

Wykład 4

Wprowadzenie do Java Script
języka wielu paradygmatów
dr inż. Grzegorz Rogus



1

DEFINICJA JAVASCRIPT

Dotychczasowa definicja

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

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);

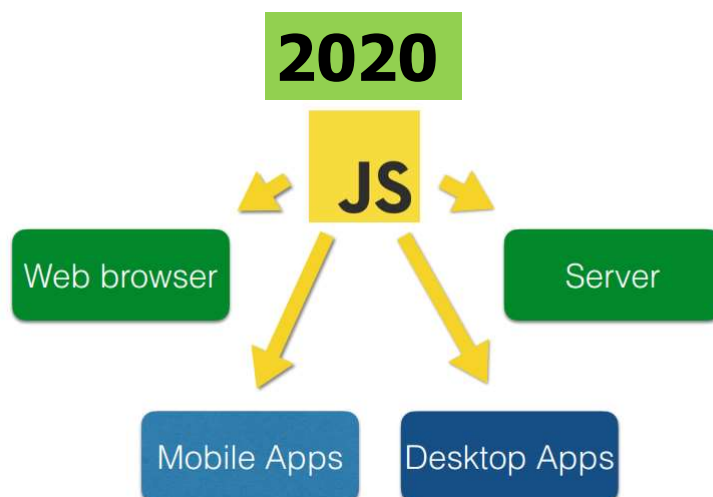
Interaktywny klej pomiędzy HTML a CSS

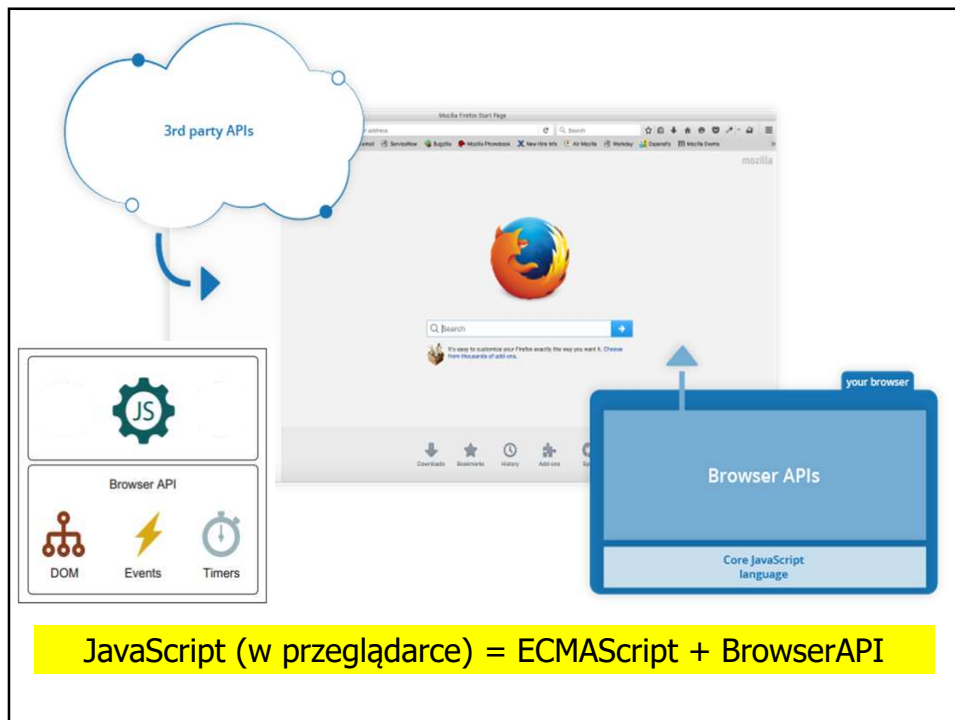


CECHY JAVASCRIPT

- **język skryptowy** - nie musi być kompilowany 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*).





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 (obiektywny model dokumentu).

Browser API:

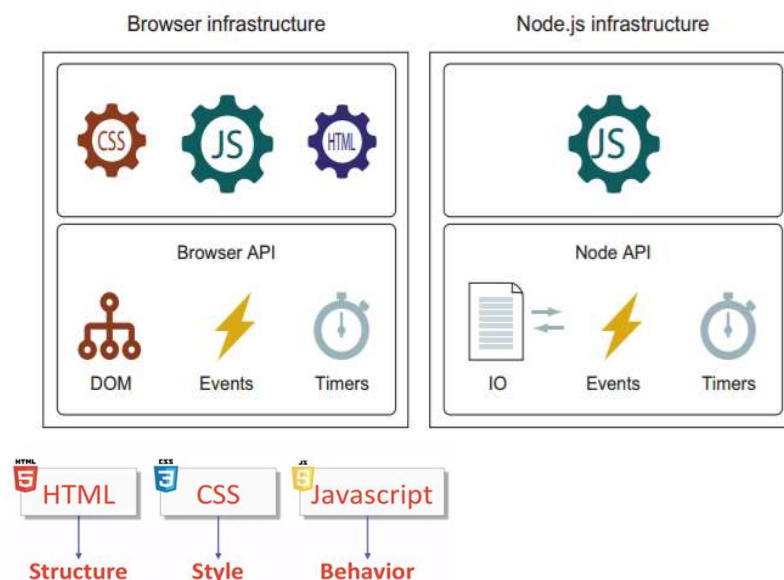
- API do manipulacji dokumentem ([DOM \(Document Object Model\) API](#))
- API do pobierania danych z serwera ([XMLHttpRequest](#) lub [Fetch API](#))
- API do rysowania i edycji grafiki ([CANVAS](#), [WebGL](#))
- Audio i Video APIs ([HTMLMediaElement](#), [Web Audio API](#), [WebRTC](#))
- Device API ([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



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



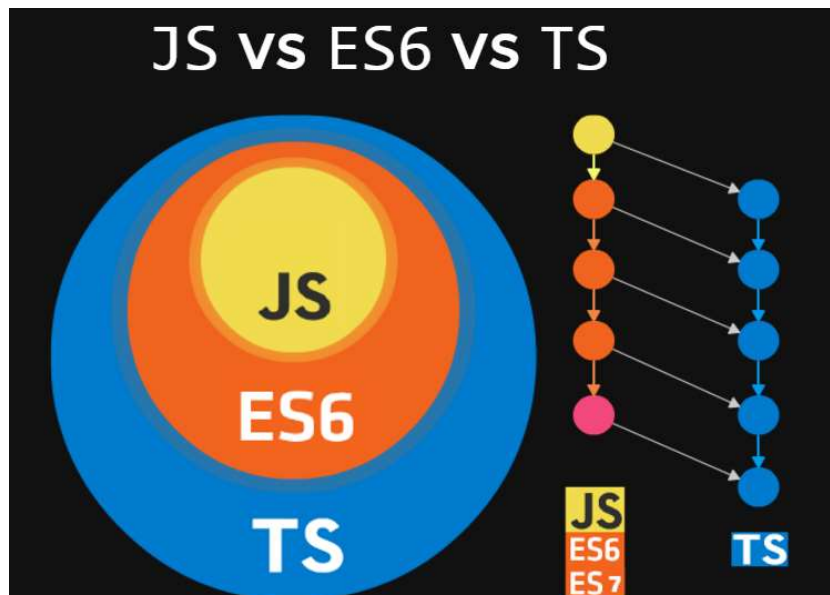
Obecnie najczęściej i najpowszechniej używaną wersją JavaScript była wersja 5.1 (ES5 / ECMA5) standaryzowana w roku 2011.

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. Paradygmat obiektowy – zarządzanie kodem

Moduły

Import

Export

Klasy

4. Nowe struktury danych

Zbiory (Sets)

Mapy (Maps)

Weak Maps oraz Weak Sets

Iteratory

Pętla for of Loop

5. Programowanie funkcyjne

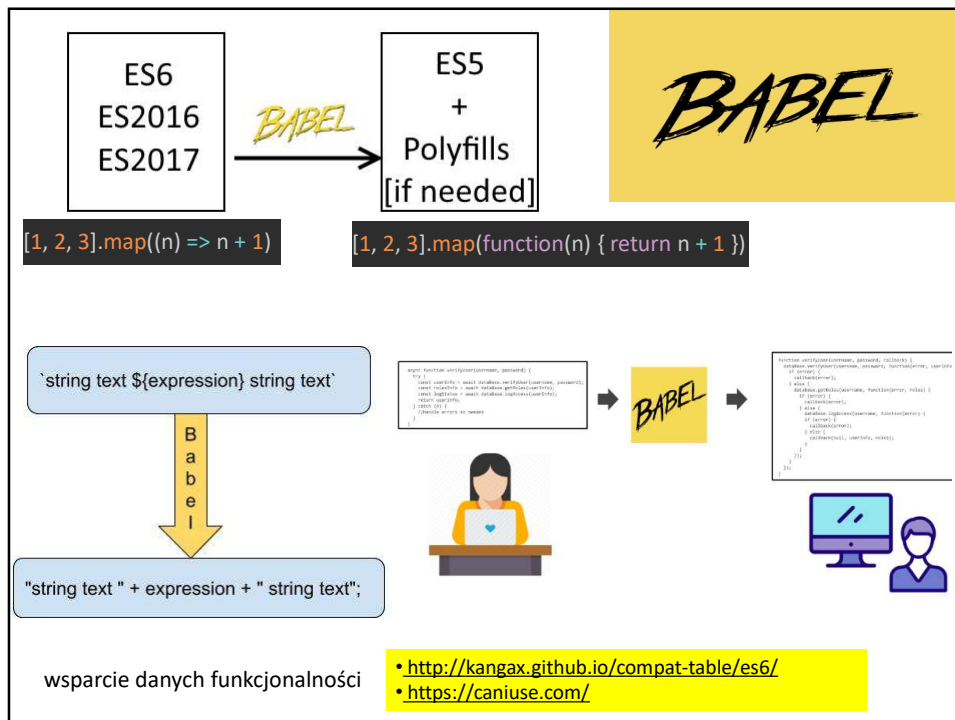
Wyrażenia Lambda (Arrow Functions)

Generatory

6. Inne elementy syntaktyczne

Obietnice (Promises)

Destrukturalizacja



JavaScript w przeglądarce - podstawowe informacje

<https://www.codeguage.com/courses/js/objects-creating-objects>

Jak załączyć JS do strony

```
<script>
...kod...
</script>

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

DOBRE PRAKTYKI JAVASCRIPT

Dobłą praktyką jest:

- zamieszczanie kodu JavaScript w zewnętrznych plikach .js;
- zamieszczanie kodu JavaScript na końcu dokumentu;
- ładowanie asynchroniczne plików .js poprzez dodawanie atrybutu `async` do znacznika `<script>` o ile nie zakłóci to działania strony.

```
<script async src="plik.js"></script>
```

```
<script src="plik.js" defer></script>
```

Zewnętrzne pliki są łatwiejsze do zarządzania, do tego poprzez umieszczenie ich w pamięci przeglądarki (cache) strony szybciej się ładują.

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 **niezmiennie**.

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

```
var 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'];  
var 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 (**windowanie – 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 fufu() {
  // "v" widoczne tu
  for ( var v = 0; v < 5; v++ ) {
    // "v" widoczne, ale żyje piętro wyżej
  }
  // "v" widoczne tutaj
}
```

```
function kuku() {
  // "l" nie ma tutaj
  for ( let l = 0; l < 5; l++ ) {
    // "l" widoczne tylko tu (i głębiej)
    // każda iteracja ma nową instancję "l"
  }
  // "l" tutaj już nie ma
}
```

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
}
```

var - Ciekawostka

```
var x = 16
var y = 17

function zmianaVars() {
  x = 24
  z = 32
  var y = 18
  var p = 2
  for (var i = 0; i < y; i++) {
    // cos tam robie
  }
  console.log(i)
}

zmianaVars()
console.log(x) // 24
console.log(y) // 17
console.log(z) // 32
console.log(p) // "ReferenceError: p nie jest zdefiniowane"
```

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 niedozwolona
```

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

WYŚWIETLANIE INFORMACJI

JavaScript nie posiada wbudowanych żadnych funkcji wyświetlających efekty działań.

Można to osiągnąć na cztery sposoby wykorzystując:

- okno z komunikatem `window.alert("cześć")`,
- dokument HTML `document.write("cześć")`,
- element HTML `innerHTML`
`document.getElementById("demo").innerHTML = "to jest demo";`
- konsolę przeglądarki `console.log("cześć")`. lub `console.table(items)`

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 (var i=0; i < zmienna.length; i++)
{
    document.write(zmienna[i] + "<br>");
}
```

while:

```
while (zmienna.length<5)
{
    document.write(zmienna[i] + "<br>");
}
```

do while:

```
do
{
    document.write(zmienna[i] + "<br>");
}
while (zmienna.length<5)
```

for in:

```
var auta = { marka: "Audi", model: "A3", pojemosc: 2500 };
for (x in auta) {
    txt = txt + auta[x];
}
```

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"
}

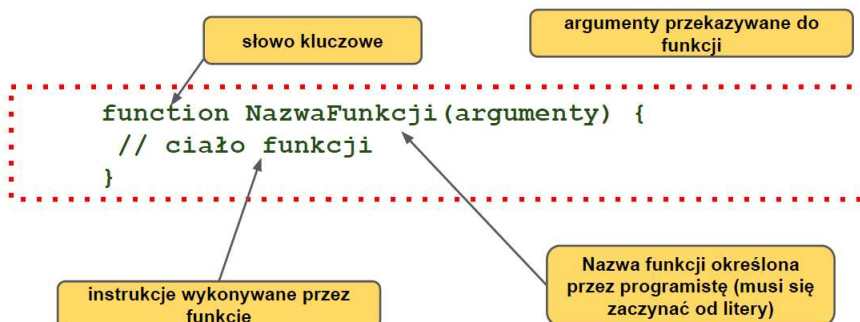
// for (let i of car) {} // error!

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

Funkcja w JS - deklaracja funkcji

Zacznijmy od zdefiniowania (nawet) jak tworzyć funkcje, które są podstawowymi narzędziami i jednostkami modularnymi wykorzystywanymi przez programistę 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 wyrażenie funkcyjnego

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

Wyrażenia lambda (Arrow function =>)

```
let func = (arg1, arg2, ...argN) => expression
```

Standard ES6 poza wśród wielu uproszczeń wprowadza również możliwość używania wyrażen lambda (funkcji strzałkowych).

```
function Dodaj(x,y){  
    return x + y;  
}  
  
var Dodaj = (x,y) => x + y;
```

Definicja tego typu funkcji składa się z dowolnej liczby parametrów (argumentów), znacznika “=>” oraz ciała funkcji umieszczonego za nim.

Tego typu funkcje są zawsze anonimowe, dlatego w przykładzie powyżej jej referencja zostaje zapisana do zmiennej Dodaj.

Gdy funkcja taka ma więcej niż jedną instrukcję jej ciało musi zostać zapisane pomiędzy parą nawiasów klamrowych {} .

```
const foo = param => doSomething(param);
```

Wyrażenia lambda (Arrow function =>)

Przykłady:

```
var f1 = () => 12;  
var f2 = x => x*2;  
var f3 = (x,y) => x + y;  
var f4 = (x,y) => {  
    var z = x*y;  
    return z/5;  
}
```

W JavaScript (ES6) wyrażenia lambda są tylko wyrażeniami funkcji, a nie jej deklaracją, które nie dysponują żadnymi referencjami które można dalej wykorzystać.

Funkcje tworzone w postaci wyrażenia lambda, mają atrakcyjną i zwięzłą formę. Ale nie zawsze w przypadku dużych wielolinijkowych funkcji jest wygodnie stosować.

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 działał!  
    const x = 2;  
    return x;  
}
```

Destrukuryzacja obiektów i tablic

Destrukuryzacja to wygodny sposób tworzenia nowych zmiennych poprzez wydobywanie pewnych wartości z danych przechowywanych w obiektach i tablicach.

```
const person = {  
  firstName: "Jan",  
  city: „Krakow”,  
  age: 35,  
  sex: "M"  
}
```

Bez destrukuryzacji:

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

Z destrukuryzacją:

```
const { firstName: first, age, city = "Paris" } = person; // Gotowe !  
  
console.log(age) // 35 -- Stworzono nową zmienną równą person.age  
console.log(first) // „Jan” -- Stworzono nową zmienną równą person.firstName  
console.log(firstName) // ReferenceError -- person.firstName istnieje, ale nowo stworzona zmienna  
nazywa się first  
console.log(city) // "Paris" -- Stworzono nową zmienną city i, ponieważ person.city jest równe  
undefined, zmienna jest równa domyślnej wartości "Paris".
```

Destrukuryzacja tablic

```
// mamy tablice z dwoma tekstami  
let arr = ["Grzegorz", "Rogus"]  
let [imie, nazwisko] = arr; //  
  destrukuryzacja  
alert(imie); // Grzegorz  
alert(nazwisko); // Rogus
```

Destrukuryzacja obiektów

```
let options = { title: „Test”, width: 100, height: 200 };  
// { sourceProperty: targetVariable }  
let {width: w, height: h, title} = options;  
// width -> w  
// height -> h  
// title -> title  
alert(title); // Test  
alert(w); // 100  
alert(h); // 200
```

Destrukuryzacja obiektów i tablic

Tablica

```
const myArray = ["a", "b", "c"];
```

Bez destrukuryzacji:

```
const x = myArray[0];  
const y = myArray[1];
```

Z destrukuryzacją:

```
const [x, y] = myArray; // Gotowe !  
console.log(x) // "a"  
console.log(y) // "b"
```

Destrukuryzacja jest często używana do rozbicia parametrów funkcji na części.

Bez destrukuryzacji:

```
function joinFirstLastName(person) {  
  const firstName = person.firstName;  
  const lastName = person.lastName;  
  return firstName + '-' + lastName;  
}  
joinFirstLastName(person); // „jan-Nowak„
```

Z destrukuryzacją:

```
function joinFirstLastName({ firstName, lastName }) {  
  // tworzymy zmienne firstName i lastName z części  
  argumentu person.  
  return firstName + '-' + lastName;  
}  
joinFirstLastName(person); // „Jan Nowak"
```

Funkcja Arrow destrukuryzacji

```
const joinFirstLastName = ({ firstName, lastName }) => firstName + '-' + lastName;  
joinFirstLastName(person); // "Jan Nowak"
```

Operatory rest i spread

- Ten sam zapis (...), ale „odwrotne” działanie
- Rest – łączy parametry funkcji w tablicę

```
function add(...numbers) {  
  return numbers.reduce((sum, elem) => sum + elem);  
}  
  
var result = add(3, 5, 8);  
console.log(result);           // 16
```

- Spread – rozбивa tablicę na listę argumentów funkcji

```
function add(n1, n2) {  
  return n1 + n2;  
}  
  
var tab = [3, 7];  
var result = add(...tab);  
console.log(result);           // 10
```

Operator rozproszenia (rozwijania) "..."

Operator rozwijania ... jest przeznaczony jest do "rozwijania" elementów obiektów iterowalnych (np. tablic) tam, gdzie można zmieścić kilka elementów

Mamy dwie tablice:

```
const arr1 = ["a", "b", "c"];  
const arr2 = [arr1, "d", "e", "f"]; // ["a", "b", "c"], "d", "e", "f"]
```

Pierwszy element tablicy arr2 jest tablicą, ponieważ arr1 został wprowadzony do arr2 bezpośrednio.

Co jednak, jeśli chcemy, by arr2 było tablicą liter? → użyć operatora rozwijania

Z operatorem rozwijania

```
const arr1 = ["a", "b", "c"];  
const arr2 = [...arr1, "d", "e", "f"]; // ["a", "b", "c", "d", "e", "f"]
```

Obsługa zdarzeń

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

Zdarzenia związane z myszą:

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

Zdarzenia związane z klawiaturą:

onKeyDown	Wciśnięcie przycisku
onKeyUp	Puszczenie przycisku
onKeyPress	Naciśnięcie i zwolnienie przycisku

Metody rejestrowania zdarzeń

Rejestrowanie zdarzenia inline

polega na określeniu zdarzenia wewnątrz znacznika – z użyciem atrybutu HTML

```
<a href="strona.html" onclick="alert(' Kliknąłeś! ')">kliknij</a>
```

Wady:

Mieszanie skryptów JS z kodem HTML (podobnie jak wpisane CSS)
rozwiązanie nie jest obiektowe – wewnątrz funkcji nie mamy dostępu do this czyli właściwości obiektu, na którym wywołano zdarzenie

Metody rejestrowania zdarzeń

Rejestrowanie zdarzenia w sekcji skryptu

```
var element = document.getElementById('Przycisk');
element.onclick = funkcja1;
element2.onmouseover = funkcja2;
```

Przykład 1 – z zastosowaniem funkcji nazwanej

```
<input type="button" id="Przycisk" value="kliknij" />
<script type="text/javascript">
    function wypisz() {
        alert('zostałem kliknięty!');
    }
    document.getElementById('Przycisk').onclick = wypisz
</script>
```

Przykład 2 – z zastosowaniem funkcji anonimowej

```
document.getElementById('Przycisk').onclick = function() {
    alert('zostałem kliknięty!');
}
```

45

Metody rejestrowania zdarzeń

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

```
var 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

```
document.addEventListener('DOMContentLoaded', function(event) {
    let btn = document.getElementById('zapisz');
    btn.addEventListener('click', function () {
        btn.textContent = 'Kliknięto!';
    });
});
```

Zabezpieczenie –
gdy cała strona
jest załadowana

46

Rozszerzona obsługa zdarzeń

Polega na pobraniu wartości pseudoparametru funkcji obsługi zdarzenia

```
document.getElementById('Przycisk').onclick = function(evt) {  
    alert('Typ zdarzenia: ' + evt.type);  
}
```

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

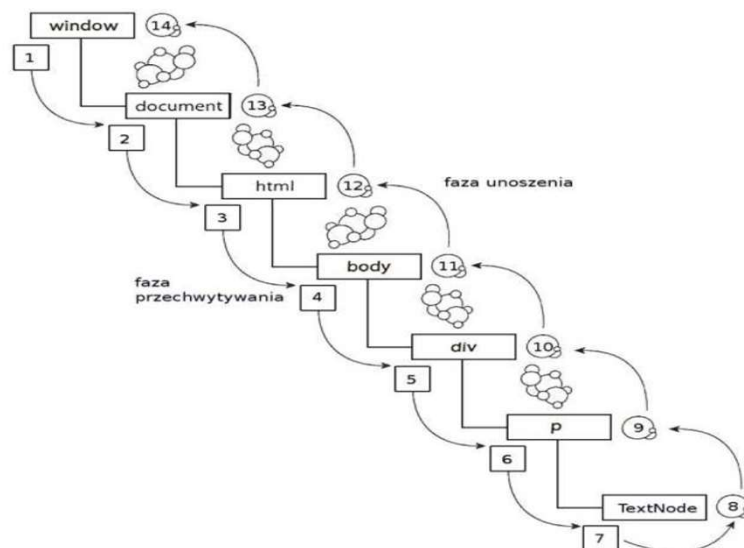
Wstrzymanie domyślnej akcji - **Prevent default**

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

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

47

Bąbelki - Kaskadowe wykonywanie zdarzeń

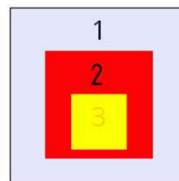


Bąbelki - Kaskadowe wykonywanie zdarzeń

Założmy 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">
document.getElementById('blok1').onclick = function() {
  alert('Kliknąłeś mnie!');
}
</script>
```



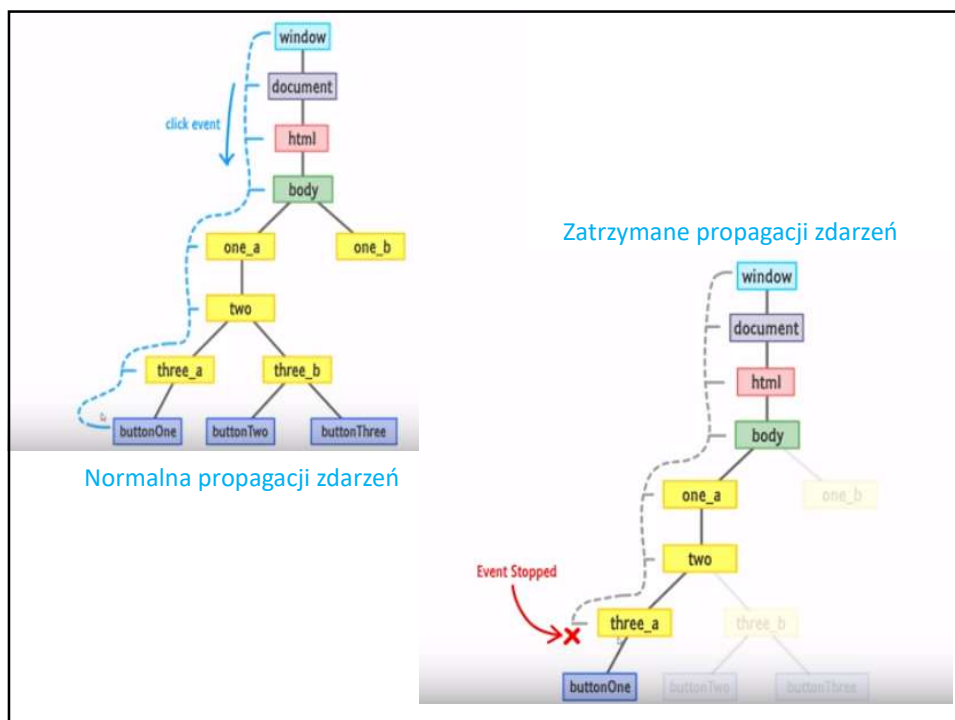
Kliknięcie w element wewnętrzny powoduje po wykonaniu zdarzenia przestanie go do elementu otaczającego (tu zawsze wywoła się alert)

Wyłączenie tego zjawiska realizujemy poprzez funkcję stopPropagation

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

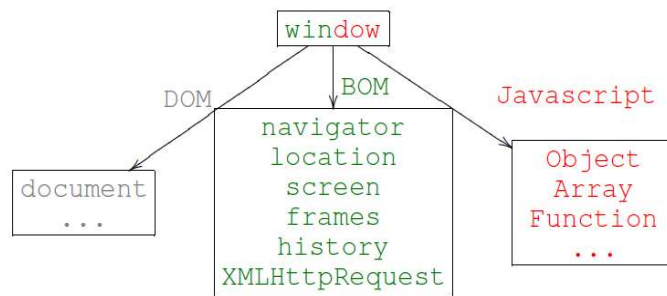
```
document.getElementById('blok31').onclick =
function() { alert('Kliknąłeś mnie!') }
document.getElementById('blok32').onclick =
function(e) { stopBubble(e); }
document.getElementById('blok33').onclick =
function(e) { stopBubble(e); }
```

49



Obiekty przeglądarki

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



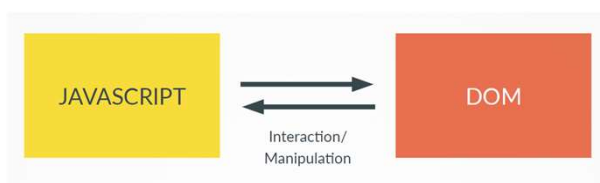
51

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

JavaScript a DOM

dopisywanie do dokumentów HTML:

```
document.write("<h1>This is a heading</h1>");
```

zmiana zawartości dokumentów HTML:

```
x = document.getElementById("IdElementu").innerHTML = "Nowa Zawartość";
```

obsługa zdarzeń:

```
<button type="button" onclick="alert('Kliknięcie')">Kliknij!</button>
```

zmiana stylów dokumentu HTML:

```
x = document.getElementById("IdElementu").style.color="#ffffff";
```

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 i usuwanie węzłów - metody DOM

appendChild() - dodaje nowy podwęzeł 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 podwęzłem.

```
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", "zdjecie.jpg");
```

ZMIANA ELEMENTÓW HTML

`element.style.property` oraz `element.className`

Dzięki możliwości modyfikowania stylu danego elementu możliwe jest uzyskanie ciekawych efektów:

- Ukrywanie elementów: `element.style.display = 'none'`
- Zmiana pozycji elementów: `element.style.left = x + 'px'`;

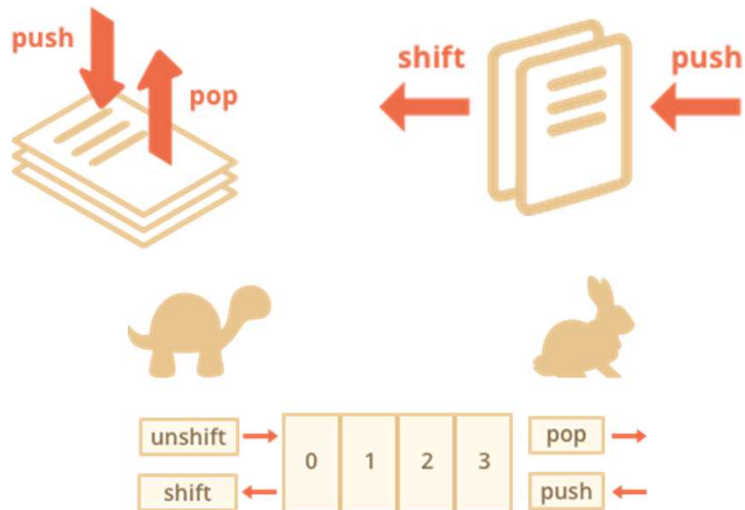
Przy używaniu nazw stylów wykorzystuje się również notację wielbłądzą (camel case).

`background-color` = `backgroundColor`

`float` = `cssFloat`

Zaleca się jednak wykorzystywanie własności `className`, w celu oddzielenia kodu JavaScript od wyglądu strony.

Przydatne metody dla obiektu Array



JavaScript Strings

charAt()	slice()
charCodeAt()	split() x
concat()	substr()
fromCharCode()	substring()
indexOf()	toLowerCase()
lastIndexOf()	toUpperCase()
length	toLocaleLowerCase()
localeCompare()	toLocaleUpperCase()
match() x	toSource()
replace() x	valueOf()
search() x	

JavaScript Arrays

concat()	slice()
join()	sort()
length	splice()
pop()	toSource()
push()	toString()
reverse()	unshift()
shift()	valueOf()

Wiecej <http://www.w3schools.com/js>

Przydatne metody dla obiektu Array

Standardowe obiekty JavaScriptowe dostarczają w pakiecie przydatne metody pozwalające na proste operacje na danym obiekcie. Dla obiektu Array najważniejsze to:

// metoda .concat()

```
var tab1 = new Array("a", "b", "c");  
var tab2 = new Array("d", "e", "f");  
  
var tab3 = tab1.concat(tab2);
```

Metoda ta kopiuje "tab1" i dołącza do niej elementy "tab2". Wynikiem jest nowa tablica "tab3" zawierająca: [a,b,c,d,e,f].

// metoda .join()

```
var tab1 = new Array("a", "b", "c");  
var string = tab1.join('|');
```

Tworzy łańcuch tekstowy z elementów tablicy z separatorem podanym jak argument metody.

Przydatne metody dla obiektu Array

// metoda .pop()

```
var tab1 = new Array("a", "b", "c");  
var ostatni = tab1.pop();
```

Usuwa ostatni element tablicy i jednocześnie zwraca jego wartość.

// metoda .push()

```
var tab1 = new Array("a", "b", "c");  
var tab2 = new Array("d", "e", "f");  
var tab3 = tab1.push(tab2);
```

Dodaje na końcu tablicy elementy zapisane jako argument metody. Zwraca długość tablicy po modyfikacji.

// metoda .reverse()

```
var tab1 = new Array("a", "b", "c");  
var tab3 = tab1.reverse();
```

Odwraca kolejność elementów tablicy.

Przydatne metody dla obiektu Array

```
// metoda .shift()
```

```
var tab1 = new Array("a", "b", "c");  
var pierwszy = tab1.shift();
```

Usuwa pierwszy element tablicy i jednocześnie zwraca jego wartość.

```
// metoda .unshift()
```

```
var tab1 = new Array("a", "b", "c");  
var tab2 = new Array("d", "e", "f");  
var tab3 = tab1.push(tab2);
```

Dodaje na początku tablicy elementy zapisane jako argument metody. Zwraca długość tablicy po modyfikacji.

```
// własność .length
```

```
var tab1 = new Array("a", "b", "c");  
var pierwszy = tab1.length;
```

To jest własność która zwraca długość tablicy.

OBIEKTY JAVASCRIPT

Obiekt najłatwiej sobie można wyobrazić jako "pojemnik", wewnątrz którego umieszczone są **zmienne (właściwości)** i **funkcje (metody)**.

Obiektem może być kot. Kot będzie miał między innymi takie właściwości, jak:

cat.age, **cat.breed**, **cat.color**

oraz takie metody jak:

cat.crazyMode() i **cat.purrMode()**

Wszystkie koty mają te same właściwości, ale różne wartości tych właściwości.

Podobnie jest z metodami – wszystkie koty je mają, jednak wywołują je w różnych momentach.



Tworzenie obiektów w JavaScript

W JavaScript obiekty można tworzyć na 3 różne sposoby:

- inicjalizator obiektu
- fabrykę
- konstruktor

inicjalizator obiektu

```
let car1 = {  
  name: 'Audi',  
  price: 35000,  
  isDiesel: true,  
  turnOn: function() {  
    console.log('silnik włączony');  
  }  
};  
console.log(car1); //Object { name:  
"Audi", price: 35000, isDiesel: true,  
turnOn: turnOn() }  
console.log(car1.name); //Audi  
console.log(car1.price); //35000  
console.log(car1.isDiesel); //true  
car1.turnOn(); //silnik włączony
```

Tworzenie obiektów w JavaScript - fabryka

```
function createCar(name, price, isDiesel) {  
  return {  
    name,  
    price,  
    isDiesel,  
    turnOn() {  
      console.log('silnik włączony');  
    }  
  };  
}
```

Dzięki powyższej fabryce możemy tworzyć obiekty w następujący sposób:

```
let car2 = createCar('Audi', 35000, true);
```

Tworzenie obiektów w JavaScript - konstruktor

```
function Car(name, price, isDiesel) {  
  this.name = name;  
  this.price = price;  
  this.isDiesel = isDiesel;  
  this.turnOn = function () {  
    console.log('silnik włączony');  
  }  
}
```

Tworzenie obiektów poprzez funkcję konstruuującą:

```
let car3 = new Car('Audi', 35000, true);  
car3.name // lub car3[„name”]
```

PRZYKŁAD OBIEKTU

```
var cat = {  
  age: 7,  
  breed: "NFO",  
  color: "black",  
  crazyMode: function() { window.alert("Au! It's crazy cat! "); },  
  purrMode: function() { window.alert("Oh, what a kitty!"); }  
};  
cat.crazyMode();
```

← Dostęp do pól obiektu

Metody obiektów

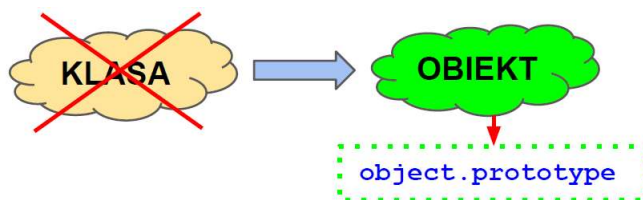
- Metody można dodawać dynamicznie do obiektów
 - Obiekty utworzone tym samym konstruktorem mogą później różnić się funkcjonalnością
- Jak zapewnić na starcie ten sam zestaw metod obiektom danego rodzaju?

- Można dodać metody w ciele konstruktora
 - Obiekty będą miały własny zestaw tych samych metod

```
function Rabbit(type) {  
  this.type = type;  
  this.speak = function(line) {  
    console.log("The " + this.type + " rabbit says '" +  
      line + "'");  
  };  
}
```

- Lepszym rozwiązaniem jest wykorzystanie prototypu
 - Zestaw metod w prototypie współdzielony przez obiekty na nim oparte

Dziedziczenie prototypowe w JS



W języku JavaScript (a tym samym we wszystkich środowiskach programistycznych opartych o ten język) dziedziczenie odbywa się w sposób prototypowy. Każdy obiekt w JavaScript można traktować jako prototyp który w dowolnym momencie można rozszerzać, a jego własności przekazywać (cedować) na inne dziedziczące obiekty.

Obiekty dziedziczą po obiektach!

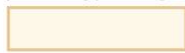
Dziedziczenie prototypowe w JS

Obiekty dziedziczą po obiektach!

`object.prototype`

Właściwość `prototype` posiadają obiekty które są tworzone przez funkcję konstruktora. Jednocześnie sam prototyp jest obiektem. Często dziedziczenie w JS określa się mianem dziedziczenia opartego o instancje.

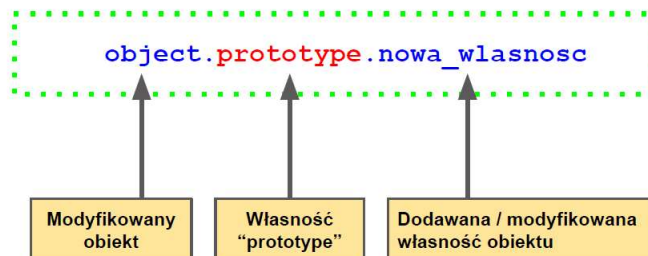
prototype object



Każdy obiekt w JavaScript ma ukryta właściwość `[[Prototype]]`, którą jest referencja na inny obiekt lub null.

Dziedziczenie prototypowe w JS

Do zmiany lub rozszerzenia utworzonego obiektu możemy użyć właściwości `prototype` (który też jest obiektem):



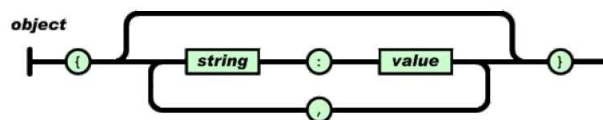
Prototypy

- Prototyp (ang. prototype) to w JavaScript to obiekt który dla danego obiektu stanowi „zapasowe” źródło właściwości
- Prototypem większości obiektów w JS jest `Object.prototype`
 - Dostarcza kilka metod np. `toString()`, `valueOf()`
 - Jego właściwości są nie-enumerowalne
- Prototypy tworzą drzewiastą hierarchię
 - Odpowiednik hierarchii dziedziczenia klas
- Odczyt prototypu: `Object.getPrototypeOf()`
- Tworzenie obiektu z prototypu: `Object.create(prototyp);`
- Zmiana prototypu (ES6): `Object.setPrototypeOf()`
 - Operacja czasochłonna, niezalecana
- Rozróżnienie własnych i odziedziczonych właściwości:
 - `hasOwnProperty()`, `Object.getOwnPropertyNames()`,

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/>



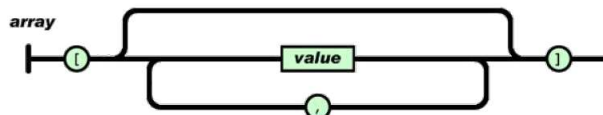
```
{
  "firstname": "Jan",
  "lastname": "Kowalski",
  "age": 20
}
```

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 rozpoczynanie 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
```

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>

XMLHttpRequest — uproszczony przykład poglądowy

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

Globalny obiekt dostępny w całej sekcji sekcji script.

```
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>
. . .
```

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;
```

Pobranie inform

Pobierz

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

Pobranie in

Pobierz

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

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

Fetch API

```
fetch(url, [options]);
```

```
fetch("https://test.com/zasob")
  .then(response => {
    console.log(response);
  })
```

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 łączyliśmy
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 <u>ArrayBuffer</u>

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
})
```

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);
```

PROMISES - OBIETNICE

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);
```

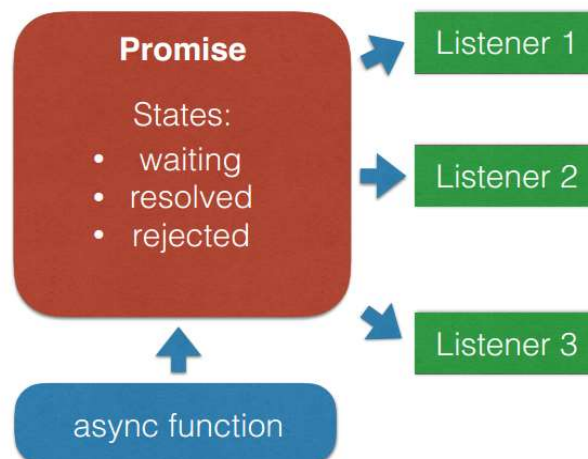


```
foo(() => {  
  bar(() => {  
    baz(() => {  
      qux(() => {  
        quux(() => {  
          quuz(() => {  
            corge(() => {  
              grault(() => {  
                run();  
              }).bind(this);  
            }).bind(this);  
          }).bind(this);  
        }).bind(this);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

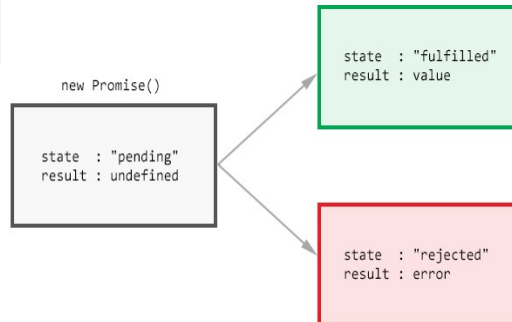
Promises



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 prześlemy poprawny rezultat.
- W przypadku błędów powinniśmy wywołać funkcję **reject()**, do której trafią 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); })  
);
```

Edytory JS

Visual Studio Code

Online editors/IDEs

JsBin.com

<http://jsbin.com/>

CodePen.io

<http://codepen.io/>

Plunker

<https://plnkr.co/>

JsFiddle

<https://jsfiddle.net/>