JavaScript

Spis rysunków

Spis tabel

1 Wstęp do JavaScript

JavaScript (JS) jest językiem programowania zorientowanym obiektowo. Głównym środowiskiem wykorzystywania JS (JScript w Windows Internet Explorer i JavaScript w innych przeglądarkach) są przeglądarki internetowe. Język ten powstał, by działać po stronie klienta w architekturze klient-serwer.

1.1 Historia

Pracę nad JavaScriptem rozpoczęto w pierwszej połowie lat 90 XX wieku w firmie Netscape, pod nadzorem Brendana Eicha. Pierwsza implementacja języka pojawiła się w przeglądarce Netscape Navigator 2.0. Początkowo język miał nosić nazwę LiveScript, jednak z powodu bardzo dużej popularności Javy i podobieństw (celowych) w składni (nie w mechanizmach rządzących językiem) firma Netscape weszła w alians programistyczny z Sun Microsystems [1]. Nazwa ta, choć marketingowo okazała się sukcesem, w praktyce powodowała przez długi czas problemy. Zbyt wielu programistów uważało, że JavaScript to skrypty napisane w Javie. Co więcej, określenie „skrypty Javy" można nawet spotkać w niektórych książkach. Winą za taki stan rzeczy można tu obarczyć tłumaczy. JavaScript zdecydowanie więcej czerpie z takich języków jak Lisp czy Scheme niż Javy [2].

Wersja 1.0 okazała się sukcesem, stąd dalsze prace nad językiem i wprowadzenie wersji 1.1 do Navigatora 3.0. W podobnym czasie ukazała się przeglądarka Microsoft Internet Explorer 3.0 z wbudowaną obsługą klonu JavaScript – JScript. Zmiana nazwy miała uniknąć sporów licencyjnych. Mimo identycznego skrótu – JS – jak i często ujednolicania nazwy – JavaScript – różnice między dwoma wersjami języka były bardzo duże. W odróżnieniu od wielu innych dojrzałych języków programowania, dla JavaScript nie istniały standardy składni i możliwości funkcjonalnych, stąd dążenia do ujednolicenia języka i wprowadzenia jednego wiodącego standardu.

W roku 1997 język JavaScript 1.1 został przedstawiony jako propozycja standardu ECMA (Europejskie Stowarzyszenie Producentów Komputerów). Komitet techniczny TC39 otrzymał zadanie standaryzacji międzyplatformowego, neutrealnego języka skryptowego [10]. Komitet TC39 składał się ze specjalistów z firm żywo zainteresowanych rozwojem języka (m.in. Sun, Microsoft, Netscape). Wynikiem prac była norma ECMA-262 [40], która definiowała nowy język skryptowy nazwany ECMAScript. Rok później ISO/IEC również zaadaptowały ECMAScript jako normę – ISO/IEC 16262 [42] .

Niestety, mimo istniejących już od kilkunastu lat norm, wersje JavaScriptu zaimplementowane w różnych przeglądarkach nadal potrafią się znacznie różnić, co potrafi skutecznie utrudniać życie programistom JavaScript, a także znacznie wydłużać czas realizacji projektów i komplikować kod podczas zabiegów mających pozwalać poprawnie uruchamiać go w dowolnej przeglądarce.

1.2 Składniki JavaScript

ECMAScript nie jest związany z żadną konkretną przeglądarką. Nie obejmuje także operacji wejścia i wyjścia. W normie ECMA-262 można znaleźć informację: „*ECMAScript może udostępniać rdzeń możliwości języka skryptów dla całego szeregu środowisk i dlatego samo sedno języka zostało oddzielone od jakichkolwiek konkretnych środowisk pracy*" [1].

Przeglądarka internetowa jest jednym z przykładów środowisk pracy dla języka ECMAScript. Innymi przykładami środowisk ECMAScript są ActionScript firmy Adobe wykorzystywany

we Flashu oraz ScriptEase firmy Nombas wykorzystywany w Director MX [1].

JavaScript używany w przeglądarkach składa się z następujących elementów:

- ECMAScript – rdzeń
- Model DOM – Document Object Model, operowanie na zawartości dokumentu
- Model BOM – Browser Object Model, operowanie przeglądarką (otwieranie okien, alerty, zmiana rozmiaru okien, przechodzenie pod inny adres, itp.)
    - document (DOM)
    - frames
    - screen
    - navigator
    - history
    - location

Każdy z tych elementów, które wspólnie tworzą pełny język JavaScript jest implementowany w każdej przeglądarce zgodnie z wolą twórców, nie zawsze w zgodnie ze standardami znanymi już od lat. O ile ECMAScript oraz DOM [49] posiadają swoje standardy, to BOM niestety nie (BOM jest de facto nadzbiorem DOM). Niestety, istnienie standardów wcale nie sprawia, że wszystkie przeglądarki interpretują kod w ten sam sposób. Co więcej, często nawet nowe wersje przeglądarek nie są kompatybilne z kodem działającym na poprzednich wydaniach.

1.3 Najbardziej niezrozumiały język na świecie

Aby móc sprawnie i dobrze programować w JavaScript należy zapamiętać jedną ważna rzecz – to osobny język, a nie mniejsza czy uproszczona Java. Tak naprawdę prócz marketingowego zabiegu z nazwą i kilku podobieństw składniowych więcej dzieli te języki niż łączy,szczególnie, kiedy zagłębić się w specyfikę JS.

Kolejnym problemem JS jest fakt, iż w nazwie występuje słowo „script”, co przez wielu uznawane jest za sygnał o prostocie (prostactwie?) tego pełnoprawnego języka programowania o olbrzymich możliwościach (co udowadniają choćby takie aplikacje webowe jak Gmail, Google Maps czy Facebook). Przez długi czas JS był wykorzystywany do bardzo małych i często niepotrzebnych rzeczy (jak choćby padający śnieg na stronie). Często skrypty pisane były przez amatorów, co skutecznie odstraszało profesjonalnych programistów od „zabawki dla dzieci”.

JavaScript jest osobnym, porządnym językiem programowania zorientowanym obiektowo, czerpiącym wiele z języków funkcyjnych takich jak Lisp czy Scheme [2][39]. Podchodząc do niego jak do miniatury Javy można się jedynie sparzyć i zniechęcić. Jednak zabierając się do JS w odpowiedni sposób można jednak odkryć pełnię możliwości tego nietypowego języka.

1.4 Podstawy JavaScript

1.4.1 Prototypy zamiast klas

JavaScript jest językiem zorientowanym obiektowo, który jednak nie posiada klas, w znaczeniu znanym z innych popularnych, „klasycznych” języków obiektowych. Zamiast klas JS posiada prototypy obiektu. Każdy tworzony obiekt posiada swój prototyp.

W takim systemie także wiele cech programowania obiektowego (OOP), choćby dziedziczenie zmienia swoje właściwości. Schemat: klasa → obiekt oraz obiekt bazowy → obiekt pochodny przestaje obowiązywać. Obiekty dziedziczą bezpośrednio po innych obiektach. Jest to chyba największy problem dla początkujących adeptów JavaScript.

W przypadku JavaScript nie możemy stworzyć klasy, według której byłyby tworzone obiekty. Mimo, że w słowach kluczowych (zastrzeżonych) występuje słowo „class" [105] to nie ma ono na razie żadnego zastosowania. Zostało zarezerwowane już teraz, aby w przyszłości móc go użyć. Tak się stało choćby z ActionScriptem („kuzynem" JavaScript, także ECMAScriptem) w wersji 3.0 [27].

W dalszej części rozdziału zostanie pokazane, w jaki sposób tworzyć konstruktory obiektów oraz w jaki sposób uzyskać dziedziczenie w JavaScript.

1.4.2 Sposoby na tworzenie obiektów

O ile w klasycznych języka obiektowych obiekty tworzy się generalnie w jeden sposób – z wykorzystaniem operatora `new' wywołującego odpowiedni konstruktor klasy, tak w przypadku JavaScript sposobów jest co najmniej 2, choć można by mieszając je uzyskać około 10 sposobów [1][2][3].

1.4.3 Ręczne budowanie obiektu

Jest to najbardziej prymitywny sposób na tworzenie obiektów. JavaScript pozwala na dowolne rozszerzanie istniejących obiektów. Tworząc ręcznie obiekt wykorzystamy elastyczność języka i jego dynamiczny charakter:

```
var oExample = new Object(); // 1
oExample['name'] = 'HelloWorldski'; // 2
oExample.introduce = function()
{
      return this.name; // 3
}
oExample.introduce(); // 4
```

W wyniku działania takiego kodu otrzymamy napis „HelloWorldski". Na powyższym listingu widzimy, że w linii 1 do zmiennej `oExample' przypisujemy nową instancję obiektu. Następnie (linie 2 i 3) „wypełniamy" obiekt przypisując do niego dynamicznie kolejne pola. Pola te potem mogą być nadpisane (zawsze, czy to przez autora kodu, czy też przez programistę – użytkownika obiektu, także obiektów natywnych JavaScript). Nie ma kontroli typów, mimo że coś takiego jak typ istnieje - jest tylko informacją dla programisty. W linii „3" użyty jest operator `this' [103], który wskazuje na obiekt `oExample', co nie jest wcale tak oczywiste, co zostanie wykazane w dalszej części pracy.

Ten sposób tworzenia obiektów ma kilka podstawowych wad:

1. Wymaga niezwykłej skrupulatności (za każdym razem, gdy chcemy stworzyć obiekt musimy ręcznie ustawić wszystkie pola).
2. Jest bardzo niewygodny i długi. Stworzenie tak prostego obiektu zajmuje 3 linie.
3. Jest podatny na możliwość nadpisania `Object'!

Punkty 1 i 2 zostaną rozwiązane przy zastosowaniu wzorca fabryki / modułu [2][3]. Problem opisany w punkcie 3 można rozwiązać używając literału obiektowego. JavaScript posiada skrótowy zapis zarówno dla `Object' (konstruktor obiektów) jak i Array (konstruktor tablic):

```
var o = new Object(); // lub new Object;
```

można zastąpić takim kodem (literałem obiektowym):

```
var o = {};
```

W przypadku tablic:

```
var a = new Array();
```

takim kodem (literałem tablicowym):

```
var a = [];
```

Sposób ten nie tylko jest szybszy, ale także pozwala na „spokojny sen”. Zarówno `Object', jak `Array' są zmiennymi, do których można przypisać inną wartość, np.:

```
Object = 2;
var example = new Object;
example;
```

Zwraca błąd: „*TypeError: Object is not a constructor*”. Jako, że `Object' jest zmienną globalną, nadpisanie jej w jakimkolwiek fragmencie kodu potrafi skutecznie uniemożliwić pracę całej aplikacji. Co więcej problem ten pojawi się w trakcie uruchomienia, nie kompilacji. W JavaScript nie ma w ogóle procesu kompilacji (jest to język interpretowany), więc wszystkie tego typu błędy ujawniają się dopiero w trakcie wykonania.

Co więcej, wykorzystując literały obiektowe można stworzyć obiekt szybciej:

```
var oExample = {
      name : 'HelloWorldski',
      introduce : function() { return this.name; }
};
oExample.introduce();
```

I w tym przypadku kod jest bardzo podatny na proste błędy, szczególnie jeśli uruchamiamy go w Internet Explorer (często określany dalej skrótem „IE”). Literał obiektowy ma bardzo prostą strukturę:

```
{ pola }
```

gdzie pola mają strukturę:

```
para,
para,

…

para
```

gdzie każda para ma strukturę:

```
klucz : wartość,
```

Jeśli `klucz' jest ciągiem znaków, który nie jest poprawną nazwą zmiennej (rozpoczyna się od litery, podkreślenia lub znaku dolara, nie zawiera spacji, jedynym dozwolonym znakiem specjalnym jest znak dolara [76]), powinien być objęty w apostrofy lub cudzysłów. Jeśli kluczem jest słowo kluczowe języka, dla pewności także lepiej jest umieścić je w cudzysłowie – niektóre przeglądarki tego wymagają. Wartość może być typem prostym (liczba, ciąg znaków, prawda/fałsz, null, undefined; [3]), obiektem (także zagnieżdżonym literałem obiektowym), funkcją, tablicą (zarówno funkcja, jak i tablica są obiektami).

W przypadku IE ostatnia para nie może posiadać przecinka na końcu, mimo że specyfikacja JavaScript zezwala na przecinek za ostatnią parą [2]. Jest to jeden z przykładów prostych błędów,

z którymi na co dzień zmagają się programiści.

Aby odwoływać się do pól można skorzystać z dwóch sposobów:

```
var o = { a : 1, b : 2 };
o.a; // 1
o['b']; // 2
```

Drugi sposób („2”) pozwala na korzystanie z nazw pól, które nie są poprawnymi nazwami zmiennych:

```
var o = {
      a : 1,
      'b-z' : 2 // #
};
o.a;
o['b-z']; // #
```

W powyższym przykładzie widać („#”), że w takim przypadku należy także inaczej zdefiniować klucz w literale obiektowym – ująć go w apostrofy lub cudzysłów.

1.4.4 Fabryka obiektów

Aby pozbyć się części problemów opisanych w poprzednim podrozdziale warto korzystać ze wzorca fabryki obiektów.

```
var createObject = function()
{
      return {
            name : 'HelloWorldski',
            introduce : function()
            {
                  return this.name;
            }
      };
};
oExample = createObject();
oExample.introduce();
```

Po raz kolejny tak prosty kawałek kodu jest podatny na niezwykle pospolity i tym razem bardzo trudny do wykrycia błąd. Źródłem jest fakt, że JavaScript nie wymaga, aby po każdej instrukcji był stawiany średnik, choć byłoby to rozwiązanie bardzo dobre [39].

Aby sprowokować wystąpienie tego błędu starczy zapisać powyższy kod jeszcze raz, z drobną zmianą w formatowaniu (położenia nawiasu klamrowego):

```
var createObject = function()
{
      return
      {
            name : 'HelloWorldski',
            introduce : function()
            {
                  return this.name;
            }
      };
};
oExample = createObject();
oExample.introduce();
```

W wyniku uruchomienia tego kodu otrzymujemy błąd: „*SyntaxError: invalid label*”. Przyczyną

tego błędu jest postawienie klamry w nowej linii, zamiast zaraz za wyrażaniem `return'. Błąd ten polega na dostawieniu automatycznie średnika na końcu każdej linii, nie zakończonej średnikiem, jeśli oczekiwany jest tam średnik. Jest to spowodowane faktem, że wyrażanie `return' może być puste. To znaczy, że funkcja ma prawo nic nie zwrócić, a `return' być użytym jedynie jako bezwarunkowe opuszczenie dalszego kodu [2].

Średniki stawiane za klamrą nie są w powyższych listingach wymagane. Warto natomiast stosować taką konwencję, ponieważ pozwala ona na odróżnienie dowolnej klamry od klamry zamykającej literał obiektowy lub konstruktor. Także zapis `var nazwaFunkcji = function() { … }', który działa jak kod `function nazwaFuncji() { … }' można zarezerwować sobie dla funkcji będących fabrykami obiektu. Metody prywatne wykorzystują „tradycyjny" sposób deklaracji funkcji. Powyższa konwencja została zaproponowana i wykorzystana w wielu projektach przez autora pracy.

1.4.5 Tworzenie obiektów z prototypów

Jak już wcześniej wspomniano JS jest pozbawiony klas. Zamiast klas występują prototypy (łańcuch prototypów [17]). Każdy obiekt tworzony jest według swojego prototypu. Prototyp może być dynamicznie zmieniany, rozszerzany lub ograniczany (przez usuwanie odpowiednich pól).

Aby stworzyć nowy obiekt należy wykorzystać funkcję:

```
var MyObject = function()  {};
oExample = new MyObject();
```

Funkcja `MyObject' jest tu konstruktorem obiektu. Średnik za klamrą zamykającą konstruktor nie jest obowiązkowy – jest częścią proponowanej konwencji.

Powyższy kod jest całkowicie prawidłowy. Jednak stworzony obiekt nie ma żadnej funkcjonalności. Poniższy kod pokazuje jak za pomocą mechanizmu prototypów stworzyć obiekt równoważny z obiektem tworzonym przez funkcję `createObject'.

```
var MyObject = function()
{
      this.name = 'HelloWorldski';
      this.introduce = function() { return this.name; }
};
oExample = new MyObject();
oExample.introduce();
```

W tym wypadku podczas wywołania konstruktora do nowo powstałego obiektu dodawane są kolejne pola (`name' oraz `introduce'). Nie są to jednak pola prototypu `MyObject', a jedynie instancji obiektu, co udowadnia poniższy kod:

```
var MyObject = function()
{
      this.name = 'HelloWorldski';
      this.introduce = function() { return this.name; }
};
oExample = new MyObject();

oExample.hasOwnProperty('introduce'); // true
MyObject.hasOwnProperty('introduce'); // false
```

Metoda `hasOwnProperty' [96] sprawdza, czy dany obiekt posiada pole podane jako argument. Metoda ta jest dziedziczona przez wszystkie obiekty zaczynając od `Object'. Nie jest sprawdzany łańcuch prototypów w górę.

W przypadku obiektu `oExample', który jest instancją `MyObject', metoda zwraca prawdę – ten

obiekt posiada pole `introduce', ponieważ jest ono dodane w konstruktorze. Obiekt `MyObject' nie posiada tego pola.

Problem można obejść w następujący sposób:

```
var MyObject = function()  {};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
     return this.name;
}
oExample.hasOwnProperty('introduce'); // false
MyObject.prototype.hasOwnProperty('introduce'); // true
```

Widzimy, że mimo iż `oExample.introduce' istnieje, to metoda `hasOwnProperty' zwraca `false'. Jest to pole dziedziczone po prototypie `MyObject' – nie jest zatem polem własnym. Aby sprawdzić, czy dane pole jest dostępne w danym obiekcie należy wykorzystać operator `in' [100] lub operator `typeof' [104].

```
var MyObject = function()  {};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
     return this.name;
}
typeof oExample.introduce !== 'undefined'; // true
introduce in oExample; // true
MyObject.prototype.hasOwnProperty('introduce'); // true
```

Mimo że obiekt `oExample' nie posiada własnego pola `introduce', pole to jest jego składową, poprzez dziedziczenie z łańcucha prototypów. Oczywiście obiekt może także posiadać swoje własne wartości pól, ustawiane w konstruktorze:

```
var MyObject = function(name)
{
      this.name = name;
};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
      return this.name;
}
oExample = new MyObject('MyExample');
oExample.introduce(); // 'MyExample'
MyObject.prototype.introduce(); // 'HelloWorldski';
```

1.4.6 Dynamiczna zmiana obiektu

Oprócz możliwości dodawania pól w konstruktorze JavaScript pozwala na dodawanie pól do już istniejących obiektów, usuwanie pól z obiektów, jak i nadpisywanie już istniejących pól – w tym metod.

```
var MyObject = function(name)
{
      this.name = name;
};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
      return this.name;
```

```
}
oExample = new MyObject('MyExample');
oExample.introduce = function()
{
      alert(this.name);
}
oExample.introduce();
```

Powyższy kod wyświetla alert z nazwą obiektu (poprzedni przykład jedynie zwracał wartość, bez ingerencji w sposób prezentacji danych).

Aby usunąć jakieś pole z obiektu należy wykorzystać operator `delete' [99].

```
var MyObject = function(name)
{
      this.name = name;
};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
     return this.name;
}
oExample = new MyObject('MyExample');
delete oExample.name;
oExample.introduce(); // 'HelloWorldski';
```

Jak pokazuje powyższy przykład, usunięte pole `oExample.name' zostaje „zastąpione” polem `MyObject.prototype.name'. Po raz kolejny daje o sobie znać łańcuch prototypów.

Dopiero taka konstrukcja:

```
var MyObject = function(name)
{
      this.name = name;
};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
      return this.name;
}
oExample = new MyObject('MyExample');
delete oExample.name;
delete MyObject.prototype.name;
oExample.introduce(); // 'HelloWorldski';
```

Powoduje, że zostaje zwrócona wartość `undefined' [97].

Warto także mieć na uwadze, że zmiana jakiegoś pola w prototypie zmienia także to pole we wszystkich obiektach dziedziczących z tego prototypu, nawet tych utworzonych przed zmianą. Jednak podmiana całego prototypu nie zmienia niczego w obiektach już istniejących.

```
var MyObject = function(name)
{
      this.name = name;
};
MyObject.prototype.name = 'HelloWorldski';
MyObject.prototype.introduce = function()
{
      return this.name;
}
oExample = new MyObject('MyExample');
oExample.introduce(); // „MyExample”
```

```
MyObject.prototype.introduce = function()
{
      return 'zmieniony';
}
oExample.introduce(); // 'zmieniony'
```

1.4.7 Operator `this' i `new'

Jak pokazano w powyższych przykładach, tworzenie obiektów w JS można podzielić na dwie główne grupy: fabrykę obiektów [2] oraz wzorzec konstruktor – prototyp [1]. W przypadku fabryki obiektów operator `new' nie jest potrzebny, można z niego skorzystać. Jeśli kod jest wolny od dziwnych i niepewnych konstrukcji wykorzystanie tego operatora nie powinno wyrządzić żadnych szkód. W przypadku wzorca konstruktor – prototyp brak użycia operatora `new' powoduje duże problemy, często uniemożliwiając dalsze działanie.

Oto prosty przykład:

```
var value = 'Hello world';
var MyThis = function()
{
      alert(this.value );
}
var exampleObject = new MyThis(); // undefined
var exampleFunction = MyThis();// 'Hello world'
alert(window.value); // 'Hello world';
```

Wywołanie funkcji `MyThis' z operatorem `new' sprawia, że operator `this' jest wiązany z kontekstem obiektu, w którego konstruktorze został wywołany [103]. W przypadku wywołania konstruktora `MyThis' bez operatora `new', operator `this' zostaje związany z kontekstem globalnego obiektu `window', podobnie jak wywołanie alert od zmiennej globalnej. Zarówno `value' jak i `window.value' odwołuje się do zmiennej globalnej `value'. W tak skonstruowanym kodzie, jak powyższy można także wywoływać `MyThis' przez 'window.MyThis'.

Zakresy, z którym związany jest operator `this' przedstawia poniższa tabelka [103]:

Typ	**Wywołane przez**	**this**
Ukryty przy wywołaniu metody	*object*.*method*([...])	*object*
Jednoznacznie poprzez `Function.prototype.call' [95]	*function*.call(*object*,...)	*object*
Jednoznanie przez `Function.prototype.apply' [94]	*function*.apply(*object*,[...])	*object*
Niejawnie przy wykorzystaniu operatora `new' [102]	new *constructor*([. . .])	Nowy kontekst

Tabela 1. Powiązania operatora `this' z zakresem

Dziwne i chwilami enigmatyczne działanie operatora `this' jest przyczyną, z powodu której wielu bardzo doświadczonych programistów JavaScript, na czele z Douglasem Crockfordem [2][85] uważa, że można sobie dobrze radzić bez operatora `new' i konsekwentnie unikają jego używania.

Literały obiektowe

Także w przypadku omawianych wcześniej literałów obiektowych można używać operatora `this', w celu odniesienia się do innych pól tego samego obiektu, np.:

```
var o = {
```

```
      a : 1,
      b : function() { alert(this.a); }
};
o.b(); // komunikat wyświetli „1”
```

Wykorzystanie „that”

W przypadku, kiedy potrzebny jest operator `this', jednak nowe domknięcie (szerzej o domknięciach w podrozdziale 1.4.9) nie pozwala na jego stosowanie wykorzystuje się prostą sztuczkę z „operatorem” that. Polega ona na przypisaniu do zmiennej o nazwie „that” wartości `this'. Jest to dozwolone (niemożliwym jest nadpisanie `this', można jednak przypisać jego wartość do innej zmiennej).

```
var o = {
      a : 1,
      b : function()
      {
            var that = this;
            function inner()
            {
                  alert(that.a); // zastąpienie this → that
            }
            inner();
      }
};
o.b(); // komunikat wyświetli „1”
```

Zmiana `that.a' na `this.a' spowoduje, że zmienna `a' będzie szukana globalnie, a nie w tym konkretnym obiekcie. Wykorzystanie operatora `this' bez operatora `new' tak działa. Nie jest to błąd języka, jest to jego właściwość. Oczywiście zmienna `that' może mieć inną nazwę. Trzy najczęściej spotykane przez autora nazwy to: „that”, „self” oraz „_this”.

Operator `this' przy zdarzeniach przypisanych przez atrybut

Innym przypadkiem, w którym operator `this' wiąże się z jeszcze innym kontekstem jest obsługa zdarzeń przypisany przez atrybuty (opisane szerzej w rozdziale 2).

```
<html>
<body>
      <a href='javascript:void()' onclick='alert(this.innerHTML)'>
            Test `this'
      </a>
</body>
</html>
```

W powyższym kodzie `this' został związany z węzłem, który został kliknięty. Stąd w wynikowym alercie pojawi się „Test `this'”.

Także taki kod, jak z powyższego listingu może być przyczyną problemów. Starczy uporządkować kod i wszystko, co działało, „nagle” przestaje:

```
<html>
<body>
      <script type='text/javascript'>
      function onClickHandler()
      {
            alert(this.innerHTML)
      }
      </script>
      <a href='javascript:void()' onclick='onClickHandler()'>Test `this'</a>
</body>
```

```
</html>
```

W alercie pojawi się „undefined”. Dlaczego? Funkcja zostanie wywołana bez związania z kontekstem. Dlatego operator `this' wskazywać będzie na obiekt globalny. Aby to sprawdzić, wystarczy zmienić kod JS:

```
var innerHTML = 'Rzeczywiście, kontekst obiektu globalnego';
function onClickHandler()
{
      alert(this.innerHTML)
}
```

Istnieje kilka rozwiązań takiego problemu:

1. używać tak jak na pierwotnym listingu inline'owych kodów JS „wstrzykniętych” w atrybut `onclick'.
2. Przekazywać `this' jako parametr do funkcji:

```
<script type='text/javascript'>
function onClickHandler(self)
{
      alert(self.innerHTML)
}
</script>
<a href='javascript:void()' onclick='onClickHandler(this)'>
      Test `this'
</a>
```

3. Wiązać ręcznie wywołaną funkcję z kontekstem.

Rozwiązanie numer 3 jest w opinii autora najsłuszniejsze i „najbardziej w duchu JavaScript”.

1.4.8 Wiązanie `this' z wybranym kontekstem

JavaScript pozwala na wiązanie wywołanej funkcji z odpowiednim kontekstem. W powyższych przykładach pokazałem, że wynik działania kodu (a w szczególności kontekst w jakim jest wykonany) może czasem zadziwiać. Dodatkowo JavaScript posiada mechanizmy pozwalające na sterowanie tym, w jakim kontekście ma zostać uruchomiona funkcja. Do tego celu wykorzystuje się metody `apply' i `call' dziedziczone przez każdą funkcję z prototypu `Function'.

W takim wypadku przykład z wywołaniem funkcji z odpowiednim kontekstem z poprzedniego podrozdziału wyglądałby tak:

```
<script type='text/javascript'>
function onClickHandler()
{
      alert(this.innerHTML)
}
</script>
<a href='javascript:void()' onclick='onClickHandler.apply(this)'>
      Test `this'
</a>
```

Powyższy przykład mógłby równie dobrze wykorzystywać `call'. Różnica między tymi metodami polega na sposobie przekazania parametrów do wywoływanej funkcji. W `apply' przekazuje się tablicę, w `call' argumenty przekazujemy jeden po drugim :

```
fun.call(thisArg[, arg1[, arg2[, ...]]])
```

1.4.9 Dziedziczenie

Dziedziczenie (ang. *inheritance*) to w programowaniu obiektowym operacja polegająca na stworzeniu nowej klasy na bazie klasy już istniejącej. Choć ECMAScript [40] rezerwuje słowo kluczowe `extends' [105], stosowane w wielu językach programowania w zapisie oznaczającym dziedziczenie, to JavaScript nie ma zaimplementowanej wprost tej funkcjonalności. Fakt ten jednak wcale nie oznacza, iż dziedziczenia nie ma. Po prostu można je uzyskać innymi sposobami.

W JavaScript istnieje co najmniej 11 sposobów na dziedziczenie [3]. Absolutnie nie oznacza to, iż jest to już górne ograniczenie możliwości języka. W swojej pracy ograniczę się do pokazania dwóch metod, opartych o wcześniejsze sposoby tworzenia obiektów. Najpierw metoda korzystająca z wiązania funkcji z odpowiednim kontekstem:

```
var Person = function( name, age )
{
      this.name = name;
      this.age = age;
};
Person.prototype.introduce = function()
{
      return 'Mam na imię ' + this.name + ', \nMam ' + this.age + ' lat';
}
var Student = function(name, age, university)
{
      Person.call(this, name, age);
      this.university = university;
};
Student.prototype.introduce = function()
{
      return 'Mam na imię ' + this.name + ', Mam ' + this.age +
            ' lat, Jestem studentem ' +  this.university;
}
var oPerson = new Person('Jan Kowalski', 30);
var oStudent = new Student('Piotr Nowak', 22, 'PG');
alert(oPerson.introduce()); // Mam na imię Jan Kowalski, Mam 30 lat
alert(oStudent.introduce()); // Mam na imię Piotr Nowak, Mam 22 lat,
                          // Jestem studentem PG
```

Powyższy fragment kodu pozwala na stworzenie prostego dziedziczenia. Istnieje jednak pewien szkopuł – obiekty `oPerson' oraz `oStudent' nie są wcale tego samego typu. Można to łatwo sprawdzić za pomocą operatora `instanceof' [101].

```
// tu kod z poprzedniego listingu
alert(oPerson instanceof Person); // true
alert(oStudent instanceof Person); // false
alert(oStudent instanceof Student); // true
```

O ile w Javie/C++/C# (słowem – językach o silnej kontroli typów) byłoby to problemem nie do rozwiązania, tak w JavaScript można to zignorować. W omawianym języku nie występuje ścisła kontrola typów, co często bywa problematyczne. Jeśli jednak zależy nam na „pełnym” dziedziczeniu, należałoby nadpisać w odpowiedni sposób pole `prototype' konstruktora obiektu `Student'.

```
var Person = function( name, age )
{
      this.name = name;
      this.age = age;
};
var Student = function(name, age, university)
{
```

```
      Person.call(this, name, age);
      this.university = university;
};
Student.prototype = new Person();
var oStudent = new Student('Tomasz Wiśniewski', 25, 'UG');
alert(oStudent instanceof Person); // true
alert(oStudent instanceof Student); // true
```

Jak pokazuje powyższy przykład, teraz wszystko działa tak, jak spodziewano by się tego w Javie. Z pewnych przyczyn część literatury fachowej [3] zalecałaby także napisanie pola `Student.prototype.constructor' przypisując do niego funkcję (konstruktor) `Student'.

Przedstawiony sposób dziedziczenia ma jeden duży problem – pozbawiony jest pól o zakresie innym niż publiczne. Zmusza to do stosowania dziwnych konwencji z podkreśleniami, które mogą być (choć nie muszą i często nie są) respektowane. W przypadku wykorzystania dziedziczenia funkcyjnego można uzyskać efekt pól prywatnych.

```
var person = function( name, age )
{
      var sName = name,
            nAge = age;

      function fIntro()
      {
            return 'Mam na imię ' + sName + ', \nMam ' + nAge + ' lat';
      }

      return {
            introduce : fIntro
      };
};
var student = function( name, age, university)
{
      // stwórz obiekt `person'
      var oSelf = person(name, age);

      // pola prywatne dla obiektu `student'
      var sUni = university,
            fSuperIntro = oSelf.introduce;

      oSelf.introduce = function()
      {
            return fSuperIntro() + ', Jestem studentem ' + sUni;
      }
      return oSelf;
};
var oPerson = person('Jan Kowalski', 23);
var oStudent = student('Piotr Nowak', 21, 'PG');

alert(oPerson.introduce());
alert(oStudent.introduce());
```

Takie rozwiązanie pozwala na odziedziczenie wszystkich publicznych pól, a także umożliwia na uzyskanie dostępu do pól prywatnych obiektu `person' – mechanizm domknięć. Mała litera w nazwach funkcji tworzących obiekty jest zabiegiem celowym, zgodnym z konwencją proponowaną przez D. Crockforda [2] – funkcje tworzące obiekty, które wymagają operatora `new' rozpoczynają się wielką literą, pozostałe małą. Zarówno obiekty tworzone za pomocą fabryki `person', jak i `student' są instancjami `object', tak jak domyślnie każdy obiekt tworzony za pomocą literału obiektowego. Nie stanowi to jednak zbyt wielkiego problemu. JavaScript podobnie

jak Python jest językiem pozwalającym na tzw. kacze typizowanie (ang. *duck typing*)[4] – "jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką". Interfejs jest więc tu ważniejszy od typu obiektu.

1.4.10 Domknięcia

Jednym z ciekawszych i coraz bardziej docenianych mechanizmów JavaScript są domknięcia (ang. *closure)*. Domknięcia występują głównie w językach funkcyjnych, w których funkcje mogą zwracać inne funkcje, wykorzystujące zmienne utworzone lokalnie [33].

W JS bardzo istotną rolę pełnią funkcje. Funkcje są obiektami, funkcje służyć mogą jako proteza przestrzeni nazw, konstruktory, czy też wreszcie są przecież zwykłymi funkcjami.

Każda zmienna ma swój zakres. Tworzenie zmiennej sprawia, że zostaje ona przypisana do jakiegoś domknięcia. Domyślnie (zmienna, która nie znajduje się w żadnej funkcji) jest to zakres globalny. Nowe domknięcie tworzone jest za pomocą nowej funkcji, np.:

```
var value = 1;
alert('value = ' + value); // 1
function closure()
{
      var value = 2;
      alert('value = ' + value); // 2
}
alert('value = ' + value); // 1
```

Usunięcie słowa kluczowego `var' z funkcji `closure' sprawia, że operujemy na zmiennej z wyższego domknięcia.

```
var value = 1;
alert('value = ' + value); // 1
function closure()
{
      value = 2;
      alert('value = ' + value); // 2
}
alert('value = ' + value); // 2 – wartość została nadpisana w funkcji
```

Takie działanie kodu sprawia, że można to bardzo ciekawie wykorzystać przy tworzeniu protezy przestrzeni nazw. Często jednak sprawia także olbrzymie problemy.

Zmienne w JS nie muszą być deklarowane za pomocą słowa kluczowego `var'. Pierwsze użycie zmiennej automatycznie tworzy ją w pamięci. Gdyby teraz nieopatrznie ktoś zapomniał napisać `var value', mogłoby to spowodować nadpisanie ważnych danych i w konsekwencji niewłaściwe działanie całego systemu. Jest to bardzo trudny do wykrycia błąd. Stąd też, często sugerowane jest, aby wszystkie zmienne były deklarowane na początku domknięcia (funkcji), a nie najbliżej pierwszego użycia [2]. Nawet narzędzie do statycznej analizy kodu JSLint [43] autorstwa D. Crockforda pozwala na ustawienie opcji „*Allow one var statement per function*", która sprawdza, czy w kodzie deklaracje zmiennych nie są rozsiane po całym ciele domknięcia [2].

Innym często spotykanym błędem, jest brak zrozumienia wiązania symbolu zmiennej z jej wartością. Domyślnie zmienna nie przyjmuje wartości jaką ma w chwili dynamicznego tworzenia funkcji, zatem kod z poniższego listingu może wydawać się poprawnym:

```
// załóżmy, że było 10 hiperlinków
var hiperlinks = document.getElementsByTagName('a');
for(var i = 0; i < hiperlinks.length; i++)
{
      hiperlinks[i].onclick = function(e) { alert(i); } // 1
```

```
}
```

Przy założeniu, że oczekiwanym jest, aby do każdego znacznika <a/> została przypisana funkcja obsługi zdarzenia `onclick' wyświetlająca wartość 9 (maksymalna wartość osiągnięta przez `i') ten kod jest poprawny. Najczęściej jednak autorowi takiego kodu chodzi o wyświetlenie indeksu aktualnego węzła hiperłącza. Dzieje się tak dlatego, że w chwili tworzenia funkcji („1”) z kodem zostaje związana nie wartość zmiennej, a referencja do niej. Wartość zmiennej ulega zmianie w czasie, przez co wynik jest inny niż mogłoby się wydawać na pierwszy rzut oka.

Aby to rozwiązać należy stworzyć nowe domknięcie i wykorzystać jego właściwości:

```
var hiperlinks = document.getElementsByTagName('a');
for(var i = 0; i < hiperlinks.length; i++)
{
      hiperlinks[i].onclick = (function(j)
      {
            return function(e) { alert(j); } // 1
      })(i); // 2
}
```

W takim wypadku podczas przypisywania funkcji obsługi zdarzenia `onclick' zostało stworzone nowe domknięcie oraz przekazano do niego wartość aktualnego `i' (linia „2”). Taki kod będzie działał bardziej intuicyjnie – kolejne znaczniki <a/> po kliknięciu będą pokazywać komunikat ze swoim indeksem (linia „1”).

1.4.11 Pseudo przestrzenie nazw

Mimo że JS nie posiada natywnego wparcia dla przestrzeni nazw, można stosując jej wbudowane mechanizmy uzyskać identyczny efekt. W tym celu wykorzystać należy wzorzec modułu [2], który de facto jest tym samym, co opisywany wcześniej wzorzec fabryki. Różnice w kodzie praktycznie nie istnieją, polegają raczej na przyjętych standardach kodowania. Główną różnicą jest idea przyświecająca powstaniu kodu – we wzorcu fabryki chodzi o kod funkcji „produkującej” gotowy obiekt. We wzorcu modułu chodzi o odpowiednie oddzielenie części systemu od reszty, stworzenie odpowiedniego modułu – klocka całości.

W celu stworzenia nowego modułu wykorzystuje się funkcje. Przykładowa implementacja:

```
var project = function()
{
      // tu pola prywatne
      return {
            ui : (function()
            {
                  // tu pola prywatne
                  return {
                        widget : function() { alert('działa!'); }
                  }
            })(),
            model : (function()
            {
                  // tu pola prywatne
                  return {
                        getData : function() { alert('dane'); }
                  }
            })()
      };
};
```

Taki kod w zastosowaniu wygląda identycznie jak kod języków wspierających przestrzenie nazw:

```
var myProject = project();
myProject.ui.widget(); // wyświetli „działa!”
```

1.4.12 Bloki kodu

Doświadczeni programiści języków dziedziczących po C (C++, Java, C#) mogą się bardzo zdziwić, gdy taki kod będzie działał:

```
var n = 10;
for (var i = 0; i < n; i++)
{
      if ( 3 === i )
      {
            var success = 'Znalazłem szukaną liczbę';
      }
}
alert(success);
```

W alercie zostanie wyświetlony komunikat „Znalazłem szukana liczbę”.

Jak widać, ani klamry (jak w C), ani wcięcia (jak w Pythonie) nie tworzą w JavaScript nowego bloku kodu (z własnym zakresem nazw). Jedynie funkcja tworzy nowe domknięcie, w którym widoczne są wszystkie zmienne z wyższych domknięć oraz zmienne lokalne.

Zmienna lokalna, jeśli jej nazwa pokrywa się z nazwą zmiennej z wyższego domknięcia uniemożliwia odwołanie się wprost do wartości tamtej zmiennej. Wyjątkiem jest domknięcie globalne, do którego w przeglądarkach można się zawsze odwołać przez obiekt globalny `window':

```
window.zmienna;
window['zmienna'];
```

Pierwszy przykład z tego podrozdziału można by napisać z wykorzystaniem funkcji anonimowej, która dałaby wrażenie, że kod działa „poprawnie” (w rozumieniu programistów C i im podobnych).

```
var n = 10;
(function()
{
      for (var i = 0; i < n; i++)
      {
            if ( 3 === i )
            {
                  var success = 'Znalazłem szukaną liczbę';
            }
      }
})();
alert(success);
```

W stworzonej funkcji anonimowej są widoczne wszystkie zmienne z domknięcia wyższego, jednak kod zawarty w tej funkcji nie propaguje się na poziom wyższy.

Jest to bardzo często stosowana metoda uniknięcia „brudzenia” przestrzeni globalnej (ang. *JavaScript global namespace pollution*)[13]. Szczególnie przydatne rozwiązanie to staje się w momencie, kiedy w naszym projekcie używamy więcej niż jednej biblioteki. Biblioteki JavaScript mają często (poniekąd zrozumiałą z chęci oszczędzenia miejsca) własność, iż wykorzystują znak `$' jako nazwę zmiennej globalnej, pod którą są dostępne. Rozwiązanie to jest bardzo wygodne. Do czasu. W momencie, kiedy w projekcie występują dwie biblioteki, korzystające z tego podejścia, jedna niechybnie nadpisze drugą. Na szczęście przeważnie twórcy takich frameworków są świadomi zagrożenia, i prócz skróconej nazwy udostępniają też funkcjonalność pod inną zmienną o dłuższej nazwie, związanej z nazwą biblioteki.

W takiej sytuacji warto wykorzystać wyżej opisaną cechę języka:

```
(function($)
{
      // tu wkleić kod wykorzystujący $ jako referencję do biblioteki1
})(biblioteka1);

(function($)
{
      // tu wkleić kod wykorzystujący $ jako referencję do biblioteki2
})(biblioteka2);

(function($)
{
      // tu wkleić kod wykorzystujący $ jako referencję do biblioteki3
})(biblioteka3);
```

1.4.13 Warunki logiczne, operatory porównań

Kolejnym aspektem JS potrafiącym sprawić wiele problemów początkującym programistom, nieświadomym pewnych logicznych i spójnych mechanizmów języka są porównania.

Wydawać by się mogło, że w przypadku warunków logicznych wszystko powinno być bardzo proste i zrozumiałe. Nie do końca tak jest. Oto kilka przykładów, których wyniki mogą zdziwić:

```
if (2 == '2')
{
      alert('równe');
}
else
{
      alert('nierówne');
}
```

W wyniku działania takiego kodu otrzymamy informację zwrotną „równe”. Jako, że JS jest językiem skryptowym o dynamicznych typach danych [4] w powyższym przykładzie następuje niejawne rzutowanie do liczby całkowitej (wartości po lewej operatora porównania).

Wchodząc jednak trochę głębiej, czy '2' jest naprawdę wartością równą 2? Często chcemy prócz samej wartości być pewnymi, że także typ się zgadza:

```
if (2 == '2' && typeof '2' == typeof 2)
{
      alert('równe');
}
else
{
      alert('nierówne');
}
```

Taki warunek zwraca nam komunikat „nierówne”. Trzeba przyznać jednak, że powyższa konstrukcja nie jest zbyt czytelna i, co ma duże znaczenie w przypadku języka przesyłanego do klienta, jest długa. Istnieje szybszy sposób – operator identyczności `===' [75][98].

```
if (2 === '2')
{
      alert('identyczne');
}
else
{
      alert('nieidentyczne');
```

```
}
```

1.4.14 Falsy values

Kolejnym ciekawym zagadnieniem są wartości uznawane za „fałsz” w warunkach logicznych. Jest ich 6 [3]:

- " – pusta zmienna typu String
- 0 – typu Number
- `false' typu Boolean
- `undefined'
- `null'
- `NaN' typu Number

Powoduje to szereg problemów w trakcie pisania kodu. Problemy te powodują najgorsze z możliwych błędy – wszystko wygląda poprawnie ani nie występują żadne komunikaty błędów. Jedynym objawem pomyłki programisty jest zły wynik działania skryptu [19].

Poniższe porównania zwrócą prawdę:

```
"0" == 0
" " == 0
[] == 0
undefined == null
false == new String("")
```

Jednak takie konstrukcje zwracają już fałsz:

```
NaN == 0
null == 0
undefined == ""
0 == {}
```

Wykorzystując operator „identyczny” (ang. *strictly equal*) poniższe przypadki zwrócą prawdę.

```
0 === 0
"" === ""
[] !== []
```

Ostatni przykład może wydawać się ewidentnym błędem mechanizmów języka. Nic bardziej mylnego. Jest to w pełni celowe działanie, a co więcej – nawet logiczne. W specyfikacji ECMAScript możemy znaleźć: *13.Return true if x and y refer to the same object or if they refer to objects joined to each other (see 13.1.2). Otherwise, return false* [40].

Dla kontrastu operator identyczności dla takich przykładów z kolei zwróci fałsz:

```
"" === 0 // typ string i number
0 === null // typ number i object
[] === 0 // typ object i number
[] === " " // typ object i string
```

Innym problemem w porównaniach jest zaokrąglanie oparte o IEEE 754 [117]:

```
(0.1 + 0.2) === 0.3 // false, także dla operatora `=='
```

1.4.15 Co zwraca alternatywa w warunku logicznym?

JavaScript w przypadku warunków logicznych ma jeszcze jedną niespodziankę dla programistów „klasycznych" języków. W przypadku alternatywy nie musi zostać zwrócona wartość logiczna.

```
var a = false;
var b = true;
if (a || b)
{
      alert('a lub b ma ma wartość `true`');
}
```

W powyższym kodzie nie ma niczego, co mogłoby nas zdziwić. Zostanie wyświetlone „a lub b ma wartość `true`".

Można jednak dokonać drobnej modyfikacji, która jednak zmieni działanie skryptu a także zdziwi niewtajemniczonych.

```
var a = false;
var b = 'jestem tekstem';
if (a || b)
{
      alert('a lub b ma wartość `true`');
}
```

Zostanie wyświetlony komunikat „a lub b ma wartość `true`", choć w rzeczywistości wcale nie jest to prawdą. Zmienna `b' nie jest typu logicznego, a w warunku tak naprawdę nie otrzymano `true' tylko 'jestem tekstem', które po (niejawnym i automatycznym) zrzutowaniu do wartości logicznej (patrz rozdział o „falsy values") daje `true' – niepusty ciąg znaków rzutowany do wartości logicznej daje w wyniku `true'.

Mechanizm ten pokazuje dużo lepiej następujący przykład:

```
alert(false || true); // wyświetlone zostanie „true"
alert(null || 'to jest napis'); // wyświetli „to jest napis"
```

JavaScript w warunkach logicznych wykorzystujących alternatywę zwraca wartość pierwszego czynnika alternatywy, którego wartość po zrzutowaniu do wartości logicznej daje `true', a nie jak można by oczekiwać wprost `true'. Jest to bardzo przydatny mechanizm pozwalający bardzo skrócić kod.

Załóżmy, ze mamy funkcję, która może, ale nie musi przyjmować parametr:

```
function example( a )
{
      a = a || 'wartość domyślna';
      return a;
}
```

Jeśli do funkcji `example' została przekazana jakaś wartość, w ciele funkcji do zmiennej `a' zostanie przypisana właśnie ta wartość. Jeśli natomiast nic nie zostanie przekazane (w JS liczba parametrów przekazywanych do funkcji nie jest sprawdzana). Jeśli przekaże się ich mniej niż w deklaracji funkcji, brakującym parametrom zostanie przekazana wartość `undefined'. Jeśli więcej – pozostałe zostaną zignorowane, choć będą dostępne w obiekcie `arguments' [92]), automatycznie zostanie przypisana „wartość domyślna".

1.4.16 Wartość `undefined'

Własność `undefined' jest bardzo problematyczna. Do wersji 1.8.5 (przynajmniej w przeglądarce FireFox) języka jest to zmienna (sic!) [97], a więc można przypisać do niej dowolną wartość.

Przykładowo:

```
var a;
if (undefined == a)
{
      alert('`a` nie ma wartości');
}
```

Wyświetli „`a` nie ma wartości”. Jeśli jednak w dowolnym miejscu kodu, choćby dla testów lub przez przypadek, ktoś wstawi taki kod:

```
var a;
undefined = 1;
if (undefined == a)
{
      alert('`a` nie ma wartości');
}
else // a != 1
{
      alert('Rzeczywiście `a` nie jest `undefined`,' +
            ' bo `undefined` jest zdefiniowane na wartość 1');
}
```

Zdecydowanie jest to błąd języka, który ma zostać naprawiony w wersji 1.8.5 (dla Firefoksa), gdzie globalna zmienna `undefined' stanie się non-writable [97].

Zmienna `undefined' nie jest jedyną zmienną, którą można nadpisać w JavaScript powodując spore problemy. Podobnie jest z `Object', `Function', czy `Array'. Dlatego lepiej stosować `{}', `function' (zamiast `new Function(„stringCode”)'), `[]' [79].

Aby uniknąć problemów wyżej opisanych zdecydowanie lepiej używać operatora `typeof' [104]. W językach dynamicznych, gdzie wartość każdej zmiennej (a więc i funkcji) może zostać nadpisana, wydaje się bardzo dobrą praktyką gdzie tylko się da stosować operatory.

```
if  (typeof a === 'undefined' )
{
      alert('`a` nie jest ustawione');
}
```

1.4.17 Wartość prosta czy obiekt `Boolean'

Wartość logiczna (true/false) jest jedną z pięciu wartości prostych w JS (obok ciągu znaków, undefined, null, NaN). Istnieje jednak także obiekt `Boolean' [93]. I w tym wypadku nie obyło się bez problemów.

```
var bool = new Boolean(false);
if (bool)
{
      alert('prawda!');
}
```

Kod z powyższego listingu zwróci komunikat „prawda”. Jest to spowodowane faktem, że do warunku przekazany został obiekt, który zostanie zrzutowany. Każdy obiekt (nawet pusty `{}') zwraca prawdę. Nie inaczej dzieje się w tym wypadku.

Aby uniknąć takich błędów należy użyć;

```
var bool = new Boolean(false);
if (bool.valueOf)
{
      alert('prawda!');
```

```
}
```

Wartość logiczna obudowana w obiekt jest przechowywana w polu publicznym `valueOf'.

`Boolean' można także wykorzystywać bez operatora `new' w celu rzutowania zmiennej przekazanej jako argument na wartość logiczną. Dokumentacja MDC odradza [93] w tym wypadku stosowania operatora `new', wykorzystując jednak funkcję `Boolean' statycznie. Nie jest to jedyny przypadek problemów wywołanych operatorem `new'. Problemy te są na tyle poważne, że spora część programistów JS unika jak może technik wymagających tego operatora [85].

Istnieją także inne sposoby na rzutowanie typów za pomocą odpowiedniego wykorzystania operatorów logicznych i bitowych [20]. Nie dość, że są one bardziej zwięzłe w zapisie, to działają szybciej, a także (z racji braku możliwości przeciążania operatorów w JavaScript) są bezpieczniejsze. Ich wadą jest nieczytelność i konieczność doskonałej znajomości mechanizmów rządzących językiem.

1.5 Sukces mimo wszystko

Mimo wszystkich problemów związanych z JavaScript język ten odniósł niezwykły sukces, stając się głównym językiem przeglądarek i jedynym wspieranym natywnie przez zdecydowaną ich większość (w tym wszystkie główne).

Problemy z jakimi na co dzień zmagają się programiści: słaba wydajność języka [36], skomplikowana i często niezrozumiała składnia, mechanizmy podobne do Lisp czy Scheme ubrane w interfejs a'la Java wcale nie odstraszają od wykorzystywania tego „najbardziej niezrozumianego języka świata” (ang. *JavaScript: The World's Most Misunderstood Programming Language*)[23].

Co stoi za sukcesem JavaScript? Wydaje się, że z jednej strony prostota z jaka można zacząć w nim tworzyć. Każdy może stworzyć nowy plik tekstowy, wpisać w nim 10 linii kodu, który bez trudu znajdzie w internecie i uruchomić ten plik w przeglądarce. W przypadku C należy zainstalować kompilator (np. g++, a dla początkującego programisty wiersz poleceń jest „czarną magią”), w przypadku Javy czy C# już na początku mamy olbrzymie środowiska programistyczne, wielkie projekty, biblioteki. Wszystko to przytłacza osobę, która chciałaby po prostu sprawdzić jak działa jakiś mały fragment kodu. Z drugiej strony – wsparcie technologii przez producentów oprogramowania (przy np. braku wsparcia Apple dla Flasha na aplikacje mobilne [38]) oraz powszechność i zaufanie wśród użytkowników (95% użytkowników ma włączoną obsługę skryptów [74]) oraz mnogość dostępnych (bardzo często darmowych) bibliotek i rozszerzeń.

Warto zauważyć, że nawet w branży webmasterskiej nadal często nie traktuje się zbyt poważnie JavaScript oczekując w wymaganiach na stanowisko „młodszego programisty PHP/Javy/C#” bardzo dobrej znajomości JS. JavaScript wydaje się jednak z każdym dniem i każdym nowym projektem udowadniać swoją wartość, za czym idzie w parze coraz większa społeczność. Jeszcze kilka lat temu Douglas Crockford mówił w trakcie jednej ze swoich prezentacji [84], że JS, mimo że jest najczęściej wykorzystywanym językiem na świecie nie posiada swojej społeczności. To powoli się zmienia. Tylko w ostatnim roku w Polsce powstało kilka naprawdę ciekawych stron „Javascripterów”.

Przyszłość JavaScriptu rysuje się w znakomitych barwach, szczególnie dzięki najnowszym trendom realizowanym w HTML 5 [21][116], a także możliwości komunikowania się z serwerem – AJAX oraz coraz popularniejszych technikach DHTML, pozwalających tworzyć bardzo przyjazne użytkownikom i przyjemne dla oka aplikacje WWW – zwane także często Web 2.0. W rzeczywistości termin ten odnosi się raczej do sposobu generowania treści – przez internautów – niż do technik wykorzystywanych na stronach. Z pewnością jednak AJAX i DHTML miały w tej rewolucji spory udział. Warto tu też wspomnieć o innych zastosowaniach JavaScriptu, jak choćby

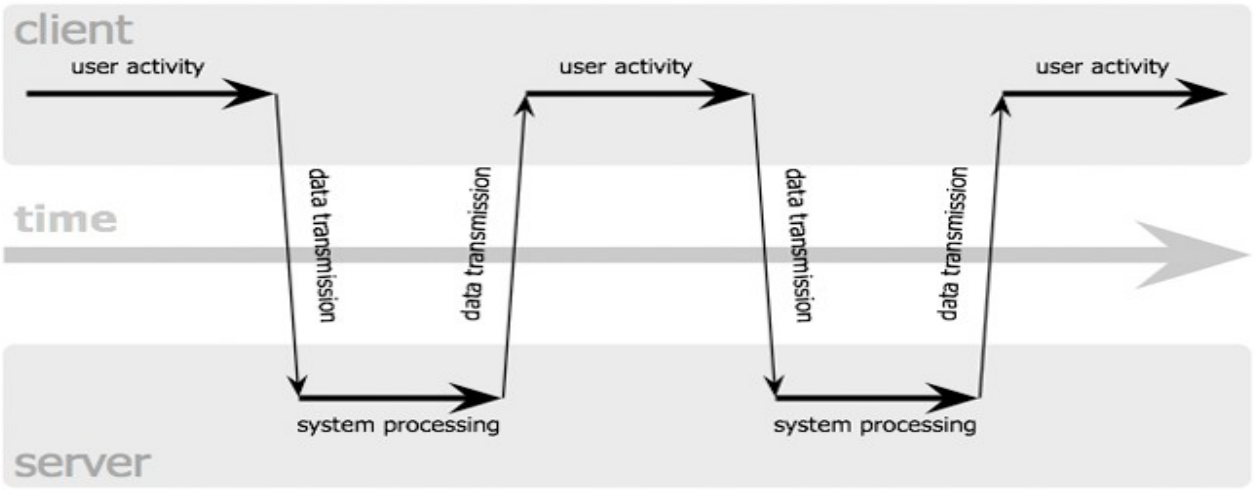
powszechne wykorzystanie w MongoDB (noSQL) jako język powłoki [45], jako język tworzenia rozszerzeń do przeglądarek z rodziny Mozilla Firefox [86], czy też w pogłoski o zastąpieniu niedalekiej przyszłości Elispa w edytorze Emacs [5][32].

2 AJAX i DHTML (Web 2.0)

2.1 Czym jest AJAX

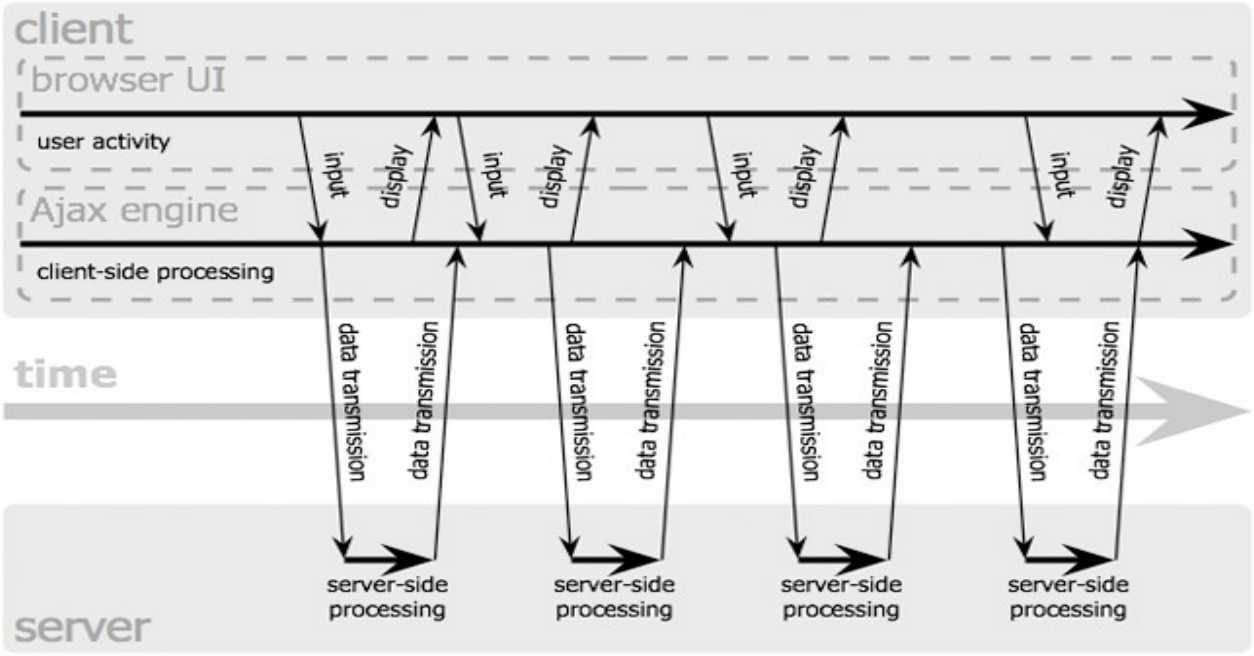
AJAX jest skrótem od „Asynchronous JavaScript and XML” (z ang. *Asynchroniczny JavaScript i XML*). Nazwa została pierwszy raz użyta na początku 2005 r. przez Jessiego Jamesa Garretta w słynnym artykule „Ajax: A New Approach to Web Applications” [37].

Mimo że często w odniesieniu do AJAX mówi się o nowej technologii, to jest to raczej bardzo ciekawa technika, wykorzystująca istniejące wcześniej technologie. AJAX polega na wysyłaniu żądań HTTP do serwera z poziomu JS, bez konieczności odświeżania strony. Tradycyjny model klient-serwer można zobrazować tak:



Rysunek 1. Tradycyjny model klient-serwer oparty o protokół HTTP (źródło: [37])

Wykorzystując AJAX (i operacje na DOM) działanie stron internetowych można zobrazować w taki sposób:



Rysunek 2. Sposób działania nowoczesnych stron WWW opartych o Ajax (źródło: [37])

Pierwszymi znanymi i bardzo popularnymi aplikacjami internetowymi w pełni opartymi o AJAX były Gmail oraz GoogleMaps.

Bardzo często pod skrótem AJAX mieści się zdecydowanie więcej niż wynikałoby z jego rozwinięcia. Nie zawsze trzeba wykorzystywać XML [69], zdecydowanie lżejszym formatem jest JSON [26]. Należałoby za to konsekwentnie stosować skrótu AJAJ (ang. *Asynchronous JavaScript and JSON*), a w przypadku czystego tekstu lub HTML należałoby mówić o AJAT (Text) bądź AJAH (HTML). Zbiorczo jednak wszystkie te rozwiązania określa się AJAX. Co więcej AJAX nie musi być także oparty o obiekt `XMLHttpRequest'. Pod tym skrótem często kryje się także całe bogactwo rozwiązań DHTML-owych (JavaScript, operacje na DOM, CSS), operujących stricte na obiektowym modelu dokumentu i nie mającym niczego wspólnego z wymianą danych z serwerem.

Chcąc nie chcąc „AJAX" stał się określeniem nowoczesnych stron WWW. Właściwie nie tyle stron, co już aplikacji webowych. Pod takimi adresami jak facebook.com, gmail.com, google.com nie znajdziemy stron opartych o statyczne pliki, HTML i synchronicznie odpowiadające kolejnymi podstronami na działania internauty. Dzisiejsze aplikacje webowe są dużo bardziej interaktywne, sprawią wrażenie inteligentnych, posiadają wiele mechanizmów bardzo ułatwiających poruszanie się po nich. To wszystko zasługa Ajaksa. Pisownia wielką literą, jak nazwy własnej, nie przypadkowa. Z powodów wymienionych powyżej „Ajax" jest po prostu nowym słowem określającym pewien zbiór rozwiązań technicznych pozwalających na dzisiejszą rewolucję internetową, a nie jak na początku skrótem od pierwszych liter technologii go tworzących.

2.2 XMLHttpRequest

Choć istnieją alternatywne techniki (w oparciu o pływającą ramkę [80][118], ciasteczka [81][118] czy JSONP [24][82], to główną techniką komunikacji z serwerem skryptów JavaScript jest ta wykorzystująca obiekt `XMLHttpRequest' [70]. Pozwala on na stworzenie odpowiedniego żądania HTTP i wysłanie go na serwer, a następnie odebranie i wyniku bez konieczności

przeładowania całej strony.

2.2.1 Interfejs

Obiekt `XMLHttpRequest' posiada interfejs [71]:

```
interface XMLHttpRequest : XMLHttpRequestEventTarget {
  attribute Function onreadystatechange;
  // states
  const unsigned short UNSENT = 0;
  const unsigned short OPENED = 1;
  const unsigned short HEADERS_RECEIVED = 2;
  const unsigned short LOADING = 3;
  const unsigned short DONE = 4;
  readonly attribute unsigned short readyState;

  void open(DOMString method, DOMString url);
  void open(DOMString method, DOMString url, boolean async);
  void open(DOMString method, DOMString url, boolean async,
                                                  DOMString? user);
  void open(DOMString method, DOMString url, boolean async,
                                  DOMString? user, DOMString? password);
  void setRequestHeader(DOMString header, DOMString value);
  void send();
  void send(Document data);
  void send([AllowAny] DOMString? data);
  void abort();

  readonly attribute unsigned short status;
  readonly attribute DOMString statusText;
  DOMString getResponseHeader(DOMString header);
  DOMString getAllResponseHeaders();
  readonly attribute DOMString responseText;
  readonly attribute Document responseXML;
};
```

2.2.2 Żądania synchroniczne

Przykładowy kod wykorzystujący Ajax wygląda w taki sposób:

```
<html>
<head>
      <title>Przykład Ajax</title>
      <meta http-equiv='content-type'
            content='text/html;charset=utf-8'/>
      <script type='text/javascript'>
      function testAjax()
      {
            var ajaxClient = new XMLHttpRequest(); // 1
            ajaxClient.open('POST', 'ajax.php', false); // 2
            ajaxClient.send(null); // 3
            alert(ajaxClient.responseText); // 4
      }
      </script>
</head>
<body>
      <a href='javascript:testAjax()'>Testuj</a>
</body>
</html>
```

Kod powyższego listingu wykonuje kolejno:

1. Tworzy obiekt `XMLHttpRequest'. Jest to obiekt zaimplementowany w każdej nowoczesnej przeglądarce. Wersje IE wcześniejsze niż 7 nie posiadały tego obiektu. Zamiast tego należało korzystać z odpowiedniej kontrolki ActiveX - `Microsft.XMLHTTP' (lub jej nowszych wersji)[6].
2. Tworzy żądanie HTTP o odpowiednich wartościach – metoda POST [51], do pliku 'ajax.php'. Adres URL [120] musi odwoływać się do zasobów w tej samej domenie, choć stosując odpowiednie sztuczki (JSONP [26], odpowiedni nagłówek po stronie serwera – Access-Control-Allow-Origin [119]) można pobierać dane ze z innej domeny. Trzeci atrybut odnosi się do sposobu w jaki ma zostać obsłużone to żądanie – synchronicznie (jeśli podany zostanie `false') lub asynchronicznie (jeśli argument ten nie zostanie ustawiony albo przekazana zostanie wartość `true') .
3. Wysłanie żądania. Przekazany parametr `null' oznacza, że nie przekazujemy w tym żądaniu żadnych parametrów do serwera. W przypadku użycia metody GET [50] (linia 2) zawsze przekazujemy tu `null', a wartość parametrów przekazujemy w URL, np. '`ajax.php?parametr1=wartość&p=1`'. Dla metody POST jest to miejsce, w którym można przesłać dane na serwer, także w formacie: „`nazwaZmiennej=wartość&zmienna=wartość`”.
4. Wyświetlenie tekstu, jaki został zwrócony jako odpowiedź na żądanie (request → response) przez serwer.

Przedstawiony kod wymaga jeszcze pliku 'ajax.php' po stronie serwera (nie musi być to plik php, może być xml, txt, lub dowolny dostępny z poziomu klienta zasób na macierzystym serwerze). Załóżmy, że kod w 'ajax.php' wygląda w ten sposób:

```
<?php echo 'Witaj świecie!'; ?>
```

Mając już taki zestaw plików (strona www oraz plik, do którego odwołuje się ajaksowy request) można wykonać test. Jeśli wszystko zadziała poprawnie powinien pojawić się komunikat „Witaj świecie!”.

W linii numer 3 kodu funkcji `testAjax' cała strona zostanie wstrzymana do czasu odebrania odpowiedzi z serwera. Jest to wynik żądania synchronicznego (3-ci parametr metody `XMLHttpRequest.open').

2.2.3 Żądania asynchroniczne

Istnieje również sposób na wywołanie żądań asynchronicznie. Jest to metoda dużo częściej spotykana i przyjemniejsza dla użytkownika – nie wstrzymuje reszty strony. W czasie takiego żądania można np. wykonać jakieś operacje na interfejsie użytkownika informując go o aktualnym działaniu.

Aby wykorzystać mechanizm żądań asynchronicznych należy wykonać taki kod:

```
<html>
<head>
      <title>Przykład Ajax</title>
      <meta http-equiv='content-type'
            content='text/html; charset=utf-8' />
      <script type='text/javascript'>
      function testAjax()
      {
            var ajaxClient = new XMLHttpRequest();
            ajaxClient.open('POST', 'ajax.php', true); // lub bez true
            ajaxClient.onreadystatechange = function()
```

```
            {
                  if (4 === this.readyState && 200 === this.status)
                  {
                        alert(this.responseText);
                  }
            }
            ajaxClient.send(null); // 3
      }
      </script>
</head>
<body>
      <a href='javascript:testAjax()'>Testuj</a>
</body>
</html>
```

Dzięki takim zmianom użytkownik nie będzie nawet świadomy, że cokolwiek zostało wysłane na serwer. W większości przypadków fakt łączenia się z serwerem jest dla użytkownika obojętny. Podstawowa zmiana to podane funkcji obsługi zdarzenia `readystatechange'. Funkcja ta zostanie wywołana kilkukrotnie – zawsze kiedy pole `ajaxClient.readyState' ulegnie zmianie. Pole to może przyjmować 5 wartości (patrz interfejs XMLHttpRequest - „states”).

Kolejną „magiczną liczbą” jest wartość „200” pola `XMLHttpRequest.status'. Jest to wartość odpowiadająca tekstowemu statusowi „OK” (dostępnemu w polu `XMLHttpRequest.statusText'). Pole `status' przyjmować może także inne wartości, np. „404” – gdy podany URL odnosi się do zasobu, który nie istnieje na serwerze lub „500” – w przypadku błędu serwera. Są to kody odpowiedzi HTTP [11].

2.2.4 Wysyłanie danych na serwer metodą POST

Aby przesłać dane na serwer, według tego co napisałem wcześniej należałoby wykonać:

```
function testAjax()
{
      var ajaxClient = new XMLHttpRequest();
      ajaxClient.open('POST', 'ajax.php', true);
      ajaxClient.onreadystatechange = function()
      {
            if (4 === this.readyState && 200 === this.status)
            {
                  alert(this.responseText);
            }
      }
      ajaxClient.setRequestHeader('Content-Type',
            'application/x-www-form-urlencoded; charset=UTF-8'); // 1
      ajaxClient.send('a=12&b=12'); // 2
}
```

Po stronie serwera do taki przesłanych danych korzystamy z metod udostępnianych przez używaną przez nas technologię do odbioru danych tradycyjnie przesłanych na serwer, np. za pomocą formularza. Aby móc przesłać dane przy pomocy metody POST musimy także ustawić odpowiedni nagłówek (linia oznaczona „1”). Dane przesyłamy jako ciąg znaków przekazany jako parametr do metody `send' (linia „2”).

2.2.5 Wysyłanie danych na serwer metodą GET

W przypadku wykorzystania metody GET w kodzie zaszły by pewne zmiany widoczne na kolejnym listingu:

```
function testAjax()
{
	var ajaxClient = new XMLHttpRequest();
	ajaxClient.open('GET', 'ajax.php?a=1&b=2'); // 1

	ajaxClient.onreadystatechange = function()
	{
		if (4 === this.readyState && 200 === this.status)
		{
			alert(this.responseText);
		}
	}
	ajaxClient.send(null); // 2
}
```

Pierwszą zmianą (linia oznaczona „1”) jest przekazanie parametrów w URL. Jak widać, brak w tej linii trzeciego parametru metody `open'. Wykorzystana zatem będzie wartość domyślna – żądanie będzie wykonane asynchronicznie. Do metody `send' (linia „2”) przekazujemy null. Po stronie serwera także dane te odbieramy w sposób właściwy dla odbioru danych przesłanych metodą GET.

Ograniczenia metody GET

Jeśli decydujemy się na stosowanie tej metody należy zawsze pamiętać o ograniczeniach, choćby limicie znaków możliwych do przesłania w adresie. [35][47][48].

2.3 XML

W poprzednich przykładach dane zwracane przez serwer nie miały żadnej struktury – był to czysty tekst. Jest to często stosowana praktyka, gdy chcemy przesłać gotowy komunikat. Sposób ten jest także często używany do przesyłania gotowego kodu HTML – fragmentu strony, który później jest wstrzykiwany w odpowiednie miejsce w strukturę DOM. Zdarza się jednak, i to często, że chcielibyśmy przekazać trochę bardziej skomplikowane dane. W takiej sytuacji jednym z dostępnych i powszechnie obsługiwanych formatów jest XML. Format ten (patrząc na nazwę) wydaje się wręcz wymaganym w Ajax, w którym przecież „X” w nazwie pochodzi od tej technologii.

2.3.1 Składnia XML

XML jest bardzo czytelnym dla człowieka i prostym w nauce formatem. Nazwa pochodzi od „E**x**tensible **M**arkup **L**anguage” - rozszerzalny język znaczników. Służy do opisywania zawartości dokumentów elektronicznych. Jego olbrzymią zaletą jest fakt, że nawet osoba niemająca na co dzień styczności z programowaniem, powinna w miarę szybko zrozumieć informacje nim zawarte. Podstawowymi składnikami dokumentu XML są: elementy oraz atrybuty [7].

Prostota XML pozwala opisać zasady jego tworzenia w niewielkim zbiorze punktów [7]:

1. Podstawowym budulcem jest element (tworzący znaczniki/tagi):

   ```
   <nazwa-elementu>treść</nazwa-elementu>
   ```

2. Element może zawierać: kolejny element, treść, atrybut lub ich kombinację.
   - Treść poprzedzona jest znacznikiem początkowym

     ```
     <nazwa-elementu>
     ```

- Za treścią występuje znacznik końcowy

  `</nazwa-elementu>`

3. Element może być pusty (nie zawierać treści, innych elementów, ani nie posiadać atrybutów). Posiada wtedy składnię:

   `<nazwa-elementu></nazwa-elementu>`

   albo:

   `<nazwa-elementu/>`

4. Jeżeli element nie jest pusty i nie jest zamknięty, to element ten jest nieprawidłowy. Co za tym idzie cały dokument XML jest niepoprawny.
5. W nazwach elementów małe i duże litery są rozróżnialne, znacznik `<a>` jest czym innym niż znacznik `<A>`.
6. Kolejność zamykania znaczników (elementów) jest odwrotna do kolejności otwierania:

   `<a><b><c>treść</c></b></a>`

7. Zasady tworzenia nazw elementów [78]:
   - nazwy elementów mogą zawierać litery, cyfry i inne znaki
   - nazwy elementów nie mogą zaczynać się od cyfry, myślnika ani kropki
   - nazwy nie mogą rozpoczynać się od „xml” („XML”, „xmL” itp.)
   - nazwy nie mogą zawierać spacji
8. Dodatkowe informacje o elemencie mogą zostać przekazane za pomocą atrybutów.
9. Atrybuty umieszcza się w znacznikach początkowych lub w znaczniku elementu pustego.
10. Atrybuty mają składnię:

    `<element atrybut='wartość' />`

11. Wartość atrybutu musi być otoczona przez cudzysłów albo apostrofy. Każdy atrybut musi mieć przypisaną wartość (choćby pusty ciąg znaków `atrybut=''`).
12. Ograniczenia nałożone na nazwy atrybutów są takie same jak przy nazwach elementów.

Być może wydaje się, że istnieje sporo zasad rządzących tym formatem. Sądzę jednak, że wystarczy prosty przykład kodu XML, aby przekonać się, iż wszystko jest logiczne i składne:

```
<ludzie>
<pracownicy>
      <kierownik dzial='finanse'>
            <imie>Stefan</imie>
            <nazwisko>Kowalski</nazwisko>
      </kierownik>
      <kierownik dzial='hr' />
      <programista dzial='startup'>
            <imie>Jan</imie>
            <nazwisko>Nowak</nazwisko>
      </programista>
</pracownicy>
<klienci>
      <klient>
            <nazwa>Firma S.A</nazwa>
      </klient>
<klienci>
```

```
</ludzie>
```

Jak widać w powyższym kodzie posiadamy strukturę drzewiastą z dwoma gałęziami. W gałęzi `pracownicy' nie posiadamy informacji o kierowniku działu „hr” - jest to znacznik pusty.

2.3.2 Wykorzystanie XML w Ajax

W tym przykładzie skorzystam z kodu XML z poprzedniego listingu, umieszczając go w pliku 'ludzie.xml'.

```
function testAjax()
{
	var ajaxClient = new XMLHttpRequest();
	ajaxClient.open('POST', 'ludzie.xml');
	ajaxClient.setRequestHeader("Content-Type", "text/xml");
	ajaxClient.onreadystatechange = function()
	{
		if (4 === this.readyState && 200 === this.status)
		{
			alert(this.responseXML.
				getElementsByTagName('kierownik')[0].
				getElementsByTagName('imie')[0].
				firstChild.nodeValue);
		}
	}
	ajaxClient.send(null);
}
```

Jak widać na powyższym kodzie nastąpiło sporo zmian w porównaniu z listingami oczekującymi czystego tekstu. Po pierwsze, wysyłamy dodatkowy nagłówek – 'Content-type' [71], który informuje jakiego typu danych oczekujemy w odpowiedzi. Po drugie zamiast `XMLHttpRequest.responseText' wykorzystujemy pole `XMLHttpRequest.responseXML' i co się z tym wiąże musimy także operować na interfejsie modelu DOM [60] zwróconego obiektu XML. Operowanie natywnie na modelu DOM nie jest ani intuicyjne, ani wygodne [2]. Zmiany w strukturze DOM strony są jedną z najczęstszych funkcjonalności implementowanych we frameworkach. Świadczyć do może z jednej strony o olbrzymiej potrzebie operowania na modelu DOM, jak z drugiej o stopniu skomplikowania tych zabiegów. Dawniej bardzo popularnym akronimem dla działań na strukturze DOM i formacie strony (CSS) był DHTML, od „Dynamiczny HTML”. Był to tak naprawdę zlepek JS, HTML i CSS. Dzisiaj skrót ten został już prawie zapomniany. Często na określenie działań z zakresu DHTML używa się (nie do końca poprawnie) określenia „Ajax”, a do określenia nowoczesnych stron WWW – Web 2.0 (choć też nie jest to do końca słuszny termin jeśli chcemy określić aspekt technologiczny).

Już dzisiaj istnieją gotowe biblioteki znacznie ułatwiające operowanie na danych w XML, choćby ObjTree [44]. W najnowszych wersjach ECMAScript jest także mowa o zdecydowanym ułatwieniu operowania na danych za pomocą rozszerzenia E4X (ECMAScript for XML [41][90]) dostarczającego natywnego wsparcia XML w JavaScript. Póki co jednak zaimplementowany jest jedynie w SpiderMonkey [108] (silniku JavaScript Gecko [91]) oraz w Rhino [107] i nadal nie jest wolny od błędów utrudniających pracę programisty. Z pewnością jednak potencjał drzemiący w tym rozszerzeniu ECMAScript jest duży i można się spodziewać, iż jego pełna implementacja zostanie bardzo dobrze przyjęta przez developerów.

2.4 JSON

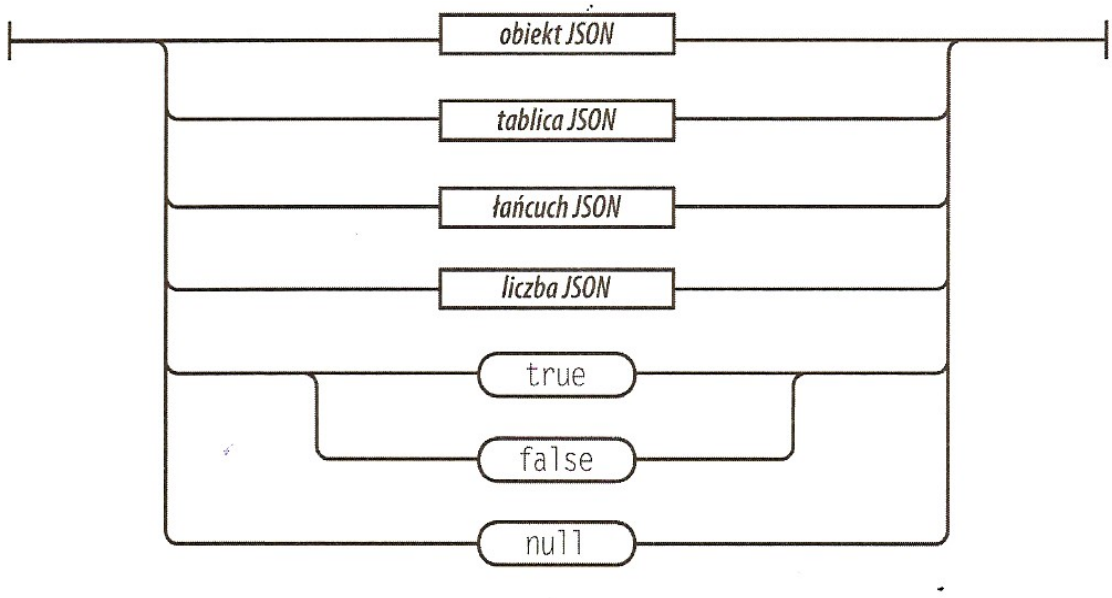
Choć w nazwie Ajaksa jawnie występuje XML, to coraz częściej (i w opinii autora słusznie)

formatem wymiany danych staje się JSON [26] – **J**ava**S**cript **O**bject **N**otation (wym. jak angielskie imię „Jason”). Format ten jest podzbiorem języka JavaScript. Oznacza to, że każdy kod JSON musi być poprawnym kodem JavaScript. JSON oparty jest na opisanych wcześniej literałach obiektowych JavaScript. Może być używany do wymiany danych między dowolnymi nowoczesnymi językami programowania. Jest wpierany przez bardzo wiele języków [26], w tym: JavaScript, PHP, Java, C, C++, C#, Python, Pearl.

2.4.1 Składnia JSON

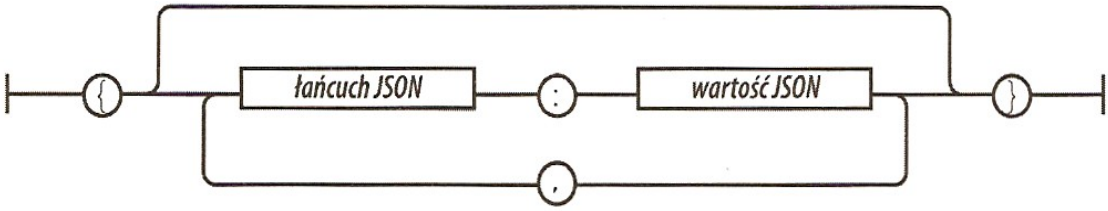
Obiekt w formacie JSON jest nieuporządkowanym zbiorem "klucz" : wartość. Klucz może być dowolnym łańcuchem. Wartość musi być jedną z dozwolonych wartości JSON [26]:

- zagnieżdżony obiekt JSON,
- tablica
- łańcuch znaków
- liczba
- wartość logiczna
- wartość null



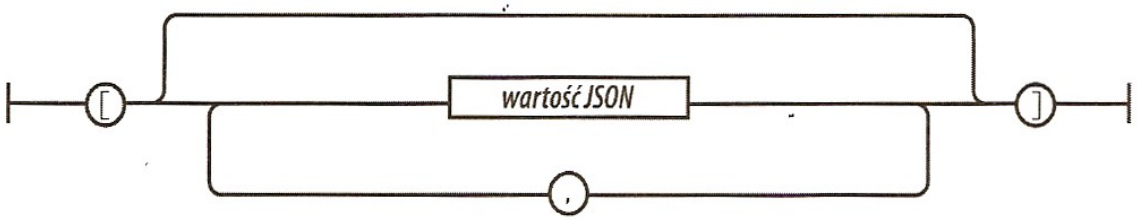
Rysunek 3. Wartość JSON (źródło: [2])

Strukturę obiektu JSON bardzo czytelnie przedstawia diagram składni:



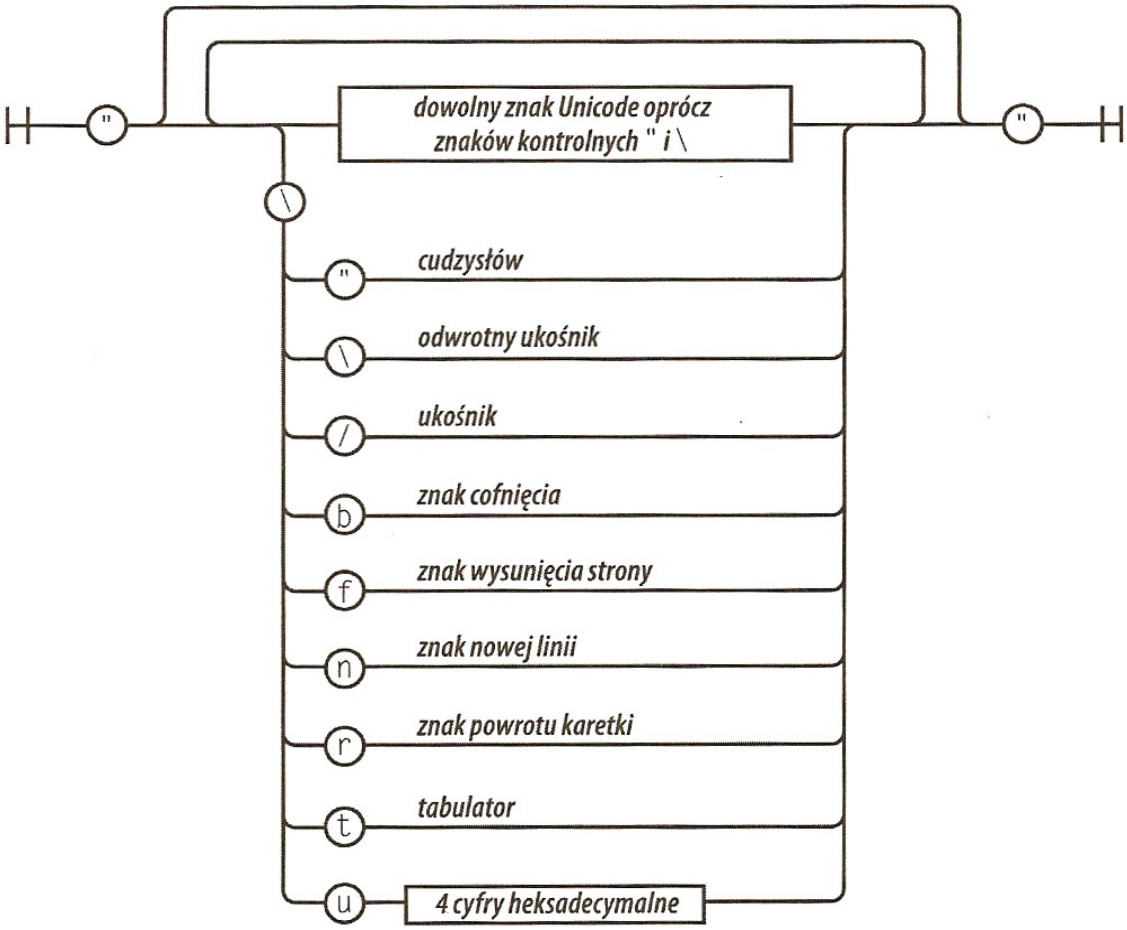
Rysunek 4. Obiekt JSON (źródło: [2])

Tablica w formacie JSON, jest uporządkowanym ciągiem wartości. Tablica (jak to w JavaScript) jest typem. Wartości poszczególnych elementów tablicy mogą mieć różne typy, wliczając w to kolejne tablice, obiekty lub typy proste.



Rysunek 5. Tablica JSON (źródło: [2])

Łańcuchem w formacie JSON jest dowolny ciąg znaków objętych w cudzysłów albo apostrofy.



Rysunek 6. Ciąg znaków JSON (źródło: [2])

Zasady rządzące JSON są celowo bardzo proste. Chodzi o to, aby nie stwarzać przestrzeni do własnej interpretacji. Sądzę, że wiele może wyjaśnić poniższy przykład kodu w formacie JSON.

Pokażę kod, przekazujący dokładnie te same dane, co te zawarte we wcześniej stworzonym pliku 'ludzie.xml'. Plik 'ludzie.json' posiadać będzie taką strukturę:

```
{
	"pracownicy" :
	[
		{
			"kierownik" :
			{
				"dzial" : "finanse",
				"imie" : "Stefan",
				"nazwisko" : "Kowalski"
			}

		},
		{
			"kierownik":
			{
				"dzial" : "hr"
			}
		},
		{
			"programista" :
			{
				"dzial" : "startup',
				"imie"  : "Jan",
				"nazwisko" : "Nowak"
			}
		},
	],
	"klienci" :
	[
		{
			"klient" :
			{
				"nazwa" : "Firma S.A."
			}
		}
	]
}
```

2.4.2 Wykorzystanie JSON w Ajax

Dla człowieka JSON jest mniej przyjazny niż XML. Nie jest on jednak przeznaczony dla oka człowieka tylko dla komputera, który bardzo dobrze radzi sobie z jego parsowaniem. Aby móc w swoim kodzie użyć danych zawartych w JSON należy go najpierw sparsować. Istnieje kilka metod, wliczając w to specjalnie napisane parsery. Najprostszą jest wykorzystanie funkcji `eval' [110], co pokazuje poniższy przykład. W bardziej zaawansowanych rozwiązaniach autor sugeruje wykorzystanie jednego z istniejących parserów, w celu zapewnienia większego bezpieczeństwa swoim skryptom.

```
function testAjax()
{
	var ajaxClient = new XMLHttpRequest();
	ajaxClient.open('POST', 'ludzie.json');

	ajaxClient.onreadystatechange = function()
	{
		if (4 === this.readyState && 200 === this.status)
```

```
            {
                  var oJson = eval( '(' + this.responseText + ')' );//1
                  alert(oJson.pracownicy[0].kierownik.imie); //2
            }
      }
      ajaxClient.send(null);
}
```

Warto zauważyć, że korzystając z JSON jako formatu wymiany danych wykorzystujemy `responseText', który to następnie parsujemy (linia „1”). Po takim zabiegu (jeśli ciąg znaków miał prawidłową strukturę) możemy korzystać z obiektu `oJson' w sposób dużo przyjaźniejszy niż stosując XML. Korzystając z JSONP („JSON with Padding” - wzorzec użycia JSON [24]) można także odczytywać dane z serwera w innej domenie [82].

Prócz pokazanej wyżej metody, uznawanej za niebezpieczną, istnieją także inne, jak choćby parser JSON autorstwa D. Crockforda [25] lub też w najnowszych przeglądarkach natywne wsparcie, czy to w Mozilla FireFox [106] czy w Windows Internet Explorer [28].

2.5 DOM i DHTML

DOM to niezależny od języka interfejs API. DOM posiada strukturę drzewiastą i jak wcześniej wspomniałem jest podzbiorem BOM [1]. Strona internetowa stworzona w HTML (lub XHTML) składa się z węzłów. Każdy znacznik jest węzłem, który może posiadać węzły potomne lub atrybuty. Interfejs DOM dostępny w JavaScript pozwala na dostęp do tych węzłów i umożliwia ich dynamiczną zmianę. Dzięki temu API webdeveloper może dowolnie wpływać zarówno na strukturę dokumentu, jak i jego wygląd (poprzez zmianę stylów CSS). Techniki umożliwiające te zmiany zwykło początkowo się nazywać DHTML (dynamiczny HTML). Jednak po pojawieniu się Ajaksa, spora część „ajaksowych” rozwiązań to tak naprawdę DHTML, bez żadnego asynchronicznego łączenia się z serwerem.

Model DOM doczekał się także odpowiedniej rekomendacji W3C [73]. W przykładach z poprzednich rozdziałów wykorzystywałem już model DOM, choćby w przypadku metody `document.getElementById'. Standard DOM podzielono na 3 poziomy: pierwszy [68], drugi [57] oraz trzeci [67]. Istnieje także pojęcie DOM Level 0 [77] określające dostępną funkcjonalność operacji na dokumencie i jego elementach w pierwszych przeglądarkach obsługujących JavaScript.

2.5.1 Korzystanie z modelu DOM

Choć obiekt `document' jest uważany za część modelu BOM, stanowi także reprezentację obiektu `HTMLDocument' w modelu HTML DOM, który jest z kolei obiektem `Document' modelu XML DOM. W większości manipulacji obiektem DOM korzysta się z obiektu `document' [1].

Aby uzyskać dostęp do znacznika elementu <html /> użyć można kodu:

```
var oHTML1 = document.documentElement;
var oHTML2 = document.getElementsByTagName('html')[0];
alert(oHTML1 === oHTML2);
```

W wyniku otrzymamy `true', co oznacza, że obie metody zwracają ten sam węzeł DOM. Z tych dwóch metod zdecydowanie częściej używa się `getElementsByTagName' [89]. Wyżej opisana metoda zwraca listę węzłów. Można z niej wybrać konkretny, interesujący nas węzeł, za pomocą odpowiedniego indeksu (według specyfikacji w3c węzły numerowane są zgodnie z kolejnością występowania w drzewie DOM [58] – drzewo trawersowane jest zgodnie z algorytmem przeszukiwania w głąb (ang. *depth-first traversal* lub *preorder traversal*) [8]). Zasadę działania tej metody i indeksacji dobrze prezentuje poniższy przykład:

```
<html>
<head>
    <title>Przykład getElementsByTagName</title>
    <meta http-equiv="content-type"
            content="text/html;charset=utf-8" />
    <script type="text/javascript">
      window.onload = function()
      {
            var divs = document.getElementsByTagName('div');
            for(var i = 0; i < divs.length; i++)
            {
                  alert(divs[i].innerHTML);
            }
      }
    </script>
</head>
<body>
    <div>1
            <div>2</div>
    </div>
    <div>3</div>
</body></html>
```

W wyniku odpalenia takiego skryptu wyświetlone zostaną kolejno „1 (...)”, „2”, „3” (prócz „1” zostanie też wyświetlony kod okalający „2”). W powyższym przykładzie kod JS zostanie odpalony w funkcji obsługującej zdarzenie (więcej o samych zdarzeniach DOM w dalszej części pracy) załadowania strony. By mieć pewność, że nasz kod, który próbuje dostać się do struktury DOM zadziała poprawnie, zawsze należy uruchamiać go po wczytaniu i sparsowaniu dokumentu.

Istnieje także inna metoda, pozwalająca pobrać jeden konkretny węzeł. Jest to metoda `getElementById' [113]. Jest to jedna z podstawowych metod wykorzystywanych przy manipulacji strukturą DOM dokumentu. Jeśli nie istnieje węzeł o zadanym id, zwracany jest null.

```
<html>
<head>
    <title>Przykład getElementById</title>
    <meta http-equiv="content-type"
            content="text/html;charset=utf-8" />
    <script type="text/javascript">
      window.onload = function()
      {
            alert(document.getElementById('exists').innerHTML); // 1
            alert(document.getElementById('not-exists').innerHTML); // 2
      }
    </script>
</head>
<body>
    <div id="exists">Istnieje</div>
</body></html>
```

W pierwszym komunikacie pojawi się napis „istnieje”. Druga linia wywoła błąd „*document.getElementById("not-exists") is null*”

2.5.2 Trawersowanie drzewa DOM

Znanym zagadnieniem algorytmicznym jest trawersowanie grafów. Struktura DOM, będąca drzewem (grafem o specjalnych właściwościach), także pozwala na odpowiednie przemieszczanie się po nim, węzeł po węźle. Model DOM Level 2 posiada zaimplementowane dwa obiekty pozwalające na trawersowanie drzewa dokumentu, są to: NodeIterator [64] oraz TreeWalker [66].

NodeIterator

Wykorzystując obiekt `NodeIterator' można rozpocząć od elementu `<html/>` obiektu `document' i przejść całe drzewo algorytmem przeszukiwania w głąb. Do utworzenia obiektu `NodeIterator' służy metoda `document.createNodeIterator', przyjmująca 4 parametry:

1. `root' jest węzłem DOM, od którego zacząć wyszukiwanie.
2. `whatToShow' jest kodem numerycznym wskazującym, które węzły należy odwiedzić (np. NodeFilter.SHOW_ALL, NodeFilter.SHOW_ELEMENT, pełną listę znaleźć można w dokumentacji [65]). Poszczególne wartości mogą być łączone za pomocą operacji bitowych.
3. `filter' – obiekt NodeFilter służący do określenia węzłów, które mają być ignorowane.
4. `entityReferenceExpansion` – wartość logiczna wskazująca, czy odnośniki do encji powinny być rozwijane.

Prosty przykład wykorzystujący `NodeIterator':

```
<html>
<head>
	<title>Przykład trawersowania DOM za pomocą NodeIterator</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
window.onload = function()
{
	var oIter = document.createNodeIterator(document,
				NodeFilter.SHOW_ELEMENT, null, false);
	for(var node = oIter.nextNode(); node; node = oIter.nextNode())
	{
		alert(node.tagName);
	}
}
	</script>
</head>
<body>
	<ul>
		<li>aaaa</li>
		<li>bbb</li>
	</ul>
	<div>cccc</div>
</body>
</html>
```

W wyniku działania powyższego kodu uzyskamy komunikaty z nazwami kolejnych trawersowanych węzłów DOM. Do funkcji `createNodeIterator' można przekazać jako 3-ci parametr filter. Jest to obiekt posiadający metodę `acceptNode', przykładem takiej metody mogłaby być:

```
var oFilter = {
	acceptNode : function(node)
	{
		return ('div' === node.tagName.toLowerCase())
			? NodeFilter.FILTER_REJECT
			:  NodeFilter.FILTER_ACCEPT;
	}
};
```

Przekazanie zmiennej `oFilter' w miejsce null sprawi, że nie zostanie wyświetlony żaden węzeł div.

Jak widać, po raz kolejny sprawdza się tu duck typing – „typ” określany przez interfejs, a nie faktyczny typ obiektu.

TreeWalker

Obiekt `TreeWalker' posiada wszystkie metody dostępne w `NodeIterator' z dodatkowymi metodami przemierzania drzewa:

- parentNode() - przejście do przodka
- firstChild() - pierwszy potomek bieżącego węzła
- lastChild() - ostatni potomek bieżącego węzła
- nextSibling() - przechodzi do następnego krewnego bieżącego węzła
- previousSibling() - przechodzi do poprzedniego krewnego bieżącego węzła

Do tworzenia obiektu `TreeWalker' służy metoda `document.createTreeWalker'. Do metody tej przekazuje się 4 parametry – takie same jak do funkcji `createIteratorNode'.

Interfejs węzła DOM

Prócz opisanym sposobów przemierzania drzewa DOM można także korzystać z zaimplementowanych właściwości (nie metod, nawet gdy pokrywają się nazwą z metodami z poprzednich przykładów) każdego węzła: `firstElementChild', `lastElementChild', `previousElementSibling', `nextElementSibling', `childElementCount', `children', `nextSibling', `firstChild', `nodeValue', `childNodes', `previousSibling', `nodeType', `lastChild', `ownerDocument', `parentNode', `attributes'. Interfejs ten także pozwala na trawersowanie drzewa dokumentu. Wymaga to jednak implementacji przez programistę odpowiedniego algorytmu trawersowania drzewa.

2.5.3 Manipulowanie strukturą drzewa DOM

DOM API dostarcza nie tylko interfejsu dla przechodzenia drzewa dokumentu. Wykorzystując odpowiednie metody można dynamicznie wpływać na strukturę DOM dodając, usuwając lub zmieniając węzły. Aby dodać dodatkowy element do struktury DOM, należy go najpierw stworzyć. Służy do tego metoda `createElement' [111].

```
var oElement = document.createElement('div');
```

Powyższy kod powoduje, że w zmiennej `oElement' znajduje się obiekt węzła DOM. Póki co taki element nie posiada żadnej zawartości, ani atrybutów. Odpowiada to takiemu kodowi XML:

```
<div/>
```

Aby dodać do tego elementu treść należy użyć metody `createTextNode':

```
var oText = document.createTextNode('Witaj świecie');
```

W tym momencie posiadamy dwa obiekty – węzły DOM. Aby je połączyć ze sobą (i wstawić na stronę) należy użyć jednej z metod:

- `elementParent.appendChild(element)' – dołącza węzeł jako element-dziecko [53]
- `elementParent.replaceChild(newEl, OldEl)' – podmienia węzeł [54]
- `elementParent.insertBefore(newEl, OldEl)' – wstawia węzeł przed [55]

```
<html>
<head>
	<title>Przykład tworzenia nowych węzłów</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	window.onload = function()
	{
		var oElement = document.createElement('div');
		var oText = document. createTextNode('Witaj świecie');
		oElement.appendChild(oText);
		// dodaje element do struktury dokumentu
		document.body.appendChild(oElement);
		var oText2 = document. createTextNode('Witaj nowy świecie');
		// podmienia węzeł tekstowy
		oElement.replaceChild(oText2, oText);
		var oElement2 = document.createElement('div');
		oElement2.appendChild(oText);
		document.body.insertBefore(oElement2, oElement);
	}
	</script>
</head>
<body></body>
</html>
```

Powyższy kod stworzy kod HTML z poniższego listingu:

```
<body>
	<div>Witaj świecie</div>
	<div>Witaj nowy świecie</div>
</body>
```

Do usuwania elementów z dokumentu służy metoda `elementParent.removeChild(elementDel)' [52].

Prócz skomplikowanego interfejsu prezentowanego wyżej (opartego o DOM Level 2), istnieje także prosty i czytelny nawet dla początkującego programisty sposób na zmianę zawartości strony. Jest to niestandardowa właściwość `innerHTML'. Powyższy przykład można przepisać w taki sposób:

```
<html>
<head>
	<title>Przykład tworzenia nowych węzłów</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	window.onload = function()
	{
		document.body.innerHTML = '<div>Witaj świecie</div>' +
					'<div>Witaj nowy świecie</div>';
	}
	</script>
</head>
<body></body>
</html>
```

Jest to specjalna właściwość [30] wprowadzona w Windows Internet Explorer 4.0, a następnie zaadoptowana przez wszystkie popularne przeglądarki. Pozwala ona podać fragment kodu HTML jako ciąg znaków, który zostaje odpowiednio sparsowany i dodany do struktury dokumentu. Podany kod może posiadać liczne atrybuty (wliczając w to atrybuty dla zdarzeń, klas CSS, itp.).

2.5.4 Operowanie na atrybutach elementów

Elementy mogą zawierać elementy–dzieci (dołączane do nich np. za pomocą `appendChild') lub atrybuty. Do operowania na atrybutach służą m. in. metody [59]:

- getAttribute('nazwa-atrybutu') – pobiera atrybut
- setAttribute('nazwa-atrybutu', 'wartość') – ustawia nową wartość atrybutu, tworzy go jeśli wcześniej nie istniał
- removeAttribute('nazwa-atrybutu') – usuwa atrybut
- hasAttribute('nazwa-atrybutu') – zwraca `true' jeśli istnieje zadany atrybut w elemencie

```
<html>
<head>
	<title>Przykład operowania na atrybutach</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	window.onload = function()
	{
		var oElem = document.getElementsByTagName('div')[0];
		alert(oElem.hasAttribute('title'));
		oElem.setAttribute('title', 'Działa!');
		alert('element ma atrybut title: ' +
			oElem.hasAttribute('title') +
			', o wartości ' + oElem.getAttribute('title'));
		oElem.removeAttribute('title');
		alert(oElem.hasAttribute('title'));
	}
	</script>
</head>
<body>
	<div>Przykład</div>
</body>
</html>
```

Powyższy przykład najpierw wyświetli informację o braku atrybuty `title', następnie ustawi go i wyświetli komunikat o jego istnieniu, wraz z wartością, po czym go usunie. Niestety kod ten nie zadziała poprawnie w IE (najnowszej – 9 – wersji) [34]. Aby to obejść w przeglądarce Microsoftu należy używać `getAttribute'. Zwrócenie przez tą metodę null bądź " (oba należą do „falsy values" - patrz rozdział I) należy interpretować jako brak tego argumentu. Dziwny jest fakt, że we wcześniejszych wersjach IE metoda ta była dostępna. Brak kompatybilności wstecznej jest częstym grzechem twórców IE.

Do atrybutów elementów można się także dostać jak do właściwości obiektu elementu:

```
<html>
<head>
	<title>Przykład atrybuty jako właściwości obiektu</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	window.onload = function()
	{
		var oElem = document.getElementsByTagName('div')[0];
		// aby działało też w IE9
		alert(!!oElem.getAttribute('title'));
		oElem.title = 'Działa!';
		alert('element ma atrybut title: ' +
```

```
                    !!oElem.getAttribute('title') +
                   ', o wartości ' + oElem.getAttribute('title'));
            oElem.removeAttribute('title');
            alert(!!oElem.getAttribute('title'));
            alert('title' in oElem);
      }
      </script>
</head>
<body>
      <div>Przykład</div>
</body>
</html>
```

Działanie widzialne dla użytkownika dwóch poprzednich listingów jest takie samo, choć wymagało to pewnej wiedzy i dostosowania się programisty do jednej z przeglądarek. Zmiany w kodzie nie były zbyt trudne. Wymagały jednak świadomości pewnych niedogodności, a w przypadku istniejącego systemu olbrzymiego nakładu pracy.

W ten sposób można także ustawić atrybuty obsługująca zdarzenia. Poniższy kod będzie działał w Firefoksie, Chromie, Operze i Safari. IE tradycyjnie zachowa się inaczej.

```
oElem.setAttribute('onclick', 'alert(1)');
```

Choć lepszym sposobem jest przypisanie zdarzenia do właściwości `oElem.onclick' (patrz rozdział o zdarzeniach DOM), który działa we wszystkich przeglądarkach lub korzystanie z mechanizmów DOM Level 2.

Wyjątek ze stylami

O ile często można zrównać atrybut elementu DOM z właściwością obiektu elementu DOM, co zostało pokazane w poprzednich przykładach, to w przypadku stylów CSS (obiektu `style') nie jest to możliwe. Obiekt ten nie może być nadpisany, można jedynie nadpisać jego właściwości [12] (o czym mowa w następnym rozdziale). Style możemy jednak „wstrzyknąć” do elementu za pomocą (w IE9 nie zadziała):

```
oElem.setAttribute('style', 'color: red');
```

Do opisywania wyglądu strony można także użyć odpowiednich klas – atrybut `class', ale nie można na niego wpłynąć za pomocą właściwości obiektu węzła. W tym wypadku jednak możliwe jest posłużenie się właściwością `className', która została szerzej opisana w następnym rozdziale.

2.6 Arkusze Stylów CSS oraz obiekt `style'

Na początku HTML [121] był językiem opisującym zarazem strukturę dokumentu, jak i wygląd. Wraz z upowszechnieniem się kaskadowych arkuszy stylów CSS [122] (ang. *Cascading Style Sheets)* na początku XXI wieku standardem stało się definiowanie w (X)HTML [123] jedynie struktury dokumentu, a w CSS jego wyglądu.

Istnieje kilka sposobów połączenia stylów CSS z HTML:

- style w osobnym pliku i załączanie za pomocą znacznika `<link/>`
- definiowanie stylów w znaczniku `<style/>`
- definiowanie stylów w atrybucie `style'
- definiowanie stylów w JavaScript, poprzez obiekty węzłów DOM

Arkusz stylów CSS to zbiór reguł ustalających w jaki sposób ma zostać wyświetlana

przez przeglądarkę internetową wybrany element (X)HTML. Za pomocą CSS można opisać cały wygląd elementu zaczynając od odległości w jakich ma on być położony od sąsiadów przez kolor obramowania, tła, czcionki, cienia, kroju czcionki, odstępu między literami, itp. Daje to niezwykle duże możliwości.

2.6.1 Inline

Najprostszym sposobem umieszczania kodu CSS jest jego wstrzykiwanie do znaczników przez atrybut `style':

```
<p style='color: red; text-decoration: underline;'>Czerwony napis</p>
```

Jak widać powyższy sposób pozwala określić styl dla jednego konkretnego węzła. W atrybucie `style' podajemy ciąg znaków (deklaracji) o strukturze [9]:

```
właściwość: wartość;
```

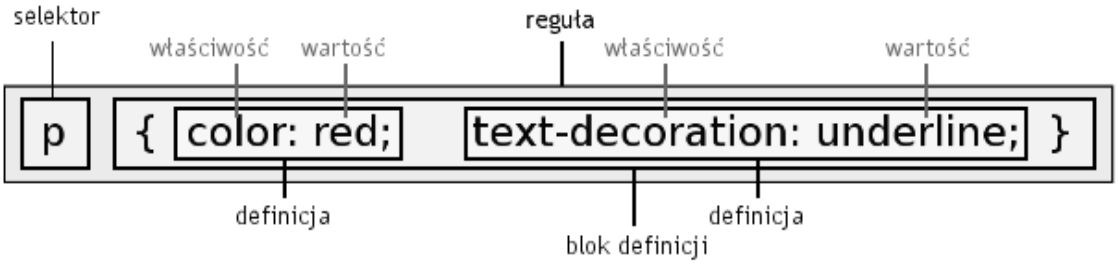
2.6.2 Definicje, selektory i klasy

Wcześniej opisany sposób niestety nie ułatwia wcale pracy programistom – nadal warstwa prezentacji jest zaszyta w warstwie danych. Aby temu zapobiec można wykorzystać definicję stylów wewnątrz znacznika <style/> tworząc odpowiednie reguły:

```
<html>
<head>
      <title>Przykład: style CSS - reguły i selektory</title>
      <meta http-equiv="content-type"
            content="text/html;charset=utf-8" />
      <style type="text/css">
      p { color: red; text-decoration: underline; }
      </style>
</head>
<body>
      <p>Czerwony napis</p>
</body>
</html>
```

Jak można zauważyć na powyższym listingu każda reguła składa się z bloku deklaracji rozpoczynającego się od `{' i kończącego `}'. Blok deklaracji z kolei składa się z deklaracji:

```
właściwość: wartość;
```



Rysunek 7. Struktura reguły CSS

Do odpowiednich elementów na stronie możemy dodać także klasy (atrybut `class').

```
<p class='extra-ordinary'>Specjalny napis</p>
```

Na taki element nadal oddziałują reguły dla znacznika <p/> (selektor `p'). Dodanie klasy pozwala

jednak na dodanie reguły tylko dla elementów, z tą klasą:

```
p.extra-ordinary { font-weight: bold; }
```

lub

```
.extra-ordinary { font-weight: bold; }
```

Drugi zapis pozwala na wykorzystanie tej reguły do elementów innego typu niż <p/>, np. tabel (znacznik <table/>) czy linków (<a/>).

Innym sposobem na wyspecyfikowanie w selektorze jaki dokładnie element ma wykorzystywać daną regułę pomaga także atrybut id (znany już z metody `getElementById'). Selektor odnoszący się do elementu z zadanym id wygląda następująco:

```
p#id { border: 1px solid red; }
```

Na stronie nie może być dwóch elementów o tym samym id, niezależnie od ich typów.

2.6.3 Obiekt `style'

Dynamiczne strony WWW nie mogą opierać się o statyczne deklaracje. Tu po raz kolejny świetnie spisuje się JavaScript, który także posiada dostęp do obiektu `style' każdego elementu strony i pozwala dopisywać do niego odpowiednie wartości.

```
<html>
<head>
	<title>Przykład: style CSS z poziomu JS – właściwości</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	window.onload = function()
	{
		var oElement = document.getElementsByTagName('p')[0];
		oElement.style.color = 'red';
		oElement.style.textDecoration = 'underline';
	}
	</script>
</head>
<body>
	<p>Czerwony napis</p>
</body>
</html>
```

W wyniku takiego działania kolor tekstu będzie czerwony, całość będzie podkreślona. Warte zauważenia jest to, iż właściwość `text-decoration' została zmieniona na `textDecoration'. Znak `-' nie jest dozwolony w nazwach zmiennych JavaScript. W takich sytuacjach znak ten znika, a w jego miejsce pojawia się notacja wielbłądzia.

2.6.4 Dynamiczne dodawanie klas

Wcześniejszy przykład pokazywał jak z poziomu JavaScript zmienić wartość pojedynczych właściwości. Istnieje także możliwość przypisywania obiektom całych klas CSS.

```
<html>
<head>
	<title>Przykład: style CSS z poziomu JavaScript – klasy</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<style type="text/css">
```

```
      .extra { color: red; text-decoration: underline; }
      .another { font-weight: bold; }
      </style>
      <script type="text/javascript">
      window.onload = function()
      {
            var oElement = document.getElementsByTagName('p')[0];
            oElement.className = 'extra'; // 1
            oElement.className += ' another'; // 2
      }
      </script>
</head>
<body>
      <p>Ten napis jest czerwony</p>
</body>
</html>
```

W linii „1” przypisałem polu `className' [88] wartość 'extra'. W linii „2” dopisałem jeszcze jedną klasę do tego elementu strony. Istotną kwestią jest tu biały znak dodany przed nazwą klasy 'another'. We właściwości `className' trzymany jest ciąg znaków. Każda kolejna klasa oddzielona jest od pozostałych białymi znakami (spacja, tabulacja, enter). Jest to dosyć kłopotliwe i w przypadku częstego operowania na tych wartościach słusznym wydaje się napisanie odpowiednich funkcji:

```
Node.prototype.addClass =  function(cls)
{
      this.className += ' ' + cls;
}
Node.prototype.removeClass =  function(cls)
{
      var classes = this.className.split(/\s/);
      this.className = '';
      for(var i = 0; i < classes.length; i++)
      {
            this.className += (classes[i] != cls) ? classes[i] : '';
      }
}
window.onload = function()
{
      var oElement = document.getElementsByTagName('p')[0];
      oElement.addClass('extra');
      oElement.addClass('another');
      oElement.removeClass('another');
}
```

Kod z powyższego listingu rozszerzył prototyp `Node' o dwie metody służące do dodawania i usuwania klas CSS. Tak prosty przykład pokazuje jak często niewiele brakuje, aby z nieprzyjemnego i operującego na niskim poziomie abstrakcji interfejsu wydobyć z niego to co najlepsze i móc wygodnie używać. W IE kod ten nie działa. Domyślna windowsowa przeglądarka nie poznaje obiektu Node.

W Firefoksie zamiast dość niskopoziomowego operowania na wartości pola `className' możemy operować na obiekcie `classList' posiadającym kilka przydatnych funkcji [87]:

- add(nazwa-klasy) – dodaje nową klasę do pola `element.classList', co z kolei powoduje dodanie nowej klasy do odpowiedniego węzła DOM
- remove(nazwa-klasy) – usuwa klasę
- toggle(nazwa-klasy) – usuwa jeśli istnieje albo dodaje klasę w przeciwnym wypadku

- contains(nazwa-klasy) – zwraca prawdę jeśli element posiada zadaną klasę

W HTML5 pole to ma stać się standardem [124].

2.7 Zdarzenia DOM

Zdarzenia są częścią modelu DOM Poziom 2 [61]. Mimo że nie znalazły się one w specyfikacji DOM Poziom 1, to mechanizm zdarzeń był implementowany w przeglądarkach (Netscape i IE 3.0 [1]) – tzw. DOM Poziom 0 [18].

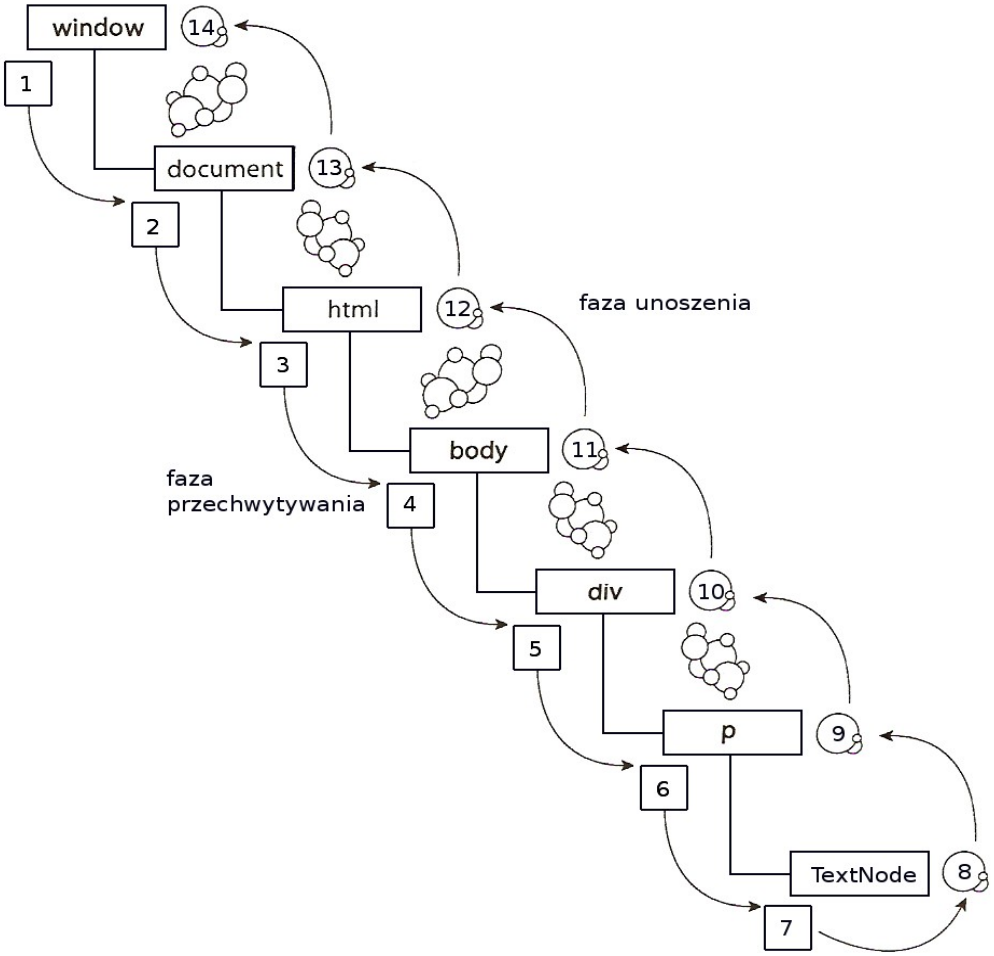
Dzięki istnieniu zdarzeń i możliwości ich obsługi poprzez wykonanie napisanych przez programistę funkcji można oszczędzić sporo czasu, który normalnie byłby poświęcona na komunikację z serwerem i renderowanie nowej strony. Jednym z najczęściej spotykanych przykładów zastosowań JavaScript i zdarzeń jest walidacja formularza.

Podobnie jak w wielu innych przypadkach, tak i specyfikacja zdarzeń powstała „po czasie". Najpierw Netscape oraz Microsoft zaimplementowali swoje firmowe rozwiązania, a dopiero potem W3C wydało odpowiednie rekomendacje. Z tego powodu, ciągle jeszcze obsługa zdarzeń nie wszędzie jest identyczna. Szczególnie IE bardzo odstawały od konkurencji posiadając swój własny „firmowy" model zdarzeń.

W wyniku braku jakiejkolwiek specyfikacji dotyczącej zdarzeń główni gracze na ówczesnym rynku Netscape oraz Microsoft starali się wypełnić lukę implementując własną formę przepływu zdarzeń.

W przypadku Internet Explorera była to technika zwana rozprzestrzenianiem. Polega ona na rozprzestrzenianiu się zdarzenia z obiektu najbardziej do najmniej specyficznego (obiekt `document', choć przeglądarki przekazują zdarzenia wyżej – ponad DOM – do obiektu `window'). Z kolei Netscape Navigator został wyposażony w mechanizm przechwytywania zdarzeń, które jest przeciwieństwem rozprzestrzeniania się; zdarzenia wywoływane są z najmniej specyficznego obiektu w modelu DOM (obiekt `document') do najbardziej specyficznego. Taki mechanizm nazywany był czasem „góra-dół" [1].

Nowoczesne popularne przeglądarki, za wyjątkiem IE, implementują zgodny ze specyfikacją W3C przepływa zdarzeń. Model DOM wspiera zarówno przechwytywanie zdarzeń, jak i ich rozprzestrzenianie. Przechwytywanie zdarzeń występuje w pierwszej kolejności. Obydwa modele przechodzą przez wszystkie elementy w strukturze DOM dwukrotnie (raz w fazie przechwytywania, raz w fazie unoszenia) zaczynając i kończąc na obiekcie `document', z którego zdarzenie jest przekazywane wyżej – do `window'. Unikatową cechą tego modelu zdarzeń jest fakt, iż węzły tekstowe także wywołują zdarzenia, choć zostało to zaimplementowane jedynie w Netscape 6 [46]. W innych przeglądarkach model ten się nie przyjął.



Rysunek 8. Przepływa zdarzeń w modelu DOM (źródło: [1])

2.7.1 Dodawanie funkcji obsługi zdarzeń – model DOM

Aby dodać obsługę zdarzenia do drzewa DOM należy użyć metody `addEventListener' [114] lub przypisać odpowiednią funkcję do atrybutu węzła DOM (np. `onclick', `onmouseover', itp.) odpowiadającego odpowiedniemu zdarzeniu. Funkcja, wywoływana w odpowiedzi na zdarzenie nazywamy procedurą obsługi zdarzenia (ang. *event_callback*) lub (według specyfikacji DOM) słuchaczem zdarzenia (ang. *event listener*). Zasadę tę ilustruje poniższy listing:

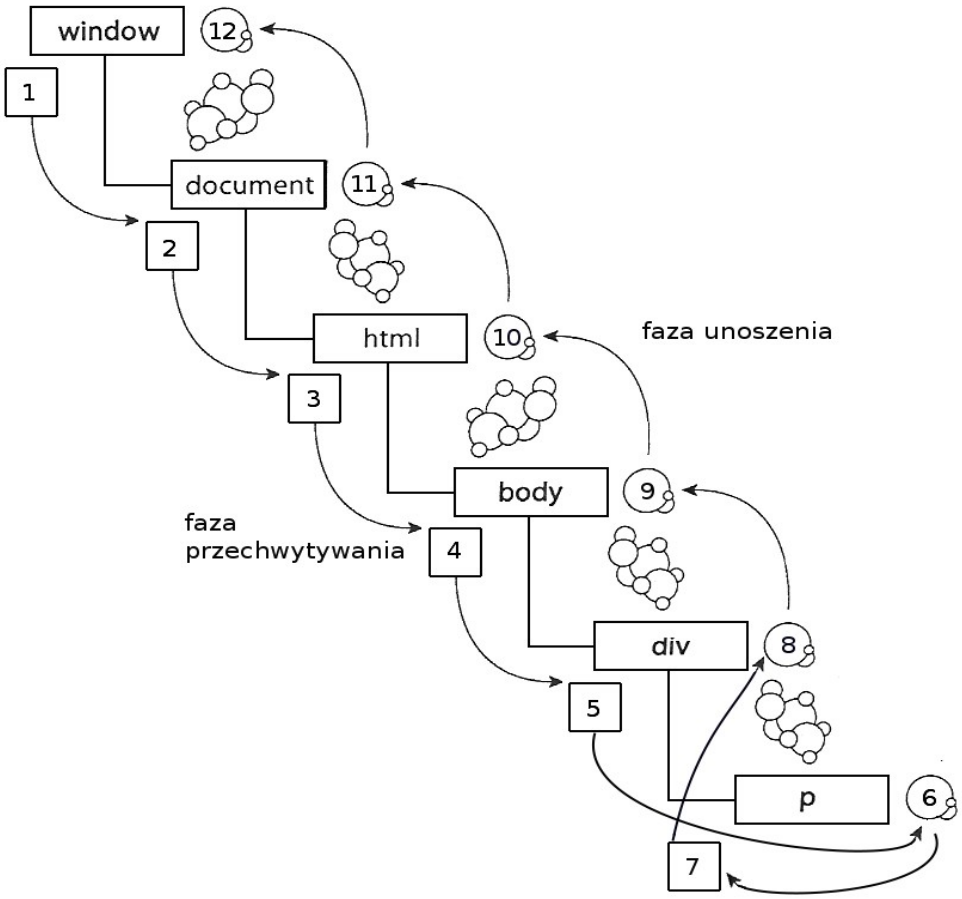
```
<html>
<head>
      <title>Przykład obsługi zdarzeń - model DOM</title>
      <meta http-equiv="content-type"
            content="text/html;charset=utf-8" />
      <script type="text/javascript">
      function addEvent( node, name )
      {
            node.addEventListener('click',
                  function(){alert(name + ' unoszenie')}, false);
            node.addEventListener('click',
                  function(){alert(name + ' przechwytywanie')}, true);
      }

      function addEventToFirstNode( tag )
      {
            var oNode = document.getElementsByTagName(tag)[0];
```

```
            addEvent(oNode, tag);
      }

      function addEventToTextNode( tag )
      {
            var oTextNode = document.getElementsByTagName(tag)[0].firstChild;
            addEvent(oTextNode, 'textNode ' + tag);
      }

      window.onload = function()
      {
            addEventToFirstNode('p');
            addEventToTextNode('p');
            addEventToFirstNode('div');
            addEventToFirstNode('body');
            addEvent(document, 'document');
            addEvent(window, 'window');
      }
      </script>
</head>
<body>
      <div>
            <p onclick='alert("p onclick!")'>test</p>
      </div>
      <div onclick='alert("działa!")'>Test atrybut</div>
</body>
</html>
```

Uruchomienie powyższego kodu w przeglądarkach FireFox, Google Chrome, Opera, Safari powoduje wyświetlenie komunikatów w kolejności:

1. window przechwytywanie
2. document przechwytywanie
3. body przechwytywanie
4. div przechwytywanie
5. p onclick
6. **p unoszenie**
7. **p przechwytywanie**
8. div unoszenie
9. body unoszenie
10. document unoszenie
11. window unoszenie

Krok 6 i 7 wydają się być niezgodne z modelem DOM (odwrotna kolejność) [63]. Z powyższego przykładu wynika, że funkcja obsługi zdarzenia przypisana za pomocą atrybutu HTML jest wywoływana przed funkcjami obsługi zdarzeń przypisanymi za pomocą `addEventListener'.



Rysunek 9. Przepływ zdarzeń w większości nowoczesnych przeglądarek

2.7.2 Usuwanie funkcji obsługi zdarzeń – model DOM

Aby usunąć funkcję obsługi zdarzenia, należy wykorzystać metodę `removeEventListener':

```
function helloWorld()
{
      alert("Kliknięto!" );
}

// dodanie obsługi zdarzeń
document.addEventListener("click", helloWorld, true );  // faza
przechwytywania
document.addEventListener("click", helloWorld, false );  // faza unoszenia

// usunięcie TYLKO obsługi zdarzenia w fazie przechwytywania
document.removeEventListener("click", helloWorld, true );
```

Jak widać na powyższym listingu, usuwając zdarzenia musimy wskazać dokładnie, o którą funkcję obsługi nam chodzi, oraz w jakiej fazie ma ona zostać usunięta.

2.7.3 Dodawanie funkcji obsługi zdarzeń – „model IE”

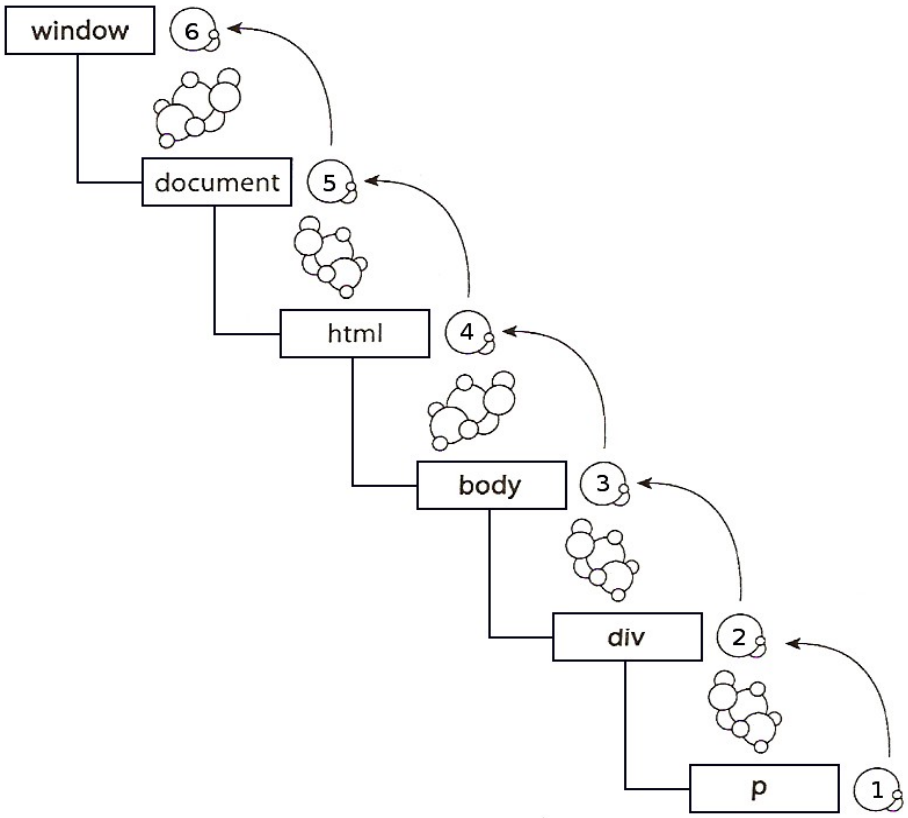
W Windows Internet Explorer kod ten nie działa poprawnie. Zadziała jedynie przypisanie zdarzenia do węzła poprzez atrybut `onclick'. Przeglądarki firmy Microsoft nadal nie wspierają modelu zdarzeń zgodnych z W3C DOM, zastępując go własnym modelem. Jest on oparty także o trzy metody:

- attachEvent – przypisuje procedurę obsługi zdarzenia do węzła
- detachEvent – usuwa procedurę obsługi zdarzenia z węzła
- fireEvent – wywołuje zdarzenie

Kod próbujący dawać te same wyniki w IE, co w przeglądarkach (lepiej lub gorzej) wspierających model DOM wyglądałby tak:

```
<html>
<head>
	<title>Przykład obsługi zdarzeń w IE</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	function addEvent( node, name )
	{
		node.attachEvent('onclick',
			function(){alert(name + ' unoszenie');});
	}

	function addEventToFirstNode( tag )
	{
		var oNode = document.getElementsByTagName(tag)[0];
		addEvent(oNode, tag);
	}

	window.onload = function()
	{
		addEventToFirstNode('p');
		addEventToFirstNode('div');
		addEventToFirstNode('body');
		addEvent(document, 'document');
		addEvent(window, 'window');
	}
	</script>
</head>
<body>
	<div>
		<p>test</p>
	</div>
	<div onclick='alert("działa!")'>Test atrybut</div>
</body>
</html>
```

Pierwsza różnica jest taka, że próba przypisania zdarzenia do węzła tekstowego wywołuje błąd. Inną, ważną różnicą, jest fakt używania metody `attachEvent', która przyjmuje tylko 2 parametry. W modelu IE nie ma fazy przechwytywania i unoszenia. Jest jedynie unoszenie, choć istnieją specjalne biblioteki, których twórcy twierdzą, że zaimplementowali model DOM w IE [31]. Różnicą, której nie widać, jest brak obsługi zdarzenia `click' na obiekcie `window'. W tej kwestii IE bardziej przypomina w działaniu model DOM od pozostałych przeglądarek, które zawsze zaczynają i kończą zdarzenie w tym obiekcie.



Rysunek 10. Przepływa zdarzeń w IE (źródło: [1])

2.7.4 Usuwanie funkcji obsługi zdarzeń – „model IE”

Do usuwania funkcji obsługi zdarzeń w IE służy metoda `detachEvent':

```
function helloWorld()
{
      alert( "Hello World" );
}
// dodaj obsługę zdarzenia
document.attachEvent("onclick", helloWorld );
// usuń obsługę zdarzenia
document.detachEvent("onclick", helloWorld );
```

Jak można zauważyć, model zdarzeń działający w IE wydaje się być prostszy.

2.7.5 Zdarzenia DOM Level 0

W powyższych przykładach położyłem nacisk na wykorzystania specjalnych metod pozwalających dodawać słuchaczy zdarzeń. Istnieje jednak prostsza – i działająca wszędzie – metoda, którą po cichu wprowadziłem w poprzednich listingach. Jest to dodawanie zdarzeń przez odpowiednie atrybuty.

```
<html onclick='alert("html")'>
<head>
      <title>Przykład obsługi zdarzeń  - atrybuty</title>
      <meta http-equiv="content-type"
            content="text/html;charset=utf-8" />
</head>
<body onload='alert("body")' onclick='alert("body")'>
```

```
      <div onclick='alert("div")'>
            <p onclick='alert("p")'>test</p>
      </div>
</body>
</html>
```

Aby móc usunąć zdarzenie należy nadpisać właściwość `onclick' odpowiedniego węzła DOM z poziomu JavaScript:

```
<html>
<head>
      <title>Przykład obsługi zdarzeń  - atrybuty, usuwanie</title>
      <meta http-equiv="content-type"
            content="text/html;charset=utf-8" />
      <script type="text/javascript">
      window.onload = function()
      {
            document.getElementById('cliker').onclick = null;
      }
      </script>
</head>
<body>
      <div id='clicker' onclick='alert("div")'>test</div>
</body>
</html>
```

Warto pamiętać, że operując bezpośrednio na atrybucie obiektu węzła DOM z poziomu JavaScript nie dopisujemy kolejnych procedur obsługi zdarzeń, ani nie możemy usunąć jednej, wybranej, funkcji obsługi zdarzenia. Zawsze nadpisujemy wszystkie funkcje obsługi danego zdarzenia przypisane do tego węzła. Plusem takiego rozwiązania jest na pewno jego przejrzystość oraz fakt, że działa we wszystkich przeglądarkach.

2.7.6 Obiekt Event

Mechanizm obsługi zdarzeń, prócz możliwości przechwytywania zdarzenia oraz jego obsługi wprowadził także specjalny obiekt `Event', dostarczający informacji dotyczących zaistniałego zdarzenia:

- obiekt, który spowodował zdarzenie
- informacje dotyczące stanu myszy w trakcie zdarzenia
- informacje dotyczące stanu klawiatury w czasie zdarzenia

Jak w całym mechanizmie obsługi zdarzeń istnieją dwa modele – zgodny ze specyfikacją DOM (obiekt przekazywany jest jako pierwszy parametr do funkcji obsługi zdarzenia) oraz model zaimplementowany w Internet Explorerze (obiekt `event' jest właściwością obiektu `window' – zmienną globalną – dostępną tylko w czasie obsługi zdarzenia i usuwaną zaraz po).

W przeglądarkach zgodnych z modelem DOM kod wyglądałby zatem:

```
element.onclick = function(event)
{
      alert(event);
}
```

W komunikacie pojawi się „*[object MouseEvent]*”.

Dla IE podobnie działający kod:

```
element.onclick = function()
```

```
{
	alert(window.event);
}
```

Wyświetli w komunikacie „*[object]*”.

Aby mieć pewność, że będziemy mieli dostęp do obiektu `Event' w różnych przeglądarkach należy zrobić coś takiego:

```
element.onclick = function(e)
{
	var evt = e || window.event;
	alert(evt);
}
```

Dzięki takiemu rozwiązaniu w obu przeglądarkach będziemy mieli wewnątrz funkcji do dyspozycji obiekt `Event'. Zastanawiającym może być, dlaczego w IE nie pojawia się błąd skoro z definicji funkcji wynika, że powinien zostać przekazany parametr. IE nie przekazuje go, jednak JavaScript nie wymaga, aby do funkcji przekazywać takiej liczby parametrów jaka została zadeklarowana. Co więcej, można zdefiniować funkcję bez żadnych parametrów, a następnie odwoływać się do nich za pomocą array like object `arguments' (jest to obiekt, choć można się odwoływać do kolejnych przekazanych parametrów za pomocą indeksów, np. arguments[0], itp.) [92].

Warto tu także zaznaczyć, że w przypadku kodu:

```
<html>
<head>
	<title>Przykład obsługi zdarzeń  - obiekt Event</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	function clickHandler(e)
	{
		var evt = e || window.event;
		alert(evt);
	}
	</script>
</head>
<body>
	<div id='clicker' onclick='clickHandler()'>test</div>
</body>
</html>
```

Model IE przechwyci obiekt `Event', podczas gdy przeglądarki zgodne z DOM już nie. W ich przypadku, dla takich funkcji obsługi zdarzeń, nie można dostać w taki sposób. Działa natomiast takie rozwiązanie:

```
<html>
<head>
	<title>Przykład obsługi zdarzeń  - obiektu Event</title>
	<meta http-equiv="content-type"
		content="text/html;charset=utf-8" />
	<script type="text/javascript">
	function clickHandler(e)
	{
		alert(e);
	}
	</script>
</head>
<body>
	<div id='clicker' onclick='clickHandler(event)'>test</div>
```

```
</body>
</html>
```

Co istotne, nazwa zmiennej przekazywanej do `clickHandler' musi być właśnie taka. Przekazanie `window.event' w Firefox daje w wyniku wartość `undefined'. Nazwa inna niż „event” („e”, „evt”, „zdarzenie”, „jakkolwiek”) powoduje błąd: „*ev is not defined*”. W przypadku Safari i Chrome obiekt `Event' przekazywany jest zarówno jako pierwszy parametr, jak i w zmiennej `window.event'. Także operator `this' zachowuje się różnie w zależności od wybranej metody przypisywania funkcji obsługi zdarzeń (patrz Rozdział 1: operator new i this).

Różnice w modelu przepływu zdarzeń, inne metody pozwalające dodawać funkcje obsługi zdarzeń oraz problemy z przekazywaniem obiektu `Event' to niestety nadal nie jedyne różnice w działaniu mechanizmu zdarzeń w przeglądarkach. Jak już wcześniej wskazałem, do przeglądarek zgodnych z modelem DOM przekazywany jest obiekt `MouseEvent', do IE trafia `object'. Co prawda JavaScript jako język o dynamicznych typach bardziej zwraca uwagę na interfejs oferowany przez obiekt niż jego typ, zgodnie z zasadą duck typing, jednak nawet nie patrząc na typ a jedynie porównując interfejs obiekty te różnią się znacznie.

Pola obiektu `Event' z określeniem modelu, w którym zostały zaimplementowane [1]:

Pole obiektu Event	Opis	IE/ DOM
bool altKey	true, gdy wciśnięty Alt	oba
bool bubbles	określa, czy zdarzenie rozprzestrzenia się	DOM
int button	Został wciśnięty klawisz myszy. Możliwe wartości: 0 – nie wciśnięto żadnego 1 – lewy przycisk 2 – prawy przycisk 4 – środkowy przycisk możliwe są wartości 3, 5, 6, 7 będące sumą odpowiednich wartości.	oba
bool cancelable	Określa, czy zdarzenie może być anulowane	DOM
bool cancelBubble	ustawienie przez programistę na true zatrzymuje rozprzestrzenianie się zdarzenia	IE
	Określa, czy rozprzestrzeniające się zdarzenie zostało anulowane (tylko do odczytu)	DOM
int charCode	Kod znaku w kodowaniu Unicode wciśniętego klawisza	DOM
int clientX	współrzędna x kursora myszki wewnątrz obszaru roboczego	oba
int clientY	współrzędna y kursora myszki wewnątrz obszaru roboczego	oba
bool ctrlKey	true, gdy ctrl wciśnięty	oba
element currentTarget	Określa element, który jest aktualnym obiektem docelowym	DOM
int detail	Określa ile razy przycisk myszki został kliknięty	DOM
int eventPhase	Faza zdarzenia, możliwe wartości: 0 – faza przechwytywania 1 – w obiekcie docelowym 2 – faza rozprzestrzeniania	DOM
element fromElement	element znad którego usuwany jest kursor w momencie występowania zdarzenia	IE
bool isChar	Określa czy wciśnięty klawisz ma skojarzony z nim znak tekstowy	DOM
int keyCode	dla zdarzenia `keypress' kod znaku w Unicode wciśniętego klawisza, dla `keydown'/`keyup' numeryczny kod wciśniętego klawisza	IE
	Numeryczny kod wciśniętego klawisza	DOM
int offsetX	współrzędna x kursora myszki względem obiektu, który wywołał zdarzenie	IE
int offsetY	współrzędna y kursora myszki względem obiektu, który wywołał zdarzenie	IE
int metaKey	Określa, czy został wciśnięty klawisz META	DOM
int pageX	Współrzędna x kursora myszki w odniesieniu do strony	DOM
int pageY	Współrzędna y kursora myszki w odniesieniu do strony	DOM
preventDefault()	Metoda możliwa do wywołania przez programistę, jeśli chce zablokować domyślne zachowanie zdarzenia	DOM
element relatedTarget	Drugorzędny obiekt docelowy zdarzenia	DOM

Pole obiektu Event	Opis	IE/ DOM
bool repeat	zwraca true, gdy zdarzenie `keydown' jest wywołane sekwencyjnie	IE
bool returnValue	ustawienie przez programistę na false w celu anulowania domyślnego zareagowania na dane zdarzenie	IE
int screenX	współrzędna x w odniesieniu do ekranu monitora	oba
int screenY	współrzędna y w odniesieniu do ekranu monitora	oba
bool shiftKey	true, gdy klawisz shift jest wciśnięty	oba
stopPropagation()	Metoda możliwa do wywołania przez programistę, jeśli chce zablokować dalszą propagację zdarzenia	DOM
Element srcElement	element, który spowodował zajście zdarzenia	IE
Element toElement	element, nad którym pojawił się kursor myszy w momencie występowania zdarzenia	IE
Element target	Element, który spowodował wystąpienie zdarzenia	DOM
int timeStamp	Czas wystąpienia zdarzenia liczony w milisekundach od 1 stycznia 1970 r.	DOM
string type	nazwa zdarzenia	oba
int x	współrzędna x kursora myszy w odniesieniu do elementu-rodzica.	IE
int y	współrzędna y kursora myszy w odniesieniu do elementu-rodzica.	IE

Tabela 2. Pola obiektu `Event' w różnych modelach zdarzeń

W przypadku modelu IE jedynie pole `repeat' jest tylko do odczytu – wartość pozostałych pól może być zmieniona. W modelu DOM jedynie wartości pól: `altKey', `button', `keyCode' mogą być zmieniane przez programistę. Pozostałe są polami tylko do odczytu.

Mimo sporych różnic w strukturze obiektu `Event' sporo spośród „brakujących” pól posiada swoje odpowiedniki w konkurencyjnym modelu:

IE	DOM
event.returnValue = false;	event.preventDefault();
event.srcElement;	event.target;
event.keyCode;	event.charCode;
event.cancelBubble = true;	event.stopPropagation();

event.eventPhase = 2; // tylko rozprzestrzenianie
de facto pole nie istnieje [== undefined]
	event.eventPhase przyjmuje: [0, 1, 2] w zależności od faktycznej fazy

Tabela 3. Różnice w interfejsach obiektu `Event' w modelu DOM i IE

2.7.7 Typy zdarzeń

W tabeli zawierającej pola obiektu `Event' często występują właściwości odnoszące się do innych zdarzeń, niż tylko tych wywoływanych myszką. Zdarzenia podzielić można na [62]:

- zdarzenia myszy
- zdarzenia klawiatury

- zdarzenia HTML
- zdarzenia interfejsu użytkownika
- zdarzenia mutacji struktury DOM

Typ	**Nazwa**	**Wywoływane, gdy**	U	C
Mysz	click	Na elemencie zostanie kliknięty przycisk. Przez „kliknięcie” rozumie się sekwencję: `mousedown', `mouseup' zachodzącą w tej samej lokalizacji. Kolejność odpalania zdarzeń: 1. mousedown 2. mouseup 3. click	T	T
	dblclick	Dwuklik	T	T
	mousedown	Przycisk myszy zostanie wciśnięty nad elementem	T	T
	mouseup	Przycisk myszy zostanie puszczony nad elementem	T	T
	mouseover	Kursor znajdzie się nad elementem	T	T
	mousemove	Kursor porusza się nad obiektem	T	T
	mouseout	Kursor znajdzie się poza obszarem elementu (jeśli wcześniej był nad nim)	T	T
Klawiatura	keydown	Klawisz klawiatury jest wciśnięty (przed `keypress')	T	T
	keypress	Klawisz klawiatury jest wciśnięty (po `keydown')	T	T
	keyup	Puszczony klawisz klawiatury	T	T
HTML	load	dla dokumentu: Przeglądarka wczyta wszystkie elementy strony (treść, object, frame, image) dla elementu: kiedy został wczytany, łącznie z treścią	N	N
	unload	Wszystkie elementy zostają usunięte z okna lub ramki	N	N
	abort	Object/image przestał się wczytywać nim została cała pobrana z serwera	T	N
	error	Object/image/frame nie został prawidłowo załadowany	T	N
	resize	Zmiana wielkości okna	T	N
	scroll	Przewijanie dokumentu	T	N
	select	Użytkownik zaznacza tekst	T	N
	change	Po zejściu z pola ma ono inną wartość, niż gdy nabywało focus	T	N
	submit	Wysłanie formularza	T	T
	reset	Reset formularza	T	N
	focus	Kontrolka formualrza nabywa focus	N	N
	blur	Kontrolka formularza traci focus	N	N
Interfejs	DOMFocusIn	Podobny do HTML focus, (stosowany nie tylko do formularzy)	T	N
	DOMFocusOut	Podobny do HTML blur, (stosowany nie tylko	T	N

		do formularzy)		
	DOMActive	Element jest aktywny (niezależnie od sposobu aktywacji – myszką, czy klawiaturą)	T	T
Mutacja	DOMSubtreeModified	Poddrzewo DOM zostało zmodyfikowane	T	N
	DOMNodeInserted	Dodanie węzła	T	N
	DOMNodeRemoved	Usunięcie węzła z drzewa DOM (wywołane przed usunięciem)	T	N
	DOMNodeRemovedFromDocument	Usunięcie węzła z drzewa dokumentu (wywołane przed usunięciem)	T	N
	DOMNodeInsertedIntoDocument	Dodanie węzła do dokumentu	T	N
	DOMAttrModified	Zmiana wartości atrybutu	T	N
	DOMCharacterDataModified	Tekst w węźle uległ zmianie	T	N

Tabela 4. Zdarzenia DOM

U – unoszenie

C – możliwość anulowania

Atrybuty HTML tworzone są od nazw zdarzenia przez dodanie przyrostka „on”, np. „click” → „onclick”. HTML pozwalał na dowolną wielkość liter w atrybutach (onCLICK było równoważne z onclick oraz OnClick). W XHTML – zgodnie z zasadami XML (patrz Rozdział 2: XML) nazwy atrybutów powinny być pisane w całości małymi literami. Zdarzenia zaczynające się od DOM nie są jeszcze w pełni wspierane przez wszystkie przeglądarki. Istnieją także zdarzenia obsługiwane jedynie przez przeglądarki z rodziny IE [29].

Jak pokazałem wcześniej, zdarzenia DOM są niezwykle przydatnym i mocnym mechanizmem działającym w JavaScript. Niestety, w wyniku początkowego braku istniejących standardów pojawiły się dwa, częściowo sprzeczne, modele obsługi zdarzeń. Powoduje to konieczność ograniczania możliwości naszych skryptów, zawężając je do „części wspólnej modeli” albo konieczność implementacji specjalnych protez pozwalających symulować część funkcjonalności. Kod taki nie jest niczym przyjemnym w tworzeniu. Jest bardzo podatny na błędy i nietypowe działania wynikające z często bardzo drobnych różnic w przeglądarkach. Także nigdy nie można być pewnym, że napisane przez nas rozwiązanie działa w każdej sytuacji, na każdej przeglądarce – wliczając w to nowe wersje już wspieranych programów. Wszystko to sprawia, że w tej dziedzinie, podobnie jak w operowaniu na strukturze DOM bardzo dużą popularnością cieszą się gotowe frameworki, pozwalające na oderwanie się od walki z przeglądarkami, a rozpoczęcie walki z faktycznymi problemami jakie stawia przed nami realizowany projekt.

2.8 Asemblaryzacja JavaScript

JavaScript, który niedawno obchodził 15 urodziny na dobre zagościł w Internecie jako „język przeglądarek”. Jego pozycja na tym polu wydaje się być przez długi jeszcze czas niezachwiana. Na jego korzyść świadczy też swoisty renesans języka, który zapoczątkowało rozpowszechnienie Ajaksa.

Mimo wielkiej popularności, przyznać trzeba, że niewielu programistów piszących skrypty w JavaScript naprawdę zna ten język! Wydaje się on być niezwykle prostym, jednak w miarę wzrostu skomplikowania systemu coraz częściej developerzy napotykają na problemy, których nie spodziewaliby się w żadnym innym języku. Stąd też taką popularnością cieszą się wszystkie

rozwiązania pozwalające na korzystanie z możliwości JavaScript, nie programując de facto w JavaScript. Choćby GWT [16] pozwalające pisać w Javie i kompilować (sic!) kod do JavaScript czy CoffeScript [22] dostarczający nowego języka, także kompilowanego do JavaScript. Innym rozwiązaniem jest ObjectiveJ [14], będącym nadzbiorem JavaScript, którego zadaniem także jest „ucieczka" od natywnego JavaScriptu i umożliwienie programowania w środowisku, do którego przywykli programiści. Coraz większą rolę odgrywa także kwestia kompresji kodu [83], pozwalająca na przyspieszenie pobierania kodu z serwera, jak i polepszenie czasu interpretacji skryptów. Pojawiają się także narzędzia do rzeczywistej kompilacji kodu JavaScript do kodu JavaScript, optymalizując go zarówno pod względem rozmiaru pliku (długość zmiennych, białe znaki) jak i szybkości wykonania (zmiany w kodzie, np. przeklejanie kodu z funkcji wykorzystywanych jednokrotnie) [15].

W nurt ten doskonale wpisują się wszelkie biblioteki i frameworki, rozszerzające możliwości (i bardzo polepszające wydajność pracy) zwykłego programisty przez nałożenie na często niewygodne mechanizmy bardzo przejrzystego interfejsu. Dzisiejsze łącza pozwalają na wymuszenie załączania dodatkowych plików frameworka, w zamian za ciekawe efekty i (za co płaci użytkownik końcowy) wygodę programisty. Także mimo faktu, iż JavaScript jest niezwykle wolnym [36] językiem programowania, dzisiejsze komputery pozwalają już na nakładanie nań kolejnych warstw abstrakcji w imię zasady: „czas pracy procesora jest tańszy niż czas pracy programisty". Dodatkowym plusem płynącym z wykorzystania frameworków JavaScript jest ominięcie jednego z największym problemów – różnic w działaniu kodu w różnych przeglądarkach. Dzięki temu programista zajmuje się rozwojem aplikacji, a nie walką ze środowiskiem, który to problem ten zostaje przeniesiony na kogoś innego. Frameworki takie jak: jQuery, dojo, ExtJS, Yahoo User Interface, Prototype, Mootools i wiele, wiele innych stale rozwijanych rozwiązań pokazują, że JavaScript jest potrzebnym językiem, który jednak nie jest zbyt wygodny w swojej podstawowej formie. Trzeba jednak przyznać, iż ostatnie propozycje zmian pozwalają patrzeć z optymizmem w przyszłość (choćby natywne wsparcie dla XML [115] czy JSON [109]).

Wszystkie te zjawiska pozwalają na wyciągnięcie prostego wniosku: JavaScript jeszcze na długo pozostanie ważnym (może nawet najważniejszym) językiem na świecie. Jednak nie oznacza, to wcale, że coraz więcej osób będzie w nim programować. Język ten stanie się swoistym „asemblerem" przeglądarek. Może nawet dojść do sytuacji, w której programista nie będzie już wcale znał JavaScript, a mimo to będzie tworzył aplikacje, które w ostateczności są oparte o ten język. Tak jak dzisiaj ma to miejsce ze zwykłym asemblerem czy byte – codem.

3 Bibliografia

[1] Zakas N.C: „JavaScript dla Webmasterów”, Helion, Gliwice 2006, ISBN: 83-246-0280-1

[2] Crockford D.: „Mocne strony JavaScript”, Helion, Gliwice 2009, ISBN: 978-83-246-1998-6

[3] Stefanov S.: „Programowanie obiektowe w JavaScript”, Helion, Gliwice 2010, ISBN: 978-83-246-2242-9

[4] Russell M. A.: „Dojo. The Definitive Guide”, O'Reilly Media, Sebastopol 2008, ISBN: 978-0-596-51648-2

[5] Seible P.: „Sztuka kodowania. Sekrety wielkich programistów, Helion, Gliwice 2011, ISBN: 978-83-246-2755-4

[6] Darie C., Brinzarea B., Cherecheş-Toşa F., Bucica M.: „AJAX i PHP. Tworzenie interaktywnych aplikacji internetowych”, Helion, Gliwice 2006, ISBN: 83-246-0644-0

[7] Kazienko P., Gwiazda K.: „XML na poważnie”, Helion, Gliwice 2002, ISBN: 83-7197-765-4

[8] Cormen T.H., Leiserson C. E, Rivest R. L., Stein C.: „Wprowadzenie do algorytmów”, Wydawnictwa Naukowo-Techniczne, Warszawa 2005, ISBN: 83-204-3149-2

[9] Meyer E. A.: „CSS Kaskadowe arkusze stylów. Przewodnik encyklopedyczny. Wydanie 3”, Helion, Gliwice 2008, ISBN: 978-83-246-0956-7

[10] Komitet TC39 - ECMAScript (formerly TC39-TG1), http://www.ecma-international.org/memento/TC39.htm (data dostępu: 18-05-2011 r.)

[11] W3C, Fragment Hypertext Transfer Protocol – HTTP/1.1 RFC 2616 Fielding, http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html (data dostępu: 18-05-2011 r.)

[12] Mozilla Developer Center, element.style, https://developer.mozilla.org/en/STYLE (data dostępu: 18-05-2011 r.)

[13] Ajaxian.com, „Security Focus: JavaScript Global Namespace Pollution” (07-02-2008), http://ajaxian.com/archives/security-focus-javascript-global-namespace-pollution (data dostępu: 18-05-2011 r.)

[14] Cappuccino.org, Learning Objective-J, http://cappuccino.org/learn/tutorials/objective-j-tutorial.php (data dostępu: 18-05-2011 r.)

[15] Google Code, Closure Compiler, http://code.google.com/intl/pl/closure/compiler/ (data dostępu: 18-05-2011 r.)

[16] Google Code, Google Web Toolkit (GWT), http://code.google.com/intl/pl/webtoolkit/ (data dostępu: 18-05-2011 r.)

[17] Soshnikov D. A.: „JavaScript. The core” (02-09-2010), http://dmitrysoshnikov.com/ecmascript/javascript-the-core/#a-prototype-chain (data dostępu: 18-05-2011 r.)

[18] Wikipedia, DOM events – DOM Level 0, http://en.wikipedia.org/wiki/DOM_events#DOM_Level_0 (data dostępu: 18-05-2011 r.)

[19] Wielgosik D.: „Falsy values i operatory porównania” (05-09-2009), http://ferrante.pl/frontend/javascript/falsy-values-i-operatory-porownania (data dostępu: 18-05-2011 r.)

[20] Wielgosik D.: „Magia operatorów w JavaScript” (03-03-2009),

http://ferrante.pl/frontend/javascript/magia-operatorow-w-javascript/ (data dostępu: 18-05-2011 r.)

[21] HTML5.org, HTML5 Standard, http://html5.org/ (data dostępu: 18-05-2011 r.)

[22] Coffee Script, Overview, http://jashkenas.github.com/coffee-script/ (data dostępu: 18-05-2011 r.)

[23] Crockford D.: „JavaScript: The World's Most Misunderstood Programming Language" (2001), http://javascript.crockford.com/javascript.html (data dostępu: 18-05-2011 r.)

[24] Getify Solutions, „JSON-P: Safer cross-domain Ajax with JSON-P/JSONP" (2010), http://json-p.org/ (data dostępu: 18-05-2011 r.)

[25] Crockford D.: „JSON in JavaScript", http://json.org/js.html (data dostępu: 18-05-2011 r.)

[26] http://json.org/json-pl.html

[27] http://livedocs.adobe.com/specs/actionscript/3/wwhelp/wwhimpl/js/html/wwhelp.htm

[28] http://msdn.microsoft.com/en-us/library/cc836466%28v=vs.85%29.aspx

[29] http://msdn.microsoft.com/en-us/library/ms533051%28v=vs.85%29.aspx

[30] http://msdn.microsoft.com/en-us/library/ms533897%28VS.85%29.aspx

[31] http://news.qooxdoo.org/qooxdoo-08-event-layer

[32] http://ole-laursen.blogspot.com/2009/11/javascript-and-emacs-lisp.html

[33] http://pl.wikipedia.org/wiki/Domknięcie_(programowanie)

[34] http://reviews.reviewboard.org/r/183/

[35] http://support.microsoft.com/kb/208427

[36] http://themaninblue.com/writing/perspective/2010/03/22/

[37] http://www.adaptivepath.com/ideas/e000385

[38] http://www.apple.com/hotnews/thoughts-on-flash/

[39] http://www.crockford.com/javascript/javascript.html

[40] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[41] http://www.ecma-international.org/publications/standards/Ecma-357.htm

[42] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33835

[43] http://www.jslint.com/

[44] http://www.kawa.net/works/js/xml/objtree-e.html

[45] http://www.mongodb.org/display/DOCS/Javascript+Language+Center

[46] http://www.nczonline.net/blog/2008/02/09/can-text-nodes-receive-events/

[47] http://www.w3.org/2001/tag/doc/get7#myths

[48] http://www.w3.org/2001/tag/doc/whenToUseGet.html#ephemeral

[49] http://www.w3.org/DOM/

[50] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.3 - metoda GET

[51] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.5 - POST

[52] http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html#ID-1734834066

[53] http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html#ID-184E7107
[54] http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html#ID-785887307
[55] http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html#ID-952280727
[56] http://www.w3.org/TR/access-control/#introduction
[57] http://www.w3.org/TR/DOM-Level-2-Core
[58] http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1938918D
[59] http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-745549614
[60] http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-BBACDC08-h2
[61] http://www.w3.org/TR/DOM-Level-2-Events/
[62] http://www.w3.org/TR/DOM-Level-2-Events/events.html
[63] http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-flow-capture
[64] http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html#Iterator-overview
[65] http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html#Traversal-NodeFilter
[66] http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html#TreeWalker
[67] http://www.w3.org/TR/DOM-Level-3-Core/
[68] http://www.w3.org/TR/REC-DOM-Level-1/
[69] http://www.w3.org/TR/xml11/
[70] http://www.w3.org/TR/XMLHttpRequest/
[71] http://www.w3.org/TR/XMLHttpRequest/#dom-xmlhttprequest-setrequestheader
[72] http://www.w3.org/TR/XMLHttpRequest/#the-xmlhttprequest-interface
[73] http://www.w3cdom.org/
[74] http://www.w3schools.com/browsers/browsers_stats.asp
[75] http://www.w3schools.com/JS/js_comparisons.asp
[76] http://www.w3schools.com/js/js_variables.asp
[77] http://www.w3schools.com/w3c/w3c_dom.asp
[78] http://www.w3schools.com/XML/xml_elements.asp
[79] http://www.yarpo.pl/2011/02/16/json-zamiast-konstruktorow
[80] http://www.yarpo.pl/2011/03/23/ajax-w-oparciu-o-plywajaca-ramke/
[81] http://www.yarpo.pl/2011/04/08/ajax-w-oparciu-o-cookies
[82] http://www.yarpo.pl/2011/05/07/json-with-padding-czyli-zdalny-ajax
[83] http://www.yarpo.pl/tag/kompresja-kodu/
[84] http://youthcoders.net/javascript/artykuly/265-js-mocne-strony-crockford.html
[85] http://yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/
[86] https://developer.mozilla.org/En/Developing_add-ons
[87] https://developer.mozilla.org/en/DOM/element.classList

[88] https://developer.mozilla.org/en/DOM/element.classname
[89] https://developer.mozilla.org/en/DOM/element.getElementsByTagName
[90] https://developer.mozilla.org/en/E4X_Tutorial/Introduction
[91] https://developer.mozilla.org/en/Gecko
[92] https://developer.mozilla.org/en/JavaScript/Reference/functions_and_function_scope/arguments
[93] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Boolean
[94] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/apply
[95] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/call
[96] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/hasOwnProperty
[97] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/undefined
[98] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Comparison_Operators
[99] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/delete
[100] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/in
[101] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/instanceof
[102] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/new
[103] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/this
[104] https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/typeof
[105] https://developer.mozilla.org/en/JavaScript/Reference/Reserved_Words
[106] https://developer.mozilla.org/en/JSON
[107] https://developer.mozilla.org/en/Rhino
[108] https://developer.mozilla.org/en/SpiderMonkey
[109] https://developer.mozilla.org/En/Using_native_JSON
[110] https://developer.mozilla.org/pl/Dokumentacja_języka_JavaScript_1.5/Funkcje/eval
[111] https://developer.mozilla.org/pl/DOM/document.createElement
[112] https://developer.mozilla.org/pl/DOM/document.createTextNode
[113] https://developer.mozilla.org/pl/DOM/document.getElementById
[114] https://developer.mozilla.org/pl/DOM/element.addEventListener
[115] https://developer.mozilla.org/pl/E4X
[116] http://www.whatwg.org/specs/web-apps/current-work/multipage/
[117] http://pl.wikipedia.org/wiki/IEEE_754
[118] Eichorn J.: „AJAX i JavaScript”, Helion, Gliwice 2007, ISBN: 978-83-246-1098-3
[119] http://www.w3.org/TR/access-control/#introduction
[120] http://pl.wikipedia.org/wiki/Uniform_Resource_Locator
[121] http://www.w3.org/TR/html401/

[122] http://www.w3.org/Style/CSS/

[123] http://www.w3.org/TR/xhtml1/

[124] http://www.whatwg.org/specs/web-apps/current-work/multipage/elements.html#dom-classlist