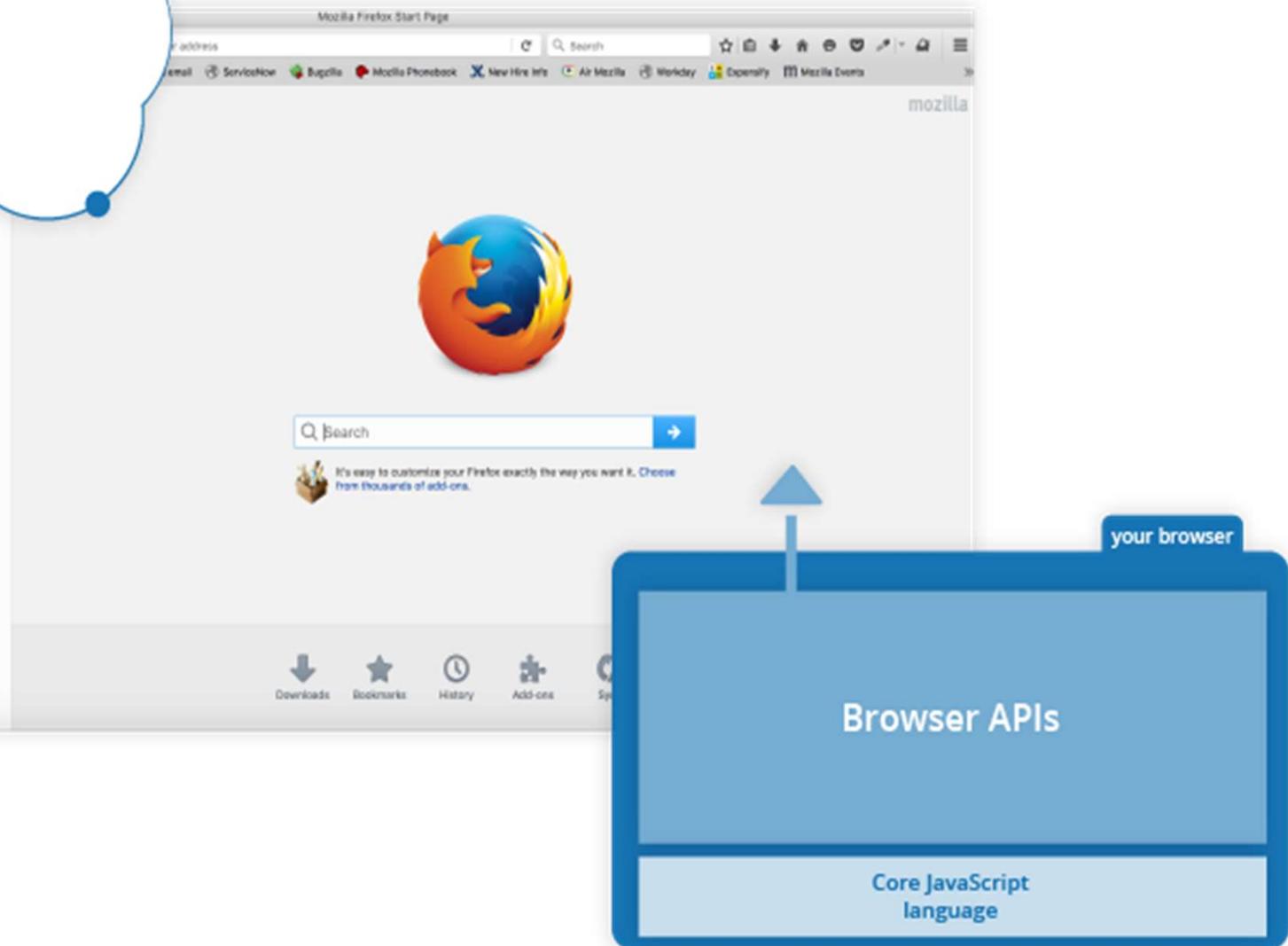
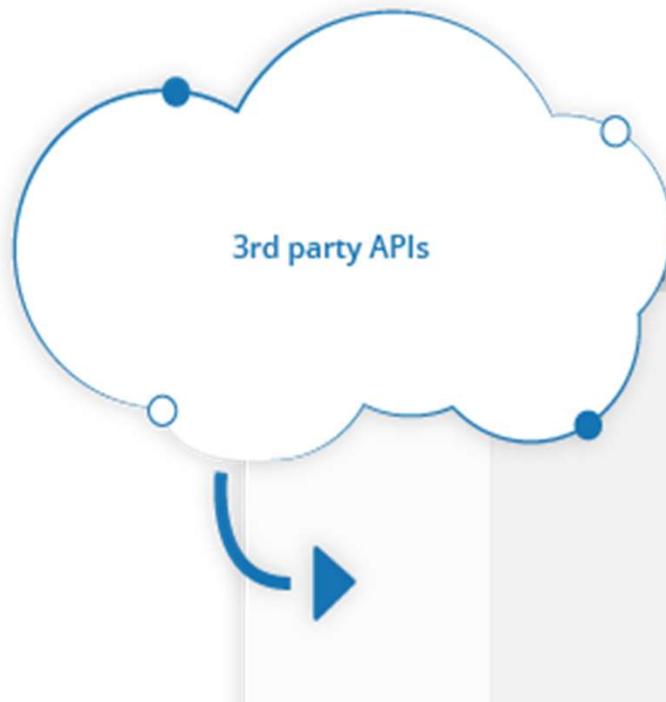
CECHY JAVASCRIPT

- **język skryptowy** - nie musi być komplikowany do kodu maszynowego;
- ze względów bezpieczeństwa nie można zapisywać na dysku komputera, na którym przeglądana jest dana strona;
- wszelkie odwołania do funkcji i obiektów wykonywane są w trakcie wykonywania programu;
- pozwala na odciążenie serwerów i ograniczenie zbędnych danych, wysyłanych przez Internet;
- działa po stronie przeglądarki użytkownika.

JavaScript posiada wszystkie podstawowe elementy poprawnego języka programowania: zmienne, instrukcje warunkowe, pętle, instrukcje wejścia/wyjścia, tablice, funkcje, a zwłaszcza obiekty. Język ten jest oparty na obiektach (ang. *object-based*) i jest sterowany zdarzeniami (ang. *event-driven*).

2021





JavaScript (w przeglądarce) = ECMAScript + BrowserAPI



ECMAScript – ustandaryzowany przez ECMA obiektowy skryptowy język programowania, którego najbardziej znane implementacje to JavaScript, JScript i ActionScript. Specyfikacja ta oznaczona jest jako ECMA-262 i ISO/IEC 16262.

Standard ten określa między innymi:

- *składnię języka* – reguły parsowania, słowa kluczowe, instrukcje, deklaracje, operatory itd.
- *typy* – typ logiczny, liczbowy, łańcuchowy, obiektowy itd.
- *prototypy i reguły dziedziczenia*
- *standardową bibliotekę wbudowanych obiektów i funkcji* – JSON, Math, metody obiektu Array, metody introspekcji wywoływane na obiektach itd.

ECMAScript nie definiuje natomiast żadnych aspektów związanych z językiem HTML, CSS, ani z sieciowymi interfejsami API, takimi jak DOM (obiektowy model dokumentu).

Browser API:

- API do manipulacji dokumentem ([DOM \(Document Object Model\) API](#))
- API do pobierania danych z serwera ([XMLHttpRequest lub Fetch API](#))
- API do rysowania i edycji grafki ([CANVAS, WebGL](#))
- Audio i Video APIs ([HTMLMediaElement, Web Audio API, WebRTC](#))
- Device API ([Notifications API, Vibration API, Geolocation API](#))
- Client-side storage API ([Web Storage API, IndexedDB API](#))

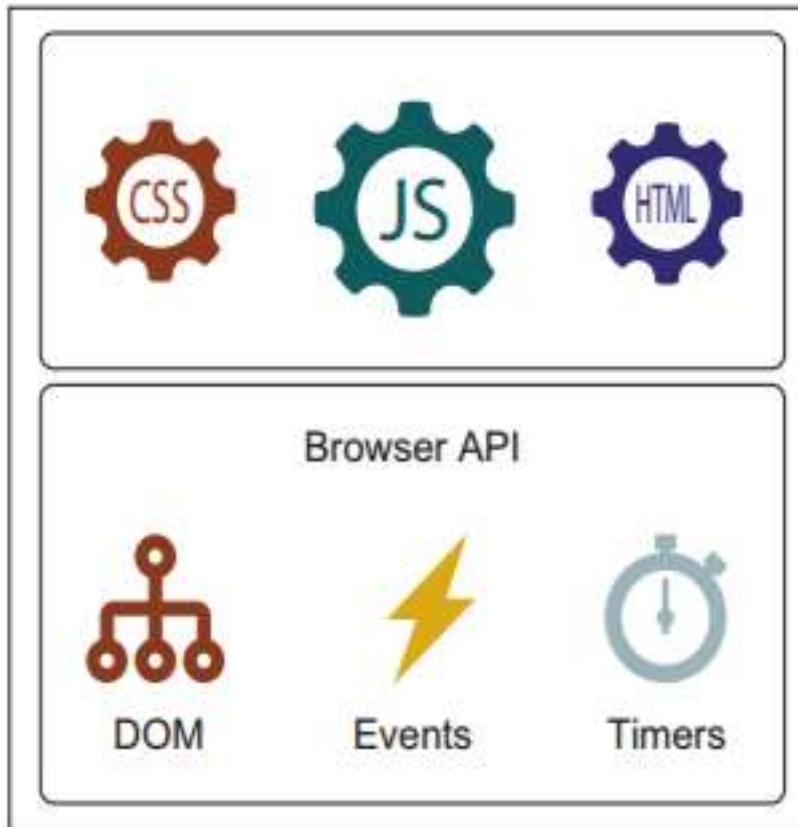
API zewnętrznych dostawców:

[TwitterAPI](#) [GoogleMapsAPI](#) [FacebookAPI](#) [YouTubeAPI](#) [AmazonS3](#)

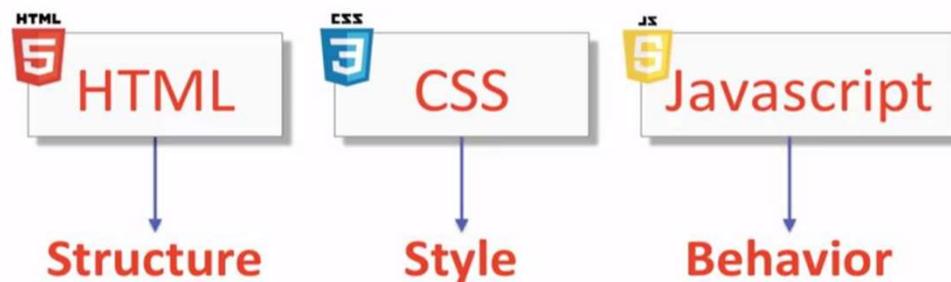
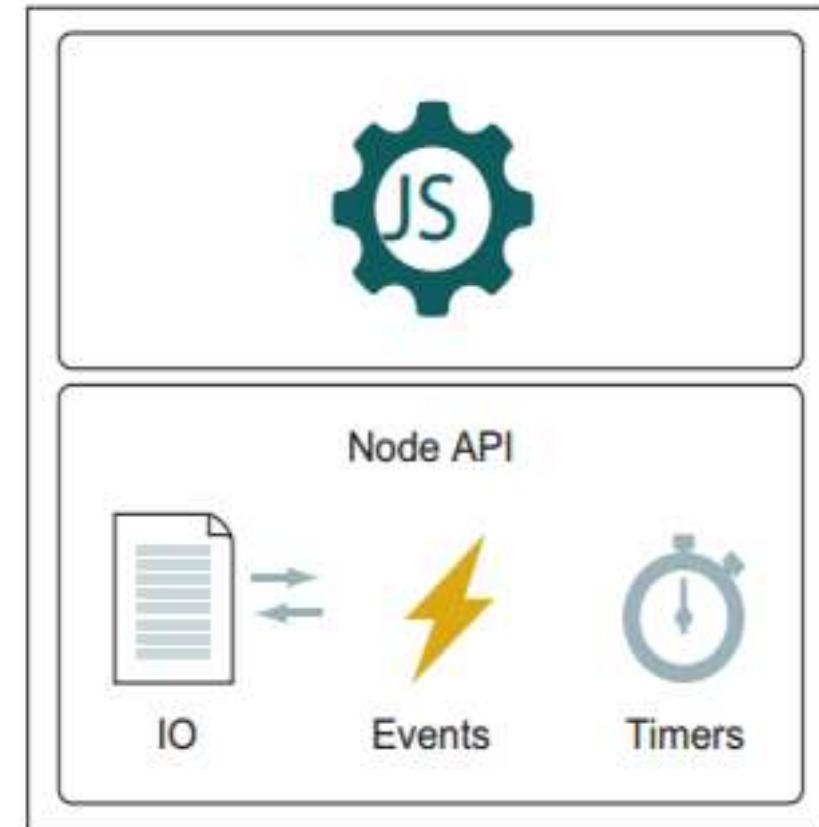
Więcej -> <https://www.programmableweb.com/category/all/apis>

Aplikacje JS w różnych środowiskach

Browser infrastructure



Node.js infrastructure



JS – cechy języka

Cechy języka JavaScript:

- Zapewnia obsługę DOM (Document Object Model),
- Brak statycznej kontroli typów zmiennych (trudności z wykryciem błędów).
- Współpraca z formatem JSON,
- Wbudowane dziedziczenie prototypowe,
- Brak klas typowych z innych języków programowania.

Opis standardu języka:

<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>

Silniki JS:

- SpiderMonkey (Mozilla Firefox)
- JavaScriptCore (Apple Safari)
- Chrome V8 (Google Chrome, Node.js)
- Chakra (Microsoft Edge)

Wersje JavaScript

Edition	Official name	Date published
ES8	ES2017	June 2017
ES7	ES2016	June 2016
ES6	ES2015	June 2015
ES5.1	ES5.1	June 2011
ES5	ES5	December 2009
ES4	ES4	Abandoned
ES3	ES3	December 1999
ES2	ES2	June 1998
ES1	ES1	June 1997

JavaScript (JS) - ECMA6

ECMA6 = ECMAScript 2016 = ES6



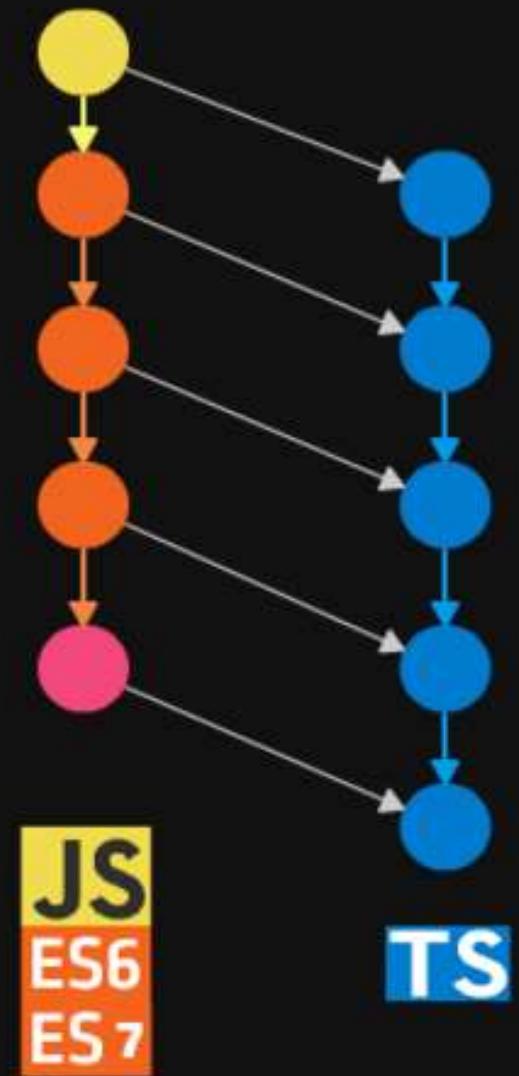
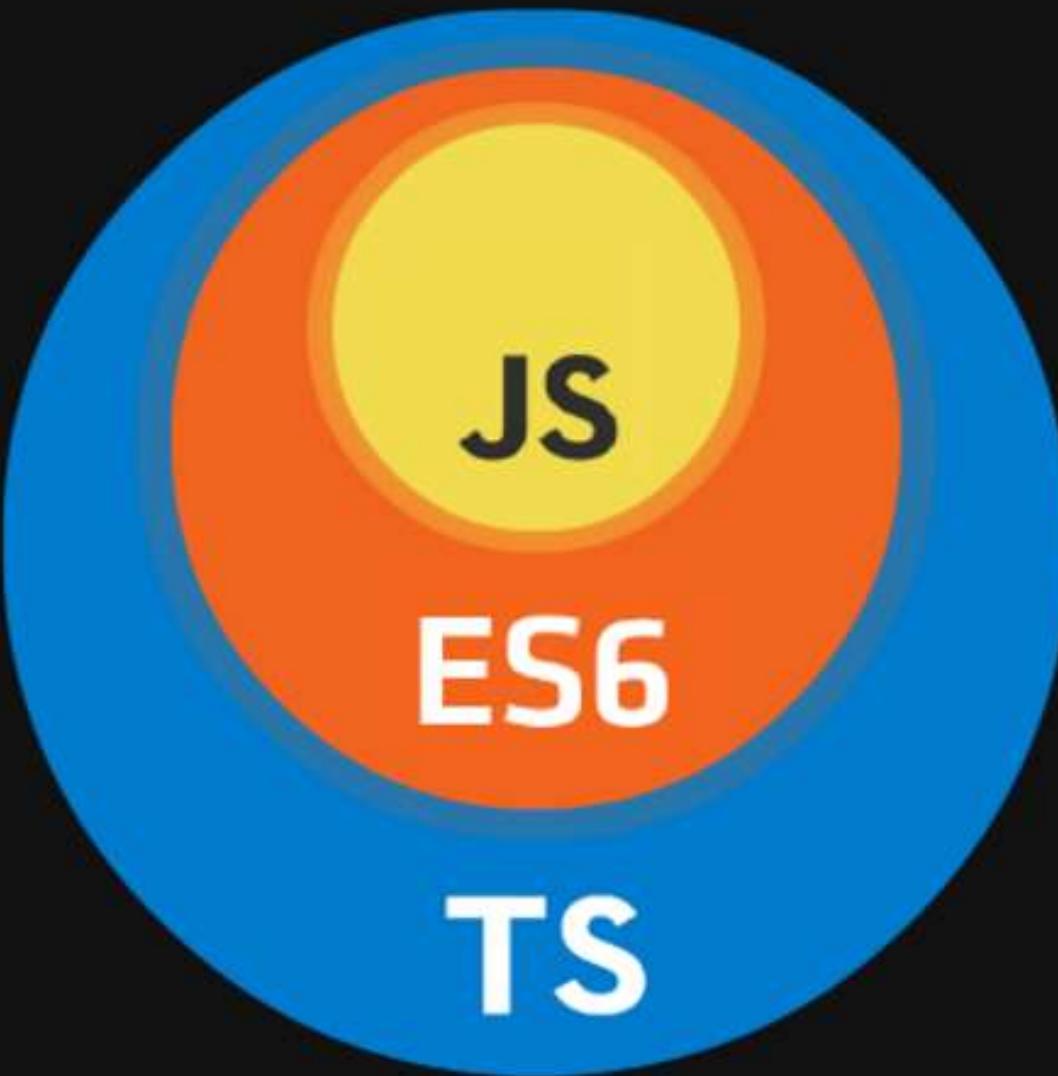
Obecnie najczęściej i najpowszechniej używaną wersja JavaScript była wersja 5.1 (ES5 / ECMA5) standaryzowana w roku 2011.

Dlaczego ES6 ?



W odróżnieniu od ES5 nowa wersja nie jest tylko drobnym ulepszeniem poprzednika (lifting), ale wprowadza zupełnie nowe jakościowe podejście do pisania kodu JavaScript. Zawiera nowe formy składniowe, nowe formy organizacji kodu, wspomaga wiele interfejsów API, które ułatwiają posługiwanie się różnymi typami danych.

JS vs ES6 vs TS



ES6 co nowego

- let/const
- template strings
- new ways to declare objects
- classes
- map, filter, reduce (ES5)
- arrow functions
- for ... of
- Promises
- Modules
- Proxy
- Iterators
- Generators
- Symbols
- Map/Set, WeakMap/WeakSet
- extended standard library (Number, Math, Array)

ES6 czego dotyczy

1. Podstawy języka

Blokowy zakres zmiennych – Let i Const

Operator rozwinięcia oraz parametry domyślne

Interpolacja stringów

2. Obiekty wbudowane

Nowe metody w String Object

Nowe metody w Number Object

Nowe metody w Array Object

3. Paradymat obiektowy – zarządzanie kodem

Moduły

Import

Export

Klasy

4. Nowe struktury danych

Zbiory (Sets)

Mapy (Maps)

Weak Maps oraz Weak Sets

Iteratory

Petla for of Loop

5. Programowanie funkcyjne

Wyrażenia Lambda (Arrow Functions)

Generatory

6. Inne elementy syntaktyczne

Obietnice (Promises)

Destrukturalizacja

Nowoczesny JS

ES6/ESNext

ES6 Modules

BABEL



ES5



webpack



Bundle

ES6
ES2016
ES2017

BABEL

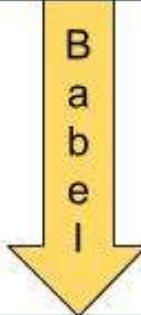
ES5
+
Polyfills
[if needed]



[1, 2, 3].map((n) => n + 1)

[1, 2, 3].map(function(n) { return n + 1 })

`string text \${expression} string text`



"string text " + expression + " string text";

```
async function verifyUser(username, password) {
  try {
    const userInfo = await database.verifyUser(username, password);
    const rolesInfo = await database.getRoles(userInfo);
    const rulesInfo = await database.getRules(userInfo);
    return userInfo;
  } catch (e) {
    //handle errors as needed
  }
}
```



```
function verifyUser(username, password, callback) {
  database.verifyUser(username, password, function(error, userInfo) {
    if (error) {
      callback(error);
    } else {
      database.getRoles(username, function(error, roles) {
        if (error) {
          callback(error);
        } else {
          callback(null, userInfo, roles);
        }
      });
    }
});
```



wsparcie danych
funkcjonalności

- <http://kangax.github.io/compat-table/es6/>
- <https://caniuse.com/>

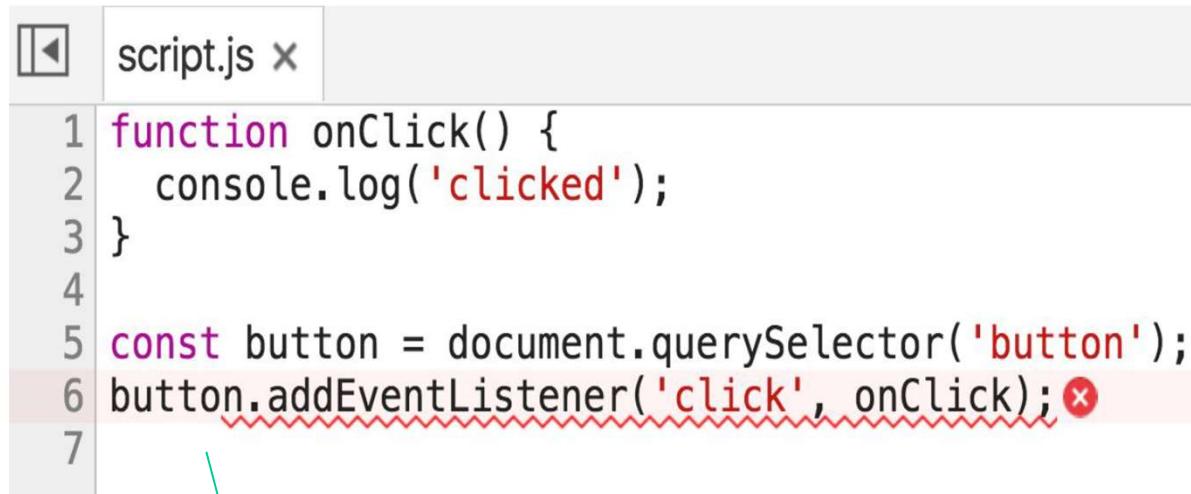
JavaScript w przeglądarce - podstawowe informacje

Jak załączyć JS do strony

```
<head>
    <script src="./js/plik.js" defer></script>
</head>
        LUB
<body>
    . . .
    <script src="./js/plik.js"></script>
</body>
        LUB
<head>    lub    <body>
<script>
...kod...
</script>
</head>    lub    </body>
```

Ładowanie JavaScript - problemy

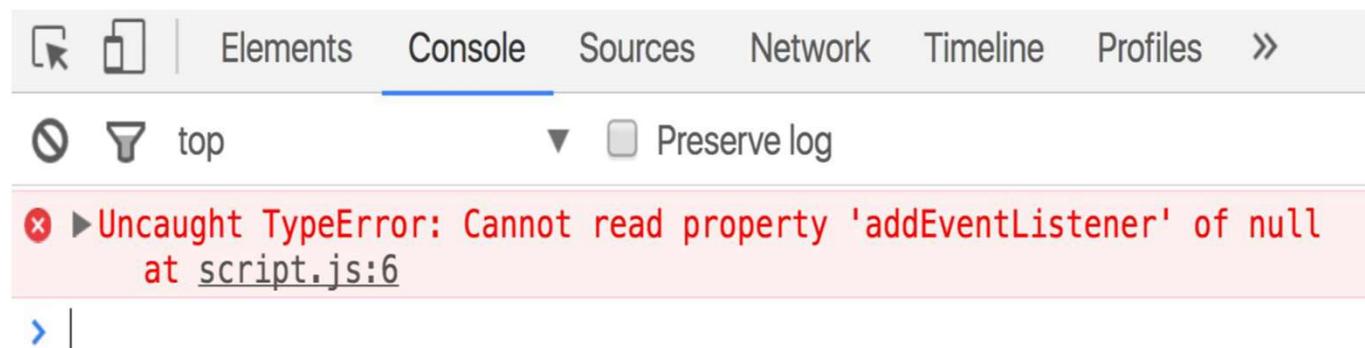
```
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
  <body>
    <button>Click Me!</button>
  </body>
</html>
```



A screenshot of a code editor showing a file named "script.js". The code contains two functions: "onClick" which logs "clicked" to the console, and "button.addEventListener('click', onClick)". The line "button.addEventListener('click', onClick)" is highlighted with a red wavy underline, indicating a syntax error. A green arrow points from the text "Błąd! Dlaczego?" below to this error line.

```
script.js x
1 function onClick() {
2   console.log('clicked');
3 }
4
5 const button = document.querySelector('button');
6 button.addEventListener('click', onClick); ×
7
```

Błąd! Dlaczego?



Rozwiążanie

```
<head>
    <script src=".js/somescript.js" defer</script>
</head>
```

HTML



Inne przestarzałe metody ładowania js do HTML (**nie rób tego**):

- Umieść tag `<script>` na dole strony
- Nasłuchuj zdarzenia „load” w obiekcie okna

W Internecie jest mnóstwo przykładów, które to robią.

Są nieaktualne.

defer jest powszechnie obsługiwany i lepszy.

JavaScript – Typy danych

W języku JavaScript wszystkie wartości poza prostymi typami liczbowymi, łańcuchowymi lub logicznymi są obiektami. (**dziedziczą po Object**)

Typy proste

Są to liczby, łańcuchy oraz wartości logiczne, posiadające metody, ale są **niezmienne**.

VS.

Obiekty

Są to asocjacyjne kolekcje klucz-wartość, które można dowolnie **modyfikować**.

W języku JavaScript:

- tablice są obiektami,
- wyrażenia regularne są obiektami,
- funkcje są obiektami,
- obiekty są obiektami.

Typy proste:
Boolean
Null
Undefined
Number
String

Zmienne

// typy proste – przekazywanie przez wartość

```
const a = 5;  
let b = a;  
b = 6;  
console.log(a);    // a = 5  
console.log(b);    // b = 6
```

// typy złożone – przekazywane przez referencje

```
let a = ['czesc', 'GR'];  
let b = a;  
b[0] = 'pa';  
console.log(a[0]);    // wynik -> 'pa'  
console.log(b[0]);    // wynik -> 'pa'
```

Deklaracja zmiennych: var, const, let

	Zakres widoczności	Można nadpisać	Można zmienić	Czasowo martwa strefa
const	Blok	Nie	Tak	Tak
let	Blok	Tak	Tak	Tak
var	Funkcja	Tak	Tak	Nie

Zakres widoczności

```
let a = 50;
let b = 100;
if (true) {
  let a = 60;
  var c = 10;
  console.log(a/c); // 6
  console.log(b/c); // 10
}
console.log(c); // 10
console.log(a); // 50
```

```
if (true) {
  let a = 40;
  console.log(a); //40
}
console.log(a); // undefined
```

ES5: var - hoisting

```
1 var foo
2
3 foo = 'OUT'
4
5 {
6   foo = 'IN'
7 }
```

W odróżnieniu od zmiennych zadeklarowanych poprzez var ([windowowanie – hoisting](#)), próba odczytu bądź nadpisania zmiennej stworzonej za pomocą let lub const przed przypisaniem wywoła błąd. Zjawisko to nazywane jest często [Czasowo martwą strefą](#)

var i let - porównanie

```
function testVar() {  
    var x = 5;  
    if (x == 5) {  
        var x = 8;          // ta sama zmienna o zasięgu funkcji  
        console.log(x);   // 8  
    }  
    console.log(x);     // 8  
}  
  
function testLet() {  
    let x = 5;  
    if (x == 5) {  
        let x = 8;    // nowa zmienna, lokalna dla bloku kodu  
        console.log(x); // 8  
    }  
    console.log(x);   // 5  
}
```

Niezmienność Const

tylko referencja jest stała!

Zmienne zadeklarowane z **const nie są niezmienne!** Konkretnie, oznacza to, że obiekty i tablice zadeklarowane poprzez const mogą podlegać modyfikacjom.

```
const myVar = "Grzegorz";
myVar = "Jan" // wywołuje błąd, ponowne przypisanie jest niedozwolone
const myVar = "Olek" // wywołuje błąd, ponowna deklaracja jest niedozwolone
```

Dla obiektów:

```
const person = {
  name: 'Grzegorz'
};
```

person.name = "Jan" // działa!

Zmienna person nie jest ponownie przypisywana, ale ulega zmianie.

person = "Sandra" // wywoła błąd,
ponieważ nie można nadpisywać
zmiennych deklarowanych poprzez const

ALE

W przypadku tablic:

```
const person = [];
```

person.push("Jan"); // działa!

Zmienna person nie jest ponownie przypisywana, ale ulega zmianie.

person = ["Olek"] // wywoła błąd,
ponieważ nie można nadpisywać zmiennych
deklarowanych poprzez const

Instrukcja **const**:

W przypadku deklaracji const dla obiektów to jej zawartość można modyfikować, nie można tylko napisać jej samej.

var, let i const - podsumowanie

- Zmienne możemy tworzyć za pomocą słów kluczowych var/let/const, przy czym zalecane są te dwa ostatnie
- Let/const różnią się od varów głównie zasięgiem oraz tym, że w jednym zasięgu (bloku) nie możemy ponownie tworzyć zmiennych o tej samej nazwie.
- Hoisting to zjawisko wynoszenia na początek skryptu zmiennych i deklaracji funkcji
- W naszych skryptach starajmy się używać jak najwięcej const. Jedynym wyjątkiem są liczniki oraz zmienne które wiemy, że zaraz zmienimy

Instrukcja sterujące

- Warunkowe: if, if/else, switch
- Pętle: while, do/while, for, for ... in, for ... of (ES6)
- Instrukcje używane w pętlach: break, continue
- Obsługa wyjątków: try/catch/finally

Instrukcja warunkowa – porównanie

if-else:

```
if (warunek)
{
    //kod wykonany gdy true
}
else
{
    //kod wykonany gdy false
}
```

switch:

```
switch (zmienna)
{
    case 0:
        x="Gdy zmienna = 0";
        break;
    case 1:
        x="Gdy zmienna = 1";
        break;
    default:
        x="Gdy zmienna różna od 0 i 1";
}
```

Pętle - porównanie

for

```
for (let i=0; i<10; i++) {  
    console.log("Iteracja numer ", i );  
}
```

while

```
let i = 0;  
while (i < 10 ) {  
    console.log("Iteracja numer ", i );  
    i++;  
}
```

do while

```
let i = 0;  
do {  
    console.log("Iteracja numer ", i );  
    i++;  
} while (i < 10 )
```

Pętle

- `for ... in` iteruje po właściwościach obiektu
 - Uwaga: tablice też są obiektami
- `for ... of` iteruje po wartościach właściwości obiektów iterowalnych (tablice, mapy, ...)

```
let arr = [3, 5, 7];

for (let i in arr) {
  console.log(i); // 0, 1, 2
}
for (let i of arr) {
  console.log(i); // 3, 5, 7
}
for (let i in arr) {
  console.log(arr[i]); // 3, 5, 7
}
```

```
let car = { marka: "Fiat", cena: 27000};
for (let i in car) {
  console.log(i); // "marka", "cena"
}

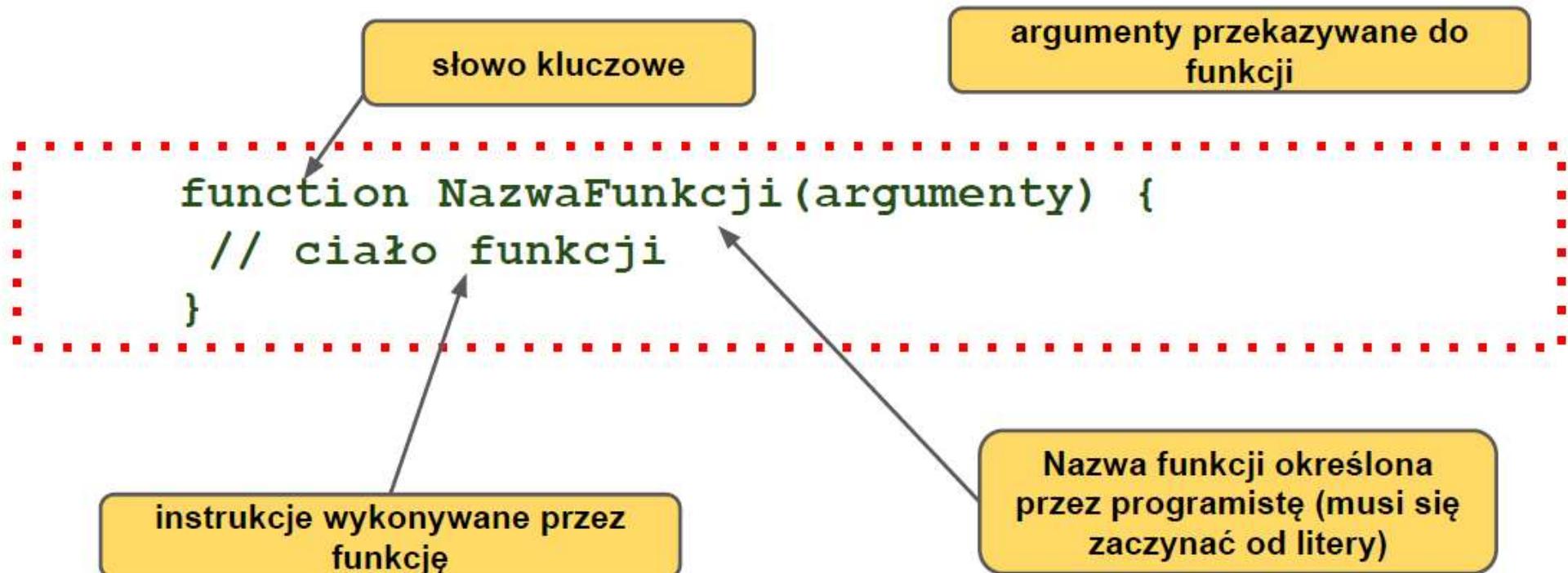
// for (let i of car) {} // error!

for (let i in car) {
  console.log(car[i]); // "Fiat", 27000
}
```

Funkcja w JS - deklaracja funkcji

Zacznijmy od zdefiniowania (narazie) jak tworzyć funkcje, które są podstawowymi narzędziami i jednostkami modularnymi wykorzystywanymi przez programstę JavaScriptu:

Funkcja nazwana:



Sposoby tworzenia funkcji

1 – Standardowa definicja funkcji

```
function displayInPage(message, value) {  
    document.body.innerHTML += message + value + "<br>";  
}  
  
displayInPage("Result: ", result);
```

2 – Użycie wyrażenia funkcyjnego

```
const displayInPage = function(message, value) {  
    document.body.innerHTML += message + value + "<br>";  
};  
  
displayInPage("Result: ", result);
```

Funkcje strzałkowe (arrow functions) - nowy sposób zapisu funkcji

```
function double(x) { return x * 2; } // Tradycyjny sposób  
console.log(double(2)) // 4
```

```
const double = x => x * 2; // Ta sama funkcja jako funkcja strzałkowa z niejawnym zwrotem  
console.log(double(2)) // 4
```

Zwracany obiekt

```
const getPerson = () => ({ name: "Nick", age: 24 })  
console.log(getPerson()) // { name: "Nick", age: 24 } -- obiekt niejawnie zwracany przez arrow function
```

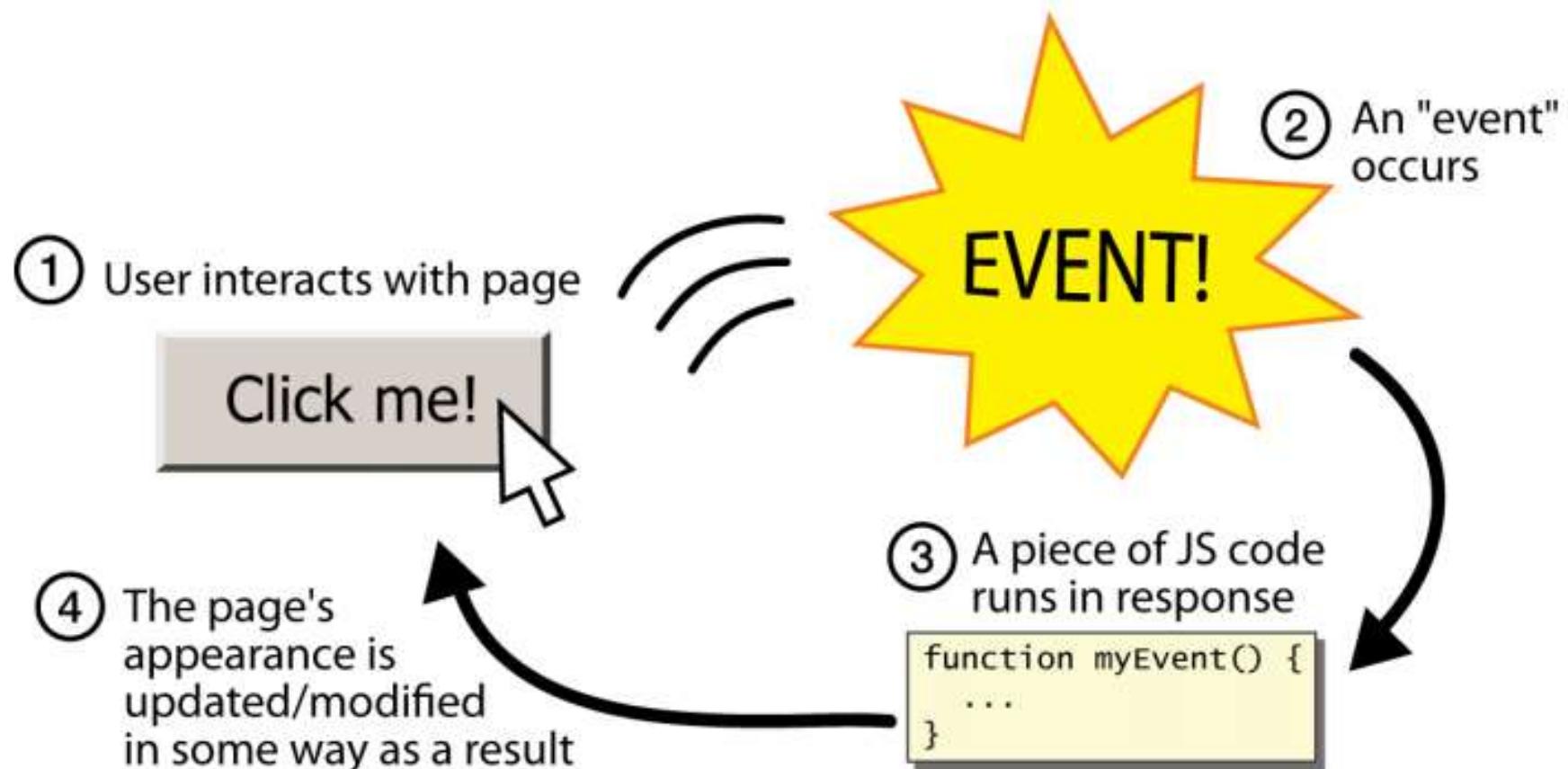
Brak argumentu

```
() => {} // są nawiasy, wszystko w porządku  
const x = 2;  
return x;  
}
```

```
=> {} // brak nawiasów, kod nie będzie działać!  
const x = 2;  
return x;  
}
```

Event-driven programming

Event-Driven Programming: kod jest uruchamiany w wyniku wystawienia zdarzeń generowanych przez użytkownika



Funkcja obsługi zdarzenia (listener) to funkcja, która ma być wykonywana po wystąpieniu danego zdarzenia

Obsługa zdarzeń

Zdarzenia związane z myszą:

onClick	Kliknięcie myszą	onMouseMove	Ruch myszy nad obiektem
onDoubleClick	Podwójne kliknięcie	onMouseOver	Wjechanie myszy nad obiekt
onMouseDown	Wciśnięcie przycisku	onMouseOut	Zjechanie myszy z obiektem
onMouseUp	Puszczenie przycisku		

Zdarzenia związane z klawiaturą:

onKeyDown	Wciśnięcie przycisku
onKeyUp	Puszczenie przycisku
onKeyPress	Naciśnięcie i zwolnienie przycisku

Co potrzebujemy do event Programming?

1. Dostęp do obiektów modelu DOM
2. Zdefiniowanie funkcji do obsługi zdarzenie (funkcja JS)
3. Powiązanie zdarzenia z funkcja JS

Użycie: event listeners

Przykład:

Wyświetlmy "Clicked" w consoli gdy użytkownik naciśnie przycisk:

```
<!DOCTYPE html>
<html>
<body>
<button>Click Me!</button>
</body>
</html>
```

Click me

Potrzebujemy zdefiniować event listener dla przycisku...

Stara szkoła (tak nie robimy!!)

```
<a href="doc.html"  
onMouseOver="document.status='test';return true">  
    Test test Test  
</a>
```

Jak dodać nowa funkcje do obsługi zdarzenia?

Click -> fufu1();

Potem chcemy dodac fufu2();

Może update fufu1()? Lecz co gdy chcemy usunąć fufu2() z kodu fufu1() w trakcie pracy aplikacji ?

Metody rejestracji zdarzeń

Rejestrowanie zdarzenia z użyciem `addEventListener`. W ten sposób możemy podpiąć kilka funkcji obsługujących zdarzenie

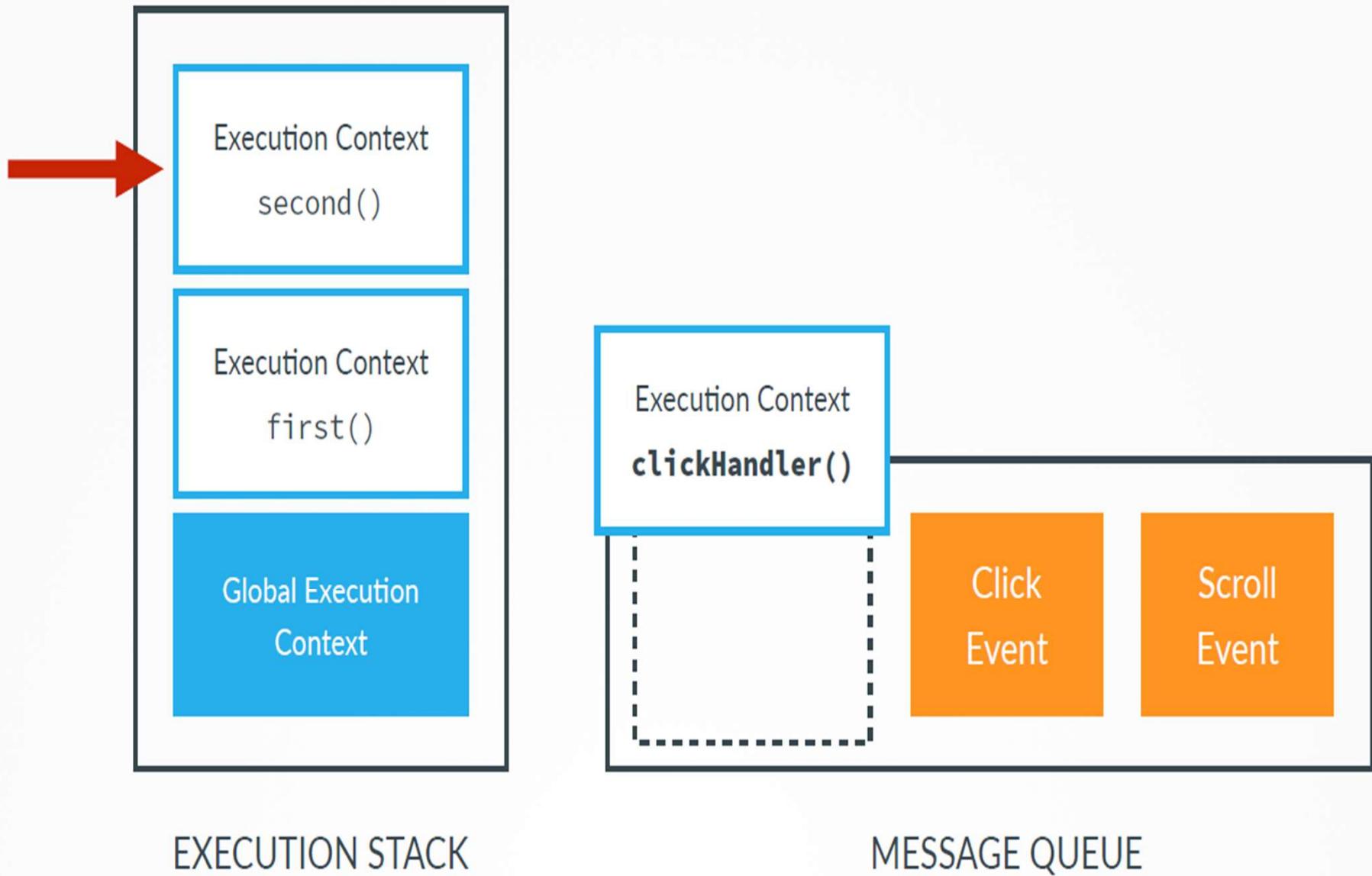
```
let element = document.getElementById('Przycisk');
element.addEventListener('click', startDragDrop, false);
element.addEventListener('click', wypiszCos, false);
element.addEventListener('click', function()
{this.style.color = 'red'; }, false);
```

Ta metoda pozwala również usuwać obsługę zdarzeń przy pomocy `removeEventListener`

```
element.removeEventListener('click', startDragDrop, false);
element.removeEventListener('click', wypiszCos, false);
```

Metoda `removeEventListener` nie będzie działała dla funkcji anonimowej

Procesowanie zdarzeń - jak to działa?



Rozszerzona obsługa zdarzeń

Polega na pobraniu wartości pseudoparametru funkcji obsługi zdarzenia

```
addBtn.addEventListener("click", addGroup);
```

```
function addGroup(e) {  
    console.log("Add button clicked!");  
    console.log(e);  
}
```

srcElement	Element, który wywołał zdarzenie
type	Typ zdarzenia
returnValue	Określa, czy zdarzenie zostało odwołane
cancelBubble	Może odwołać kaskadowe wywołanie zdarzeń
screenX, screenY	Współrzędne kurSORA myszy (względem okna)
pageX, pageY	Współrzędne kurSORA myszy (względem elementu)
button	Czy wciśnięto jakiś przycisk myszy?
altKey, ctrlKey, shiftKey	Czy trzymano przyciski Alt, Ctrl lub Shift
keyCode	Wartość unicode wciśniętego klawisza

Wstrzymanie domyślnej akcji - Prevent default

```
element.addEventListener('click', function (e) {  
    alert('Ten link nigdzie nie przeniesie.');//  
    e.preventDefault();  
}, false);
```

Powstrzymuje domyślną akcję odpalaną na danym eventie (np. nawigację do nowej strony po kliknięciu na link, albo submit formularza)

Event Object Przykład

Add button clicked!

[groupizer.js:100](#)

```
▼ MouseEvent {isTrusted: true, screenX: 301, screenY: 661, clientX: 301, clientY: 559, ...} ⓘ
  altKey: false
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 301
  clientY: 559
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 1
  eventPhase: 0
  fromElement: null
  isTrusted: true
  layerX: 301
  layerY: 559
  metaKey: false
  movementX: 0
  movementY: 0
```

Obiekty związane ze zdarzeniami myszy

Właściwość	Opis
<i>altKey</i>	Informuje, czy podczas zdarzenia kliknięcia wciśnięty jest klawisz ALT na klawiaturze.
<i>button</i>	Informuje, który przycisk myszy został naciśnięty (0 - lewy, 1 środkowy, 2 - prawy przycisk myszy).
<i>buttons</i>	Zwraca numer, który identyfikuje wciśnięty klawisz myszy.
<i>clientX</i>	Zwraca współrzędną x wskaźnika myszy względem danego okna.
<i>clientY</i>	Zwraca współrzędną y wskaźnika myszy względem danego okna.
<i>ctrlKey</i>	Informuje, czy podczas zdarzenia kliknięcia wciśnięty jest klawisz CTRL na klawiaturze.
<i>detail</i>	Informuje ile razy klawisz myszy był naciśnięty.
<i>metaKey</i>	Brak opisu.
<i>pageX</i>	Zwraca współrzędną x wskaźnika myszy względem całego dokumentu.
<i>pageY</i>	Zwraca współrzędną y wskaźnika myszy względem całego dokumentu.
<i>relatedTarget</i>	Brak opisu.
<i>screenX</i>	Zwraca współrzędną x wskaźnika myszy względem ekranu.
<i>screenY</i>	Zwraca współrzędną y wskaźnika myszy względem ekranu.
<i>shiftKey</i>	Informuje, czy podczas zdarzenia kliknięcia wciśnięty jest klawisz SHIFT na klawiaturze.
<i>wwhich</i>	Informuje, który przycisk został naciśnięty.

Mouse Event – inny przykład

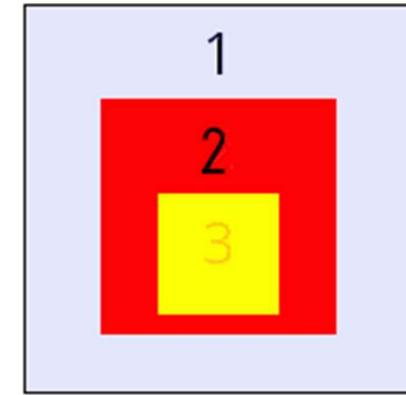
```
let button = document.querySelector("button");
button.addEventListener("mousedown", event => {
    if (event.button == 0) {
        console.log("Left button");
    } else if (event.button == 1) {
        console.log("Middle button");
    } else if (event.button == 2) {
        console.log("Right button");
    }
});
```

Bąbelki - Kaskadowe wykonywanie zdarzeń

Załóżmy istnienie zagnieżdżonych bloków

```
<div style="..." id="blok1">1
  <div style="..." id="blok2">2
    <div style="..." id="blok3">3</div>
  </div>
</div>

<script type="text/javascript">
  let blok1 = document.querySelector("#blok1");
  blok1.addEventListener("click", function() {
    alert('Kliknąłeś mnie!');
  });
</script>
```



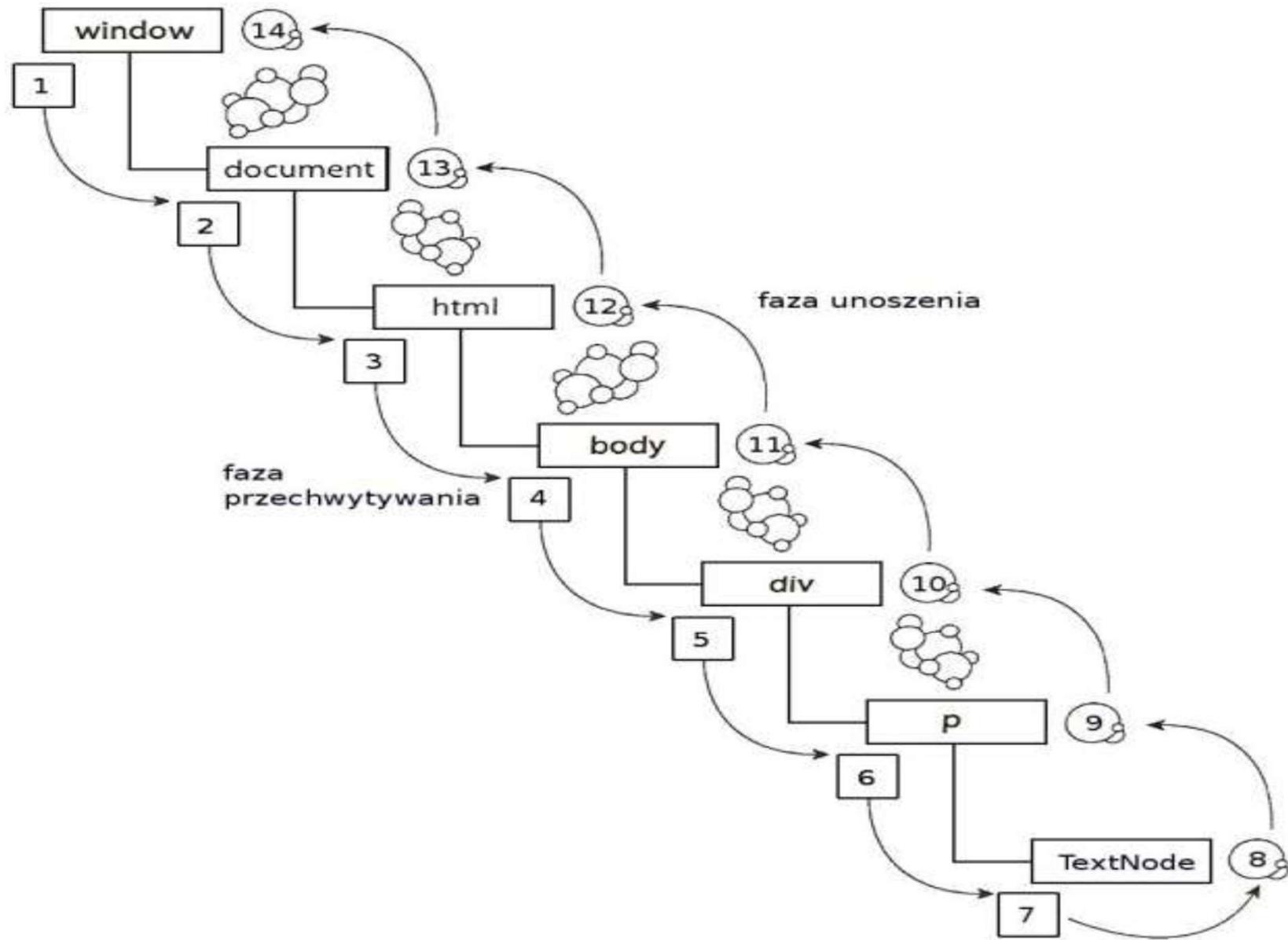
Kliknięcie w element wewnętrzny powoduje po wykonaniu zdarzenia przesłanie go do elementu otaczającego (tu zawsze wywoła się alert)

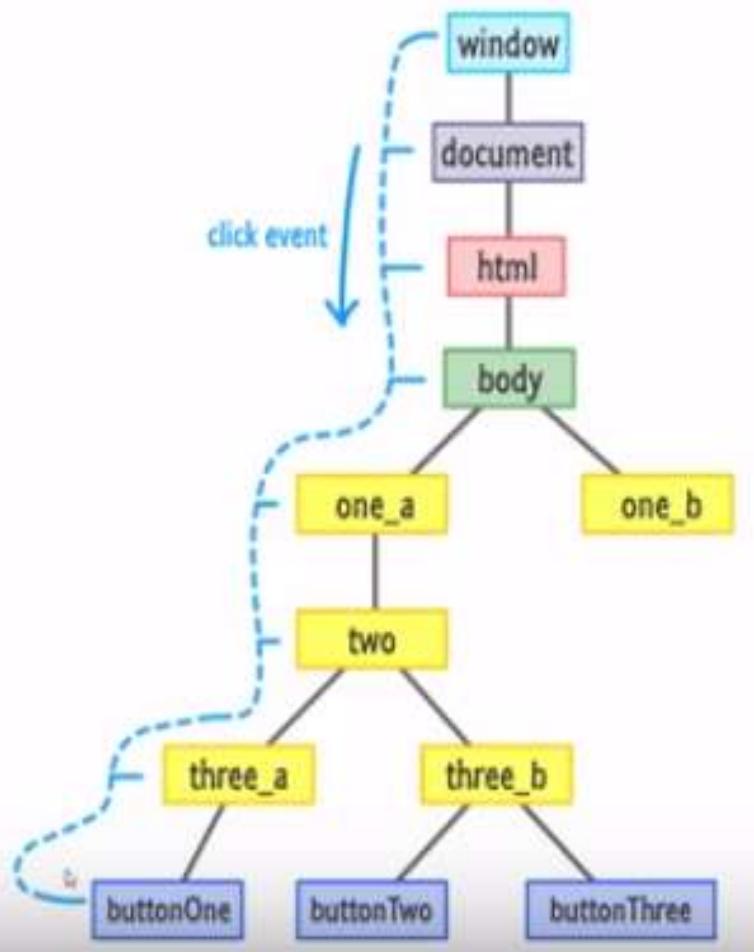
Wyłączenie tego zjawiska realizujemy poprzez funkcję stopPropagation

```
function stopBubble(e) {
  if (!e)
    let e = window.event;
  e.cancelBubble = true;
  if (e.stopPropagation)
    e.stopPropagation();
}
```

```
blok1.addEventListener("click", function() {
  alert('Kliknąłeś mnie!');
});
blok2.addEventListener("click", function(e){
  stopBubble(e);
});
blok3.addEventListener("click", function(e){
  stopBubble(e);
});
```

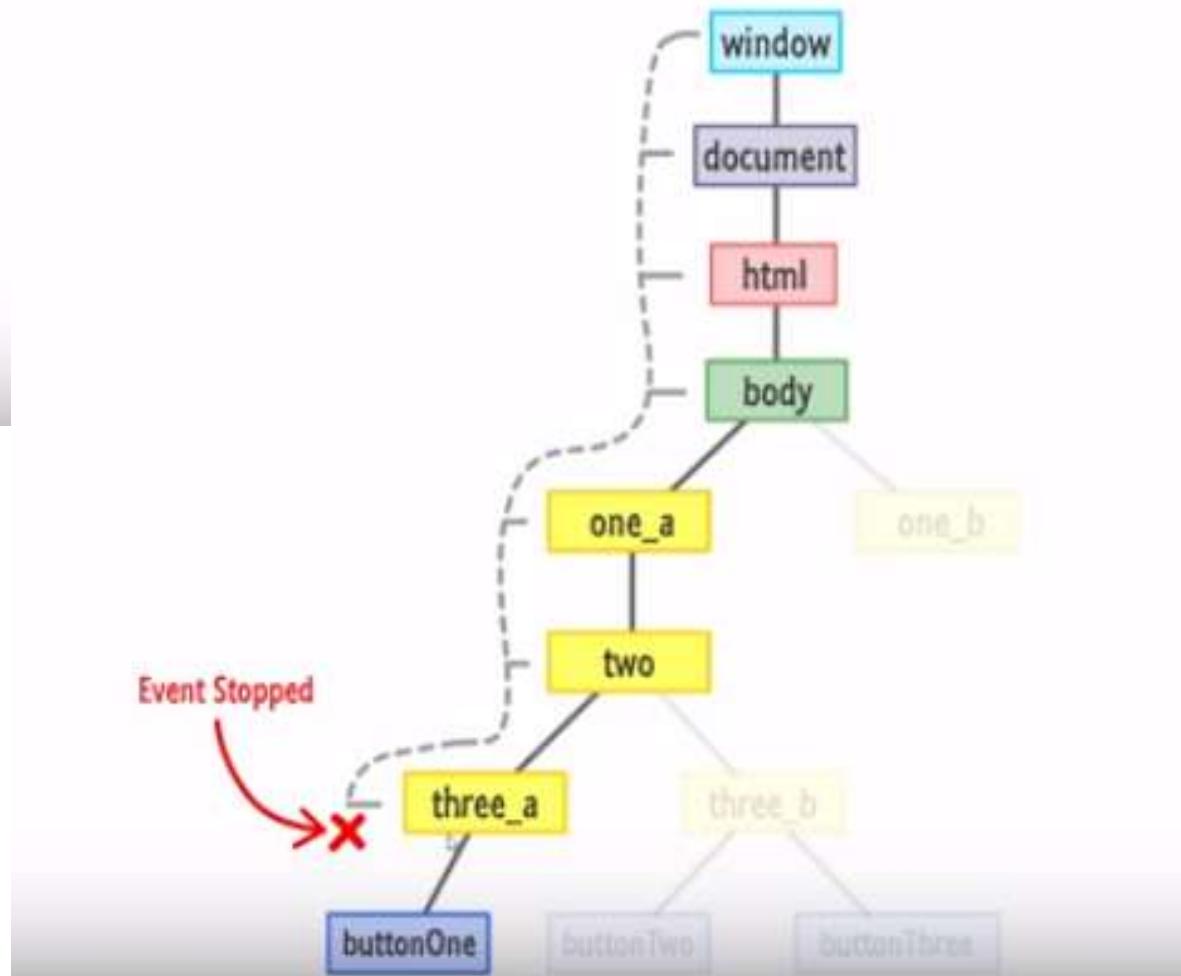
Bąbelki - Kaskadowe wykonywanie zdarzeń





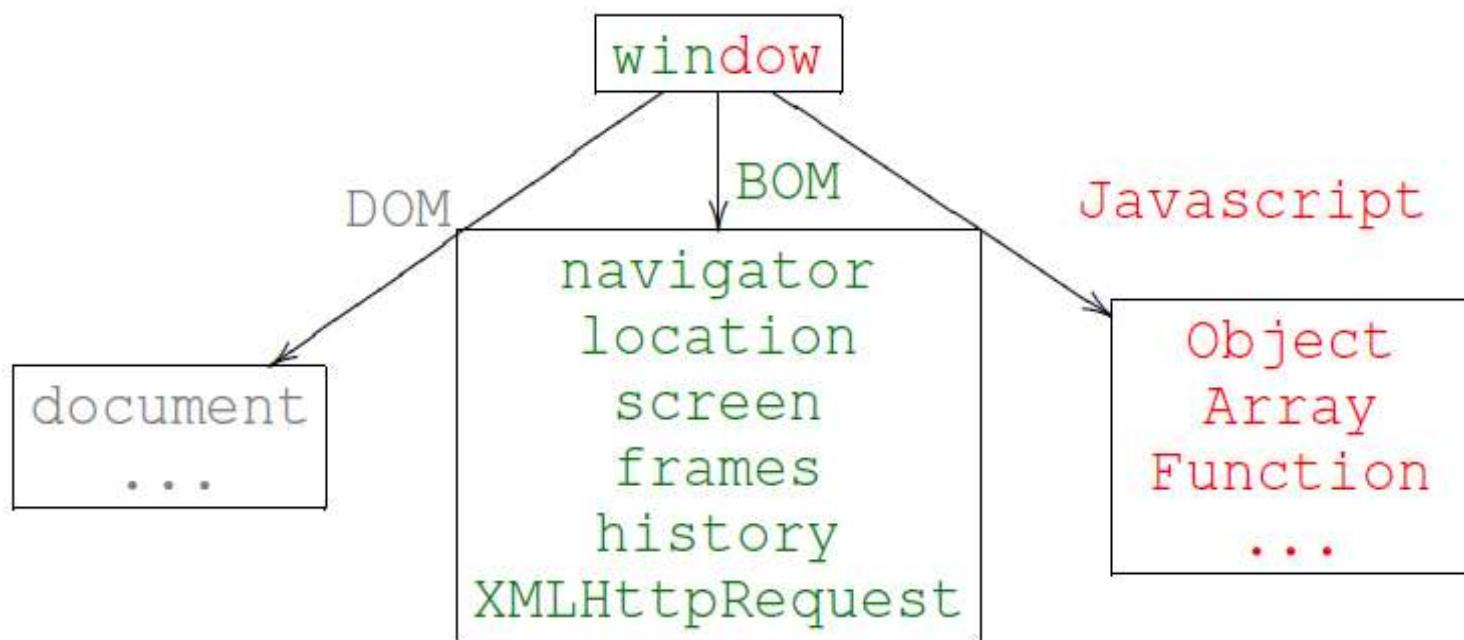
Normalna propagacji zdarzeń

Zatrzymane propagacji zdarzeń



Obiekty przeglądarki

- Przeglądarka pozwala ma dostęp do hierarchii obiektów
- Trzy typy obiektów globalnych



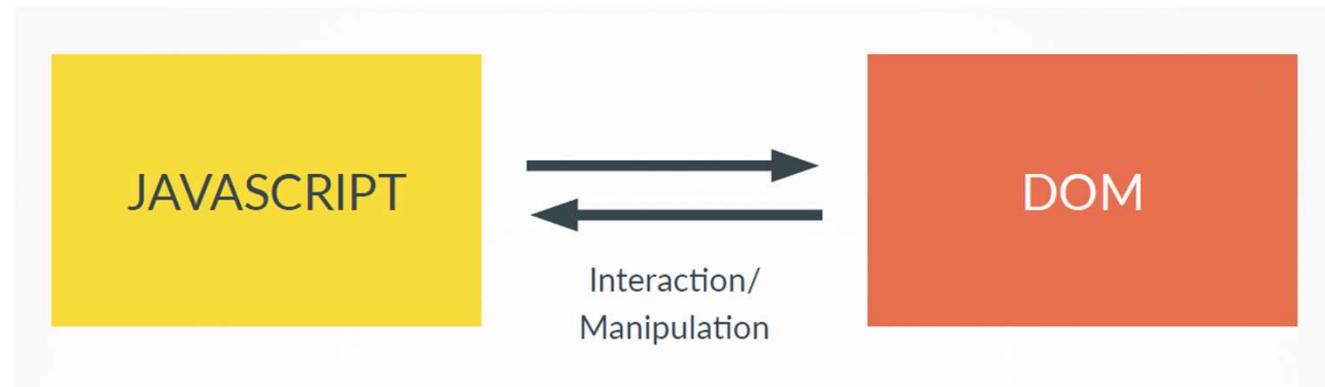
SCHEMAT DOM

Głównym, globalnym obiektem DOM przeglądarki jest `window`. W tym obiekcie przechowywane są wszystkie globalne zmienne i funkcje.

W nim jest także obiekt `document`, który reprezentuje całą stronę www.

W oparciu o DOM JavaScript może:

- Dodawać, zmieniać i usuwać wszystkie elementy HTML i ich atrybuty na stronie;
- Zmieniać wszystkie style i klasy CSS na stronie;
- Dodawać i reagować na wszystkie zdarzenia HTML na stronie;



Wyszukiwanie elementów HTML

Nazwa metody	Po czym szuka	wynik
querySelector	CSS-selector	Pierwszy znaleziony
querySelectorAll	CSS-selector	kolekcja
getElementById	id	element
getElementsByName	name	element
getElementsByTagName	Znacznik lub „*”	kolekcja
getElementsByClassName	class	kolekcja

Dostęp do elementu DOM

```
let els = document.querySelectorAll  
('ul li:nth-child(even)');
```

Zmiana stylu wybranego elementu

```
let p = document.querySelector('#paragraph1');  
p.style.color = 'red';
```

Modyfikacja zawartości elementu

```
let elem = document.querySelector('#myElem');  
elem.innerHTML = 'GR ';
```

Dodanie nowego elementu do DOM

```
let img = document.createElement('img');  
img.width = 200;  
let el = document.querySelector("#test");  
el.append(img);
```

Usunięcie elementu z DOM

```
let list = document.getElementById("myel");  
list.removeChild(list.childNodes[0]);
```

Dodawanie nowego elementu na stronie

```
const el = document.createElement("div");

el.id = „GRdiv”;
el.innerText = „DIV jest moim włascicielem”;
el.setAttribute("title", „Przykład”);
el.classList.add(„GRclass”);
el.style.setProperty("background-color", "#FF6633");
```

```
//do tego div wstawie el div.appendChild(el);
const div = document.querySelector(".test-GR");
```

Właściwości elementów

innerHTML	zwraca lub ustawia kod HTML danego element
outerHTML	zwraca lub ustawia kod HTML wraz z tagiem
innerText	zwraca lub ustawia tekst znajdujący się w elemencie (bez html)
tagName	zwraca nazwę tagu
getAttribute	pobiera atrybut elementu
setAttribute	ustawia atrybut elementu
hasAttribute	sprawdza czy element ma dany atrybut
toggleAttribute	dodaje lub usuwa dany atrybut
dataset	zwraca (obiekt) dataset, który przetrzymuje zdefiniowane przez programiste atrybuty (data-...).

Strategie modyfikacji DOM

1. Zmień zawartość istniejących elementów HTML na stronie
 - innerHtml, innerText

Dobry do prostych aktualizacji tekstowych
2. Dodaj elementy poprzez createElement i appendChild lub usunięcie z modelu DOM.
3. Ukrywanie elementów (właściwość display:none)

Dodawanie i usuwanie węzłów - metody DOM

appendChild() - dodaje nowy podwózeł do danego węzła,

```
body.appendChild(element);
```

removeChild() - usuwa węzeł,

```
body.removeChild(element);
```

replaceChild() - odmienia węzeł,

```
element.replaceChild(nowy_el, stary_el);
```

insertBefore() - wstawia nowy węzeł przed wybranym podwężłem.

```
element.insertBefore(nowy_el, dany_el);
```

Atrybuty i elementy - metody DOM

createAttribute() - tworzy węzeł atrybutu,

```
document.createAttribute("class");
```

createElement() - tworzy węzeł elementu,

```
document.createElement("div");
```

createTextNode() - tworzy węzeł tekstowy,

```
document.createTextNode("napis");
```

getAttribute() - zwraca wartość danego atrybutu,

```
element.getAttribute(nazwaAtrybutu);
```

setAttribute() - ustawia lub zmienia wartość atrybutu.

```
document.getElementById("zdjecie1").setAttribute("src", "zdjcie.jpg");
```

Array i Object

Dwie najczęściej używane struktury danych w JavaScript to:
Obiekt i tablica.

Obiekty pozwalają nam stworzyć pojedynczą jednostkę, która przechowuje elementy danych według klucza.

Tablice pozwalają nam zebrać elementy danych w uporządkowaną listę.

Wyświetlenie właściwości Obiektu

```
const person = {  
    name: "John",  
    age: 30,  
    city: "New York"  
};  
document.getElementById("demo").innerHTML = person.name + "," + person.age +  
"," + person.city;
```

Wyświetlenie obiekt za pomocą pętli

```
const person = {    name: "John",    age: 30,    city: "New York" };  
  
let txt = "";  
for (let x in person) {  
    txt += person[x] + " ";  
}  
  
document.getElementById("demo").innerHTML = txt;
```

Kolekcja danych- Array

Jak tworzymy tablice w JS?

```
let arr = new Array(ele0, ele1, ..., eleN)  
let arr = Array(ele0, ele1, ..., eleN)  
let arr = [ele0, ele1, ..., eleN]; // preferowany
```

```
let fruits = ["Apple", "Orange", "Plum"];
```

dostep do elementu tablicy:

```
arr[0] = "Mango",
```

```
let x = arr[1];
```

Konwersja obiektu w tablice

- Object.keys(obj) – wynik -> tablica kluczy (keys)
- Object.values(obj) – wynik -> tablica wartości (values).
- Object.entries(obj) – wynik -> tablica par klucz, wartość [key, value]

```
let user = {  
    name: "John",  
    age: 30  
};
```

```
Object.keys(user) = ["name", "age"]
```

```
Object.values(user) = ["John", 30]
```

```
Object.entries(user) = [ ["name","John"], ["age",30] ]
```

Array - Lista operacji

Method	Description
<code>list.push(element)</code>	Add <i>element</i> to back
<code>list.unshift(element)</code>	Add <i>element</i> to front

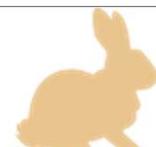
Method	Description
<code>list.pop()</code>	Remove from back
<code>list.shift()</code>	Remove from front

Method	Description
<code>list.indexOf(element)</code>	Returns numeric index for <i>element</i> or -1 if none found

slow method



Fast method



Array – Lista operacji część 2

slice(start_index, upto_index)

```
let myArray = ['a', 'b', 'c', 'd', ,e'];
myArray = myArray.slice(1, 4) // zwraca elementy od 1 do 4 indeksu
tablicy [ "b", "c", "d"]
```

splice(index, count_to_remove, addElem1, addElem2, ...)

```
let myArray = ['1', '2', '3', '4', ,5];
myArray.splice(1, 3, 'a', 'b')
// myArray -> ["1", "a", "b", "5"]
// wycina n elementów od podanego indeksu ( opcjonalnie zastępuje je
podanymi wartościami) .
```

concat()

```
let myArray = ['1', '2', ,3];
myArray = myArray.concat('a', 'b', 'c')
// myArray -> ["1", "2", "3", "a", "b", "c"]
```

Przegląd tablicy

1. Stara szkoła

```
const tab = ["Marcin", "Ania", "Agnieszka"];
for (let i=0; i<tab.length; i++) {
    console.log(tab[i]); // "Marcin", "Ania"...
}
```

2. Lepiej for.each

```
const tab = ["Marcin", "Monika", "Magda"];

tab.forEach(el => {
    console.log(el.toUpperCase()); // "MARCIN", "MONIKA", ...
});
```

3. najlepiej Loop for of

```
const tab = ["Marcin", "Ania", "Agnieszka"];

for (const el of tab) { // el - variable name by us
    console.log(el); // "Marcin", "Ania"...
}
```

Przeglądanie tablicy – zalecane metody

- `Array.includes` // true/false
- `Array.find` // finding element or null
- `Array.indexOf` // index number
- `Array.findIndex` // index number
- `Array.lastIndexOf` // index number
- `Array.filter` // new array
- `Array.Map()` // new array
- `Array.some()` // true/false
- `Array.every()` // true/false

Problem 1 – selekcja elementów na podstawie kryterium

```
const ages = [11, 34, 8, 9, 23, 51, 17, 40, 14];
let tab = [];
for (let i = 0; i < ages.length; i++) {
  if(ages[i] > 18){
    tab.push(ages[i])
  }
}
console.log(tab); // [34, 23, 51, 40]
```

Old fashion –
tak nie robimy

```
const ages = [11, 34, 8, 9, 23, 51, 17, 40, 14];
let tab = ages.filter((age) => age > 18);
console.log(tab); // [34, 23, 51, 40]
```

Nowoczesne
podejście -
Tak robimy

Filter function

```
let results = arr.filter(function(item, index, array)
{ // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});
```

```
let tab = ages.filter((age) => age > 18);
```

Arrow function
dzieki ES5!!!

function fufu(arg1) { expresion} == (arg1, arg2, ...argN) => expression

```
const students = [
  { name: 'Quincy', grade: 96 }, { name: 'Jason', grade: 84 }, { name: 'Alexis', grade: 100 },
  { name: 'Sam', grade: 65 }, { name: 'Katie', grade: 90 }
];
```

```
const studentGrades = students.filter(student => student.grade >= 90);
return studentGrades;
// [ { name: 'Quincy', grade: 96 }, { name: 'Alexis', grade: 100 }, { name: 'Katie', grade: 90 } ]
```

Problem 2 – jak zmodyfikować wszystkie elementy w kolekcji

Metoda map() służy do tworzenia nowej tablicy z istniejącej, stosując funkcję modyfikującą do każdego elementu pierwszej tablicy.

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(item => item * 2);
console.log(doubled);    // [2, 4, 6, 8]
```

```
const tab = ["Marcin", "Monika", "Magda"];
const tab2 = tab.map(el => el.toUpperCase());
console.log(tab); // [Marcin, Ania, Agnieszka]
console.log(tab2); // [MARCIN, ANIA, AGNIESZKA]
```

```
function multiple3(number) {
  return number * 3;
}
var ourTable = [1, 2, 3];
console.log(ourTable.map(multiple3)); // [3, 6, 9]
```

Problem 3 – agregacja wartości danych

- Metoda `reduce()` uruchamia funkcję na każdym elemencie tablicy, aby wytworzyć (zredukować) pojedynczą wartość.
- Metoda `reduce()` działa w tablicy od lewej do prawej. `ReduceRight()` działa w odwrotnym kierunku.
- Metoda `reduce()` nie zmniejsza oryginalnej tablicy.

```
const tab = [3, 2, 4, 2];

const result = tab.reduce((a, b) => a * b);

//1 iteration => prev = 3, curr = 2
//2 iteration => prev = 6, curr = 4
//3 iteration => prev = 24, curr = 2
//result = 48
```

Problem 4 – jak sprawdzić, czy element istnieje

Metody `indexOf()`, `lastIndexOf`, `include` przeszukuje tablicę w poszukiwaniu wartości elementu i zwraca jego pozycję

- `arr.indexOf(item, from)` – szuka elementu zaczynając od indeksu od i zwraca indeks, w którym został znaleziony, w przeciwnym razie -1.
- `arr.lastIndexOf(item, from)` - to samo, ale szuka od prawej do lewej..
- `arr.includes(item, from)` – szuka elementu zaczynając od indeksu od, zwraca true, jeśli został znaleziony.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
```

```
let position = fruits.indexOf("Apple") + 1; // 1
```

```
let position = fruits.lastIndexOf("Apple") + 1; // 3
```

```
fruits.includes("Mango"); // is true
```

Problem 4 – jak sprawdzić, czy element istnieje

Sprawdź, czy każdy element przechodzi test: every()

Sprawdź, czy przynajmniej jeden element przechodzi test: some()

Metoda every() sprawdza, czy wszystkie wartości tablicy przechodzą test.

Metoda some() sprawdza, czy niektóre wartości tablicowe przechodzą test..

W rezultacie obie metody zwracają wartość true lub false.

Every () zwróci wartość true, gdy funkcja przekazana w parametrze zwróci wartość true dla każdego elementu w tablicy.

```
const numbers = [45, 4, 9, 16, 25];  
  
let someOver18 = numbers.some(el => el >= 18 );  
  
let allOver18 = numbers.every(el => el >= 18 );
```

Ten przykład sprawdza, czy niektóre wartości tablicy są większe niż 18: // true

Ten przykład sprawdza, czy wszystkie wartości tablicy są większe niż 18: // false

Problem 5 – jak znaleźć element w kolekcji

Metoda `find()` zwraca wartość pierwszego elementu tablicy, który przeszedł funkcję testową

```
const tab = [12, 5, 8, 130, 44];  
  
const bigNr = tab.find(el => el >= 15);  
console.log(bigNr); //130
```

```
const tab = [  
    { name : "Karol", age: 10, gender: "m" },  
    { name : "Beata", age: 13, gender: „f" },  
    { name : "Marcin", age: 18, gender: "m" },  
    { name : "Ania", age: 20, gender: „f" }  
];  
  
const firstWoman = tab.find(el => el.gender === „f");  
console.log(firstWoman); //{ name : "Beata", age: 13, gender: „f" }
```

Podejście funkcyjne – sklejanie funkcji

Łańcuch metod to sposób na uruchamianie kolejnych metod (funkcji), które piszemy po kropce.

Uruchomienie funkcji zwraca pewną wartość.

Jeśli taka zwrócona wartość pasuje do następnej metody, możemy uruchomić tę metodę natychmiast po kropce:

```
const tab = ["Marcin", "Monika", "Magda"];  
  
const newTab = tab  
  .map(el => el.toLowerCase())    //return new array...  
  .filter(el => el.endsWith("a"))  //...so I can filter  
  .map(el => el + "!")           //...filter return array which I use with map  
  .forEach(el => console.log(el)) //...map return array so forEach match to do it  
  
console.log(newTab)
```

Array of objects Method chaining

```
const students = [
  { name: "Nick", grade: 10 },
  { name: "John", grade: 15 },
  { name: "Julia", grade: 19 },
  { name: "Nathalie", grade: 9 },
];
const aboveTenSum = students
  .map(student => student.grade) // we map the students array to an array of their
grades
  .filter(grade => grade >= 10) // we filter the grades array to keep those 10 or above
  .reduce((prev, next) => prev + next, 0); // we sum all the grades 10 or above one by
one
console.log(aboveTenSum) // 44 -- 10 (Nick) + 15 (John) + 19 (Julia), Nathalie below 10
is ignored
```

Destrukturyzacja

- Destrukturyzacja to specjalna składnia, która pozwala nam „rozpakować” tablice lub obiekty do wielu zmiennych
- Destrukturyzacja działa również świetnie w przypadku złożonych funkcji, które mają wiele parametrów, wartości domyślnych i tak dalej.

```
const person = {  
    firstName: "Nick",  
    lastName:  
    "Anderson",  
    age: 35,  
    sex: "M"  
}
```

Bez destrukturyzacji

```
const first = person.firstName;  
const age = person.age;  
const city = person.city || "Paris";
```

Z destrukturyzacją - wszystko w jednej linii:

```
const { firstName: first, age, city = "Paris" } = person; // Działa!
```

Podsumowanie

- Kiedy musimy iterować po tablicy – możemy użyć `forEach`, `for` lub `for..of`.
- Kiedy musimy iterować i zwracać dane dla każdego elementu – możemy użyć `map`
- Gdy potrzebujemy sprawdzić, czy elementy istnieją, możemy użyć: `indexOf`, `include`, `every`, `some`
- Kiedy potrzebujemy uzyskać eleemnty, które spełniają określone warunki, używamy `filter`.

Problem

dane są nadal osadzone w naszym kodzie.

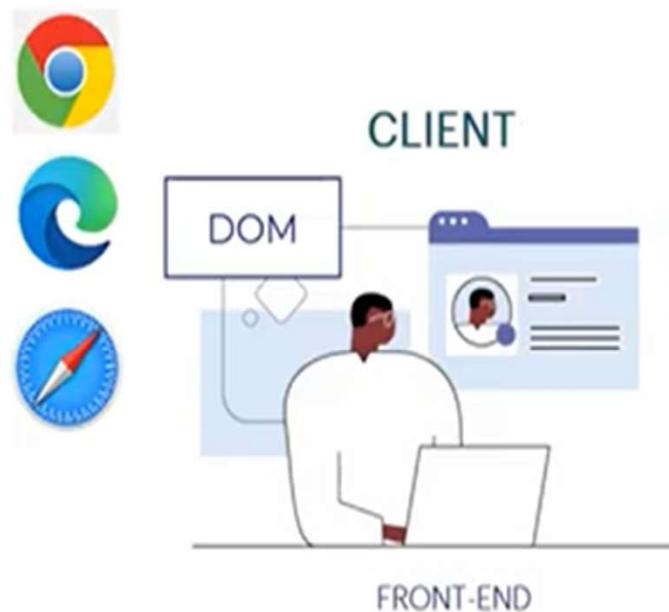
Chcielibyśmy uzyskać dane ze środowiska zewnętrznego.

Jak ładować dane z zewnątrz do naszej aplikacji?

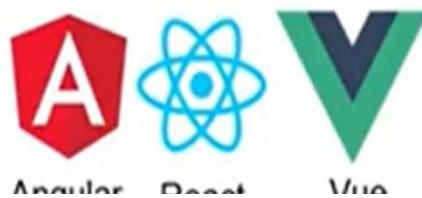
Pamietasz?

Nowoczesne SPA aplikacje ładują pliki json.

Single page applications (SPAs)



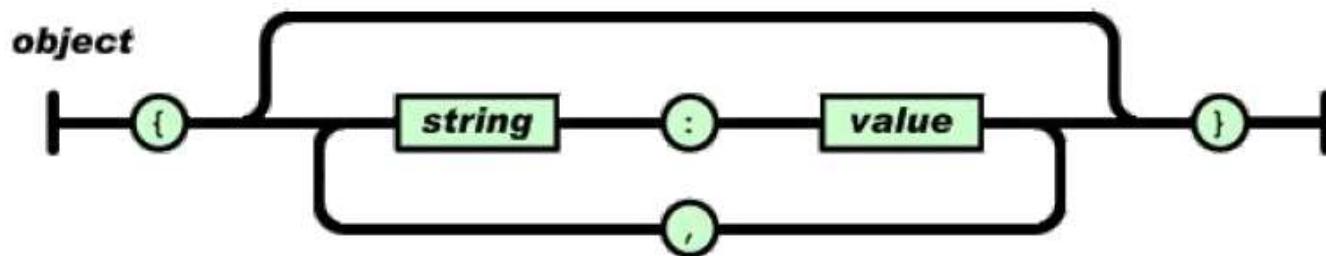
SINGLE PAGE APP
(SPA)



JSON

JSON (JavaScript Object Notation) [Notacja Obiektowa JavaScriptu] - jest to “lekki” format do przenoszenia danych oparty o literały obiektowe JavaScriptu. Jest podzbiorem JS, ale kompletnie niezależnym i może być używany do wymiany danych w zasadzie w każdym współczesnym języku programowania.

<http://www.json.org/>



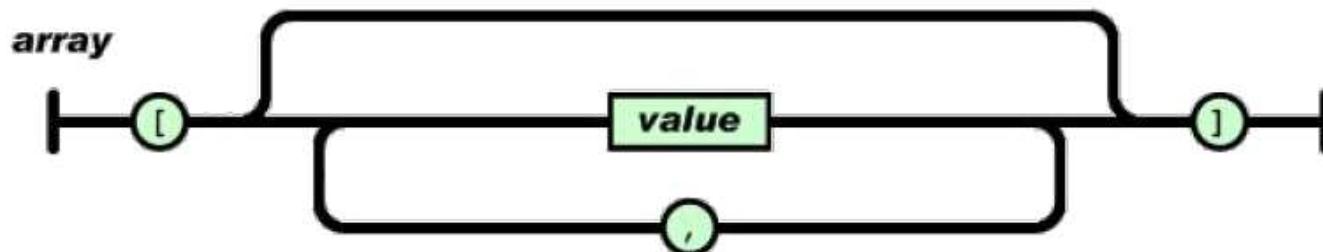
```
{  
  "firstname": "Jan",  
  "lastname": "Kowalski",  
  "age": 20  
}
```

Obiekt w formacie JSON jest nieuporządkowanym zbiorem klucz-wartość, gdzie klucz może być dowolnym łańcuchem, natomiast wartość jednym z dowolnych typów (integer, string etc) włączając w to tablice i inne obiekty.

JSON

W formacie JSON możemy także posługiwać się tablicami, które tworzą uporządkowane ciągi wartości o dowolnych typach dozwolonych przez JSONa (w tym tablice i obiekty).

<http://www.json.org/>



```
[ {  
    "firstname": "Jan",  
    "lastname": "Kowalski",  
    "age": 20  
,  
{  
    "firstname": "Anna",  
    "lastname": "Nowak",  
    "age": 25  
} ]
```

Uwaga:

Podobnie jak w JavaScript niedopuszczalne jest rozpoczęwanie liczb całkowitych od "zera" np.:

```
{ "liczba": 023 }
```

W niektórych przypadkach zapis taki może zostać zinterpretowany jako liczba w formacie ósemkowym.

Tablica Obiektów

Przykład tablicy obiektów:

```
{ "samochod": [  
    {  
        "Marka": "VW",  
        "Model": "Golf",  
        "Rocznik": 1999  
    },  
    {  
        "Marka": "BMW",  
        "Model": "S6",  
        "Rocznik": 2007  
    },  
    {  
        "Marka": "Audi",  
        "Model": "A4",  
        "Rocznik": 2009  
    }  
]
```

Format JSON do złudzenia przypomina klasyczne obiekty w JavaScript

Obiekt JSON

W pracy z formatem JSON w JavaScript bardzo pomocny okaże się obiekt JSON.

Udostępnia on nam 2 metody: **stringify()** i **parse()**.

Pierwsza z nich zamienia dany obiekt na tekstowy zapis w formacie JSON.

Druga z nich zamienia zakodowany wcześniej tekst na obiekt JavaScript:

```
const ob = { name : "Grzegorz", surname : "Rogus" }
```

```
const obStr = JSON.stringify(ob);
console.log(obStr); //>{"name":"Grzegorz","surname":"Rogus"}
```

```
console.log( JSON.parse(obStr) ); //nasz wcześniejszy obiekt
```

JSON stringify/parse example

```
> let point = {x: 1, y: 2, z: 3}
> point
> {x: 1, y: 2, z: 3}

> let s = JSON.stringify(point);
> s
> '{"x":1,"y":2,"z":3}"

> s = s.replace("1", "4");
> '{"x":4,"y":2,"z":3}"

> let point2 = JSON.parse(s);
> point2
> {x: 4, y: 2, z: 3}
```

Gdzie używamy JSON?

Dane JSON pochodzą z wielu źródeł w sieci:

- usługi internetowe używają JSON do komunikacji
- serwery WWW przechowują dane w postaci plików JSON
- bazy danych czasami używają JSON do przechowywania, odpytywania i zwracania danych

JSON to de facto uniwersalny format wymiany danych

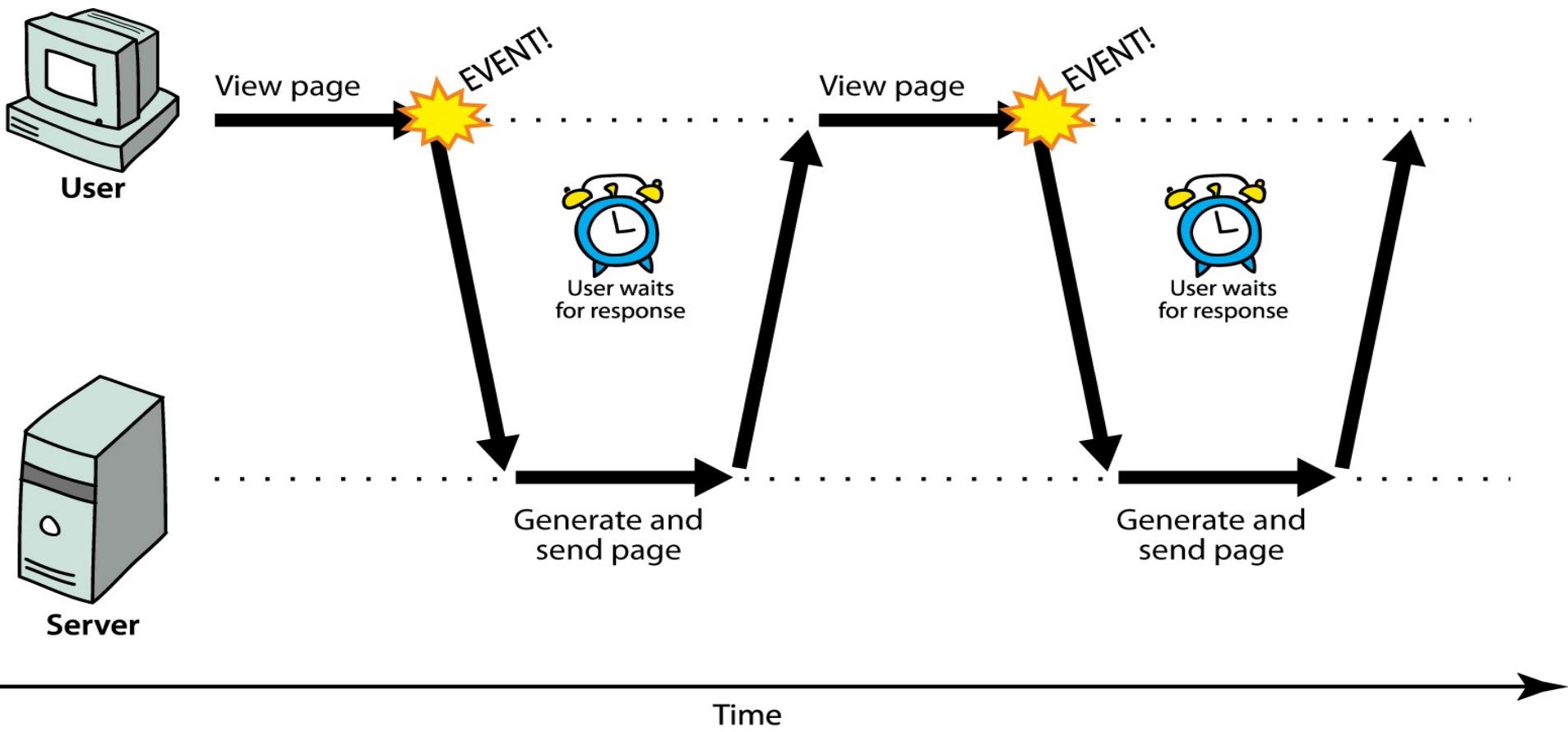
Serwer zewnętrzny

- Serwer lokalny
 - npm i http-server -g
 - //lub
 - npm i live-server -g
- json-server
 - npm install json-server -g
 - json-server --watch nazwa-pliku.json
- Fakowy serwer w chmurze np. <https://jsonplaceholder.typicode.com/posts>

AJAX

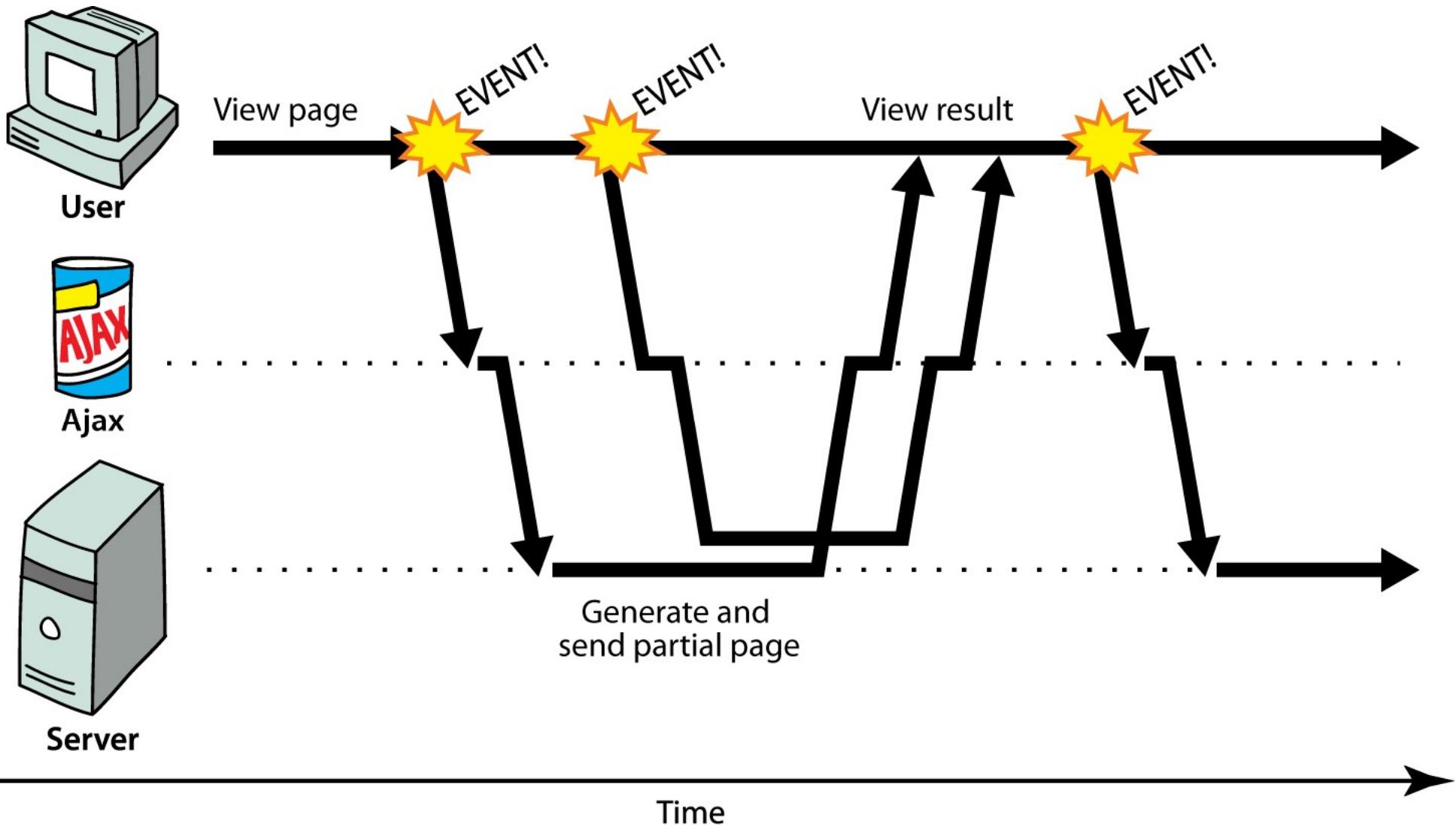
Asynchronous JavaScript and XML

Zapytania synchroniczne



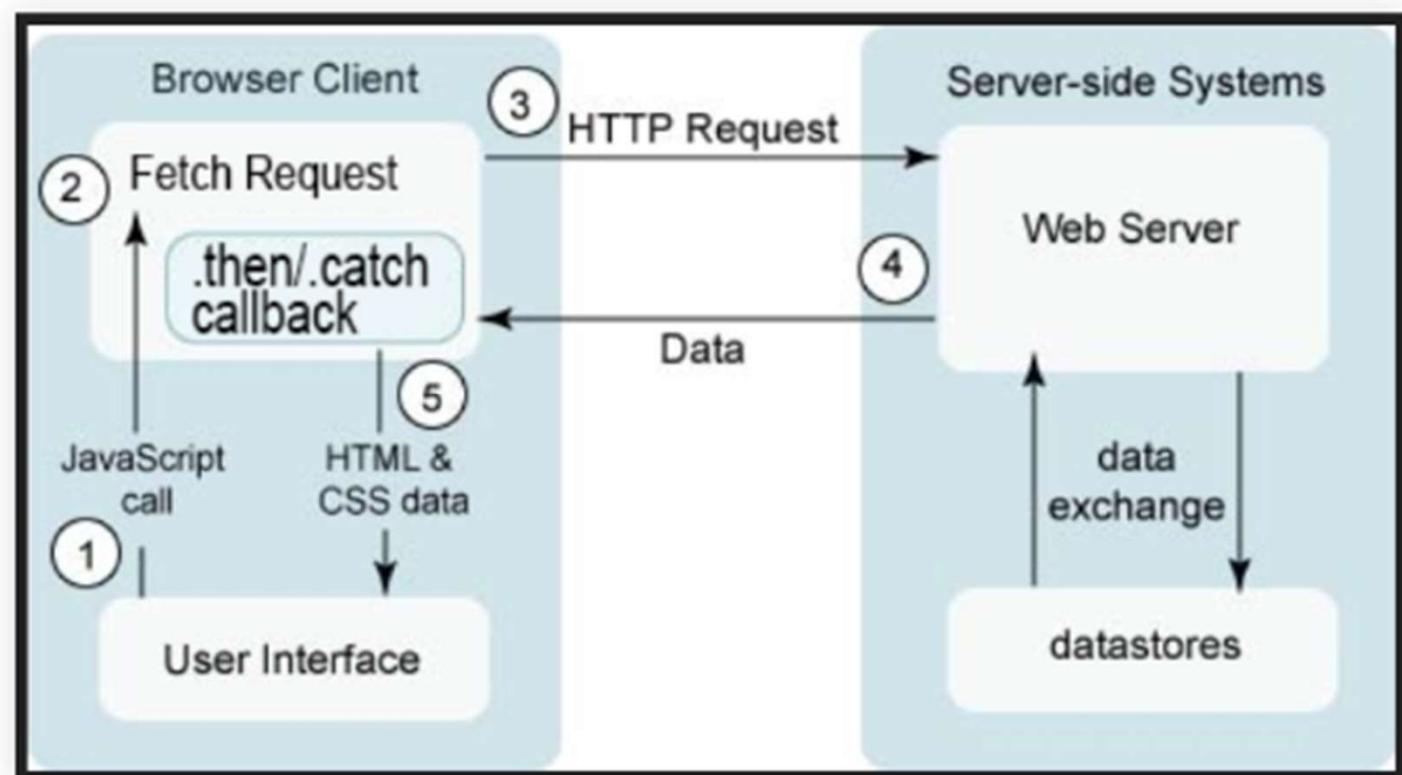
Dlaczego zapytania synchroniczne są problemem?

Zapytania asynchroniczne

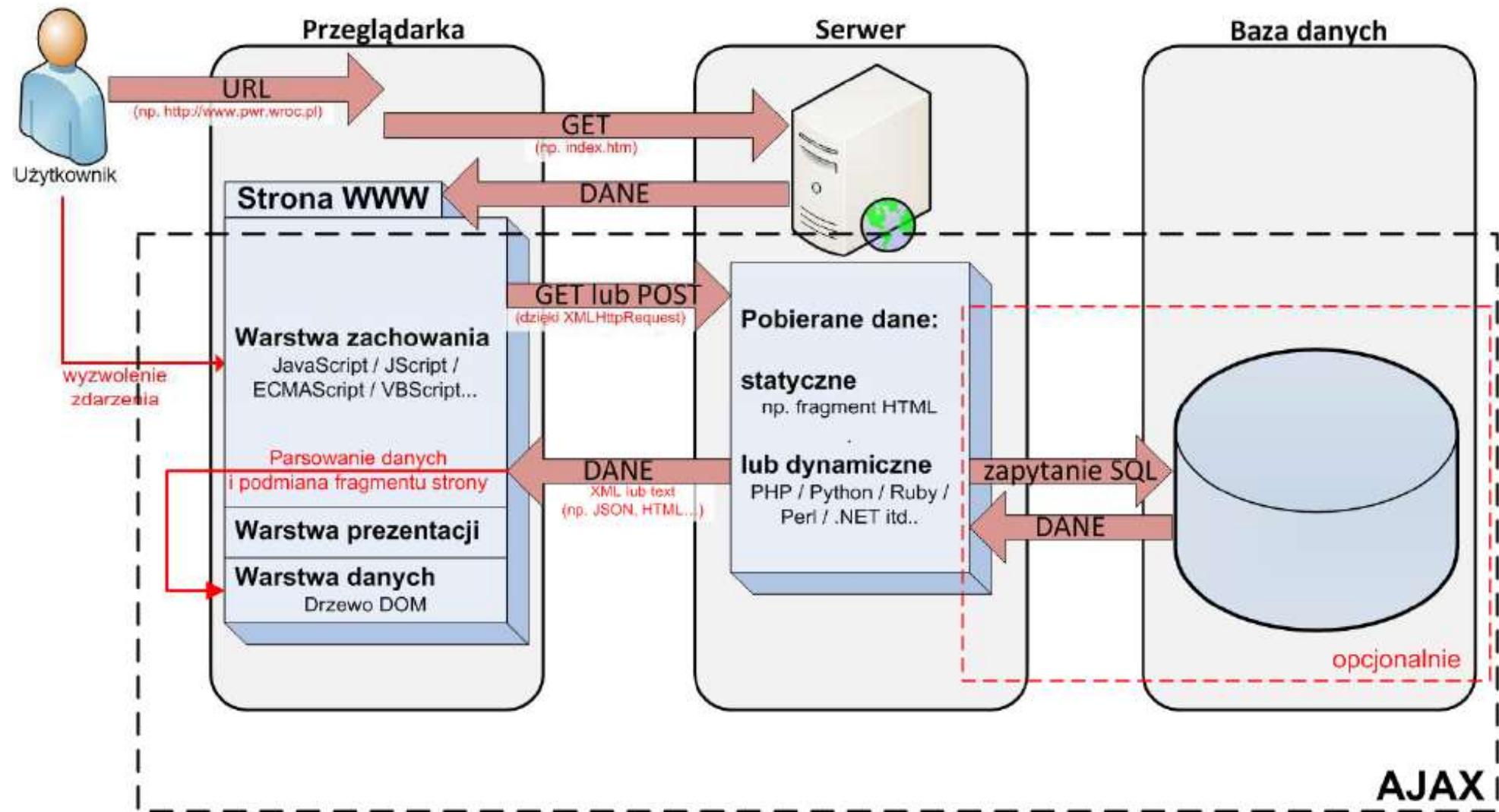


Dlaczego AJAX?

- Możesz użyć AJAX do pobierania informacji z serwera w tle
- Umożliwia dynamiczne aktualizowanie strony bez zmuszania użytkownika do czekania
- Unika „kliknij-czekaj... odśwież”, co mogłoby frustrować użytkowników



Jak działa AJAX?



AJAX (asynchroniczność)

Aby rozpocząć korzystanie z AJAX należy utworzyć obiekt klasy

XMLHttpRequest():

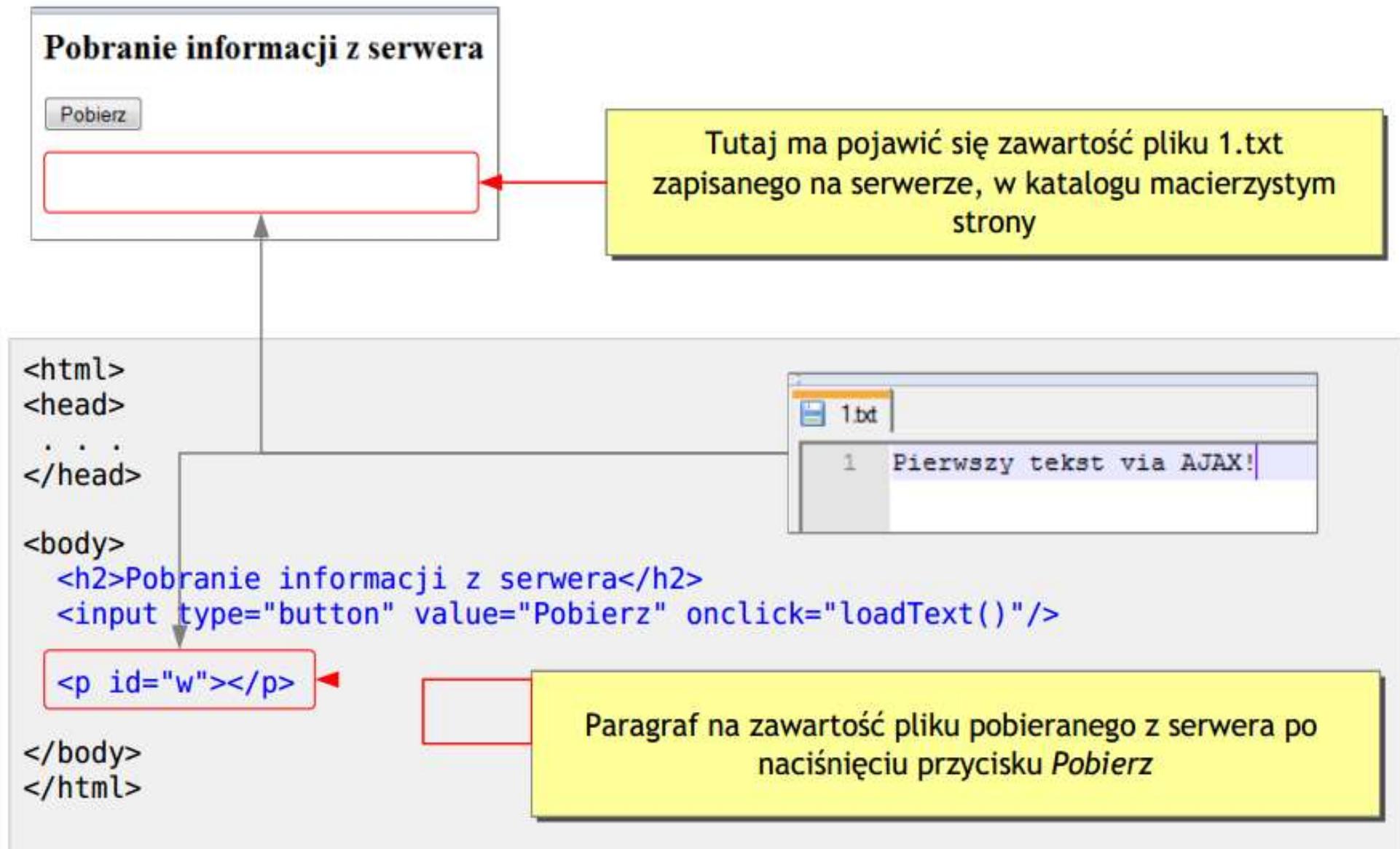
```
var request = new XMLHttpRequest();
```

Od tego miejsca mamy dostęp do zestawu metod i właściwości przynależących do klasy XMLHttpRequest, które pozwalają na obsługę połączenia.

Zestaw właściwości:

- **onreadystatechange** - zdarzenie odpalone w chwili zmiany stanu danego połączenia
- **readyState** - zawiera aktualny status połączenia
- **responseText** - zawiera zwrócone dane w formie tekstu
- **responseXML** - zawiera zwrócone dane w formie drzewa XML (jeżeli zwrócone dane są prawidłowym dokumentem XML)
- **status** - zwraca status połączenia np 404 - gdy strona nie istnieje, itp)
- **statusText** - zwraca status połączenia w formie tekstowej - np 404 zwróci Not Found

XMLHttpRequest — uproszczony przykład poglądowy, cel



XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();
```

Utworzenie obiektu realizującego komunikację
via HTTP

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();  
.  
.  
requester.open( "GET", "1.txt", true );
```

O określenie typu żądania (GET, POST), adresu URL którego żądanie będzie dotyczyć, oraz czy żądanie będzie wykonywane asynchronicznie (true), czy synchronicznie (false).

Uwaga – wywołanie open tak naprawdę nie powoduje „ruchu sieciowego”. Jest to po prostu „skonfigurowanie” parametrów żądania.

Uwaga – w trybie synchronicznym po wysłaniu żądania trwa oczekiwanie, inne akcje są wstrzymane.

AJAX == wywołania asynchroniczne

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();  
.  
.  
.  
requester.open( "GET", "1.txt", true );  
requester.send();
```

Wysłanie żądania, dla żądań GET funkcja nie otrzymuje parametrów.

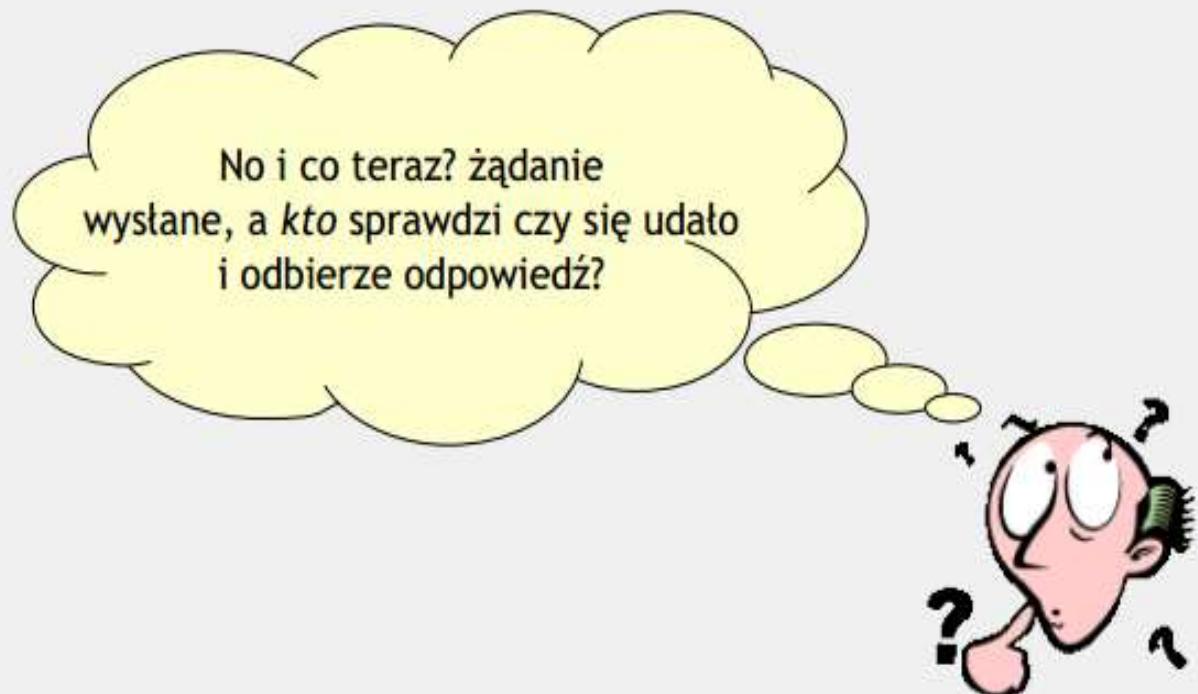
Funkcja send otrzymuje parametr w postaci łańcucha znaków dla żądań POST.

Spotyka się wywołania w postaci:

```
requester.send( null );
```

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();  
  
...  
  
requester.open( "GET", "1.txt", true );  
requester.send();
```



XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();

requester.onreadystatechange = ??? // ←

requester.open( "GET", "1.txt", true );

requester.send();
```

Ustalenie funkcji nasłuchującej – będzie ona wywoływana po każdej zmianie statusu obsługi żądania, w tym jego zakończeniu (odebraniu danych).

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();
```

```
requester.onreadystatechange = function()  
{  
    . . .  
}
```

```
requester.open( "GET", "1.txt", true );  
requester.send();
```

Anonimowa funkcja wywoływana po zmianie statusu żądania.

Jest to funkcja typu *callback*.

Oczywiście nie musi to być funkcja anonimowa, to tylko prosty przykład.

Uwaga – funkcja nasłuchująca jest bezparamterowa!

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();

requester.onreadystatechange = function()
{
    if( requester.readyState == 4 )
        . .
}

requester.open( "GET", "1.txt", true );
requester.send();
```

Właściwość *readyState* obiektu klasy XMLHttpRequest zawiera informację o jego stanie w trakcie obsługi żądania.

Znaczenie wartości w *readyState*:

- 0 - niezainicjowany (zapewne przed *open*)
- 1 - zainicjowany (po *open* a przed *send*)
- 2 - żądanie wysłane, w trakcie przetwarzania,
- 3 - odbieranie danych
- 4 - zakończono obsługę żądania

Uwaga – *readyState == 4* oznacza że serwer zakończył obsługę żądania. Ale to nie znaczy, że żądany zasób został dostarczony!

Przeglądarki mogą różnie ustawiać stan właściwości *readyState*.

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();

requester.onreadystatechange = function()
{
    if( requester.readyState == 4 )
        if( requester.status == 200 )
    {
        . . .
    }
}

requester.open( "GET", "1.txt", true );
requester.send();
```

Właściwość *status* obiektu klasy XMLHttpRequest zawiera informację o kodzie statusu żądania HTTP.

Znaczenie wartości w *status*, np:
200: OK
401: Unauthorized
403: Forbidden
404: Not Found

Uwaga – nie wystarczy tylko sprawdzenie czy *readyState == 4*

Kontrola czy status HTTP jest równy 200 jest konieczna

XMLHttpRequest — uproszczony przykład poglądowy

```
var requester = new XMLHttpRequest();

requester.onreadystatechange = function()
{
  if( requester.readyState == 4 )
    if( requester.status == 200 )
    {
      var out = document.getElementById( "w" );
      out.innerHTML += "<br \/> " + requester.responseText;
    }
}

requester.open( "GET", "1.txt", true );
requester.send();
```

var out = document.getElementById("w");
out.innerHTML += "<br \/> " + requester.responseText;

Odpowiedź serwera może byćłańcuchem znaków, dostępnym poprzez właściwość *responseText*.

Odpowiedź serwera może być postać XML dostępą poprzez właściwość *responseXML*.

Tak naprawdę o tym, to, co będzie odpowiedzią rozstrzyga się po stronie serwera i pracującego tam softu.

XMLHttpRequest — uproszczony przykład poładowy

```
var requester = new XMLHttpRequest();

requester.onreadystatechange = function()
{
    if( requester.readyState == 4 )
        if( requester.status == 200 )
        {
            var out = document.getElementById( "w" );
            out.innerHTML = requester.responseText;
        }
}

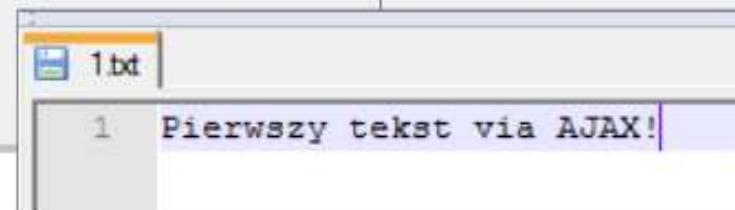
requester.open( "GET", "1.txt", true );
requester.send();
```

W naszym przypadku odpowiedzią ma być zawartość pliku 1.txt zapisanego w katalogu macierzystym serwera.

Pobranie informacji z serwera

Pobierz

Pierwszy tekst via AJAX!



XMLHttpRequest — uproszczony przykład poglądowy

```
<html>
<head>
<script type="text/javascript">
function loadText()
{
    var requester;
    requester = new XMLHttpRequest();
    requester.onreadystatechange = function() {
        if( requester.readyState == 4 )
            if( requester.status == 200 )
            {
                var out = document.getElementById( "w" );
                out.innerHTML = requester.responseText;
            }
    }
    requester.open( "GET", "1.txt", true );
    requester.send( null );
}
</script>
</head>
<body>
    <h2>Pobranie informacji z serwera</h2>
    <input type="button" value="Pobierz" onclick="loadText()" />
    <p id="w"></p>
</body>
</html>
```

XMLHttpRequest — uproszczony przykład poglądowy

```
<html>
<head>
<script type="text/javascript">
var requester;

function processText()
{
  if( requester.readyState == 4 )
    if( requester.status == 200 )
    {
      var out = document.getElementById( "w" );
      out.innerHTML = requester.responseText;
    }
}

function loadText()
{
  requester = new XMLHttpRequest();

  requester.onreadystatechange=processText;
  requester.open( "GET", "1.txt", true );
  requester.send( null );
}

</script>
</head>
. . .
```

Globalny obiekt dostępny w całej sekcji
sekcji script.

Funkcja nasłuchująca pod lupą

```
var requester;

function processText()
{
    var out = document.getElementById( "w" );
    out.innerHTML += "<br \/>Readystate " + requester.readyState;

    if( requester.readyState == 4 )
        if( requester.status == 200 )
        {
            out.innerHTML += "<br \/>Status " + requester.status;
            out.innerHTML += "<br \/> " + requester.responseText;
        }
    else
    {
        out.innerHTML += "<br \/>Status " + requester.status;
        out.innerHTML += "<br \/>Niepowodzenie!";
    }
}
.

requester.onreadystatechange = processText;
```

Brak żądanego zasobu, mimo że właściwość `readyState == 4` zasób na serwerze nie został znaleziony – status HTTP == 404

Pobranie informacji

Pobierz

Readystate 1
Readystate 1
Readystate 2
Readystate 3
Readystate 4
Status 200
Pierwszy tekst via AJAX!

Pobranie informacji

Pobierz

Readystate 1
Readystate 1
Readystate 2
Readystate 3
Readystate 4
Status 404
Niepowodzenie!

FETCH API

nowy interfejs do dynamicznego
pobierania zasobów

Fetch API

fetch(url, [options]);

```
fetch("./data/list.json")
  .then((response) => {
    console.log("OK", response);
  })
  .catch((err) => {
    console.log("error", err);
  });
}
```

Składowe obiektu Response

ok	czy połączenie zakończyło się sukcesem i możemy zacząć pracować na danych
status	statusy połączenia (200, 404, 301 itp.)
statusText	status połączenia w formie tekstowej (np. Not found)
type	typ połączenia
url	adres na jaki się łączymy
body	właściwe ciało odpowiedzi

response.text()

zwraca odpowiedź w formacie text

response.json()

zwraca odpowiedź jako JSON

response.formData()

zwraca odpowiedź jako FormData

response.blob()

zwraca odpowiedź jako blob

response.arrayBuffer()

zwraca odpowiedź jako ArrayBuffer

Pobieranie danych

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(resp =>
    { console.log(resp); }
  )
```

```
fetch("https://jsonplaceholder.typicode.com/users")
)
  .then(resp => {
    console.log(resp.headers.get("Content-Type"));
    console.log(resp.headers.get("Date"));
    console.log(resp.status);
    console.log(resp.statusText);
    console.log(resp.type);
    console.log(resp.url);
    console.log(resp.body);
    ...
  })
```

```
▼ (10) [Object, Object, Object, Object, Object, Object, Object, Object, Object, Object]
  ▼ 0: Object
    ► address: Object
    ► company: Object
      email: "Sincere@april.biz"
      id: 1
      name: "Leanne Graham"
      phone: "1-770-736-8031 x56442"
      username: "Bret"
      website: "hildegard.org"
    ► __proto__: Object
  ▶ 1: Object
  ▶ 2: Object
```

```
▼ Response ⓘ
  ▼ body: ReadableStream
    locked: false
  ► __proto__: Object
  bodyUsed: false
  ▼ headers: Headers
    ► __proto__: Headers
    ok: true
  redirected: false
  status: 200
  statusText: ""
  type: "cors"
  url: "https://jsonplaceholder.typicode.com/posts"
  ► __proto__: Response
```

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(resp => resp.json())
  .then(resp => {
    console.log("Przykład:");
    console.log(resp);
  })
```

Przetwarzanie zwracanych danych

Teraz, gdy wykonaliśmy pobieranie, musimy zrobić coś z danymi, które wracają z serwera.

Ale nie wiemy, jak długo to potrwa i czy w ogóle wróci poprawnie!

Wywołanie pobierania zwraca obiekt Promise, który pomoże nam z tą niepewnością.

Jak stworzyć zapytanie do własnego serwera za pomocą Fetch API?

```
fetch('/api/content/all')
  .then(function (response) {
    // response jest instancją interfejsu Response
    if (response.status !== 200) {
      return Promise.reject('Zapytanie się nie powiodło');
    }
    // zwracamy obiekt typu Promise zwracający dane w postaci JSON
    return response.json();
  })
  .then(this._doSomethingWithJson)
  .catch(this._catchError);
```

Jak stworzyć zapytanie do własnego serwera za pomocą Fetch API?

```
fetch("...", {  
    method: 'POST', // *GET, POST, PUT, DELETE, etc.  
    mode: 'cors', // no-cors, *cors, same-origin  
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached  
    credentials: 'same-origin', // include, *same-origin, omit  
    headers: {  
        'Content-Type': 'application/json'  
        // 'Content-Type': 'application/x-www-form-urlencoded',  
    },  
    redirect: 'follow', // manual, *follow, error  
    referrerPolicy: 'no-referrer', // no-referrer, *client  
    body: JSON.stringify(data) // treść wysyłana  
})
```

Wysłanie danych

Żeby wysłać dane musimy je ustawić we właściwości body.

```
fetch("...", {
  method: "post",
  body: "name=Grzegorz&surname=Rogus"
})
.then(res => res.json())
.then(res => {
  console.log("Dodałem użytkownika:");
  console.log(res);
})
```

Wysyłka Jsona

```
const ob = {
  title: "Nazwa posta",
  body: "Lorem ipsum dolor",
  userId: 1
}.
```

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "post",
  headers: {
    "Content-type": "application/json; charset=UTF-8"
  },
  body: JSON.stringify(ob)
})
.then(res => res.json())
.then(res => {
  console.log("Dodałem użytkownika:");
  console.log(res);
})
```

Jak stworzyć zapytanie do zewnętrznego serwisu za pomocą Fetch API?

```
const headers = new Headers({  
    'Content-Type': 'text/plain'  
});  
  
const request = new Request({  
    method: 'POST',  
    mode: 'cors',  
    headers: headers  
});  
  
fetch('https://test.pl/api/content/all', request)  
    .then(this._handleResponse)  
    .catch(this._catchError);
```

Kolejny przykład użycia FetchAPI

```
fetch("./data/GR_cities.json")
  .then((response) => {
    if (response.status !== 200) {
      console.log("są błędy");
    }
    console.log("OK", response);
  })
  .catch((err) => {
    console.log("błąd podczas pobierania danych", err);
  });
});
```

Kolejny przykład użycia FetchAPI – async/await

```
const API_URL = 'https://jsonplaceholder.typicode.com/users';

async function fetchUsers() {
  try {
    const response = await fetch(API_URL)
    const users = await response.json();
    return users;
  } catch(err) {
    console.error(err);
  }
}

fetchUsers().then(users => {
  users; // zwrócenie użytkownicy
});
```

PROMISES - OBIETNICE

Promise to relatywnie nowa konstrukcja javascript wprowadzona w EcmaScript 6.

Jest to odpowiedź na składnię wykorzystującą obiekty konstruktora XMLHttpRequest, który to powodował bałagan w kodzie i tzw. Callback Hell.

Piekło Callback (callback hell)

```
doSomething(function(result) {  
    doSomethingElse(result, function(newResult) {  
        doThirdThing(newResult, function(finalResult) {  
            console.log('Wreszcie się udało otrzymać: ' + finalResult);  
        }, failureCallback);  
    }, failureCallback);  
}, failureCallback);
```

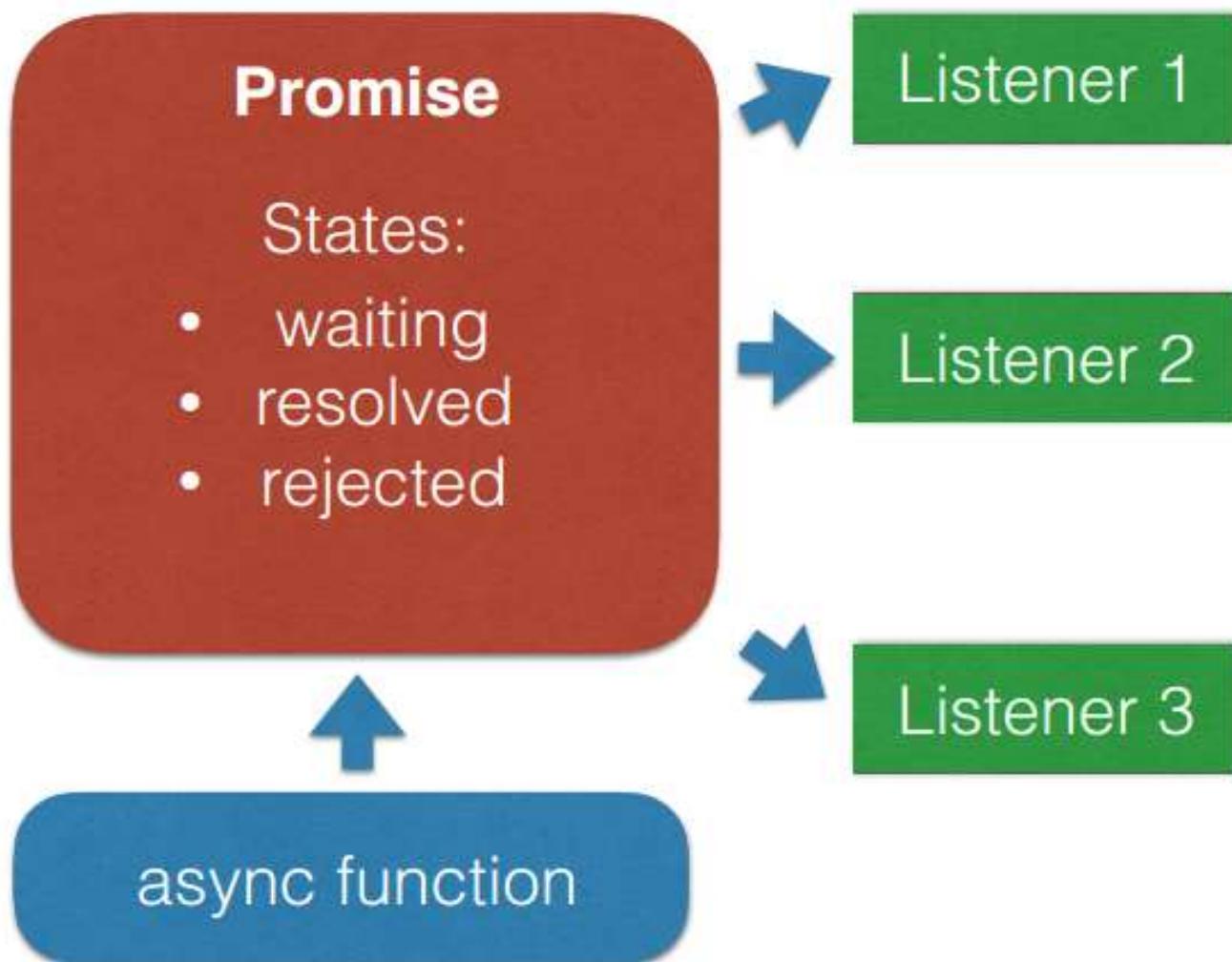


```
foo(() => {  
    bar(() => {  
        baz(() => {  
            qux(() => {  
                quux(() => {  
                    quuz(() => {  
                        corge(() => {  
                            grault(() => {  
                                run();  
                            }).bind(this);  
                        }).bind(this);  
                    }).bind(this);  
                }).bind(this);  
            }).bind(this);  
        }).bind(this);  
    }).bind(this);  
}).bind(this);  
});
```

callbacks – co w tym złego?

```
1  fs.readdir(source, function (err, files) {  
2    if (err) {  
3      console.log('Error finding files: ' + err)  
4    } else {  
5      files.forEach(function (filename, fileIndex) {  
6        console.log(filename)  
7        gm(source + filename).size(function (err, values) {  
8          if (err) {  
9            console.log('Error identifying file size: ' + err)  
10         } else {  
11           console.log(filename + ' : ' + values)  
12           aspect = (values.width / values.height)  
13           widths.forEach(function (width, widthIndex) {  
14             height = Math.round(width / aspect)  
15             console.log('resizing ' + filename + 'to ' + height + 'x' + height)  
16             this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {  
17               if (err) console.log('Error writing file: ' + err)  
18             })  
19             }.bind(this))  
20           }  
21         })  
22       })  
23     })  
24   })  
25 }
```

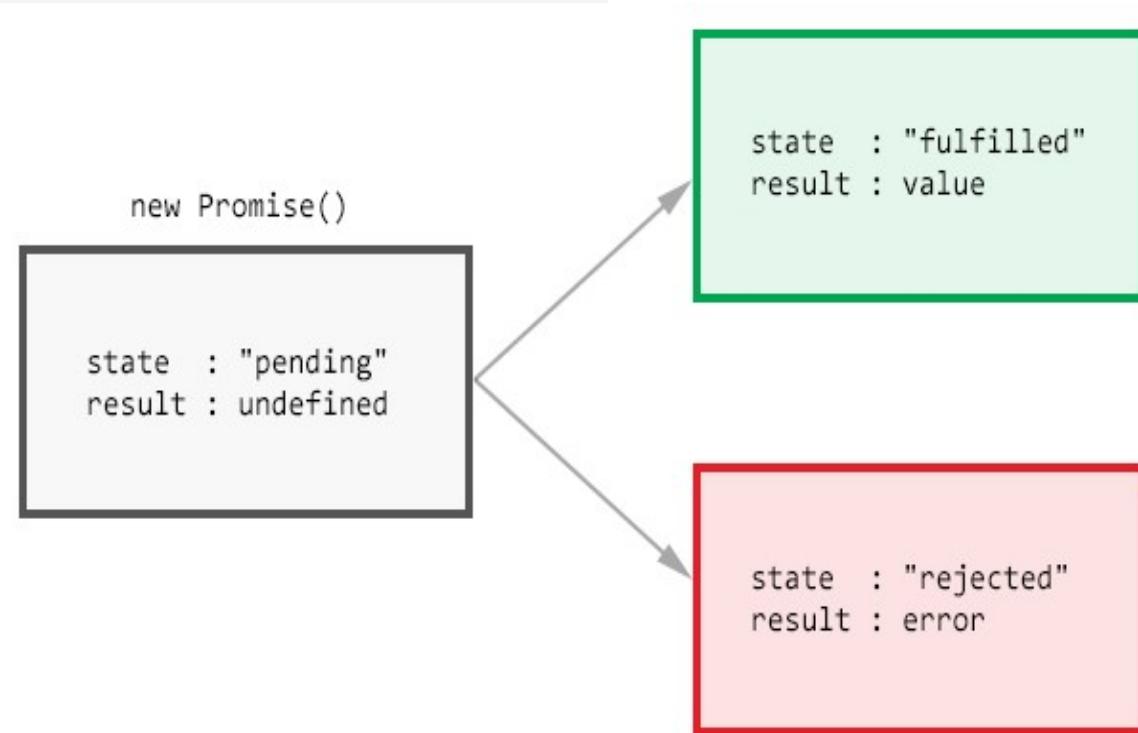
Promises



Obietnica = obiekt reprezentujący zakończenie w przyszłości asynchronicznego przetwarzania (pozytywne lub z błędem)

Obietnica

```
const promise = new Promise((resolve, reject) => {
    resolve("Wszystko ok");
    reject("Nie jest ok");
});
```



- Każda obietnica może zakończyć się na dwa sposoby – powodzeniem i niepowodzeniem.
- Gdy obietnica zakończy się powodzeniem (np. dane się wczytają), powinniśmy wywołać funkcję `resolve()`, do której przekażemy poprawny rezultat.
- W przypadku błędów powinniśmy wywołać funkcję `reject()`, do której trafiają błędne dane.

Obietnice (promises)

```
setTimeout(function() {  
    console.log('I promised to run after 1s')  
    setTimeout(function() {  
        console.log('I promised to run after 2s')  
    }, 1000)  
}, 1000)
```

callback

obietnica

```
const wait = () => new Promise((resolve, reject) => {  
    setTimeout(resolve, 1000)  
})  
  
wait().then(() => {  
    console.log('I promised to run after 1s')  
    return wait()  
})  
.then(() => console.log('I promised to run after 2s'))
```

Przykład użycia obietnicy

```
function getAsync(url) {
  return new Promise((resolve, reject) => {
    var httpReq = new XMLHttpRequest();
    httpReq.onreadystatechange = () => {
      if (httpReq.readyState === 4) {
        if (httpReq.status === 200) {
          resolve(JSON.parse(httpReq.responseText));
        } else {
          reject(new Error(httpReq.statusText));
        }
      }
    }
    httpReq.open("GET", url, true);
    httpReq.send();
  });
}

getAsync('https://jsonplaceholder.typicode.com/posts/1')
  .then((data) => {
    const post = 'Title: ' + data.title + '\n\nBody: ' + data.body; alert(post)
  })
  .catch((err) => { alert(err); })
};
```