Język JavaScript

programowanie „DOM-owe”

DOM – Document Object Model

- **Prehistoria**
  - „DOM Level 0”
    - reprezentacja jedynie wybranych elementów dokumentu – odsyłacze, obrazki, elementy formularzy
    - dostępność
      - obsługa za pośrednictwem JavaScript
      - Netscape 3.0+ IE 3.0+
      - powszechnie dostępna – na zasadzie „wstecznej zgodności”
    - użyteczna w prostych zastosowaniach:

```
document.links[],
document.forms[],
document.images[]
```


DOM – Document Object Model

- **Historia**
  - **W3C** DOM (Level 1 → Level 2 → Level 3)
    - pełna reprezentacja struktury dokumentów (**X**|**HT**)**ML**
    - implementacje:
      - Java, Perl, PHP, Ruby, Python, C++, JavaScript, ...
    - najważniejsze możliwości
      - nawigowanie po strukturze dokumentu
      - dostęp do atrybutów, stylów i treści elementów
      - obsługa „zdarzeń”
    - obsługa w przeglądarkach – „pewne problemy”
  - **WHATWG** DOM – Living Standard

`http://dom.spec.whatwg.org/`

DOM a JavaScript

- Elementy DOM
  - reprezentowane jako obiekty typu „Host”
  - dostępne głównie w przeglądarkach
  - nieliczne implementacje „zewnętrzne”, np.

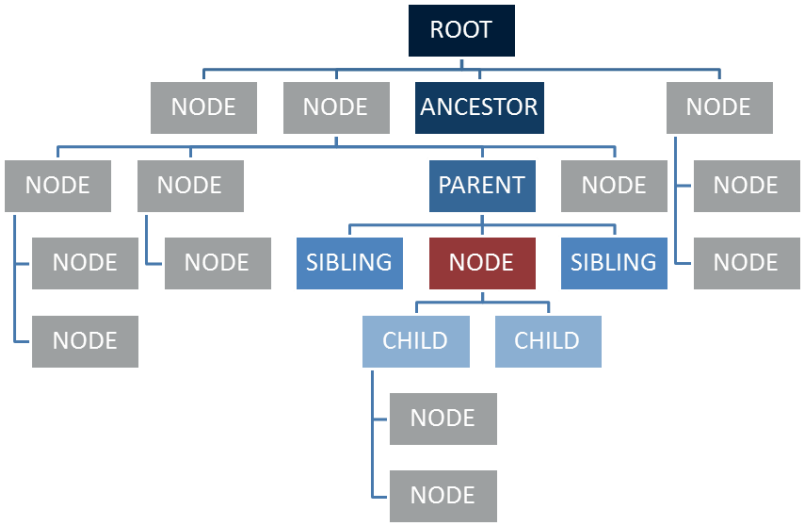
`https://github.com/tmpvar/jsdom`

przeznaczona głównie do współpracy z `node.js`

Poruszanie się po drzewie DOM

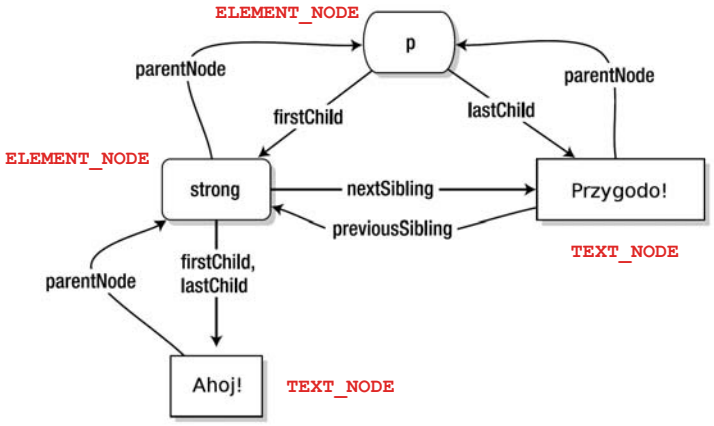
- Reprezentacja – graf skierowany
  - terminologia „genealogiczna”
    - rodzice, dzieci, rodzeństwo
  - inaczej niż w genealogii drzewo dokumentu zawsze rozpoczyna się od pojedynczego węzła – korzenia
  - połączenia pomiędzy węzłami prowadzące do:
    - potomków/dzieci
    - rodzica (dla węzłów różnych od korzenia)
    - rodzeństwa
  - każdy węzeł posiada typ

Terminologia genealogiczna



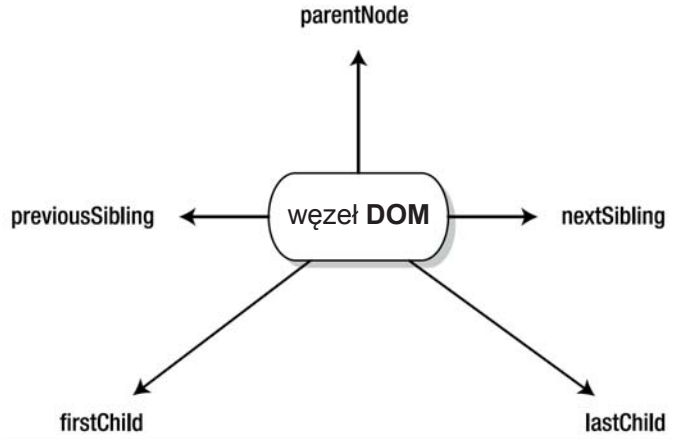
Poruszanie się po drzewie DOM

```
<p><strong>Ahoj!</strong> Przygodo!</p>
```



DOM API – podstawowe atrybuty węzłów

- „Wskaźniki sąsiedztwa” (mogą mieć wartość null)



Nawigacja DOM-owa

- docieramy do korzenia dokumentu
  - ▹ `document.documentElement`
- prosty przykład

```
<html>
  <head>
    <title>Prosty dokument</title>
  </head>
  <body>
    <h1>Wstęp</h1>
    <p class="t">DOM jest wygodny.</p>
    <script type="text/javascript" src="skrypt1.js"></script>
  </body>
</html>
```

`skrypt1.js`:

```
alert('|'+document.documentElement.innerHTML+'|');
```


Nawigacja DOM-owa - przykład

- Rezultat zadziałania skryptu `skrypt1.js` (Opera)

dla innych przeglądarek efekt jest analogiczny

Nawigacja DOM-owa

- próbujemy dotrzeć do nagłówka **h1**

```
<html>
  <head>
    <title>Prosty dokument</title>
  </head>
  <body>
    <h1>Wstęp</h1>
    <p class="t">DOM jest wygodny.</p>
    <script type="text/javascript" src="skrypt2.js"></script>
  </body>
</html>
```

`skrypt2.js`:

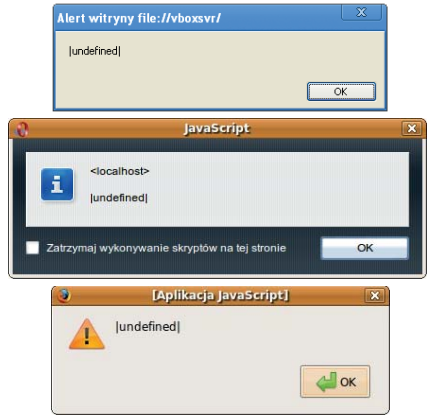
```
(function () {
  var elem = document.documentElement.lastChild.firstChild;
  alert('|' + elem.innerHTML + '|');
}());
```


Nawigacja DOM-owa – testujemy

Google Chrome

Opera

Firefox



Internet Explorer

Nawigacja DOM-owa – wyniki testu

- **Podsumowanie**
  - Internet Explorer jako jedyny sobie „poradził” (sic!)
  - pozostałe przeglądarki zachowały się „dziwnie”
- **Wyjaśnienie**
  - „pozostałe” przeglądarki stosują reprezentację DOM zgodną z zasadami XML-a zachowując „białe znaki” jako elementy tekstowe (tam, gdzie XML na to pozwala)
  - Internet Explorer radośnie je pomija :)
- ***Houston, we've got a problem…***

„Białe znaki” w DOM-u

- Drzewo DOM z reprezentacją „białych znaków”

Problem białych znaków – podejście 1

- Możemy napisać funkcję „czyszczącą” białe znaki:

```
/*jshint browser: true */
var cleanWhitespace;

cleanWhitespace = function (elem) {
    elem = elem || document;
    var cur = elem.firstChild;
    while (cur !== null) {
        if (cur.nodeType === 3 && !/\S/.test(cur.nodeValue)) {
            elem.removeChild(cur);
        } else if (cur.nodeType === 1) {
            cleanWhitespace(cur);
        }
        cur = cur.nextSibling;
    }
};
```

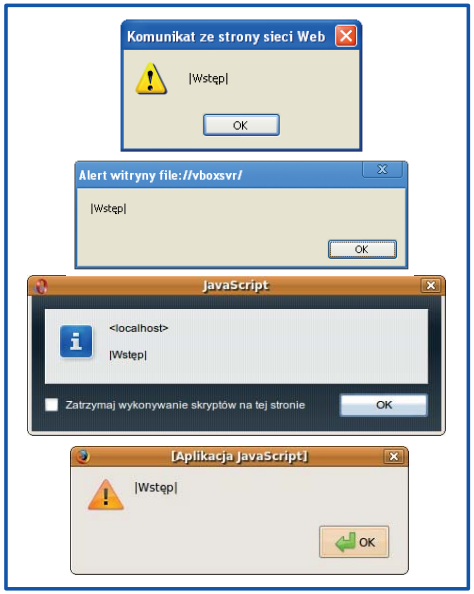

Nawigacja DOM-owa – testujemy ponownie

Internet Explorer

Google Chrome

Opera

Firefox



Problem białych znaków

- Dyskusja rozwiązania z **`cleanWhitespace`**
  - plusy
    - działa zarówno dla Internet Explorera, jak i dla „pozostałych” przeglądarek
  - minusy
    - rozwiązanie „siłowe”, wymagające przejrzenia całego drzewa dokumentu
    - w przypadku jakichkolwiek zmian w drzewie musimy ponownie wywołać funkcję **`cleanWhitespace`**, na tej części drzewa, która uległa zmianie

Typy węzłów w drzewie DOM

- **`cleanWhitespace`** wykorzystuje typy węzłów
- Najczęściej wykorzystywane typy
  - element (**`nodeType === 1`**)
  - tekst (**`nodeType === 3`**)
  - dokument (**`nodeType === 9`**)
- Przeglądarki inne niż IE (oraz IE9+) oferują stałe
  - **`ELEMENT_NODE`**
  - **`TEXT_NODE`**
  - **`DOCUMENT_NODE`**
  - plus 9 innych …

Problem białych znaków – podejście 2

- Idea: zestaw funkcji do sensownego podróżowania po drzewie DOM
  - skorzystamy z atrybutów DOM API:
    - **`firstChild`**, **`lastChild`**, **`parentNode`**, **`nextSibling`**, **`previousSibling`**
  - zdefiniujemy funkcje:
    - **`first`**, **`last`**, **`parent`**, **`next`**, oraz **`prev`**

    „pomijające zbędne białe znaki”, a ściślej – służące do poruszania się w drzewie DOM pomiędzy węzłami typu element (z pominięciem innych typów węzłów, w tym węzłów tekstowych)

Problem białych znaków – podejście 2

```
/*jshint browser: true */
var next, prev, first, last, paren;

next = function (elem) {
    do {
        elem = elem.nextSibling;
    } while (elem && elem.nodeType !== 1);
    return elem;
};

prev = function (elem) {
    do {
        elem = elem.previousSibling;
    } while (elem && elem.nodeType !== 1);
    return elem;
};
```

Problem białych znaków – podejście 2

```
first = function (elem) {
    elem = elem.firstChild;
    return elem && elem.nodeType !== 1 ? next(elem) : elem;
};

last = function (elem) {
    elem = elem.lastChild;
    return elem && elem.nodeType !== 1 ? prev(elem) : elem;
};

paren = function (elem, num) {
    var i;
    num = num || 1;
    for (i = 0; i < num; i += 1) {
        if (elem !== null) {
            elem = elem.parentNode;
        }
    }
    return elem;
};
```


Nawigacja DOM-owa

- próbujemy dotrzeć do nagłówka **h1**

```
<html>
  <head>
    <title>Prosty dokument</title>
  </head>
  <body>
    <h1>Wstęp</h1>
    <p class="t">DOM jest wygodny.</p>
    <script type="text/javascript" src="mylib.js"></script>
    <script type="text/javascript" src="skrypt3.js"></script>
  </body>
</html>
```

skrypt3.js:

```
alert('|'+ first(last(document.documentElement)).innerHTML +'|');
```

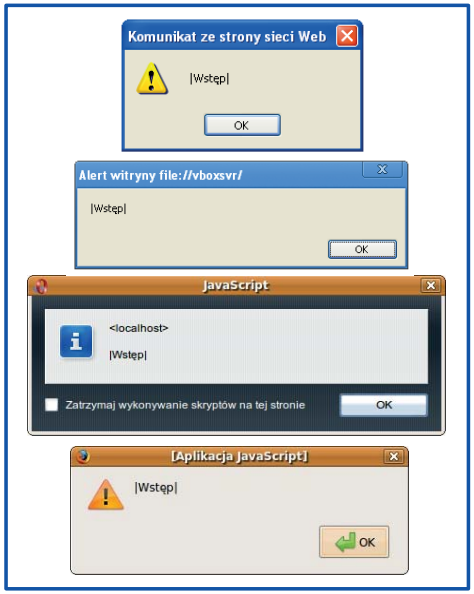

Nawigacja DOM-owa – podejście 2

Internet Explorer

Google Chrome

Opera

Firefox



Ulepszamy Podejście 2

- Zamiast

```
first(last(document.documentElement)).[...]
```

znacznie bardziej „obiektowo” byłoby pisać:

```
document.documentElement.last().first().[...]
```

wystarczy do prototypu obiektu **HTMLElement** dodać odpowiednie metody, np.

```
1. HTMLElement.prototype.next = function() {
2.   var elem = this;
3.   do {
4.     elem = elem.nextSibling;
5.   } while ( elem && elem.nodeType !== 1 );
6.   return elem;
7. };
```

„Drobny problem” ...

- Możliwość modyfikowania prototypu **`HTMLElement`** udostępniają jedynie „pozostałe przeglądarki”
  - Firefox, Google Chrome, Safari, Opera ...
- Możliwe (częściowe) rozwiązanie
  - użyć narzędzia, które „symuluje” **`HTMLElement`** w przeglądarce Internet Explorer
  - jedna z dostępnych implementacji:
    - Element Prototype Extension
    - http://www.jslab.dk/epe.introduction.php
- Uwaga! Problem z **`HTMLElement`** w IE to dopiero „wierzchołek góry lodowej”...

Podstawowe metody DOM API

- Najczęściej wykorzystywane metody DOM API to
  - **`getElementsByTagName( el )`**
    - znajduje w dokumencie wszystkie elementy „**`el`**”
    - przykład: **`getElementsByTagName('p')`** – wyszukuje w dokumencie wszystkie akapity
    - wynik wywołania jest „prawie tablicą” – nie można na nim wykonywać pewnych operacji tablicowych, jak **`push()`**, **`pop()`**, **`shift()`**, ...
  - **`getElementById( id )`**
    - znajduje w dokumencie element o identyfikatorze „**`id`**”

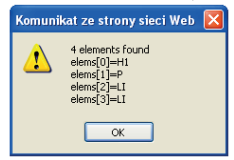
Przykłady

```
/*jshint browser: true */
var tags, hasClass;

// Szukamy w dokumencie elementów rodzaju type
tags = function (type, ctxt) {
    return (ctxt || document).getElementsByTagName(type);
};

// Wyszukujemy elementy należące do podanej klasy CSS
hasClass = function (name, type) {
    var found = [],
        re = new RegExp("(^|\\s)" + name + "(\\s|$)"),
        e = document.getElementsByTagName(type || "*"),
        j;
    for (j = 0; j < e.length; j += 1) {
        if (re.test(e[j].className)) {
            found.push(e[j]);
        }
    }
    return found;
};
```

Dynamicznie tworzone wyrażenie regularne

hasClass – przykład wykorzystania

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Prosty dokument</title>
    <script src="hasClass.js" type="text/javascript"></script>
  </head>
  <body>
    <h1 class="abc test">Wstęp</h1>
    <p class="abc test cde">Dom jest wygodny bo jest:</p>
    <ul>
      <li class="test">rozpowszechniony</li>
      <li>dobrze znany</li>
      <li class="test">niezbyt rozbudowany</li>
    </ul>
    <script src="skrypt.js" type="text/javascript"></script>
  </body>
</html>
```



skrypt.js

```
/*jshint browser: true, devel: true */
/*globals hasClass: false */
(function () {
    var elems = hasClass('test'),
    res = elems.reduce(function (prev, curr, idx) {
        return prev + '\nelems[' + idx + ']=' + curr.nodeName;
    }, '');
    alert(elems.length + 'elems found' + res);
}());
```

Jak zrobić coś „po utworzeniu DOM” ?

- **Podstawowe możliwości**
  - czekamy, aż załaduje się cała strona – korzystamy z mechanizmu „**`onload`**” obiektu „**`window`**”
    - niepotrzebnie czekamy np. na obrazki
  - używamy atrybutów zdarzeń języka (X)HTML i przypisujemy im jako wartości kod JavaScript
    - potencjalnie dostępny jest jedynie fragment drzewa DOM
    - mieszanie kodu (X)HTML i JavaScript nie jest wskazane
  - polecenia ładowania skryptów JavaScript umieszczamy „w odpowiednich miejscach” kodu strony
    - ograniczenia/wady podobne jak wyżej

Jak zrobić coś „po utworzeniu DOM” ?

- **Rozwiązanie „bardziej inteligentne”**
  - przygotowujemy listę czynności do wykonania
  - czekamy aż dostępne będą ważne elementy struktury drzewa DOM:
    - obiekt **`document`**
    - standardowe metody
      - **`document.getElementsByTagName`**
      - **`document.getElementById`**
    - element **`document.body`**
  - do zorganizowania „czekania” używamy standardowej funkcji **`setInterval(fun,time_int)`** pozwalającej na opóźnione wywołanie funkcji

Jak zrobić coś „po utworzeniu DOM” ?

- funkcja **`domReady(f)`**

```
/*jshint browser: true */
var addEvent, isDomReady, domReady;

domReady = function domReady(f) {
    if (domReady.done) {
        return f();
    }
    if (domReady.timer !== undefined) {
        domReady.callbacks.push(f);
    } else {
        addEvent(window, "load", isDomReady);
        domReady.callbacks = [f];
        domReady.timer = setInterval(isDomReady, 13);
    }
};
```


Jak zrobić coś „po utworzeniu DOM” c.d.

- funkcja **`isDOMReady()`**

```
isDomReady = function isDomReady() {
    if (domReady.done) {
        return false;
    }
    if (document && document.getElementsByTagName &&
                    document.getElementById && document.body) {
        clearInterval(domReady.timer);
        domReady.timer = undefined;
        domReady.callbacks.forEach(function (cb) {
            cb();
        });
        domReady.callbacks = null;
        domReady.done = true;
    }
};
```

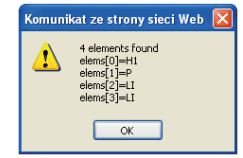
Jak zrobić coś „po utworzeniu DOM” c.d.

- funkcja pomocnicza **`addEvent(obj,type,fn)`**

```
addEvent = function addEvent(obj, type, fn) {
    if (obj.addEventListener) {
        obj.addEventListener(type, fn, false);
    } else if (obj.attachEvent) {
        obj["e" + type + fn] = fn;
        obj[type + fn] = function () {
            obj["e" + type + fn](window.event);
        };
        obj.attachEvent("on" + type, obj[type + fn]);
    }
};
```


domReady – przykład zastosowania

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Prosty dokument</title>
    <script src="hasClass.js" type="text/javascript"></script>
    <script src="domReady.js" type="text/javascript"></script>
    <script src="skrypt2.js" type="text/javascript"></script>
  </head>
  <body>
    <h1 class="abc test">Wstęp</h1>
    <p class="abc test cde">Dom jest wygodny bo jest:</p>
    <ul>
      <li class="test">rozpowszechniony</li>
      <li>dobrze znany</li>
      <li class="test">niezbyt rozbudowany</li>
    </ul>
  </body>
</html>
```



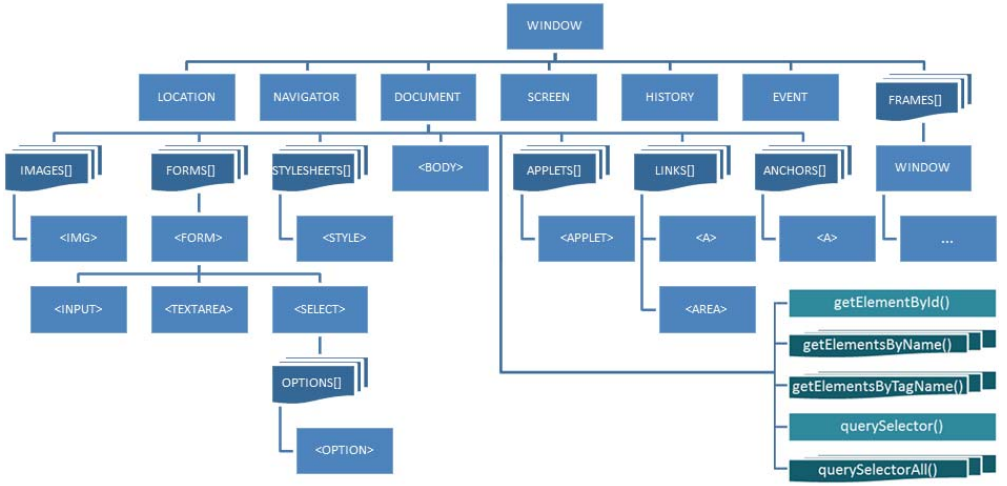
skrypt2.js

```
/*jshint browser: true, devel: true */
/*globals hasClass: false */
domReady(function () {
    var elems = hasClass('test'),
    res = elems.reduce(function (prev, curr, idx) {
        return prev + '\nelems[' + idx + ']=' + curr.nodeName;
    }, '');
    alert(elems.length + 'elems found' + res);
});
```

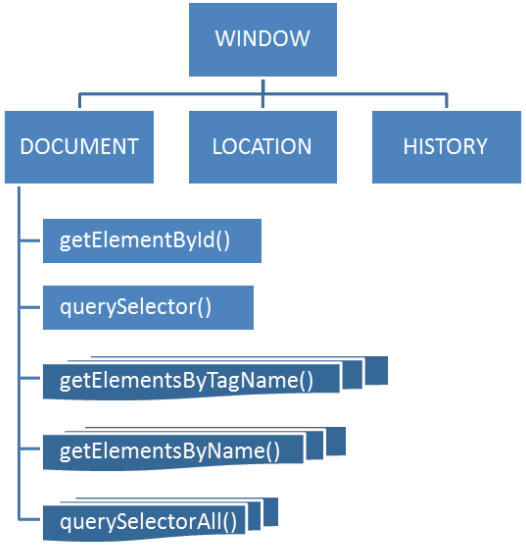

DOM API

„operacje na drzewie dokumentu”

Uproszczona struktura DOM API



Najważniejsze elementy



DOM – Document Object Model

- Manipulowanie węzłami
  - **appendChild**(nowy)
    - dodanie elementu na koniec listy childNodes
  - **insertBefore**(nowy, nast)
    - nast === **null** : na końcu childNodes
    - nast === w.**firstChild** : na początku childNodes
  - **replaceChild**(nowy, stary)
    - zastąpienie elementu
  - **removeChild**(węzeł)
    - usunięcie elementu
  - **cloneNode**(głębokie?), **normalize**()

DOM – Document Object Model

- Manipulowanie atrybutami
  - **getAttribute**(nazwa)
  - **setAttribute**(nazwa, wartość)
  - **removeAttribute**(nazwa)
- Tworzenie nowych węzłów
  - **document.createElement**(…)
  - **document.createTextNode**(…)

```
var e = document.createElement('div'),
    txt = document.createTextNode('A qq!');
 e.id = 'content';
 e.className = 'main';
 e.appendChild(txt);
 document.body.appendChild(e);
```


DOM – Document Object Model

- Styl elementu
  - **el.style.**nazwaCechy
  - nazwy cech CSS
    - CSS: **nazwa-pewnej-cechy**
    - DOM: **nazwaPewnejCechy**
    - wyjątek: **float** → **cssFloat** (**styleFloat** w IE8-)

DOM – Document Object Model

- Wyławianie elementów z użyciem selektorów
  - **`document.querySelector`**`(selektor)`
    - zwraca pierwszy węzeł pasujący do selektora
    - lub **`null`** – gdy takiego nie ma
  - **`document.querySelectorAll`**`(selektor)`
    - zwraca listę wszystkich węzłów pasujących do selektora
  - dostępność: IE8+ i „pozostałe przeglądarki”

DOM – Document Object Model

- Proste trawersowanie drzewa dokumentu
  - **`childElementCount`**
    - zwraca liczbę elementów potomnych (bez elementów tekstowych i komentarzy)
  - (**`first`**|**`last`**)**`ElementChild`**
    - zwraca pierwszy/ostatni element potomny
  - (**`previous`**|**`next`**)**`ElementSibling`**
    - zwraca poprzedni/kolejny element potomny
  - dostępność: IE9+ i „pozostałe przeglądarki”

DOM – Document Object Model

- Trawersowanie dokumentu – **`NodeIterator`**
  - **`createNodeIterator`**`(root,wts,filt)`
    - `root` : węzeł początkowy
    - `wts` : „maska bitowa” typów węzłów, np.
      `NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT`
    - `filt` : funkcja „filtrująca węzły”
  - zwraca obiekt typu **`NodeIterator`**
    - **`nextNode()`**
    - **`previousNode()`**
    - …
  - dostępność: IE9+ i „pozostałe przeglądarki”

DOM – Document Object Model

- **`NodeIterator`** – prosty przykład wykorzystania

```
/*jshint browser: true */
/*global NodeFilter: false */
var filter = function (node) {
    return node.nodeName.toLowerCase() !== 'script' ?
            NodeFilter.FILTER_ACCEPT :
            NodeFilter.FILTER_SKIP;
};

var root = document.documentElement,
    toShow = NodeFilter.SHOW_ELEMENT,
    iter = document.createNodeIterator(root, toShow, filter),
    node = iter.nextNode();

while (node !== null) {
    console.log(node.nodeName);
    node = iter.nextNode();
}
```

DOM – Document Object Model

- Trawersowanie dokumentu – **`TreeWalker`**
  - **`createTreeWalker`**`(root,wts,filt)`
    - parametry – jak dla **`NodeIterator`**
  - zwraca obiekt typu **`TreeWalker`**
    - **`nextNode()`**`,` **`previousNode()`**
    - **`parentNode()`**
    - **`firstChild()`**
    - **`lastChild()`**
    - **`previousSibling()`**
    - **`nextSibling()`**
  - dostępność: IE9+ i „pozostałe przeglądarki”

Problemy z DOM API

- Niemal dla każdej metody DOM API istnieje jakaś wersja przeglądarki (**IE**), w której metoda ta jest błędnie lub „niestandardowo” zaimplementowana
- **`getElementsByTagName(name)`**
  - dla „**`*`**” nie zwraca żadnych elementów w IE 5.5
  - dla „**`*`**” nie zwraca elementów <object> w IE 7
  - IE: atrybut **`length`** wyniku zostaje zamazany jeśli wśród znalezionych znajduje się element o **`id`** „length” (**sic!**)

DOM API a biblioteki JavaScript

- Używanie „gołego” DOM API jest wciąż stosunkowo uciążliwe (choć „idzie ku lepszemu”)
  - błędy/różnice w implementacjach standardu API w obecnie używanych przeglądarkach
  - konieczność śledzenia rozwoju przeglądarek
- Efektywniejszym rozwiązaniem jest wykorzystanie jednej z wielu dostępnych bibliotek JavaScript
  - ukrywają różnice pomiędzy przeglądarkami
  - oferują często znacznie bardziej koherentne API niż oryginalny DOM

Biblioteki i nawigowanie DOMowe

- Biblioteka **cssQuery**
  - modularna struktura i wsparcie dla wszystkich współczesnych przeglądarek
  - obsługa selektorów CSS 1-3

```
01. // Znajduje wszystkie dzieci elementów <div> będące
02. // akapitami
03. cssQuery("div > p");
04.
05. // Znajduje wszystkie elementy <div>, <p>, oraz <form>
06. cssQuery("div,p,form");
07.
08. // Znajduje w dokumencie wszystkie elementy <p> i <div>,
09. // a następnie w nich wyszukuje odsyłacze <a>
10. var p = cssQuery("p,div");
11. cssQuery("a",p);
12.
13. // Dodaje ramkę wokół odsyłaczy do Google
14. var g = cssQuery("a[href*='google.com']");
15. for ( var i = 0; i < g.length; i++ ) {
16.     g[i].style.border = "1px dashed red";
17. }
```

Biblioteki i nawigowanie DOMowe

- Biblioteka **jQuery**
  - wsparcie dla wszystkich współczesnych przeglądarek
  - obsługa selektorów CSS 1-3 oraz podzbioru XPath

```
01. // Znajduje wszystkie elementy <div> posiadające klasę
02. // "links" oraz element <p> wewnątrz
03. $("div.links[p]")
04.
05. // Znajduje wszystkich potomków elementów <p> lub <div>
06. $("p,div").find("*")
07.
08. // Znajduje co drugi odsyłacz wskazujący na Google
09. $("a[@href*='google.com']:even")
10.
11. // Dodaje ramkę wszystkim odsyłaczom do Google
12. $("a[@href*='google.com']").css("border","1px dashed red");
```


Biblioteki i nawigowanie DOMowe

- Biblioteka **YUI Selector Utility** (Yahoo)
  - wsparcie dla wszystkich współczesnych przeglądarek
  - obsługa selektorów CSS 1-3
  - jeden z modułów cenionej za swoją wysoką jakość biblioteki YUI

```
01. // Poczynając od pierwszego <ul> wewnątrz elementu
02. // o id = "nav" znajduje wszystkie <li>, które nie
03. // należą do klasy "selected"
04. Selector.query("#nav ul:first-of-type > li:not(.selected)");
05.
06. // Poczynając od pierwszego <ul> wewnątrz elementu
07. // o id = "nav" znajduje pierwszy <li> należący do
08. // klasy "selected"
09. Selector.query("ul:first-of-type > li.selected", "nav", true);
10.
11. // Dodaje klasę "odd" wszystkim nieparzystym wierszom
12. // elementu o id = "data"
13. Dom.addClass(Selector.query("#data tr:nth-child(odd)"), "odd");
```


Struktura strony a szybkość jej ładowania

- **Cuzillion**

http://stevesouders.com/cuzillion/

  - przydatne narzędzie do eksperymentów
  - przykładowy układ:
    - `http://stevesouders.com/cuzillion/?ex=10008`
- **YSlow**

http://developer.yahoo.com/yslow

DOM – dalsze informacje

http://www.w3schools.com/jsref/

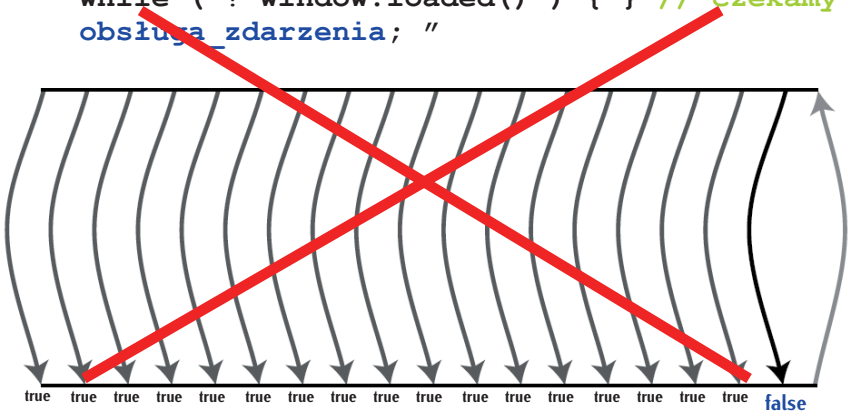
Język JavaScript

„DOM – obsługa zdarzeń”

JavaScript – brak obsługi wątków

- JavaScript **nie działa** wielowątkowo

```
" while ( ! window.loaded() ) { } // czekamy
  obsługa_zdarzenia; "
```



Asynchroniczna obsługa zdarzeń

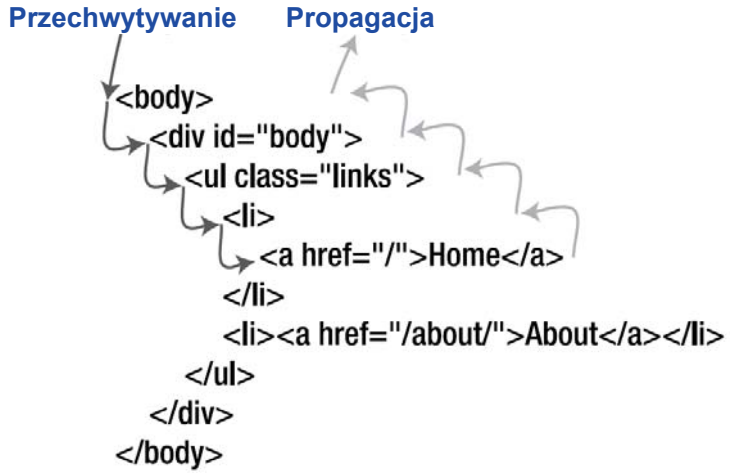
- Schemat „obsługi” zdarzenia
  - przygotowanie kodu „procedury obsługi”
  - „rejestracja” procedury obsługi zdarzenia

```
/*jshint browser: true */

// definiujemy „procedurę obsługi” zdarzenia
var loaded = function () {
    document.getElementById('body').style.border = '1px solid';
};

// Rejestrujemy procedurę przypisując ją do zdarzenia
// załadowania strony do okna przeglądarki
window.onload = loaded;
```

- Ilekroć dane zdarzenie zajdzie wykonana zostanie procedura jego obsługi

Asynchroniczna obsługa zdarzeń

- Każde zdarzenie ma dwie fazy



Prosty przykład

```
domReady(function () {
    var li = document.getElementsByTagName("li");
    Array.prototype.forEach.call(li, function (el) {
        el.onmouseover = function () {
            this.style.backgroundColor = 'blue';
        };
        el.onmouseout = function () {
            this.style.backgroundColor = 'white';
        };
    });
});
```

- Przykład ciągu zdarzeń dla **<li>** zawierającego **<a>**:
  - **<li>** mouseover: najeżdżamy myszą na **<li>**
  - **<li>** mouseout: mysz wędruje nad <a> „opuszczając” **<li>**
  - **<a>** mouseover: mysz najeżdża na **<a>** zmiana koloru tła **<li>** na niebieski
  - **<li>** mouseover: propagacja zdarzenia z **<a>** na **<li>**
- Użyta powyżej, tradycyjna metoda przypisywania obsługi zdarzeń przez atrybutu „on<zdarzenie>” obsługuje jedynie fazę propagacji

Prosty przykład poprawiony

```
/*jshint browser: true */

/*global domReady: false */

domReady(function () {
    var li = document.getElementsByTagName('li'),
        toBlue = function () {
            this.style.backgroundColor = 'blue';
        },
        toWhite = function () {
            this.style.backgroundColor = 'white';
        };

    Array.prototype.forEach.call(li, function (el) {
        el.onmouseover = toBlue;
        el.onmouseout = toWhite;
    });
});
```


Ważniejsze klasy zdarzeń w DOM

- zdarzenia „myszowe”
  - **mouseover**, **mouseout**, **mouseup**, **click**, ...
- zdarzenia „klawiaturowe”
  - **keyup**, **keydown**, **keypress**
- zdarzenie „interfejsowe”
  - **focus**, **blur**
- zdarzenia „formularzowe”
  - **submit**, **change**, **select**, **reset**
- zdarzenia ładowania dokumentu
  - **load**, **unload**, **beforeunload**, ...