Technologie internetowe *client-side* na przykładzie języka JavaScript

Andrzej Skowron

Spis treści

1 Wprowadzenie

W 1995r. Brendon Eich rozpoczął pracę nad pierwszą wersją języka **JavaScript** (wtedy projekt ten nazywał się **LiveScript**). Został on zaimplementowany w przeglądarce Netscape. Niedługo potem doszło do porozumienia między firmami Netscape i Sun Microsystems, w wyniku którego na początku roku 1996 w przeglądarce Netscape 2 pojawił się JavaScript w wersji 1.0.

Wkrótce firma Microsoft wyposażyła przeglądarkę Internet Explorer 3 w swoją implementację tego języka, którą określono jako **JScript**. W krótkim czasie, po stronie klienta, pomiędzy językami JavaScript a JScript istniało tak dużo różnic, że pisanie skryptów przeznaczonych do wykorzystywania na wielu platformach nie było łatwe. Powstał pomysł stworzenia wspólnego standardu, specyfikację języka Javascript przekazano do Europejskiego Stowarzyszenia Producentów Komputerów, tak powstał język **ECMAScript**.

ECMAScript to de facto międzynarodowy standard języka JavaScript, stanowi on istotny krok na drodze do większej jednolitości języka JavaScript. Jednak w języku JavaScript ciągle znajdują się elementy, które nie są częścią specyfikacji ECMAScript, częściowo po to, aby zachować zgodność ze starszymi przeglądarkami. Najnowsza specyfikacja języka ECMAScript to "3r'd Edition - December 1999". Możemy ją ściągnąć ze strony: http://www.ecma-international.org/.

2 Co to jest JavaScript?

Język JavaScript możemy określić jako wieloplatformowy, obiektowy język skryptowy.

Język wieloplatformowy. Mówiąc, że język JavaScript jest wieloplatformowy mamy na myśli fakt, że kod w nim zapisany będzie w większości przypadków działał na różnych systemach operacyjnych (MS Windows, Linux, Mac OS i in.), dając ten sam wynik.

Język obiektowy. Język Javascript traktuje dane zapisane na stronie jako obiekty o hierarchicznej strukturze. Wykorzystywany jest standard W3C DOM (*World Wide Web Consortium Document Object Model* - Obiektowy Model Dokumentu Konsorcjum WWW). Standard ten określa w jaki sposób można uzyskać dostęp do elementów dokumentów HTML z poziomu języka JavaScript i jak wykonywać na nich operacje, co umożliwia łatwiejszą organizację stron WWW.

Język skryptowy. Języki skryptowe są interpretowane, oznacza to, że poddawane są analizie za pomocą interpretera. Jest to narzędzie wbudowane w przeglądarkę WWW. Interpreter analizuje kod za każdym razem, kiedy go uruchamiamy. Tego typu języki działają zazwyczaj wewnątrz innego programu lub aplikacji, np. przeglądarki WWW. Typowy skrypt

JavaScript jest zawarty wewnątrz strony WWW napisanej w języku HTML lub XHTML. Cechą wyróżniającą języki skryptowe jest to, że wymagają one zapisania znacznie mniejszej ilości kodu niż w przypadku pisania niezależnego programu. W przypadku JavaScript-u dzieje się tak dlatego, że przeglądarka WWW zawiera wiele użytecznych funkcji obsługujących ten język. Skrypty są więc łatwiejsze do napisania, ale są wykonywane wolniej niż skompilowany kod.

Java i JavaScript: różnice i podobieństwa:

JavaScript	**Java**
Interpretowany przez klienta	kompilowany do tzw. b-kodu wykonywanego potem za pomocą wirtualnej maszyny Javy na komputerze klienta
oparty na predefiniowanych obiektach	zorientowany obiektowo z obsługą wszystkich mechanizmów obiektowości
kod programu zagnieżdżony w kodzie HTML	kod programu jest niezależny od kodu HTML i znajduje się w oddzielnych plikach
zmienne nie muszą być deklarowane przed użyciem	zmienne muszą być deklarowane przed użyciem
nie ma możliwości zapisu na dysk twardy	aplety (w przeciwieństwie do aplikacji) nie mają możliwości zapisu na dysk twardy

3 Do czego służy JavaScript?

Język JavaScript na wiele zastosowań. Podstawowym jest ulepszanie wyglądu i działania stron WWW. Umożliwia on wprowadzenie do statycznych stron efekt ruchu oraz elementów interaktywnych. Możemy również w ten sposób zaimplementować obsługę błędów dla danych wprowadzanych do formularzy HTML.

Wykonywanie zadań po stronie klienta. Kiedy używamy tylko technologii po stronie serwera (*CGI - Common Gateway Interface*, *ASP - Active Server Pages*, *JSP - Java Server Pages* czy też *PHP - PHP: Hypertext Preprocessor*), komputer-klient często bezczynnie oczekuje na załadowanie strony, podczas gdy serwer po drugiej stronie ledwo nadąża obsługiwać wszystkie żądania. Oczywistym rozwiązaniem problemu jest wykonywanie przynajmniej części zadań na komputerze-kliencie. Pierwszą korzyścią, jaką uzyskamy po przesunięciu pewnej części obciążenia na stronę klienta jest zmniejszenie konieczności częstego ładowania strony. Na przykład, jeśli sprawdzenie poprawności danych wprowadzanych w formularzu HTML nastąpi po stronie klienta, z wykorzystaniem skryptów JavaScript, unikniemy całkowicie opóźnień sieciowych – przynajmniej do czasu, kiedy dane będą sprawdzone i przygotowane do przesłania na serwer w celu dalszego przetwarzania.

Ważną zaletą wykorzystywania skryptów po stronie klienta jest uzyskanie możliwości programowania w obrębie samej strony WWW. W efekcie, zyskujemy możliwość tworzenia dynamicznych stron WWW odpowiadających na działania użytkowników, którzy przeglądają stronę i wykonują na niej jakieś działania.

Oczywiście, w rzeczywistości komputer-klient nie może wykonywać całego przetwarzania. We wszystkich usługach wykorzystujących technologię WWW konieczne jest przesłanie danych na serwer w celu ich przechowywania lub dalszego przetwarzania. W praktyce więc wykorzystuje się zarówno technologie po stronie serwera, jak i po stronie klienta. Najlepsze efekty uzyskuje się poprzez równomierne rozłożenie obciążenia pomiędzy serwer a komputery klienckie.

4 Podstawowe informacje

Rozróżnianie małych i wielkich liter. W języku JavaScript wielkość liter ma znaczenie. Stosując zatem wielkie i małe litery w nazwach zmiennych, funkcji i stałych musimy być konsekwentni.

Średniki. Dobrą praktyką jest zwyczajowe kończenie każdej instrukcji JavaScript średnikiem. Nie stanowi to wymogu, w rzeczywistości potrzebne jest tylko wtedy, gdy chcemy umieścić co najmniej dwie instrukcje w jednym wierszu.

Gdzie umieszczamy kod JavaScript? Istnieją trzy miejsca, gdzie możemy umieścić skrypty JavaScript na naszych stronach WWW:

- w bloku wpisywanym wewnątrz kodu HTML,
- w oddzielnym pliku zawierającym kod JavaScript,
- w obrębie znacznika HTML.

Blok skryptu. Jest to fragment kodu JavaScript otoczony parą znaczników HTML `<script>` oraz `</script>`.

```
<html>
<head>
<title>Mój pierwszy skrypt w JavaScript!</title>
</head>
<body>
To jest normalny dokument HTML. <br>
<script language="javascript">
    document.write("To jest JavaScript!")
```

```
</script>
I znowu dokument HTML.
</body>
</html>
```

Zewnętrzne pliki skryptów JavaScript. W celu załączenia skryptu z zewnętrznego pliku należy do znacznika `<script>` dodać atrybut **src**.

```
<html>
<head>
<title>Mój pierwszy skrypt w JavaScript!</title>
</head>
<body> To jest normalny dokument HTML. <br>
<script src="ścieżka/do/pliku/nazwapliku.js"></script>
I znowu dokument HTML.
</body>
</html>
```

W języku XHTML istnieje opcja pozwalająca na użycie pustego znacznika `<script>`. Ma on postać `<script />`. Niestety, w niektórych przeglądarkach użycie pustego znacznika powoduje błąd, więc bezpieczniej jest używać pary znaczników `<script></script>`.

W obrębie otwierającego znacznika HTML. Umieszczenie kodu JavaScript w obrębie otwierającego znacznika HTML, to przypadek specjalny. Jedynymi konstrukcjami języka JavaScript, które wykorzystujemy w ten sposób, są procedury obsługi zdarzeń (np. zdarzenie onMouseOver). Do tego aspektu powrócimy przy omawianiu tych procedur.

Określenie języka skryptowego. Do niedawna atrybut **language** był zatwierdzonym sposobem informowania przeglądarki, że skrypt napisano w języku JavaScript. W tym przypadku przyjmował on wartość "javascript". Atrybut ten niestety nie znajduje się w specyfikacji najnowszych wersji języków HTML i XHTML. W specyfikacji języków HTML 4.01 oraz XHTML 1.0 wewnątrz otwierającego znacznika `<script>` występuje atrybut **type**. Jeśli korzystamy z języka JavaScript atrybut ten powinien przyjąć wartość **"text/javascript"**. Aby uzyskać maksymalną zgodność, najlepiej wykorzystywać zarówno atrybut language jak i type. Dzięki temu nasz skrypt będzie można wykorzystać w starych przeglądarkach jak również w przyszłości nie trzeba będzie zmieniać kodu i dodawać atrybutu type.

```
<script language="javascript" type="text/javascript">
// Tu znajduje się
// kod skryptu
</script>
```

Stare przeglądarki. Ciągle istnieją przeglądarki, które nie rozpoznają znacznika `<script>`. Aby zapobiec wyświetlaniu w takim przypadku kodu w oknie przeglądarki powinniśmy używać następującej składni:

```
<html>
<head>
<title>Ukrywanie skryptów za pomocą komentarzy.</title>
<script language="javascript" >
<!-- komentarz w języku HTML

Tu będzie kod JavaScript ...
//--> zamykamy komentarz w języku HTML
</script>
</head>
<body>
</body>
</html>
```

Komentowanie kodu. Aby utworzyć komentarz obejmujący pojedynczy wiersz, wystarczy na jego początku wprowadzić dwa kolejne ukośniki (//). Wtedy interpreter JavaScript zignoruje wszystkie znaki w tym wierszu. Aby rozpocząć komentarz, który obejmuje kilka wierszy, wprowadzamy znaki ukośnik i gwiazdka (/*) na jego początku i znaki gwiazdka i ukośnik (*/) na jego końcu. Tego rodzaju komentarzy nie powinniśmy zagnieżdżać.

```
//Ten cały wiersz jest komentarzem
var x=10 //ten komentarz wprowadzono po fragmencie kodu
/* Cały
tekst znajdujący się w takim bloku
jest komentarzem */
```

5 Typy danych

Język JavaScript jest językiem o dynamicznej i słabej kontroli typów. Słaba kontrola typów oznacza, że podczas wprowadzania elementu w skrypcie nie musimy deklarować jego typu. Dynamiczna kontrola typów pozwala na zmianę typu elementów podczas wykonywania skryptu.

Nie oznacza to jednak, że w przypadku języka JavaScript możemy zignorować typy danych. Jest on po prostu bardziej elastyczny pod tym względem od niektórych innych języków.

Musimy zdawać sobie sprawę, w jaki sposób interpretowane są dane w tym języku. Na przykład zapis

```
10 + 0
```

będzie zinterpretowany przez JavaScript jako liczba 10, natomiast

```
10 + "0"
```

interpreter języka JavaScript przekształci w ciąg "100".

Liczby. W języku JavaScript różne rodzaje liczb nie są rozróżniane, mogą być one interpretowane jako całkowite (np. liczba 7) lub zmiennoprzecinkowe (np. 1,55). Język JavaScript obsługuje dodatnie i ujemne liczby w zakresie od -2^{1024} do 2^{1024}, co w przybliżeniu odpowiada zakresowi -10^{307} do 10^{307}.

Oprócz liczb dziesiętnych, w języku JavaScript obsługuje też liczby ósemkowe i szesnastkowe. Liczby te mogą się np. przydać do obsługi kolorów obiektów, które jak wiemy w HTML-u czy XHTML-u przedstawia się powszechnie w formie szesnastkowej. Jeśli nie określimy jawnie, że wprowadzona liczba jest liczbą ósemkową lub szesnastkową, to język JavaScript będzie ją interpretował jako liczbę dziesiętną. Aby oznaczyć liczbę jako ósemkową, wystarczy rozpocząć ją od cyfry 0. Aby oznaczyć liczbę jako szesnastkową zaczynamy ją od znaków "0x". Warto zwrócić uwagę na fakt, że wynikiem przetwarzania liczb ósemkowych i szesnastkowych będą liczby dziesiętne. Na przykład

```
alert(010+010);
```

wyświetli 16.

Istnieją trzy specjalne wartości liczbowe. Są to: **infinity**, **-infinity** i **NaN**. Wynik równy nieskończoności **infinity** lub minus nieskończoności **-infinity** otrzymamy, jeśli liczba przekroczy maksymalną wartość obsługiwaną przez JavaScript lub jeśli spróbujemy podzielić liczbę przez zero. Przy czym dostaniemy **infinity**, jeśli dzielnik był liczbą dodatnią, a **-infinity** jeśli był on liczbą ujemną.

Wartość **NaN** jest skrótem *Not a Number* - to nie liczba. Wartość tę uzyskamy jeśli wykonamy nieprawidłowe działanie z liczbą. Na przykład

```
alert(10/"kot");
```

zwróci wynik NaN.

Typ Boolean. Dane typu Boolean przyjmują jedną z dwóch wartości: **true** lub **false**.

Ciągi znaków. W języku JavaScript do oznaczania ciągów znaków możemy użyć zarówno cudzysłowów, jak i apostrofów. Ciągi *"Jestem ciągiem znaków"* oraz *'Jestem ciągiem znaków'*, to równoważne sposoby oznaczania danych typu string. Musimy pamiętać o tym, by początek ciągu był oznaczony tym samym znakiem, co koniec. Np.

```
"Jestem ciągiem znaków'
```

spowoduje powstanie błędu. Jeśli chcemy w ciągu umieścić znak, którym go oznaczamy, musimy ten znak poprzedzić znakiem backslash (\) Np.

```
'it\'s five o\'clock!'
```

Znaki poprzedzone znakiem backslash są tzw. znakami sterującymi. Oto niektóre z nich:

\b	znak backspace
\n	nowy wiersz
\r	powrót karetki
\t	tabulacja
\'	apostrof
\"	cudzysłów
\\	znak backslash
\xNN	znak według kodowania Latin 1 (NN - szesnastkowy kod znaku)
\uNNNN	znak według kodowania Unicode (NNNN - szesnastkowy kod znaku)

6 Przydatne funkcje

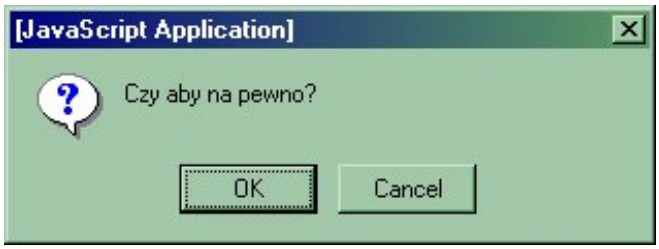
Funkcja alert(). Wyświetla okno ostrzeżenia na naszej stronie WWW, jako argument podajemy tekst, który będzie wyświetlonym komunikatem.

```
alert("Witam wszystkich!");
```

Pamiętajmy, że argument jest łańcuchem znaków, więc powinien być otoczony apostrofami lub cudzysłowami. Jednak w przypadku, gdy argument to liczba, znaki te możemy opuścić.

Funkcja confirm(). Jest to funkcja nieco bardziej zaawansowana niż `alert()`. Wyświetla okno z komunikatem i przyciskami Ok i Cancel. Po wybraniu Ok zwraca wartość `true`, a po wybraniu Cancel wartość `false`.

```
alert(confirm("Czy aby na pewno?"));
```

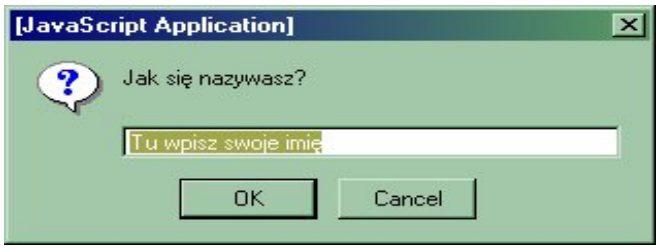


Powyższy przykład wyświetli false jeśli wybierzemy Cancel i true jeśli wybierzemy Ok.

Funkcja prompt(). Działanie tej funkcji polega na wyświetleniu prośby o wprowadzenie tekstu, który może być dalej wykorzystany przez skrypt. Przyjmuje ona dwa argumenty typu łańcuchowego, pierwszy jest komunikatem wyświetlonym w oknie, a drugi to domyślna wartość wstawiona w polu edycji. Jeśli wciśniemy Ok, to zwrócony będzie wpisany tekst, a jeśli wciśniemy Cancel, to funkcja zwróci wartość false.

```
alert("Witam Cię"+prompt("Jak się nazywasz?","Tu wpisz swoje imię"));
```

Powyższy fragment kodu poprosi nas o podanie imienia, a następnie wyświetli ciąg znaków "Witam Cię podane imię".



7 Praca z danymi

Zmienne. Nazwy zmiennych mogą składać się z liter i cyfr oraz znaku podkreślenia i znaku dolara. Pierwszym znakiem nazwy zmiennej nie może być cyfra. Aby utworzyć zmienna mojaZmienna bez definiowania jej wartości, zapiszemy następującą instrukcję

```
var mojaZmienna;
```

Natomiast, aby utworzyć zmienną i nadać jej od razu wartość używamy następującego kodu:

```
var mojaZmienna="Ciąg znaków zapisany w zmiennej";
```

Uwaga, możemy pominąć słowo kluczowe **var** jeśli w tym samym wierszu, w którym wpisujemy nazwę zmiennej, przypisujemy do niej również wartość.

Stałe. Stałe pojawiły się w wersji 1.5 języka JavaScript. Stałe definiujemy za pomocą słowa kluczowego **const** i ich wartość nie może być zmieniona.

```
const STALA=10; alert(STALA);
```

Nazwy stałych muszą spełniać te same warunki, co nazwy zmiennych, zwyczajowo nazwy stałych pisze się z dużej litery.

Operatory.

Operatory arytmetyczne:

Symbol	Składnia	Opis
-	x-y	Wykonuje operację odejmowania
-	-x	Wykonuje negację zmiennej
%	x%y	Zwraca resztę z dzielenia pierwszej zmiennej przez drugą (dzielenie modulo)
*	x*y	Wykonuje operację mnożenia
/	x/y	Wykonuje operację dzielenia
?:	x?:y	Sprawdza prawdziwość warunku
+	x+y	Wykonuje operację dodawania

Operatory inkrementacji i dekrementacji:

Symbol	Składnia	Opis
++	x++	Zwraca wartość x i zwiększa x o 1
++	++x	Zwiększa x o 1 i zwraca wartość x
--	x--	Zwraca wartość x i zmniejsza x o 1
--	--x	Zmniejsza x o 1 i zwraca wartość x

Operatory porównania:

Symbol	Składnia	Opis
!=	x!=y	Zwraca true, jeśli zmienne nie są równe
<	x<y	Zwraca true, jeśli pierwsza zmienna jest mniejsza niż druga
<=	x<=y	Zwraca true, jeśli pierwsza zmienna jest mniejsza niż druga lub jej równa
==	x==y	Zwraca true, jeśli zmienne są równe
>	x>y	Zwraca true, jeśli pierwsza zmienna jest większa niż druga
>=	x>=y	Zwraca true, jeśli pierwsza zmienna jest większa niż druga lub jej równa
===	x===y	Zwraca true, jeśli zmienne są równe i tych samych typów
!==	x!==y	Zwraca true, jeśli zmienne są różne lub różnych typów

Operatory logiczne:

Symbol	Składnia	Opis
!	!x	Ten operator logiczny neguje wyrażenie
&&	x&&y	Operator logiczny AND zwraca true, jeśli obie zmienne są prawdziwe (true)
||	x||y	Operator logiczny OR zwraca true, jeśli co najmniej jedna ze zmiennych jest prawdziwa (true)

Operatory przypisania:

Symbol	Składnia	Opis
%=	x%=y	Wykonuje przypisanie x=x%y
*=	x*=y	Wykonuje przypisanie x=x*y
/=	x/=y	Wykonuje przypisanie x=x/y
+=	x+=y	Wykonuje przypisanie x=x+y
-=	x-=y	Wykonuje przypisanie x=x-y
=	x=y	Przypisuje wartość y do zmiennej x

Operatory bitowe:

Symbol	Składnia	Opis
<	x<y	bitowa operacja AND, która wyświetla 1, jeśli obie zmienne wynoszą 1
^	x^y	bitowa operacja XOR, która wyświetla 1, jeśli jedna ze zmiennych (ale nie obie jednocześnie) wynosi 1
|	x|y	bitowa operacja OR, która wyświetla 1, jeśli jedna ze zmiennych wynosi 1
<<	x<<y	przesunięcie bitów w lewo o podaną liczbę miejsc
>>	x>>y	przesunięcie bitów w prawo o podaną liczbę miejsc

Przypisania bitowe:

Symbol	Składnia	Opis
<=	x<=y	Wykonuje przypisanie x=x<y
^=	x^=y	Wykonuje przypisanie x=x^y
|=	x|=y	Wykonuje przypisanie x=x|y
<<=	x<<=y	Wykonuje przypisanie x=x<
>>=	x>>=y	Wykonuje przypisanie x=x>>y
>>>=	x>>>=y	Wykonuje przypisanie x=x>>>y

Operator łańcuchowy (konkatenacji):

Symbol	Składnia	Opis
+	"text1"+"text2"	Łączy dwa ciągi znaków w jeden

Zakres zmiennych. W języku JavaScript mamy dwa rodzaje zasięgu: zasięg globalny i lokalny.

Zmienna globalna jest dostępna w dowolnym miejscu skryptu w obrębie tego samego dokumentu HTML. Ponadto, zmiennych globalnych można użyć w skryptach działających w innej ramce lub w innym oknie przeglądarki. Wszystkie zmienne zadeklarowane w bloku skryptów, ale na zewnątrz funkcji, mają zakres globalny.

Jeśli zmienna jest deklarowana w funkcji, to jej zakres zależy od tego czy do jej zadeklarowania użyliśmy słowa kluczowego *var* i tak, jeśli wewnątrz funkcji zadeklarujemy zmienną bez słowa kluczowego *var*, jej zakres będzie globalny, jeśli zaś deklarujemy zmienną ze słowem kluczowym *var*, to jej zakres będzie lokalny. Przeanalizuj następujący kod:

```
var zmiennaTestowa="To jest zmienna Globalna!<br>";
function testowanieZakresu() {
    document.write(zmiennaTestowa);
    var zmiennaTestowa1="To jest zmienna Lokalna!<br>";
    document.write(zmiennaTestowa1); }

function testowanieZakresu1() {
    document.write(zmiennaTestowa);
    zmiennaTestowa="To jest inna zmienna Globalna!<br>";
    document.write(zmiennaTestowa);
}
testowanieZakresu();
testowanieZakresu1();
```

8 Podstawy składni języka

Struktury warunkowe. Instrukcja `if` (jeżeli) jest jednym z najważniejszych mechanizmów dostępnych w wielu językach z JavaScript włącznie. Pozwala na wyodrębnienie fragmentu kodu, który zostanie wykonany pod określonym warunkiem. Instrukcja `if` w JavaScript jest bardzo podobna do swojego odpowiednika z języka C:

```
if (warunek)
            wyrażenie
```

Jeśli warunek jest równy `true` wyrażenie zostanie wykonane; w przeciwnym razie zostanie pominięte. Często potrzeba, aby więcej niż jedna instrukcja była wykonana pod przyjętym warunkiem. Nie ma oczywiście potrzeby umieszczać każdej z tych instrukcji w osobnej strukturze `if`. Zamiast tego należy zgrupować te instrukcje za pomocą instrukcji grupującej. Na przykład poniższy kod wyświetli `a jest większe niż b`, jeżeli `a` jest większe niż `b`, i przypisze wartość `a` do `b`:

```
if (a > b) {
    alert("a jest większe niż b");
    b = a;}
```

Często potrzeba wykonać jedną instrukcję programu, gdy pewien warunek jest spełniony, lub inną, gdy ten sam warunek nie jest spełniony. Służy do tego instrukcja `else`. `else` rozszerza możliwości instrukcji `if` do sytuacji kiedy warunek opisany przy instrukcji `if` ma wartość `false`. Na przykład poniższy kod wyświetli `a jest większe niż b` jeżeli `a` jest większe niż `b`, lub `a NIE jest większe niż b` w przeciwnym przypadku:

```
if (a > b) { alert("a jest większe niż b"); }
   else {    alert("a NIE jest większe niż b"); }
```

Kod zawarty w instrukcji `else` wykonywany jest tylko wówczas, kiedy wyrażenie logiczne przy instrukcji `if` ma wartość `false`.

Instrukcja `switch` jest podobna do serii instrukcji `if` z warunkiem na to samo wyrażenie. W wielu przypadkach istnieje potrzeba porównania jednej zmiennej (lub wyrażenia) z wieloma różnymi wartościami i wykonania różnych fragmentów kodu, w zależności od wyniku porównania tej zmiennej z różnymi wartościami. Do tego właśnie służy instrukcja switch.

```
switch (i) {
    case 0: alert("i jest równe 0");
            break;
    case 1: alert("i jest równe 1");
            break;
    case 2: alert("i jest równe 2");
        break;
    default: alert("nie wiem ile jest równe i");
}
```

Instrukcja `switch` jest wykonywana linia po linii (dokładnie wyrażenie po wyrażeniu). Na początku żaden fragment kodu nie jest wykonywany. Dopiero kiedy zostaje odnalezione wyrażenie `case`, którego wartość odpowiada wyrażeniu przy instrukcji `switch`, JavaScript rozpoczyna wykonywanie kodu od miejsca, gdzie znajduje się ta instrukcja `case`. JavaScript

wykonuje instrukcje aż do momentu kiedy blok `switch` się skończy, lub do momentu znalezienia instrukcji `break`. Jeśli nie napiszemy instrukcji `break` na końcu instrukcji w danym wyrażeniu `case` to JavaScript będzie wykonywać dalej instrukcje z następnego wyrażenia `case`. W instrukcji `switch` wartość wyrażenia jest obliczana tylko raz, a następnie jest porównywana z każdym z wyrażeń przy etykiecie `case`. Dlatego, jeśli wyrażenie jest bardziej skomplikowane od zwykłego porównania, `switch` może być szybszy. Specjalną etykietą jest etykieta warunku domyślnego - `default`. Etykieta ta dotyczy sytuacji, w której wyrażenie nie pasowało do wartości przy innych etykietach typu `case`. W instrukcji `switch` ta etykieta powinna być ostatnia z listy.

Instrukcje iteracyjne. Pętla `while` jest najprostszym typem pętli w JavaScript. Zachowuje się ona identycznie jak jej odpowiednik z języka C. Jej podstawowa forma wygląda następująco:

```
while (wyrażenie) instrukcja
```

Znaczenie instrukcji `while` jest bardzo proste. Wykonuje określone instrukcje tak długo, jak długo wyrażenie przy słowie `while` ma wartość `true`. Wartość tego wyrażenia jest sprawdzana za każdym razem na początku wykonywania nowej iteracji pętli, więc jeśli jego wartość zmieni się w trakcie wykonywania instrukcji, wykonanie całej pętli nie skończy się do momentu zakończenia całej iteracji. Jedna iteracja jest to jednokrotne wykonanie wszystkich instrukcji w pętli. Jeśli wyrażenie logiczne ma wartość `false` już na samym początku, instrukcje wewnątrz pętli nie będą w ogóle wykonane. Podobnie jak w instrukcji `if`, w pętli `while` można grupować instrukcje za pomocą nawiasów klamrowych.

Poniższe przykłady są identyczne i obydwa wyświetlają liczby od 1 do 10:

```
var i = 1;
while (i <= 10){
    alert(i++)}
```

Pętla `do...while` zachowuje się bardzo podobnie do pętli `while`, z wyjątkiem tego, że wartość wyrażenia logicznego sprawdzana jest na końcu iteracji, a nie na początku. Wynikającą z tego główną różnicą jest to, że pierwsza iteracja w pętli `do...while` na pewno zostanie wykonana (gdyż wyrażenie logiczne będzie sprawdzone dopiero na koniec iteracji). Natomiast w pętli `while`, gdzie wyrażenie logiczne jest sprawdzane na początku iteracji, może dojść do sytuacji, że pętla w ogóle nie zostanie wykonana, jeśli to wyrażenie będzie miało wartość `false` od początku.

```
var i = 0;
do {
```

```
   alert(i)}
while ($i>0);
```

Powyższa pętla zostanie wykonana tylko raz, gdyż po pierwszej iteracji, wartość wyrażenia logicznego wynosić będzie `false` i pętla zostanie zakończona.

Pętla `for` jest najbardziej skomplikowanym rodzajem pętli w JavaScript. Zachowuje się identycznie jak jej odpowiedniki z C. Jej składnia wygląda następująco:

```
for (wyrażenie1; wyrażenie2; wyrażenie3) instrukcje
```

`wyrażenie1` jest wykonywane tylko raz, na początku pętli. Na początku każdej nowej iteracji, obliczana jest wartość logiczna wyrażenia `wyrażenie2`. Jeśli wynikiem obliczenia jest `true`, to pętla kontynuuje i następuje wykonanie instrukcji umieszczonych w pętli. Jeśli jednak wyrażenie ma wartość false, to wykonanie pętli zostaje przerwane. Na końcu każdej iteracji zostaje wykonane `wyrażenie3`.

Poniższy przykład wyświetla liczby od 1 do 10:

```
for (var i = 1; i <= 10; i++) {
    alert(i);}
```

`break` kończy wykonywanie aktualnej instrukcji `for`, `while`, `do...while` lub `switch`.

`continue` używane jest wewnątrz instrukcji pętli do przerwania wykonywania danej iteracji pętli i rozpoczęcia nowej iteracji.

9 Funkcje użytkownika

Funkcje w języka JavaScript tworzymy podobnie jak w innych językach programowania. Używamy słowa kluczowego *function*, po którym występuje identyfikator funkcji. Identyfikator ten podlega tym samym regułom, co identyfikator zmiennych.

```
function nazwaFunkcji(lista_argumentow) {
 kod
}
```

Zwracanie wartości przez funkcje. Aby uzyskać wyniki przetwarzania danych przez funkcje, używamy instrukcji *return*.

```
function obliczModul(liczba) {
    return (liczba>=0)?liczba:-liczba;
}
var naszaLiczba=prompt("Podaj liczbę","");
alert("Wartość bezwzględna z liczby "+naszaLiczba+
      " to:"+obliczModul(naszaLiczba));
```

10 Predefiniowane funkcje globalne

Funkcje te można wywoływać w dowolnej części skryptu, dokładnie w ten sam sposób, w jaki wywołuje się funkcje użytkownika. Do tej pory mieliśmy już do czynienia z niektórymi predefiniowanymi funkcjami, takimi jak `alert()`, `confirm()` oraz `prompt()`. Funkcje te są wykorzystywane od tak dawna, że obsługuje je większość przeglądarek, choć w zasadzie nie należą one do rdzenia języka JavaScript. Oznacza to, że w innych środowiskach niż przeglądarki WWW mogą, lecz nie muszą być dostępne. Oto lista predefiniowanych funkcji JavaScript należących do rdzenia języka:

- `decodeURI()`
- `decodeURIComponent()`
- `encodeURI()`
- `encodeURIComponent()`
- `escape()`
- `unescape()`
- `eval()`
- `isFinite()`
- `isNaN()`
- `Number()`
- `parseFloat()`
- `toString()`
- `watch()`
- `unwatch()`

Funkcje `watch()` oraz `unwatch()` wspomagają uruchamianie diagnostyczne dla właściwości obiektów. Funkcje te omówimy w dalszej części wykładu.

Kodowanie i dekodowanie adresów URI. Jednolite identyfikatory zasobów (*Uniform Resource Identifier*) - to najbardziej ogólna nazwa dla adresów wykorzystywanych w celu uzyskania dostępu do plików w sieci. "Zasób" określa jednostkę informacji, może to być np.

strona WWW. Adresów URI używamy również do przesyłania informacji, które mają być wykorzystane na tej stronie.

Kiedy do przesyłania ciągu zapytania wykorzystujemy formularz, w którym stosujemy metodę `HTTP GET`, przeglądarka automatycznie przekształci znaki niedozwolone w adresie URI na zakodowane wartości szesnastkowe. Aby tego dokonać za pomocą języka JavaScript, należy wykorzystać funkcje wbudowane.

Początkowo do kodowania i dekodowania adresów URI w języku JavaScript stosowano funkcje `escape()` i `unescape()`.

```
var ciagZnakow="To kosztuje 5$";
var zakodowanyCiag=escape(ciagZnakow);
var zdekodownayCiag=unescape(zakodowanyCiag);
alert(zakodowanyCiag);
alert(zdekodownayCiag);
```

Funkcje `escape()` i `unescape()` kodują również znaki, które nie muszą być kodowane i dla różnych przeglądarek kodowane są różne znaki. W związku z tym w wydaniu 3 ECMAScript zdefiniowano cztery nowe funkcje: `decodeURI()`, `decodeURIComponent()`, `encodeURI()`, `encodeURIComponent()`, które miały zastąpić `escape()` i `unescape()`.

Funkcja `encodeURI()` powoduje zakodowanie wszystkich znaków, za wyjątkiem znaków alfanumerycznych oraz następujących:

! # $ & ' ( ) * + , - . / : ; = ? @ _ ~

Funkcja `encodeURIComponent()` działa podobnie. Koduje wszystkie znaki kodowane przez `encodeURI()`, a oprócz nich:

$ & + , / : ; = ? @

Funkcja ta nie koduje znaków alfanumerycznych oraz znaków:

! ' ( ) * - . _ ~

Powodem istnienia dwóch różnych funkcji służących do kodowania jest fakt, że czasem trzeba zakodować cały adres URI, łącznie z takimi elementami jak "*http://*", natomiast innym razem wystarczy, jeśli zakodujemy tylko ciąg zapytania.

Odpowiednikami tych funkcji, służącymi do dekodowania, są funkcje `decodeURI()` oraz `decodeURIComponent()`. Działanie funkcji `decodeURIComponent()` jest podobne do `unescape()`. Powoduje ona dekodowanie wszystkich wcześniej zakodowanych znaków. Natomiast funkcja `decodeURI()` pozostawia bez zmian następujące znaki:

$ & + , / : ; = ? @

Zamiana ciągu znaków na kod. Do tego celu służy funkcja `eval()`, wymusza ona zmianę ciągu znaków na kod JavaScript i wykonanie go.

```
var mojKod="alert(\"to był ciąg znaków\")";
//uwaga, możemy również zapisać
// var mojKod="alert('to był ciąg znaków')";
eval(mojKod);
```

Funkcje arytmetyczne.

Funkcja `isFinite()` sprawdza, czy argument nie ma jednej ze specjalnych wartości `Infinity` lub `-Infinity`. Jeżeli liczba jest skończona (i można przeprowadzić obliczenia), funkcja zwraca wartość `true`, w przeciwnym razie zwraca wartość `false`.

Funkcja `isNaN()` pozwala na określenie, czy wykorzystywana przez nas dana to liczba. Zwraca wartość `false`, jeżeli podana wartość jest liczbą oraz `true` w przeciwnym przypadku.

> Zaskakującym jest fakt, że wartość `NaN` nie jest równoważna sama sobie.
> Mamy więc:
> `NaN == NaN` //przyjmuje wartość false
> `NaN === NaN` //przyjmuje wartość false
> Ale:
> `isNaN(NaN)` //przyjmuje wartość true

Funkcja `isNaN()` przydaje się wówczas, gdy pobieramy dane liczbowe od użytkownika korzystającego z naszej strony WWW.

```
function obliczModul() {
    do
        var liczba=prompt("Proszę podać liczbę","");
    while (isNaN(liczba));
    alert((liczba>=0)?liczba:-liczba);
}
obliczModul()
```

Konwersja pomiędzy ciągami znaków a liczbami.

Funkcje `Number()` i `parseFloat()` przyjmują jeden argument i przetwarzają go w podobny sposób. Jeśli argument jest liczbą zwracają go w niezmienionej postaci, natomiast gdy jest to ciąg znaków, wykonają próbę przekształcenia go na liczbowy typ danych.

Funkcja `parseFloat()` użyta do wartości innych niż liczba lub ciąg znaków, w którym znajdują się znaki i cyfry, zawsze przyjmie wartość `NaN`. Z kolei funkcja `Number()` zamienia wartości `false` i `null` na wartość `0`, wartość `true` na `1`, jedynie wartość `undefined` będzie przekształcona na wartość `NaN`.

Jeżeli ciąg znaków rozpoczyna się od cyfr, a po nich występują inne znaki, to funkcja `Number()` zwróci wartość `NaN`. W takim przypadku funkcja `parseFloat()` usunie nienumeryczne znaki z końca ciągu i zwraca znaki numeryczne jako liczbę.

```
alert(Number("3.1415 tu nie jest liczbą"));
alert(parseFloat("3.1415 tu jest liczbą"));
```

Jeśli chcemy pozbyć się części ułamkowej, możemy użyć funkcji `parseInt()`. Funkcja ta przekształca argumenty na typ liczbowy, a ponadto zaokrągla je w dół, do najbliższej liczby całkowitej.

```
alert(parseInt("3.1415 tu jest liczbą"));
alert(parseInt("3.99999 też będzie 3"));
```

Funkcja `parseInt()` ma jeszcze inną, przydatną czasem właściwość. Za jej pomocą możemy przekształcać liczby w systemach o podstawie pomiędzy 2 a 36 na liczby dziesiętne. W tym przypadku funkcja przyjmuje dwa argumenty, pierwszy to łańcuch znaków do przetworzenia, a drugi to podstawa.

```
alert(parseInt("31.415 tu jest liczbą",4));
alert(parseInt("1111","2 podstawa"));
alert(parseInt("ff.ff",16));
alert(parseInt("0xf",16));
alert(parseInt("010",8));
alert(parseInt(100,3));
alert(parseInt("100",3));
```

Jeżeli podstawa jest większa niż 10, to pierwszy argument powinien być ciągiem znaków, wynika to z faktu, że w systemach o podstawie większej niż 10 liczby mogą zawierać znaki alfabetu.

Ostatnią funkcją do konwersji jest `toString()`. Działa ona dokładnie odwrotnie do wyżej opisanych. Przekształca liczby na ciągi znaków. Składnia tej funkcji jest nieco inna od funkcji poznanych do tej pory. Funkcja ta na ogół nie przyjmuje argumentów, aby ją wywołać musimy użyć notacji z kropką (ponieważ jest to właściwie metoda obiektu Number).

```
var jakasZmienna=3.141592;
  alert(52+jakasZmienna.toString());//wyświetli 523.141592 a nie 55.141592
```

Funkcja `toString()` ma jeszcze jedno zastosowanie, może ona przekształcać liczby w systemie dziesiętnym na liczbę w systemie o podanej podstawie od 2 do 36 (a więc działa odwrotnie do funkcji `parseInt()`).

```
var x=10,y=0.5;
  alert(x.toString(8));//wyświetli 12
  alert(y.toString(16));//wyświetli 0.8
```

Dzięki funkcjom `parseInt()` oraz `toString()` możemy przekształcać liczby z jednego systemu (o podstawie od 2 do 36) na drugi (też o podstawie od 2 do 36). Przy czym powinniśmy pamiętać, że `parseInt()` usuwa kropkę dziesiętną, dlatego dla liczb, które nie są całkowite dostaniemy wynik przybliżony.

```
function zamianaPodstawy(liczba,podstawa1,podstawa2) {
    var liczba10=parseInt(liczba,podstawa1);
    alert(liczba10.toString(podstawa2));
}
zamianaPodstawy("0xff",16,2);
```

11 JavaScript jako język obiektowy

W języku JavaScript wszystkie elementy dokumentu HTML oraz niektóre elementy przeglądarki są udostępniane w postaci obiektów. W praktyce umożliwia to łatwy dostęp i zmianę właściwości dokumentu. Na przykład możemy w ten sposób pobierać informacje na temat rysunków na stronie WWW, na przykład odczytywać ich szerokości. W ten sam sposób możemy zmieniać zawartość paska statusu.

Wszystkie obiekty JavaScript należą do obiektu **Global**. Kiedy dokument zawierający kod JavaScript jest ładowany lub pobierany do środowiska obsługującego ten język, wszystkie obiekty, właściwości i metody budowane są na podstawie obiektu **Global**. W środowisku przeglądarki, po stronie klienta, obiektem **Global** jest obiekt **window**. Tym samym, kiedy korzystamy z dowolnego innego obiektu JavaScript, robimy to za pośrednictwem obiektu **window**.

```
window.status=prompt("Wprowadź nowy tekst na pasek stanu","");
```

Obiekt **window** posiada kilka obiektów potomnych (czyli jest dla nich obiektem macierzystym). Można je podzielić na trzy grupy:

- obiekt **document**,
- obiekty środowiskowe,
- obiekty rdzenia języka JavaScript.

Dzięki obiektowi **document** możliwe jest uzyskanie dostępu do dokumentów załadowanych w przeglądarce WWW. Obiekt ten zawiera właściwie wszystko to, co znajduje się pomiędzy znacznikami `<html>` i `</html>`.

Standard ustanowiony przez model obiektowy dokumentu (*Document Object Model*) organizacji W3C (WWW Consortium) - w skrócie DOM - opisuje logiczne przypisanie właściwości i metod do obiektów. Standard ten zostanie omówiony w dalszym ciągu.

Obiekty środowiskowe dostarczają nam informacji o przeglądarce oraz środowisku w jakim ona pracuje. Mamy tu cztery obiekty potomne obiektu **window**, które umożliwiają uzyskanie tych wiadomości. Oto ich lista:

- obiekt **location**,
- obiekt **history**,
- obiekt **navigator**,
- obiekt **screen**.

Obiekty **location** i **history** umożliwiają uzyskanie dostępu odpowiednio do bieżącego adresu URI oraz historii przeglądarki. Obiekty **navigator** i **screen** dostarczają informacji o przeglądarce oraz o pozycji okna przeglądarki na ekranie użytkownika.

Rdzeń JavaScript, to część języka, która jest dostępna w każdym środowisku, w którym język jest wykorzystywany. Istnieje 11 obiektów rdzenia języka JavaScript. Oto one:

- obiekt **Array**,
- obiekt **Boolean**,
- obiekt **Date**,
- obiekt **Error**,
- obiekt **Function**,
- obiekt **Global**,
- obiekt **Math**,
- obiekt **Number**,
- obiekt **Object**,
- obiekt **RegExp**,

- obiekt **String**.

W środowisku przeglądarki obiektem **Global** jest oczywiście obiekt **window**.

Eksploracja obiektu. Do tego celu przydatne są operatory `in` i `typeof`. Pierwszy pozwala stwierdzić, czy określona właściwość lub obiekt potomny istnieją, natomiast drugi pozwala określić typ danych lub obiektu, z którym mamy do czynienia.

```
alert("document" in window); //zwróci wartość true
```

Operator `in` może występować w połączeniu z operatorem pętli `for`

```
for (var property in object){
 kod }
```

W każdej iteracji pętli do zmiennej `property` przypisane będą kolejne nazwy właściwości obiektu `object` począwszy od pierwszej, a skończywszy na ostatniej.

Operator `typeof`, w odniesieniu do fragmentu danych, przyjmuje następujące wartości: `boolean`, `function`, `number`, `object`, `string`, `undefined`.

```
alert(typeof window.document); //zwróci wartość object
alert(typeof 5); //zwróci wartość number alert(typeof
document.write); //zwróci wartość function
```

Następujący przykład wyświetla wszystkie właściwości i ich typy obiektu **window.navigator**.

```
var objekt=window.navigator;
document.write("<table border=1>
                <tr><th>Nazwa właściwości</th>
                <th>Typ właściwości</th>
                <th>Wartość</th></tr>");
for(var wlasc in objekt){
document.write("<tr><td>"+wlasc+"</td>");
document.write("<td>"+(typeof objekt[wlasc])+"</td>");
document.write("<td>"+objekt[wlasc]+"</td>");
document.write("</tr>");
}
document.write("</table>");
```

Tworzenie własnych obiektów. Nowy obiekt tworzymy w języku JavaScript przy pomocy funkcji, która nazywa się konstruktorem. Funkcja ta wygląda właściwie tak samo jak każda inna funkcja. Nazwą tej funkcji jest nazwa obiektu, który mamy zamiar utworzyć.

```
function obiektKilent(parametry){ kod konstruktora }
```

Aby utworzyć właściwości obiektu, stosujemy słowo kluczowe `this`. Oznacza ono, że tworzone właściwości odnoszą się do obiektu, który jest tworzony w momencie wywołania konstruktora.

```
function obiektKilent(nazwisko, adres, nr_telefonu, email){
this.nazwisko=nazwisko;
this.adres=adres;
this.telefon=nr_telefonu;
this.adres_email=email;
}
```

Do tworzenia nowego egzemplarza naszego obiektu używamy słowa kluczowego `new`.

```
var nowyKlient=new obiektKlient("Jan Kowalski", "Zakopane,Polska",
                                "88020020","kowal@kowalski.com");
alert("Witaj "+nowyKlient.nazwisko+"!");
```

Możemy również najpierw utworzyć egzemplarz obiektu, a następnie przypisać wartości poszczególnym właściwościom.

```
var innyKlient=new obiektKlient();
  innyKlient.nazwisko="Adaś Nowak";
  innyKlient.adres="Rawa Maz., Polska";
  innyKlient.telefon="604029292";
  innyKlient.adres_email="adnow@nowak.org";
alert("Witaj "+innyKlient.nazwisko+"!");
```

Tworzenie metod obiektu. Metody obiektów tworzymy jak zwyczajne funkcje. Jedyna różnica polega na tym, że możemy korzystać z właściwości obiektu za pomocą operatora `this`.

```
function wyswietlKlienta(){
document.write("<b>Imię i nazwisko klienta: </b>"+this.nazwisko+"<br>");
document.write("<b>Adres:</b>"+this.adres+"<br>");
document.write("<b>Numer telefonu:</b>"+this.telefon+"<br>");
document.write("<b>Adres e-mail:</b>"+this.adres_email+"<br>");
}
```

Funkcja ta nie musi być zdefiniowana przed konstruktorem. Następnie musimy wstawić tą funkcję jako metodę do definicji obiektu, robimy to podobnie, jak w przypadku definiowania właściwości:

```
function obiektKilent(nazwisko, adres, nr_telefonu, email){
this.nazwisko=nazwisko;
this.adres=adres;
this.telefon=nr_telefonu;
this.adres_email=email;
this.wyswietlKlienta=wyswietlKlienta;
}
```

Metoda nie musi mieć tej samej nazwy, co nazwa funkcji. Metody mogą przyjmować argumenty.

```
function noweNazwisko(nazwisko){
    this.nazwisko=nazwisko;
}
```

Łączenie obiektów. Właściwości obiektów mogą być innymi obiektami. Załóżmy, że mamy dwa obiekty, ten który utworzyliśmy poprzednio **obiektKlient** oraz **obiektFirma**, zdefiniowany następująco:

```
function obiektFirma(nazwa,adres,nr_telefonu){
this.nazwa=nazwa;
this.adres=adres;
this.telefon=nr_telefonu;
this.wyswietlFirma=wyswietlFirma;
}
function wyswietlFirma(){
document.write("<b>Nazwa firmy: </b>"+this.nazwa+"<br>");
document.write("<b>Adres: </b>"+this.adres+"<br>");
document.write("<b>Numer telefonu: </b>"+this.telefon+"<br>");
}
```

Teraz tworzymy egzemplarz obiektu **objektFirma** i możemy dodać ten egzemplarz do właściwości obiektu **obiektKlient**:

```
var jakasFirma=new obiektFirma("ArtKomp","Piotrków Tryb.","911113233");
function obiektKilent(nazwisko, adres,nr_telefonu, email){
```

```
this.nazwisko=nazwisko;
this.adres=adres;
this.telefon=nr_telefonu;
this.adres_email=email;
this.wyswietlKlienta=wyswietlKlienta;
this.obiektFirma=jakasFirma;
}
```

Możemy wówczas w egzemplarzach obiektu **obiektKlient** używać metod i właściwości obiektu **objektFirma**:

```
var nowyKlient=new obiektKlient("Jan Kowalski", "Zakopane,Polska",
                     "88020020","kowal@kowalski.com", jakasFirma);
nowyKlient.obiektFirma.wyswietlFirma();
```

12 Obiekt window

Obiekt **window** znajduje się najwyżej w hierarchii obiektów i reprezentuje on okno przeglądarki.

Aby otworzyć nowe okno, używamy metody `open()`, trzeba przy tym określić adres URL i nazwę okna:

```
window.open("http://onet.pl","onet");
```

Jako nazwy można użyć jednej ze specjalnych wartości:

Atrybut	Opis
`_blank`	Otwarcie strony w nowym dokumencie
`_parent`	Otwarcie strony w ramce macierzystej
`_self`	Otwarcie strony w bieżącym oknie lub ramce
`_top`	Otwarcie strony w bieżącym oknie i nadpisanie istniejących ramek

Wygodniej jest wartość zwracaną przez metodę `open()` przypisać do zmiennej, wtedy będziemy mogli odwoływać się do tego okna za pomocą tej zmiennej:

```
var okno=window.open("http://onet.pl","onet");
```

Podczas otwierania nowego okna możemy określić dokładniej jego wygląd. To tego celu służą atrybuty zebrane w trzecim argumencie metody. Oto lista atrybutów:

Atrybut	Opis
`height`	Określa w pikselach wysokość nowego okna
`width`	Określa w pikselach szerokość nowego okna
`left`	Liczba pikseli od lewej krawędzi ekranu do lewej krawędzi okna
`top`	Liczba pikseli od górnej krawędzi ekranu do górnej krawędzi okna
`directories`	Wyświetla pasek Łącza w IE i pasek Personal w Netscape
`location`	Wyświetla pole adresu
`menubar`	Wyświetla pasek menu
`resizable`	Wskazuje, czy użytkownik może zmienić rozmiar okna
`scrollbars`	Wyświetla paski przewijania
`status`	Wyświetla pasek statusu
`toolbar`	Wyświetla pasek narzędzi

```
var okno=window.open("http://onet.pl","onet",
"height=300,width=200,directories=0,location=0,status=1,toolbar=0,menubar=0");
```

Aby zamknąć okno używamy metody `window.close()`:

```
okno.close();
```

Aby zmienić tekst w pasku statusu okna, ustawiamy właściwość `status`:

```
okno.status="To jest nowy tekst w pasku stanu";
```

12.1 Obiekt location

Obiekt **location** zawiera informacje dotyczące adresu URL, z którego załadowano stronę. Jeśli chcemy zmienić ten adres dla danego dokumentu możemy wywołać kod:

```
window.location.href=
    "http://www.iret.com:80/sciezka/docs.html#kotwica?cos=1&inne=2"
```

Poniżej zostały zestawione właściwości obiektu **location** z przykładowymi wartościami dla wyżej zdefiniowanego adresu:

Właściwość	Wartość
protocol	`http:`
hostname	`www.iret.com`
port	`80`
host	`www.iret.com:80`
pathname	`/sciezka/docs.html`
hash	`#kotwica`
href	`http://www.iret.com:80/sciezka/docs.html#kotwica`
search	`?cos=1&inne=2`

Oto lista metod obiektu **location**:

- `assign("URL")` - metoda ta przypisuje do właściwości `href` podany adres URL. Jest to sposób przestarzały i nie zaleca się używania tej metody.
- `reload()` - powoduje odświeżenie strony, jeśli strona ma zostać bezwarunkowo jeszcze raz pobrana ze zdalnego węzła (a nie wczytana z pamięci cache) to jako argument podajemy wartość `true`.
- `replace("URL")` - powoduje zamianę adresu na podany URL, dzięki temu strona nie jest przechowywana w historii przeglądarki. Oznacza to, że jeśli klikniemy przycisk **Wstecz**, strona, na której znajdowało się wywołanie metody `replace()` w rzeczywistości nie jest zapisana w historii i ładowana jest strona poprzednia.

12.2 Obiekt history

Obiekt ten daje nam dostęp do stron, które w danym oknie odwiedziliśmy. Obiekt ten ma jedną właściwość `length`, która przechowuje ilość elementów w historii. Obiekt **history** ma następujące metody:

- `back()` - powoduje załadowanie w oknie poprzednio odwiedzonej strony.
- `forward()` - powoduje załadowanie w oknie następnej strony w historii.
- `go()` - umożliwia przejście do strony zapisanej w historii, za pomocą liczbowego argumentu specyfikujemy pozycję w historii względem bieżącego adresu, np. `window.history.go(-2)` oznacza przejście o dwie pozycje wstecz, `window.history.go(1)` oznacza przejście o jedną pozycję do przodu, a `window.history.go(0)` oznacza "miękkie" odświeżenie strony.

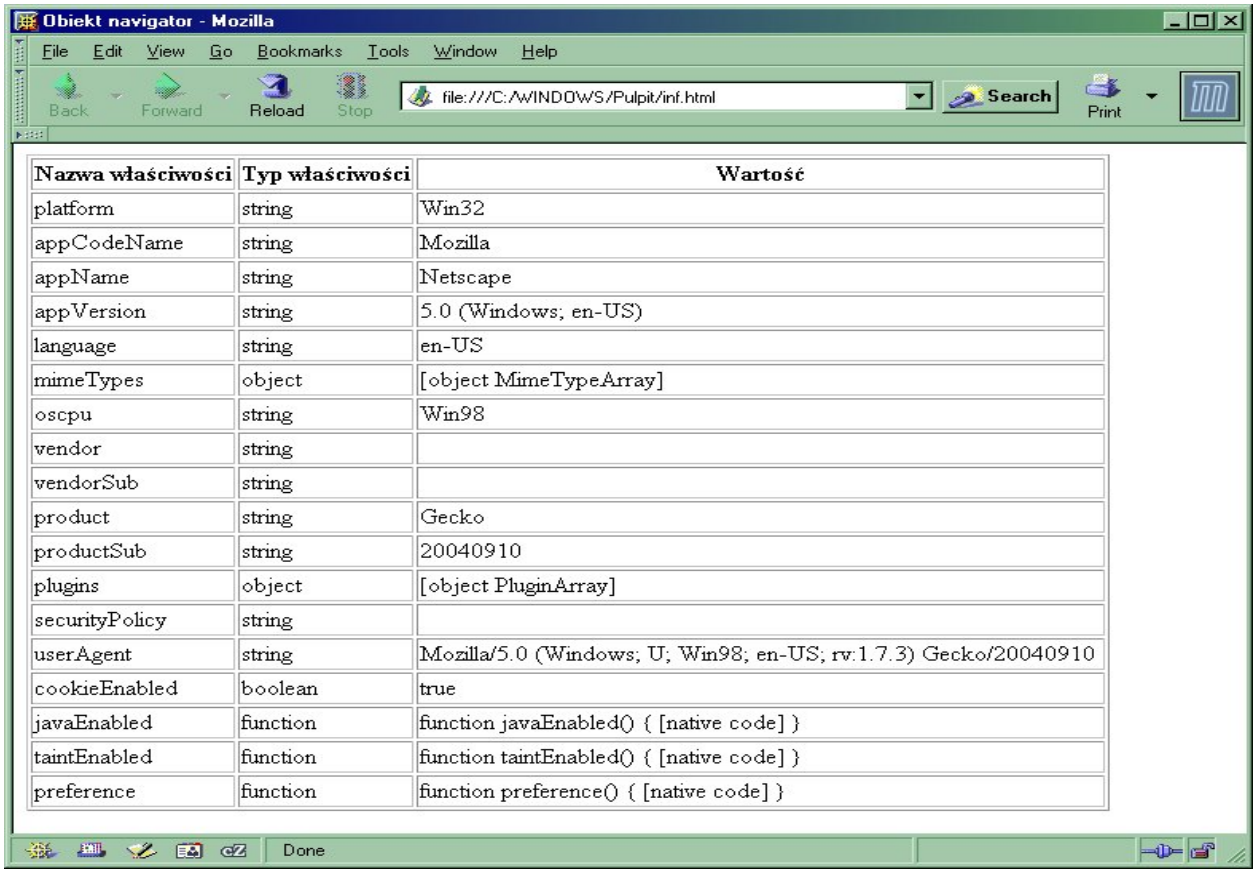
12.3 Obiekt navigator

Obiekt **navigator** jest mechanizmem pozwalającym uzyskać informacje na temat przeglądarki. Obiekt ten ma wiele właściwości, większość z nich jest specyficzna dla konkretnej przeglądarki. Oto ich pełna lista:

- `appCodeName`
- `appMinorVersion`
- `appName`
- `appVersion`
- `browserLanguage`
- `cookieEnabled`
- `cpuClass`
- `language`
- `mimeTypes`
- `onLine`
- `oscpu`
- `platform`
- `plugins`
- `product`
- `productSub`
- `securityPolicy`
- `systemLanguage`
- `userAgent`
- `userLanguage`
- `userProfile`
- `vendor`

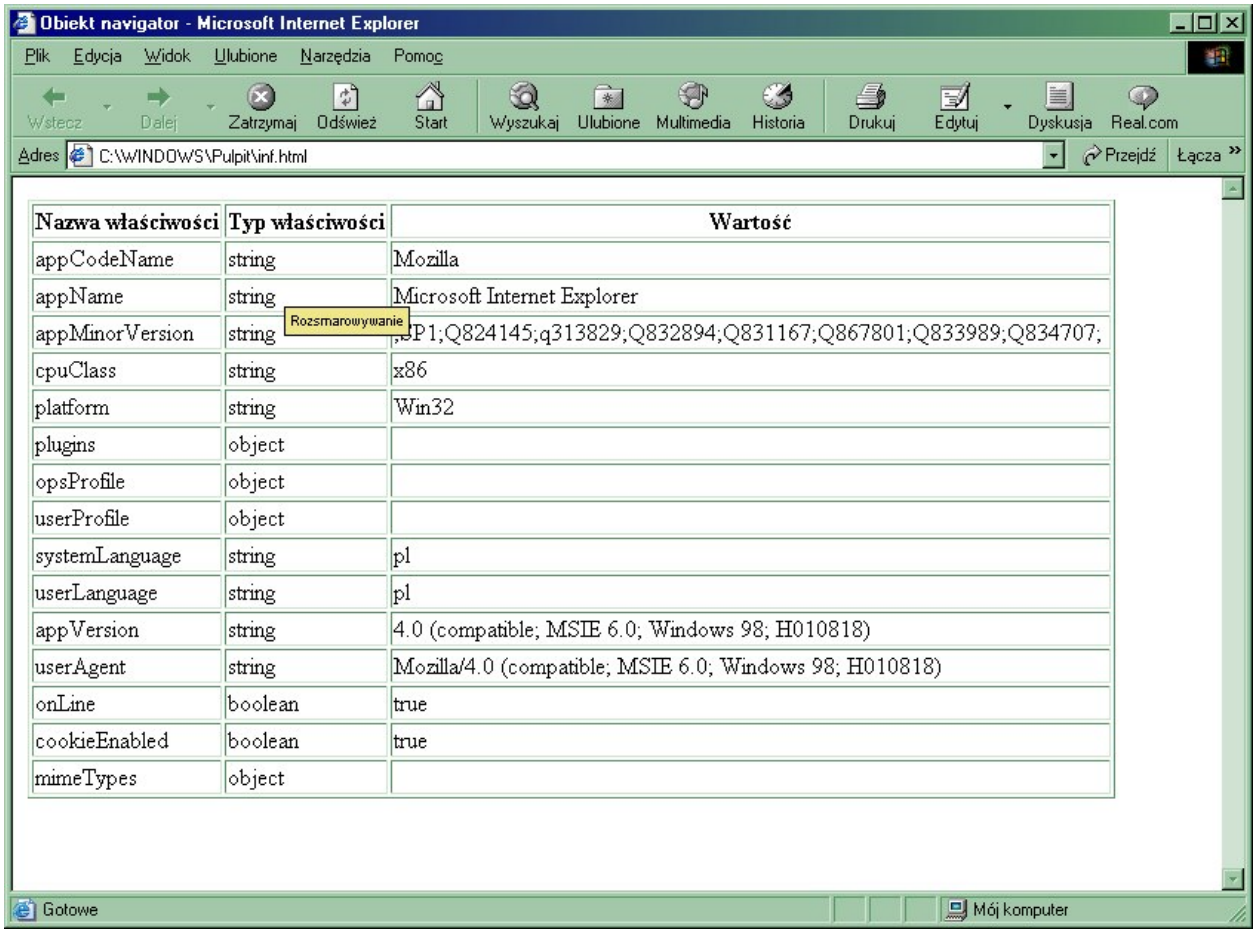
- `vendorSub`

Pełna lista właściwości obiektu navigator dla przeglądarki Mozilla:



Nazwa właściwości	Typ właściwości	Wartość
platform	string	Win32
appCodeName	string	Mozilla
appName	string	Netscape
appVersion	string	5.0 (Windows; en-US)
language	string	en-US
mimeTypes	object	[object MimeTypeArray]
oscpu	string	Win98
vendor	string	
vendorSub	string	
product	string	Gecko
productSub	string	20040910
plugins	object	[object PluginArray]
securityPolicy	string	
userAgent	string	Mozilla/5.0 (Windows; U; Win98; en-US; rv:1.7.3) Gecko/20040910
cookieEnabled	boolean	true
javaEnabled	function	function javaEnabled() { [native code] }
taintEnabled	function	function taintEnabled() { [native code] }
preference	function	function preference() { [native code] }

Pełna lista właściwości obiektu navigator dla przeglądarki Internet Explorer:



Nazwa właściwości	Typ właściwości	Wartość
appCodeName	string	Mozilla
appName	string	Microsoft Internet Explorer
appMinorVersion	string	;SP1;Q824145;q313829;Q832894;Q831167;Q867801;Q833989;Q834707;
cpuClass	string	x86
platform	string	Win32
plugins	object	
opsProfile	object	
userProfile	object	
systemLanguage	string	pl
userLanguage	string	pl
appVersion	string	4.0 (compatible; MSIE 6.0; Windows 98; H010818)
userAgent	string	Mozilla/4.0 (compatible; MSIE 6.0; Windows 98; H010818)
onLine	boolean	true
cookieEnabled	boolean	true
mimeTypes	object	

12.4 Obiekt screen

Obiekt **screen** umożliwia dostęp do informacji na temat rozdzielczości i innych opcji wyświetlania monitora, z których korzysta użytkownik. Oto lista właściwości obiektu:

- `availHeight, height` - dostępna wysokość ekranu w pikselach.
- `availWidth, width` - dostępna szerokość ekranu w pikselach.
- `colorDepth, pixelDepth` - liczba kolorów z jaką pracuje monitor.
- `fontSmoothingEnabled` - zawiera wartość `true` jeśli włączony jest antyaliazing czcionek ekranowych.

13 Obiekt document

Oto lista właściwości obiektu **document** według standardu W3C DOM:

- charset - specyficzna dla Internet Explorer, zawiera nazwę strony kodowej dokumentu.

- characterSet - specyficzna dla Netscape, zawiera nazwę strony kodowej dokumentu.
- cookie - umożliwia ustawienie cookie z poziomu JavaScript.
- domain - pozwala na uzyskanie dostępu do dokumentów wyświetlanych w innych ramkach lub oknach, które załadowano z tej samej domeny, co dokument, w którym znajduje się skrypt, ale na innym serwerze. Domyślnie nie jest to możliwe ze względów bezpieczeństwa.
- height i width - specyficzne dla Netscape, zawierają wysokość i szerokość okna w pikselach w momencie, w którym są sprawdzane. Przeglądarka IE też ma te właściwości, ale należą one do obiektu **body**.
- lastModified - data i godzina ostatniej modyfikacji dokumentu.
- referrer - adres URL strony, na której znajdowało się łącze do dokumentu (o ile użytkownik użył takiego sposobu, by dostać się na stronę).
- title - zawiera tytuł dokumentu.
- URL - adres, którego użyto do pobrania dokumentu.

Obiekt **document** zawiera wiele metod, my omówimy te najbardziej użyteczne dla nas. Oto ich lista:

- `close()`
- `getElementById()`
- `getElementsByName()`
- `getElementsByTagName()`
- `open()`
- `write()`
- `writeln()`

Metody `open()`, `write()`, `writeln()` i `close()` umożliwiają tworzenie nowych dokumentów za pomocą języka JavaScript. Nie można natomiast za ich pomocą dodawać elementów do dokumentu, który został już załadowany w oknie przeglądarki. Metoda `open()` pozwala na otwarcie nowego strumienia danych w oknie przeglądarki lub ramce. Jako argument podajemy typ MIME nowego dokumentu (np. `text/html`, `image/jpeg` i. in.). Aby utworzyć

dokument używamy metody `write()` `writeln()` (różnią się one tym, że ta druga dodaje na końcu znak nowego wiersza). Konstruowanie dokumentu kończymy poprzez użycie metody `close()`, która powoduje zamknięcie strumienia danych.

```
document.open("text/html"); document.write("<html><head>");
document.write("<meta http-equiv=\"Content-type\"
                  content=\"text/html; charset=ISO-8859-2\" >");
document.write(" <title>Tytuł dokumentu</title></head><body>");
document.write("Zawartość strony</body></html>");
document.close();
```

13.1 Model W3C DOM

Model DOM opisuje relacje pomiędzy wszystkimi obiektami na stronie HTML. Relacje te przypominają strukturę drzewa. Każdy obiekt w modelu DOM jest tzw. węzłem (ang. *node*). Pierwszym elementem drzewa jest obiekt **document** - od niego rozchodzą się poszczególne gałęzie drzewa. I tak, znacznik `<HTML>` jest węzłem obiektu **document**, a znaczniki `<body>` i `<title>` są węzłami potomnymi znacznika `<HTML>`. Obiekt **document**, mimo iż nie znajduje się w kodzie HTML, jest traktowany jako pierwszy element na stronie.

Każdy element na stronie jest węzłem. Jest nim również zwykły tekst. Węzeł zawierający inne węzły, nazywa się węzłem rodzicem (*parent node*). Węzeł zawarty w innym węźle określa się mianem węzeł dziecko lub węzeł potomny (*child node*). Np., następujący kod HTML tworzy dwa różne węzły:

`<b>To jest pogrubiony tekst<b>`

Mamy tu węzeł **b** oraz węzeł potomny typu **Text** o zawartości *To jest pogrubiony tekst.*

Typy węzłów. Poniżej zestawiona została tabela typów węzłów według standardu W3C DOM:

Typ	**Numer**	**nodeName**	**nodeValue**	**Opis**
Element	1	nazwa znacznika	null	dowolny znacznik HTML
Attribute	2	nazwa atrybutu	wartość atrybutu	para atrybut-wartość
Text	3	#text	tekst	fragment tekstu
Comment	8	#comment	komentarz	komentarz HTML
Document	9	#document	null	główny obiekt document
DocumentType	10	DOCTYPE	null	specyfikacja DTD

Właściwości węzłów. Każdy węzeł ma wiele właściwości, większość z nich to opis zależności pomiędzy innymi węzłami. Oto lista niektórych właściwości typów węzłów:

Właściwość	Typ wartości	Opis
nodeName	String	Zależy od typu węzła
nodeValue	String	Zależy od typu węzła
nodeType	Integer	Stała reprezentująca typ węzła
parentNode	Object	Referencja do węzła macierzystego
childNodes	Array	Wszystkie węzły potomne
firstChild	Object	Referencja do pierwszego węzła potomnego
lastChild	Object	Referencja do ostatniego węzła potomnego
previousSibling	Object	Referencja do poprzedniego węzła na danym poziomie
nextSibling	Object	Referencja do następnego węzła na danym poziomie
attributes	NodeMap	Tablica atrybutów węzła

Dostęp do elementów HTML za pomocą węzłów. Rozważmy bardzo prosty dokument HTML:

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html;
         charset=ISO-8859-2" />
        <meta name="Description" content="[ Opis dokumentu ]" />
        <meta name="Author" content="[ Autor dokumentu ]" />
        <meta name="Generator" content="EdHTML" />
        <title>To jest tytuł dokumentu</title>
    </head>
    <body>
        <!-- A to jest komentarz -->
        <p name="ala" id="akapit" title="cos">
            A to akapit
                <em>
                    i coś jeszcze
                </em>
        </p>
    </body>
</html>
```

Aby uzyskać dostęp do akapitu możemy wykorzystać następującą strukturę węzłów:

```
document.childNodes[1].childNodes[1].childNodes[1]
\\          <html>          <body>          <p>
```

Umieśćmy w `<body>`, ale na końcu żeby nie zmienić struktury dokumentu sekcję <script> o następującej zawartości:

```
alert(document.childNodes[1].childNodes[1].childNodes[1].nodeName);
```

Jeśli nasza przeglądarka obsługuje standard W3C DOM, to powinniśmy zobaczyć **P**. Natomiast kod:

```
alert(document.childNodes[1].childNodes[1].firstChild.nodeValue);
```

powinien wyświetlić treść komentarza (tzn. "A to jest komentarz").

Jeśli chcemy wyświetlić tekst akapitu, to musimy użyć takiego kodu:

```
alert
(document.childNodes[1].childNodes[1].childNodes[1].firstChild.nodeValue);
```

A żeby wyświetlić tekst wyróżniony, użyjemy kodu:

```
alert
(document.childNodes[1].childNodes[1].childNodes[1].lastChild.firstChild.
nodeValue);
```

Żeby wyświetlić wszystkie ustawione atrybuty dla elementu **P** możemy użyć składni:

```
var zm=document.getElementById("akapit");
  for (var i=0;i<zm.attributes.length;i++){
    if(zm.attributes[i].nodeValue)
    alert(zm.attributes[i].nodeName+"="+zm.attributes[i].nodeValue);
   }
```

Taka składnia jest dość kłopotliwa i długa, ale można ją skrócić używając metod dostępnych do grupowania i wyszukiwania węzłów dostępnych dla obiektu **document**.

Metoda `getElementById()` to mechanizm służący do uzyskiwania dostępu do elementu, któremu w dokumencie przypisano wartość atrybutu `id`. Zwraca ona adres obiektu o określonym atrybucie `id`. Wartości atrybutu `id` w obrębie dokumentu powinny być unikalne.

Metody `getElementsByName()` oraz `getElementsByTagName()` pełnią podobną rolę, ale w odróżnieniu do `getElementById()`, mogą zwracać kolekcje złożone z więcej niż jednego obiektu (tablice). W przypadku `getElementsByName()` zwracane są te obiekty dokumentu, których atrybut `name` ustawiony jest na podaną wartość. Metoda `getElementsByTagName()` zwraca elementy o nazwie znacznika zgodnej z nazwą podaną jako parametr.

Tak więc, aby uzyskać dostęp do paragrafu w naszym dokumencie możemy użyć składni

```
zm=document.getElementById("akapit"); alert(zm.nodeName);
```

Ten sposób jest dużo łatwiejszy i bardziej uniwersalny (działa w większości przeglądarek). W tym wypadku żeby wyświetlić tekst wyróżniony, użyjemy kodu:

```
alert(zm.lastChild.firstChild.nodeValue);
```

Jest to na pewno sposób krótszy i przez to czytelniejszy.

Metody dostępne dla węzła. Teraz omówimy metody dostępne z poziomu węzłów. Umożliwią one manipulację węzłami: ich kasowanie, tworzenie, zmianę zawartości.

Oto ich lista:

Metoda	**Opis**
appendChild(newChild)	Dodaje węzeł potomny na końcu aktualnego węzła
cloneNode(deep)	Kopiuje aktualny węzeł (opcjonalnie z potomkami)
hasChildNodes()	Sprawdza czy węzeł ma potomków
insertBefore(new, ref)	Wstawia węzeł przed inny węzeł
removeChild(old)	Usuwa węzeł
replaceChild(new, old)	Zamienia stary węzeł na nowy

Warto zauważyć, że wszystkie powyższe metody zakładają iż punktem ich wywołania jest węzeł macierzysty do tego, który ma być modyfikowany.

Usuwanie węzła. Aby usunąć węzeł używamy metody `removeChild()`. Następująca funkcja usuwa węzeł, o podanym **id**:

```
function removeElement(elemID) { var elem =
document.getElementById(elemID) elem.parentNode.removeChild(elem)
}
```

Więc, aby usunąć akapit w naszym dokumencie wystarczy wywołać funkcję `removeElement("akapit")`. Wtedy zostaną usunięte wszystkie węzły potomne. Aby usunąć tylko wyróżnienie (fragment `<em>`), musimy użyć składni:

```
var em=document.getElementById("akapit").childNodes[1];
em.parentNode.removeChild(em)
//lub document.getElementById("akapit").removeChild(em);
```

Tworzenie nowego węzła. Aby utworzyć nowy węzeł w standardzie DOM, trzeba użyć odpowiedniej metody obiektu **document**. Dwie najważniejsze z nich to `createElement()` i `createTextNode()`. Pierwsza generuje znacznik HTML o podanej nazwie, druga tworzy węzeł tekstowy o podanej zawartości.

Kiedy tworzymy nowy element, istnieje on tylko w pamięci przeglądarki i nie jest częścią dokumentu. Co więcej, wynik wywołania metody `createElement()` to odwołanie do pustego

elementu (mamy wtedy tylko nazwę znacznika). Na przykład, aby stworzyć nowy akapit użyjemy składni:

```
var newElem = document.createElement("P")
```

Ten element nie ma ustawionych ani atrybutów, ani zawartości. Żeby ustawić pary atrybut-wartość używamy metody `setAttribute()` (jest to metoda każdego elementu). Aby ustawić **id**, wywołamy kod:

```
newElem.setAttribute("id", "nowy Akapit");
```

lub

```
newElem.id="nowy Akapit";
```

Obydwa sposoby są poprawne. Jednak w dalszym ciągu element nie jest częścią dokumentu i nie można uzyskać dostępu do niego za pomocą metody `getElementById()`.

Żeby w końcu stworzyć zawartość akapitu, tworzymy nowy obiekt tekstowy:

```
var newText = document.createTextNode("To jest drugi akapit.");
```

Aby połączyć dwa elementy, możemy wpisać tekst jako element potomny dla istniejącego tylko w pamięci akapitu.

```
newElem.appendChild(newText)
```

Teraz możemy już wstawić paragraf do dokumentu. Wstawiamy nowy akapit po pierwszym za pomocą metody `appendChild()` (czyli wstawiamy go na końcu) dla obiektu body:

```
document.body.appendChild(newElem)
```

Teraz nowy akapit jest już częścią naszego dokumentu i możemy się do niego odwoływać tak samo jak do innych elementów. Jeśli chcielibyśmy wstawić ten akapit w innym miejscu możemy użyć metody `insertBefore()`. Na przykład, jeśli chcemy wstawić tekst przed pierwszym akapitem, użyjemy kodu:

```
document.body.insertBefore(newElem,document.getElementById("akapit"));
```

Zmiana zawartości węzła. Jeśli chcemy zmienić zawartość węzła tekstowego możemy użyć metody `replaceChild()`, bądź po prostu zmienić zawartość właściwości `nodeValue`. Zmienimy zawartość pierwszego akapitu za pomocą `replaceChild()`. Wpierw musimy stworzyć nowy węzeł tekstowy:

```
var newText = document.createTextNode("Nowy tekst akapitu " );
```

Potem wywołujemy po prostu `replaceChild()`:

```
var oldChild = document.getElementById("akapit").firstChild;
document.getElementById("akapit").replaceChild(newText, oldChild)
```

Jeśli chcemy przechwycić stary węzeł, możemy to zrobić ponieważ metoda `replaceChild()` zwraca odwołanie do zmienianego węzła, który jest już tylko w pamięci w tej chwili i nie jest częścią dokumentu. Możemy więc metodę przyporządkować do zmiennej i używać, jeśli chcemy. W przypadku węzłów tekstowych prostszym sposobem zmiany zawartości węzła jest modyfikacja właściwości `nodeValue`. Ten sam efekt, co poprzednio uzyskamy następująco:

```
document.getElementById("akapit").childNodes[0].nodeValue =
    "Nowy tekst akapitu ";
```

Aby zamienić tekst wyróżniony na tekst pogrubiony w naszym akapicie, możemy więc użyć składni:

```
var newBold = document.createTextNode(" tekst pogrubiony" ); var
newBoldEl = document.createElement("B");
newBoldEl.appendChild(newBold); var
em=document.getElementById("akapit").lastChild;
document.getElementById("akapit").replaceChild(newBoldEl, em);
```

Standard W3C DOM daje nam dużo większe możliwości, aby je poznać możemy odwiedzić stronę http://www.w3.org/TR/DOM-Level-3-Core gdzie znajduje się aktualna specyfikacja tego standardu. Aktualna wersja to **DOM Level 3 Core Version 1.0** datowana na 07.04.2004.

14 Zdarzenia i ich obsługa

Za każdym razem, kiedy w przeglądarce coś się dzieje, JavaScript modyfikuje zawartość odpowiedniego obiektu. Na przykład jeśli użytkownik kliknie na umieszczony na stronie przycisk, to dla tego przycisku zajdzie zdarzenie `Click`, jeśli zmodyfikujemy zawartość formularza zajdzie zdarzenie `Change`. Za pomocą JavaScript-u możemy tworzyć kod służący do obsługi tych zdarzeń, kod ten określa się mianem *procedury obsługi zdarzenia*.

Następujący przykład, to procedura obsługi zdarzenia `Click` dla przycisku, która wyświetla okno informacyjne.

```
<input type="button" id="but" value="Naciśnij mnie"
       onClick="alert('Nacisnąłeś właśnie przycisk')">
```

Zamiast pisać cały kod wewnątrz znacznika HTML, możemy wywołać w nim funkcję:

```
<script language="JavaScript" type"text/javascript">
function pokaz(){
alert("Nacisnąłeś właśnie przycisk")
}
</script>
<input type="button" id="but" value="Naciśnij mnie" onClick="pokaz()">
```

Innym sposobem na przechwytywanie zdarzeń jest utworzenie kodu JavaScript, który wyczekuje na zajście określonego zdarzenia. Ten rodzaj obsługi zdarzenia nadaje się szczególnie do obiektów, które nie mają znaczników HTML (np. dla obiektu **document**), możemy go również stosować do dowolnych innych zdarzeń.

Aby wykorzystać ten sposób musimy utworzyć następujące przyporządkowanie w kodzie:

```
nazwaObiektu.nazwaZdarzenia=proceduraObslugiZgarzenia;
```

Wartość `nazwaObiektu` określa obiekt, dla którego chcemy przyporządkować procedurę obsługi, `nazwaZdarzenia` określa, które zdarzenie chcemy śledzić, natomiast `proceduraObslugiZgarzenia`, to funkcja JavaScript do obsługi zdarzenia.

Prześledźmy następujący przykład:

```
<script language="JavaScript" type="text/javascript">
          window.onresize=wyswietlKomunikat;

function  wyswietlKomunikat(){
    window.alert("Zmieniasz właśnie rozmiar okna");
} </script>
```

Kod ten spowoduje wyświetlenie komunikatu `Zmieniasz właśnie rozmiar okna` w sytuacji, gdy zmieniamy wielkość okna przeglądarki. W ten sam sposób możemy kontrolować zdarzenie w przykładzie pierwszym, kod będzie wyglądał następująco:

```
<body>
<input type="button" id="przycisk" value="Naciśnij mnie">
<script language="JavaScript" type="text/javascript">
document.body.childNodes[0].onclick=pokaz;
//możemy również tak:
//document.getElementById("przycisk").onclick=pokaz;
function pokaz(){
alert("Nacisnąłeś właśnie przycisk") }
</script>
```

W tym przypadku, sposób ten wydaje się bardziej skomplikowany i trzeba pamiętać o tym, że przyporządkowanie procedury musi nastąpić po załadowaniu odpowiedniego węzła.

Rodzaje zdarzeń. Oto lista zdarzeń dostępnych w JavaScript:

Zdarzenie	**Procedura**	**Opis**
Abort	onabort	Występuje w momencie anulowania ładowania strony lub grafiki
Blur	onblur	Występuje w momencie zakończenia aktywności i uaktywnienia innego obiektu
Change	onchange	Występuje w momencie modyfikacji pola w formularzu
Click	onclick	Występuje w momencie kliknięcia obiektu
DblClick	ondblckick	Występuje w momencie dwukrotnego kliknięcia obiektu
Error	onerror	Występuje w momencie wystąpienia błędu na stronie
Focus	onfocus	Występuje w momencie, kiedy obiekt staje się aktywny
KeyDown	onkeydown	Występuje w momencie wciśnięcia klawisza na klawiaturze
KeyPress	onkeypress	Występuje, gdy klawisz zostaje naciśnięty
KeyUp	onkeyup	Występuje w momencie,gdy klawisz zostaje zwolniony
Load	onload	Występuje w momencie ładowania strony lub grafiki
MouseDown	onmousedown	Występuje w momencie wciśnięcia dowolnego przycisku myszy
MouseMove	onmousemove	Występuje gdy, przesuwamy kursorem myszy
MouseOut	onmouseout	Występuje gdy, opuszczamy kursorem myszy obiekt
MouseOver	onmouseover	Występuje gdy, kursor myszy znajdzie się nad obiektem
MouseUp	onmouseup	Ma miejsce, gdy zwalniany jest przycisk myszy
Move	onmove	Ma miejsce, gdy przesuwane jest okno przeglądarki
Reset	onreset	Ma miejsce, gdy wciśnięty jest przycisk `reset` w formularzu
Resize	onresize	Ma miejsce, gdy zmieniamy rozmiar okna przeglądarki
Select	onselect	Ma miejsce, gdy zaznaczamy tekst na stronie
Submit	onsubmit	Ma miejsce, gdy wysyłamy formularz, tzn. wciskamy przycisk `submit`
UnLoad	onunload	Ma miejsce w momencie usuwania strony z pamięci

Przykłady obsługi zdarzeń. Poniżej zamieszczone zostały przykłady obsługi niektórych zdarzeń. Inne przykłady, dotyczące obsługi formularzy znajdują się w paragrafie **Formularze HTML i obiekt String**.

Przechwytywanie wciśnięcia przycisku klawiatury:

```
<script language="JavaScript" type="text/javascript">
document.onkeypress=wyswietlKomunikat;
function  wyswietlKomunikat(){
    window.alert("Wciśnąłeś jakiś przycisk");
}
</script>
```

> Zdarzenia związane z obsługą klawiatury mogą nie być generowane, gdy wciskamy klawisze nie posiadające kodów ASCII, np. strzałki lub klawisze funkcyjne, dzieje się tak w IE.

Wyświetlanie komunikatu podczas ładowania strony:

```
<body onload="alert('Witamy na naszej stronie')">
```

Przechwytywanie kliknięć myszą:

```
<script language="JavaScript" type="text/javascript">
function sprawdz(url){
return (confirm("Czy chcesz odwiedzić serwis WWW"+url+"?"))?true:false;
}
</script>
<body>
<a href="http://www.onet.pl" onclick="return sprawdz('Onet')">Onet<a><br>
<a href="http://www.o2.pl" onclick="return sprawdz('O2')">O2<a>
</body>
```

Powyższy kod wymaga pewnych wyjaśnień. Została tu użyta specjalna konstrukcja obsługi zdarzeń. W momencie kliknięcia na dany odnośnik wywoływana jest funkcja obsługi zdarzenia, która w zależności od wybranej opcji zwraca wartość `true` lub `false`. Wartość ta w połączeniu ze słowem kluczowym `return` w procedurze obsługi zdarzenia, umożliwia kontrolę, czy użytkownik chce na pewno otworzyć dane łącze, i tak jeśli funkcja `sprawdz()` zwróci wartość `true` - łącze będzie otwarte, a jeśli `sprawdz()` zwróci wartość `false` odnośnik nie będzie otwarty.

Kontrola wskazania obiektu myszą:

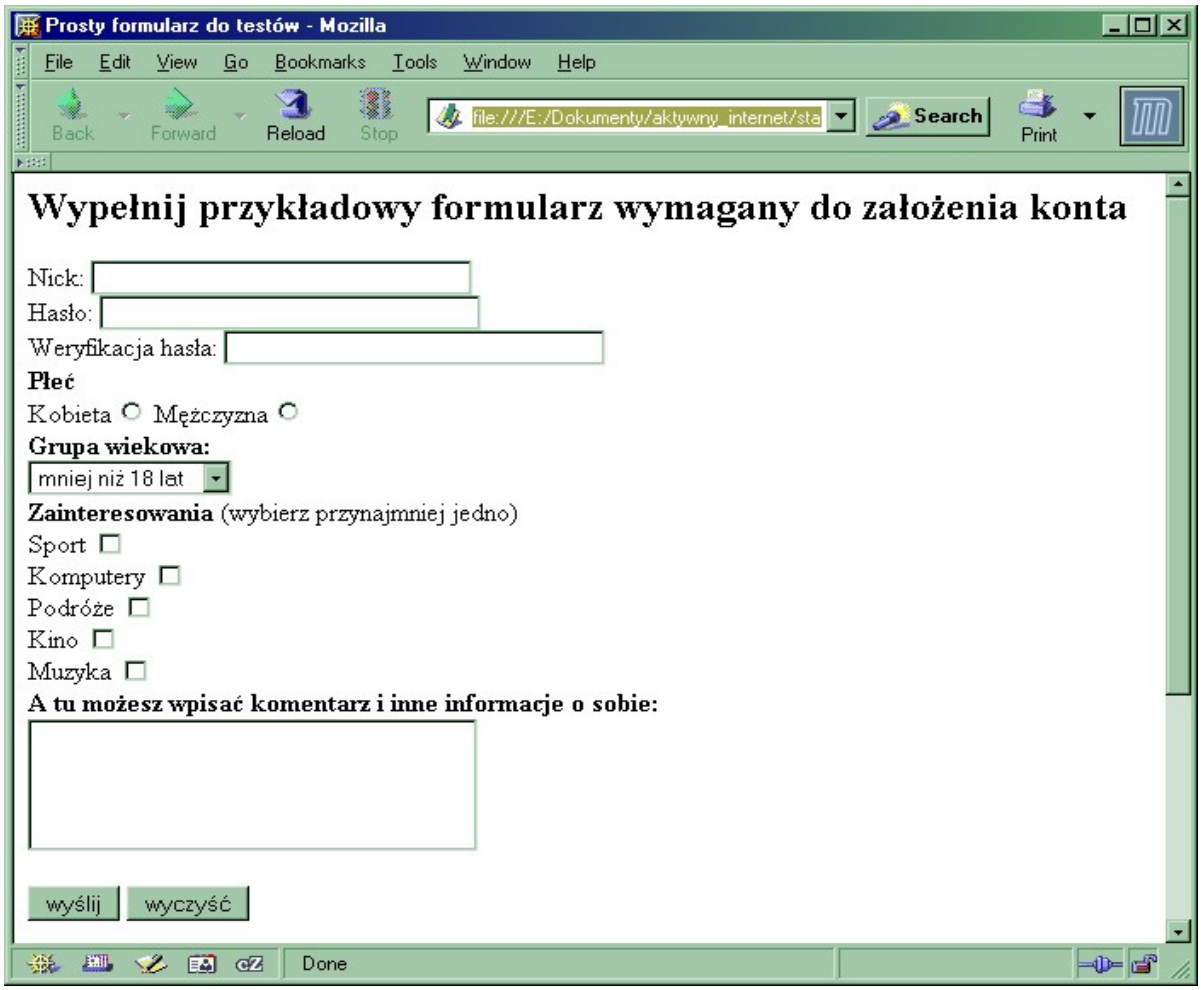
```
<h2 onmouseover="alert('Wjechałeś myszką na nagłówek')"
    onmouseout="alert('Nie wychodź stąd')">
To jest tekst tytułu stopnia drugiego</h2>
```

15 Formularze HTML i obiekt String

Na początku warto wyjaśnić, do czego służy język JavaScript w formularzach. Bardzo często się zdarza, że w przypadku tworzenia projektów WWW opartych o technologię **server-side** pobieramy pewne istotne dane od użytkownika. Rozważmy dla przykładu następujący formularz, przydatny do zakładania konta pocztowego na jakimś portalu:

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<meta http-equiv="Content-type" content="text/html;charset=windows-1250">
<title>Prosty formularz do testów</title>
</head>
<body>
<h2>Wypełnij przykładowy formularz wymagany do założenia konta</h2>
<form name="formularz" action="jakis_zdalny_skrypt" method="post"
      enctype="multipart/form-data">
Nick: <input type="text" size="30" name="nick"><br>
Hasło: <input type="password" size="30" name="haslo1"><br>
Weryfikacja hasła: <input type="password" size="30" name="haslo2"><br>
<b>Twoja płeć</b><br>
Kobieta<input type="radio" name="plec" value="kobieta">
Mężczyzna<input type="radio" name="plec" value="mezczyzna"><br>
<b>Grupa wiekowa:</b><br>
<select name="wiek">
<option value="1">mniej niż 18 lat
<option value="2">18-25 lat
<option value="3">więcej niż 25 lat
</select><br>
<b>Zainteresowania</b> (wybierz przynajmniej jedno)<br>
Sport <input type="checkbox" name="sport" value="lubie"><br>
Komputery <input type="checkbox" name="komputery" value="lubie"><br>
Podróże <input type="checkbox" name="podroze" value="lubie"><br>
Kino  <input type="checkbox" name="kino" value="lubie"><br>
Muzyka  <input type="checkbox" name="muzyka" value="lubie"><br>
<b>A tu możesz wpisać komentarz i inne informacje o sobie:</b><br>
<textarea name="komentarz" rows="4" cols="30"></textarea><br>
<input type="submit" value="wyślij"><input type="reset" value="wyczyść">
```

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



Co się stanie, gdy użytkownik poda nazwę username, która nie spełnia wymogów portalu? Co jeśli hasła się nie zgadzają, jeśli nie została wybrana ani płeć, ani grupa wiekowa i żadne z zainteresowań? W takiej sytuacji dane będą i tak wysłane na serwer, który musi je przetworzyć i wygenerować błąd. Wszystko to możemy w takim przypadku zrobić po stronie klienta używając języka JavaScript. Możemy sprawdzić, czy pola są wypełnione, czy się zgadzają z ustaloną składnią i dopiero wtedy wysyłać je do przetworzenia za pomocą zdalnych skryptów, co z pewnością przyspieszy działanie serwisu i zachęci do jego odwiedzin.

Dostęp do formularza z poziomu języka JavaScript. Język JavaScript udostępnia kolekcję **forms**, która umożliwia dostęp do poszczególnych formularzy na stronie (pierwszy formularz na stronie to **document.forms[0]**, kolejny **document.forms[1] itd.**). Ponadto, jeśli formularz ma ustawiony atrybut `name`, to formularz jest dostępny pod obiektem o nazwie:

```
document.forms["nazwa_formularza"]
```

lub

```
document.forms.nazwa_formularza
```

Dla zachowania wstecznej zgodności, zachowano również następującą notację obiektu formularza:

```
document.nazwa_formularza
```

Sposób ten jest ciągle obsługiwany przez przeglądarki, jednak nie zaleca się tej formy dostępu do danych formularza. W naszym przypadku formularz będzie dostępny pod każdą z nazw:

```
document.forms[0]
document.forms["formularz"]
document.forms.formularz
document.formularz
```

Właściwości obiektu form. Obiekt **form** ma wiele właściwości, większość z nich odpowiada atrybutom znacznika `<form>`, między innymi są to:

- `name` - wartość atrybutu `name` formularza,
- `action` - wartość atrybutu `action`
- `enctype` - wartość atrybutu `enctype` formularza,
- `method` - wartość atrybutu `method` formularza,
- `length` - ilość pól formularza.

Aby wyświetlić wartości tych pól dla naszego formularza musimy użyć składni:

```
<script language="JavaScript" type="text/javascript">
document.write("<br>Nazwa formularza"+document.forms.formularz.name+
               "<br>");
document.write("<br>Action formularza"+document.forms.formularz.action+
               "<br>");
document.write("<br>Method formularza"+document.forms.formularz.method+
               "<br>");
document.write("<br>Ilość pól formularza"+document.forms.formularz.length+
               "<br>");
document.write("<br>Wartość enctype formularza"+document.formularz.enctype+
               "<br>");
</script>
```

Metody obiektu form. W języku JavaScript są dwie metody obiektu **form**. Metody te odpowiadają standardowym przyciskom formularzy wyślij (*submit*) oraz wyczyść (*reset*). Metody te mają takie same nazwy jak przyciski `submit()` i `reset()`. Jeśli chcemy wysłać nasz formularz, to użylibyśmy składni:

```
document.forms.nazwa_formularza.submit();
```

Jeśli chcielibyśmy wyczyścić pola formularza, to wywołalibyśmy kod:

```
document.forms.nazwa_formularza.reset();
```

Dostęp do elementów formularza. Każdy formularz należący do kolekcji **forms** posiada inną kolekcję - **elements**. Kolekcja ta zawiera wszystkie pola formularza i umożliwia dostęp do tych pól za pomocą tablicy.

Jeśli znamy typ pola i jego nazwę to dostęp do elementu jest prostszy. W przypadku **pól tekstowych** (typ `text`, `password` oraz pole `textarea`) zawartość tych pól znajduje się we właściwości:

```
document.forms.nazwa_formularza.nazwa_pola_tekstowego.value
```

Żeby wyświetlić wpisane przez użytkownika pola `nick` i `komentarz` użyjemy kodu

```
alert("Twój nick to "+document.forms.formularz.nick.value);
alert("Podałeś komentarz"+document.forms.formularz.komentarz.value);
```

Jeśli typ pola o nazwie **poleWyporu** to **checkbox**, to element ten ma dwie właściwości `value` - zawierającą wartość atrybutu `value` pola i `checked`, która przyjmuje wartość `true` jeśli pole było zaznaczone i `false` w przeciwnym razie.

Następujący kod zwróci tekst "lubie":

```
alert(document.forms.formularz.sport.value);
```

Aby sprawdzić, czy użytkownik lubi sport użyjemy składni:

```
if (document.forms.formularz.sport.checked)
    alert("Lubisz sport");
```

Żeby natomiast sprawdzić, które pola **checkbox** były wybrane musimy użyć kolekcji **elements**:

```
var form=document.forms.formularz;
for (var j=0;j<form.elements.length;j++)
if (form.elements[j].type=="checkbox" && form.elements[j].checked)
    alert("Lubisz " +form.elements[j].name);
```

Jeśli chcemy testować pola typu **radio**, możemy użyć kolekcji o nazwie takiej jak nazwa grupy **radio** (pola typu **radio** różnią się od pól typu **checkbox** tym, że w grupie pól **radio** można wybrać tylko jeden przycisk). Kolekcja ta jest dostępna pod nazwą:

`document.forms.nazwa_formularza.nazwa_grupy_radio`

Każdy element kolekcji ma właściwość `checked`, która informuje, czy pole było wybrane. Żeby sprawdzić jaką płeć wybrał użytkownik, użyjemy kodu:

```
var form=document.forms.formularz;
for (var i=0;i<form.plec.length;i++)
if (form.plec[i].checked) break;
if (i==form.plec.length) alert("Nie wybrałeś płci!");
```

W przypadku pól typu **select**, musimy wiedzieć o dwóch rzeczach, numer wybranego pola jest zapamiętany we właściwości `selectedIndex`, a każdy element tego typu zawiera kolekcję **options** złożoną ze wszystkich opcji pola **select**. Ponadto każdy element **options** ma dwie właściwości `value` i `text`, które zawierają odpowiednio wartość atrybutu `value` i tekst opcji. Żeby sprawdzić, którą grupę wiekową wybrał użytkownik, użyjemy kodu:

```
var form=document.forms.formularz;
var wybrany=form.wiek.selectedIndex;
var str="Masz"+form.wiek.options[wybrany].text;
alert(str);
```

Uwaga! Jeśli mamy do czynienia z polem `select` z włączoną opcją multiple, to nie wystarczy skorzystać z właściwości `selectedIndex`, ponieważ może być wybrana grupa elementów. W takiej sytuacji korzystamy z właściwości `selected` każdej opcji pola. Ma ona wartość `true`, jeśli pole było wybrane i `false` w przeciwnym razie.

Rozważmy następujący przykład:

```
var poleSelect=document.forms.nazwaFormularza['nazwaPolaSelect'];
var wybrany=false;
for (var j=0;j<poleSelect.length;j++)
if (poleSelect.options[j].selected){
alert('Wybrana opcja:'+poleSelect.options[j].value);
wybrany=true;
}
if (!wybrany) alert('Nie wybrałeś żadnej opcji!');
```

Przykład - testowanie poprawności danych w formularzu. A oto nasz formularz z początku paragrafu wyposażony w funkcje kontroli danych.

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<meta http-equiv="Content-type" content="text/html;charset=windows-1250">
<title>Prosty formularz do testów</title>
</head>
<script language="JavaScript" type="text/javascript">
function wyswietl(i){
var form=document.forms.formularz;
alert("Twój nick to"+form.nick.value);
alert("Hasło to "+form.haslo1.value);
alert("Jesteś "+form.plec[i].value);
var wybrany=form.wiek.selectedIndex;
var str="Masz"+form.wiek.options[wybrany].text;
alert(str);
}
function poprawny() {
var form=document.forms.formularz;
    if (!form.nick.value){
    alert("Nie podałeś imienia")
    return false;
    }
    if (form.haslo1.value!=form.haslo2.value || form.haslo1.value==''){
    alert("Błędne hasło!");
    return false;
    }
for (var i=0;i<form.plec.length;i++)
     if (form.plec[i].checked) break;
     if (i==form.plec.length) { alert("Nie wybrałeś płci!!");
return false;
}
var wybrany=false;
for (var j=0;j<form.elements.length;j++)
if (form.elements[j].type=="checkbox" && form.elements[j].checked) {
    wybrany=true;
    alert("Lubisz " +form.elements[j].name);
}
if (!wybrany) { alert("Nie masz żadnego hobby?!!");
```

```
return false;
}
    wyswietl(i);
}
</script>
<body>
<h2>Wypełnij przykładowy formularz wymagany do założenia konta</h2>
<form name="formularz" action="mailto:askox@o2.pl" method="post"
      enctype="multipart/form-data" onsubmit="return poprawny()">
Nick: <input type="text" size="30" name="nick"><br>
Hasło: <input type="password" size="30" name="haslo1"><br>
Weryfikacja hasła:<input type="password" size="30" name="haslo2"><br>
<b>Płeć</b><br>
Kobieta<input type="radio" name="plec" value="kobieta">
Mężczyzna<input type="radio" name="plec" value="mezczyzna"><br>
<b>Grupa wiekowa:</b><br>
<select name="wiek">
<option value="1">mniej niż 18 lat
<option value="2">18-25 lat
<option value="3">więcej niż 25 lat
</select><br>
<b>Zainteresowania</b>(wybierz przynajmniej jedno)<br>
Sport <input type="checkbox" name="sport" value="lubie"><br>
Komputery <input type="checkbox" name="komputery" value="lubie"><br>
Podróże <input type="checkbox" name="podroze" value="lubie"><br>
Kino  <input type="checkbox" name="kino" value="lubie"><br>
Muzyka  <input type="checkbox" name="muzyka" value="lubie"><br>
<b>A tu możesz wpisać komentarz i inne informacje o sobie:</b><br>
<textarea name="komentarz" rows="4" cols="30"></textarea><br><br>
<input type="submit" value="wyślij"><input type="reset" value="wyczyść">
</form>
</body> </html>
```

15.1 Obiekt String

Aby stworzyć egzemplarz obiektu **String** możemy użyć zwykłej instrukcji przypisania:

```
var myString="To jest ciąg znaków";
```

możemy również użyć konstruktora obiektu:

```
var myString=new String("To jest ciąg znaków");
```

Obiekt **String** ma kilka właściwości, najczęściej używaną jest `length` - informuje ona o długości łańcucha. Teraz omówimy najważniejsze metody obiektu **String**:

- `charAt()` - zwraca ciąg jednoliterowy, składnia `string.charAt(index)` - używamy tej metody aby odczytać pojedynczą literę z ciągu `string` na pozycji `index` (aby dostać ostatnią literę ciągu `myString` użyjemy składni `myString.charAt(myString.length - 1)`).
- `charCodeAt()` - zwraca całkowity kod ASCII litery na podanej pozycji, składnia `string.charCodeAt(index)`.
- `fromCharCode()` - zwraca scalony łańcuch znaków o podanych kodach, składnia `string.fromCharCode(num1 [, num2 [, ...,numn]])`.
- `concat()` - wykonuje konkatenację łańcuchów, składnia `string.concat(string2)`.
- `indexOf()` - zwraca pozycję pierwszego wystąpienia podciągu w ciągu znaków, składnia `string.indexOf(searchString [, startIndex])`.
- `lastIndexOf()`- zwraca pozycję ostatniego wystąpienia podciągu w ciągu znaków, składnia `string.lastIndexOf(searchString[,startIndex])`
- `localeCompare()` - zwraca liczbę całkowitą, składnia `string.localeCompare(string2)` - metoda pozwala na porównywanie ciągów znaków w stronie kodowej **Unicode**, zwraca zero jeśli ciągi są równe, liczbę mniejszą od zera jeśli `string` jest przed `string2` i liczbę dodatnią w przeciwnym razie.
- `match()` - zwraca tablicę pasujących ciągów, składnia `string.match(regExpression)`, parametr jest wyrażeniem regularnym, metoda zwróci tablicę jeśli w ciągu przynajmniej jeden fragment będzie pasował do wzorca i null w przeciwnym razie.
- `replace()` - zwraca zmieniony ciąg, składnia `string.replace(regExpression, replaceString)`.
- `search()` - zwraca liczbę całkowitą, składnia `string.search(regExpression)`, metoda podobna do `indexOf()`, ale w tym przypadku argument może być wyrażeniem regularnym.
- `slice()` - zwraca łańcuch znaków, składnia `string.slice(startIndex [, endIndex])` - zwracany łańcuch znaków zaczyna się w `startIndex` a kończy w opcjonalnym parametrze `endIndex` lub na końcu łańcucha `string` jeśli go nie podano.

- `split()` - zwraca tablicę z kawałkami ciągu, składnia `string.split("delimiterCharacter" [,limitInteger])` - metoda dzieli ciąg `string` w miejscach, gdzie pojawia się znak `delimiterCharacter`, fragmenty te są kolejnymi elementami tablicy, jeśli podano drugi liczbowy argument , to liczba elementów tablicy będzie ograniczona do `limitInteger`, a w ostatnim elemencie będzie reszta ciągu.

- `substr()` - zwraca łańcuch znaków, składnia `string.substr(start [, length])` - zwraca podciąg ciągu `string` zaczynający się w `start` o długości `length`.

- `substring()` - zwraca łańcuch znaków, składnia `string.substring(indexA, indexB)` - parametry to początkowy i końcowy indeks.

- `toLocaleLowerCase()` (toLocaleUpperCase()), składnia `string.toLocaleLowerCase()` (`string.toLocaleUpperCase()`) - zamienia ciągi znaków w stronie kodowej Unicode na małe litery (duże litery).

- `toLowerCase()` (toUpperCase()), składnia `string.toLowerCase()` (`string.toUpperCase()`) - zamienia ciągi znaków w stronie kodowej Latin 1 na małe litery (duże litery).

16 Wyrażenia regularne i obiekt RegExp

Wyrażenie regularne jest sposobem opisu wzorca (ang. *pattern*) szukanego we fragmencie tekstu (od lewej do prawej). Najprostszym wyrażeniem jest zwykły ciąg znaków np. "to jest mój dom", który pasuje do dowolnego łańcucha znaków zawierającego ten tekst. Korzystając z wyrażeń regularnych możemy się posługiwać specjalnymi znakami (ang. *meta characters*), pozwalającymi na wyszukiwanie nie tylko dokładnych dopasowań. Są one interpretowane w sposób szczególny, przy ich pomocy możemy wskazać, czy wzorzec musi się pojawić na początku lub końcu łańcucha, że musi się składać z określonych znaków itp. Mamy dwa zbiory meta znaków: te które są rozpoznawane wszędzie poza nawiasami prostokątnymi [] i te, które są interpretowane w nawiasach prostokątnych.

Tworzenie wyrażeń regularnych. Wyrażenie regularne tworzymy wstawiając je między znaki /,

```
var mojeWyrReg=/opis wzorca/;
```

Możemy również użyć konstruktora objektu **RegExp**:

```
var mojeWyrReg=new RegExp("opis wzorca");
```

W typ przypadku wzorzec jest zamknięty w znaki " (traktujemy go jako zwykły łańcuch znaków).

Poniżej zestawiamy znaki specjalne, które mogą służyć do opisu wzorca:

Metaznak	**Dopasowanie**	**Wzorzec**	**Pasuje do**	**Nie pasuje do**
\b	ograniczenie słowa	/\bba/	"bak", "badyl"	"abakan"
		/ba\b/	"huba"	"barbakan"
\B	nie na granicy słowa	/\Bba/	"kabała"	"badyl"
		/\Bba\B/	"kabanos"	"bak", "huba"
\d	cyfra od 0 do 9	/\d\d\d/	"999"	"65a"
\D	nie cyfra	/\D\D\D/	"abc"	"b17"
\s	znak biały	/nie\scoś/	"nie coś"	"niecoś"
\S	nie znak biały	/alt\Sracja/	"alteracja"	"alt racja"
\w	znak alfanumeryczny	/A\w/	"A1" ,"A_"	"A+"
\w	nie litera, cyfra i znak _	/A\W/	"A+"	"AB"
.	dowolny znak za wyjątkiem znaku nowego wiersza	/.as/	"bas", "las"	"\nas"
^	początek ciągu lub wiersza	/^W3C/	"W3C DOM"	"Strona W3C"
$	koniec ciągu lub wiersza	/DOM$/	"W3C DOM"	"DOM i CSS"
|	rozgałęzienie	/ala|kot/	"ala", "kot"	"kola"

Jeśli chcemy, żeby wzorzec pasował do pewnej grupy znaków możemy wymienić tę grupę w nawiasach []. Poniżej przedstawiamy przykłady użycia klas znaków:

Klasa	**Opis**	**Wzorzec**	**Pasuje do**	**Nie pasuje do**
[abc]	litery a,b,c	/[abc]bc/	"abc", "cbc"	"dbc"
[^abc]	nie litery a,b,c	/[^abc]bc/	"dbc	"bbc"
[A-Z]	duże litery	/[A-Z]BC/	"ABC"	"4BC"
[0-9]	cyfry	/[0-9]BC/	"4BC"	"ABC"
[A-Z0-9]	duże litery i cyfry	/[A-Z0-9]BC/	"4BC", "ABC"	"+BC"
[^A-Z0-9]	nie duże litery ani cyfry	/[^A-Z0-9]BC/	"+BC"	"ABC"

Wyrażenie regularne może składać się z podwyrażeń, z których każde powinno być zamknięte w nawiasy (). Jeśli chcemy żeby pewien fragment lub podwyrażenie "powtórzyło się" kilka razy w wyszukiwanym łańcuchu musimy użyć metaznaków wyliczeniowych (ang. *counting metacharacters*). Poniżej przedstawiamy tabelę tych znaków:

Metaznak	Opis	Wzorzec	Pasuje do
*	zero lub więcej powtórzeń	/(ba)*/	"", "baba"
+	jedno lub więcej powtórzeń	/(ba)+/	"ba", "baba"
?	zero bądź jedno powtórzenie	/(ba)?/	"", "ba"
{n}	dokładnie n powtórzeń	/(ba){2}/	"baba"
{n,m}	od n do m powtórzeń	/(ba){1,2}/	"ba", "baba"
{n,}	n lub więcej powtórzeń	/(ba){2,}	"bababa"

Jeśli chcemy dopasowywać któryś ze znaków specjalnych:

```
\ ^ $ * + ? . ( ) { } [ ] |
```

to musimy poinformować JavaScript, że chodzi nam o zwykły znak, a nie o jego funkcję. Używamy do tego znaku unikatowego \, każdy znak poprzedzony znakiem \ jest traktowany jako zwykły znak. Np. wyrażenie:

```
var wyr=/\./;
```

pasuje do ciągu zawierającego kropkę.

Właściwości obiektu RegExp. Obiekt **RegExp** ma kilka właściwości, opiszemy dwie najważniejsze z nich:

- `ignoreCase` - wartość logiczna, jeśli wartość ustawiona jest na `true`, to w wyrażeniu nie istotna jest wielkość znaków.
- `source` - wartość typu łańcuchowego, zawiera łańcuchową reprezentację wyrażenia regularnego.

Metody obiektu RegExp. Najczęściej używane metody obiektu **RegExp**, to:

- `test()` - zwraca wartość logiczną, składania `wyrazReg.test(string)` - zwróci wartość `true`, jeśli wyrażenie `wyrazReg` pasuje do ciągu `string` i `false` w przeciwnym razie.
- `exec()` - zwraca tablicę trafień lub wartość `null`, składnia `wyrazReg.exec(string)`, wynik wywołania zawiera kolejne części łańcucha `string`, które pasują do wzorca `wyrazReg`.

Kilka przykładów.
Testowanie numeru **PESEL**. Numer ten składa się z 11 cyfr.

```
var wyrNrPesel=/^\d{11}$/;
```

Nazwa **username** na wielu serwerach powinna się składać małych liter, powiedzmy od trzech do ośmiu. Wyrażenie do testowania tego ciągu może wyglądać następująco:

```
var wyrNick=/^[a-z]{3,8}$/;
```

Wyrażenie regularne do **testowania numeru karty kredytowej**. Numer ten składa się z szesnastu cyfr, mogą one być pogrupowane po cztery i rozdzielone spacją lub znakiem -.

```
var wyrNrKarty=/^(\d{4}[ -]?){3}\d{4}$/;
```

Wyrażenie będzie pasowało do ciągu, który zaczyna się od czterech cyfr i spacji lub znaku -, fragment ten powinien się powtórzyć trzy razy, a potem znów mamy cztery cyfry.

```
var ciag1="1234-1234-4445-7665";
var ciag2="1212 12211212-1122";
var ciag3="1221_1112_1112_1211";
alert(wyrNrKarty.test(ciag1));//zwróci true
alert(wyrNrKarty.test(ciag2));//zwróci true
alert(wyrNrKarty.test(ciag3));//zwróci false
```

Wyrażenie do testowania poprawności adresu **e-mail**. Adres e-mail powinien się składać z ze znaku @ poprzedzonego znakami alfanumerycznymi (pierwszy powinien być literą), po którym powinny wystąpić inne znaki alfanumeryczne i ewentualnie znak ".".

```
var wyrEmail=/^[A-Za-z]\w+@(\w+\.)+\w+$/;

var ciag1="ktos@gdzies.pl";
var ciag2="1ktos@.pl";
var ciag3="a_@aaa.aaa.gov.pl";
alert(wyrEmail.test(ciag1));//zwróci true
alert(wyrEmail.test(ciag2));//zwróci false
alert(wyrEmail.test(ciag3));//zwróci true
```

Oczywiście jest to sposób, który nie wyklucza wszystkich błędnych możliwości, ale na pewno pozwoli na wykrycie wielu błędów.

17 Zasoby sieciowe związane z językiem JavaScript

Poniżej przedstawione zostały serwisy WWW zawierające specyfikacje, przykłady skryptów i wskazówki dotyczące programowania w JavaScript:

- http://www.ecma-international.org/ - specyfikacja języka ECMAScript.

- JavaScript Kit - wiele przykładów wykorzystania języka JavaScript oraz CSS.
- JavaScript Source - duża ilość gotowych do wykorzystania na własnych stronach skryptów.
- WebSite Tips - ćwiczenia i przykładowe skrypty.
- Microsoft JScript Reference - opis języka JScript.
- Netscape JavaScript Developer Central - zasoby informacyjne o języku JavaScript, grupy dyskusyjne oraz przykładowe kody.
- W3C World Wide Web Consortium - strona zawiera opisy standardów języka HTML, CSS i DOM.
- WebReference.com - praktyczne uwagi dla osób korzystających z języka JavaScript oraz innych technologii, m. in. DHTML, HTML, XML oraz Perl.
- WEBHELP - polski serwis zawierający wiele artykułów i innych materiałów dotyczących m. in. php, mysql, html, css, flash, javascript, asp, cgi.
- JavaScript Planet PL - polskie tłumaczenie serwisu JavaScript Planet.
- Kurs języka HTML - poradnik webmastera - bardzo dobry kurs HTML z elementami CSS i JavaScript.