MODUŁ 5:
JavaScript - programowanie w środowisku przeglądarki

Cel modułu

Niniejszy moduł stanowi wprowadzenie do podstawowej technologii dostępnej dla programowania funkcjonalności ze strony klienckiej, a mianowicie – języka skryptowego JavaScript[1]. Niekiedy mówi się, że technologie klienckie aplikacji WWW są ustandaryzowane wg trzech „części mowy":

- rzeczowniki – HTML (struktura i treść dokumentu)
- przymiotniki – CSS (wygląd i układ dokumentu)
- czasowniki – JS (modyfikacja zawartości i wyglądu dokumentu, interakcja z użytkownikiem i serwerem)

Ze względu na miejsce funkcjonowania skryptów JS, równie ważne jak konstrukcja samego języka jest środowisko, na którym skrypty te operują. W związku z tym zapoznamy się również z interfejsem programistycznym DOM (Document Object Model), określającym, w jaki sposób z poziomu języka JavaScript będzie widzialna i aktualizowalna struktura dokumentu WWW. W aktualnym module skupimy się na standardowych elementach środowiska w przypadku uruchamiania skryptu w kontekście strony w przeglądarce. Specyficzną cechą dzisiejszych aplikacji WWW jest to, że ich funkcjonalność nie tylko obsługuje interakcję w architekturze klient-serwer jaką stanowi WWW, ale też, że sama w sobie jest rozproszona zgodnie z tą architekturą. Innymi słowy, funkcjonalność ta może działać w dwóch wzajemnie uzupełniających się kontekstach: w środowisku serwera oraz w środowisku klienta.

Efekty kształcenia

Niniejsza lekcja służy osiągnięciu następujących efektów kształcenia.

Wiedza

Po ukończonym kursie student / studentka:

- Charakteryzuje interfejs programistyczny DOM HTML,
- Objaśnia ograniczenia dotyczące skryptu działającego po stronie przeglądarki,
- Wskazuje sposoby powiązywania kodu JS ze stroną oraz motywy ich doboru, w tym wykorzystanie Content Delivery Network.

Umiejętności

Po ukończonym kursie student / studentka:

- Buduje prostą funkcjonalność aktywnej strony WWW przy użyciu języka JavaScript,
- Uzupełnia wiedzę na temat języka JavaScript stosując polskojęzyczne i anglojęzyczne źródła dostępne online.

Uzasadnienie

JavaScript aspiruje obecnie do roli najważniejszego języka programowania w konstrukcji aplikacji WWW. Stanowi powszechni i dobrze ustandaryzowany „wspólny mianownik" dla technologii WWW

[1] W dalszej części tekstu skrótowo określany JS

działających po stronie klienta umożliwiających tworzenia dokumentów aktywnych WWW. Bogate biblioteki i ramy programistyczne oparte na JavaScript zapewniają programistom podniesienie poziomu abstrakcji programowania przy realizacji typowych zadań. Kolejną przesłanką dla bliższego zapoznania się z JavaScript jest dostępność dojrzałych rozwiązań umożliwiających stosowanie tego języka również po stronie serwera, do tworzenia dynamicznych stron WWW.

Kontekst działania JavaScript

Mieliśmy już okazję zapoznać się z dwiema kluczowymi technologiami funkcjonującymi po stronie klienckiej aplikacji WWW, a więc – wspieranymi przez przeglądarkę: językom znaczników HTML i XHTML, wyznaczającym strukturę i – częściowo – semantykę dla treści dokumentów WWW, oraz językowi CSS – pozwalającemu określić sposób prezentacji (formatowanie) dokumentu. JS stanowi kolejny element tego ekosystemu. Z punktu widzenia celów i zasięgu działania tego języka w środowisku przeglądarki – wykazuje on pewne analogie do CSS; mianowicie kod JS, podobnie jak CSS:

- Może być osadzany wewnątrz dokumentu znacznikowego, lub też być umieszczany w osobnym pliku połączonym poprzez hiperłącze z obsługiwanym dokumentem,
- Dysponuje dostępem do hierarchicznej struktury danych reprezentującej bieżący dokument WWW,
- Jego interpretacja może przesądzać o tym, w jakiej postaci dokument zostanie zaprezentowany użytkownikowi, oraz jak będzie się zmieniał zależnie od działań użytkownika oraz parametrów okna przeglądarki.

O ile jednak w przypadku zastosowania CSS mamy raczej do czynienia z przetwarzaniem deklaratywnym, o tyle JS, jako typowy język programowania, operuje na strukturze dokumentu w sposób imperatywny.

Łączenie skryptu z dokumentem WWW

Fakt, że kod JS działa po stronie klienta, każe zadać przede wszystkim pytanie o bezpieczeństwo. Problem ograniczonego zaufania do treści odwiedzanych stron internetowych rozwiązano przyjmując, że środowiskiem działania JS nie jest maszyna klienta, ani nawet cała przeglądarka, a jedynie bieżące okno przeglądarki wraz z załadowanym doń dokumentem. Tym samym kod JS pracuje przede wszystkim na strukturze bieżącego dokumentu, mając do dyspozycji parametry bieżącego okna przeglądarki, oraz na zasobach zdalnych udostępnianych przez serwer, z którego tę stronę pobrano. JS nie może samodzielnie wykonać tzw. operacji uprzywilejowanych, do których zaliczamy m.in. dostęp do lokalnych plików czy drukowanie.

Oznacza to, że kod JS, podobnie jak CSS, trafia do klienta wraz z dokumentem, z którym go powiązano. Powiązaniu temu służy znacznik[2]:

```
<script type="text/javascript">
  …tutaj wiersze kodu…
</script>
```

Ze względu na łatwość pielęgnacji, a zwłaszcza – możliwość wielokrotnego użycia tego samego kodu na różnych stronach, częściej izoluje go w osobnych dokumentach@@@:

[2] W HTML5 atrybut type nie jest konieczny: domyślnie założono, że w tym wypadku będzie to skrypt w JS

```
<script type="text/javascript" src="nazwaPliku.js">
</script>
```

(Uwaga – także i wówczas znacznik zamykający jest niezbędny.)

Taki element skryptowy może zostać umieszczony w dokumencie HTML zarówno wewnątrz elementu `head` jak i w ramach elementu `body`. Zwykle skrypty łączy się z dokumentem w sekcji nagłówkowej (to bardziej odpowiednie z punktu widzenia semantyki elementu `head`), jednakże łączenie skryptów w elemencie `body` (np. pod jego koniec) można rozważyć, gdy:

- Zależy nam na jak najszybszym wyświetleniu użytkownikowi dokumentu (pobieranie skryptów nie opóźni wówczas pobrania treści dokumentu[3]),
- Zamierzamy umieścić instrukcje wywołujące funkcjonalność pracującą na strukturze dokumentu (w przeciwnym razie, ze względu na kolejność interpretowania dokumentu, odpowiednie struktury dokumentu mogłyby nie być jeszcze dostępne w momencie uruchomienia instrukcji).

Kontrola bieżąca	( )	*komentarz odpowiedzi*
Kod JavaScript możemy umieszczać:		
Tylko w plikach zewnętrznych względem dokumentu HTML – by nie naruszyć reguł poprawności składniowych HTML		Nie. Ma to istotne zalety, ale nie jest niezbędne.
W plikach zewnętrznych oraz wewnątrz elementu `body` dokumentu HTML.		Nie tylko.
W plikach zewnętrznych oraz wewnątrz elementu `head` dokumentu HTML.		Nie tylko.
W plikach zewnętrznych oraz wewnątrz elementu `head` lub `body` dokumentu HTML.	x	Tak.

Przekazywanie rezultatów użytkownikowi

Aby zyskać możliwość budowy pierwszego przykładu, musimy wiedzieć, w jaki sposób kod jest w stanie wyświetlać rezultaty dla użytkownika.

W przypadku kodu umieszczonego w ciele dokumentu, możemy sięgnąć po najbardziej bezpośrednią formę wyprowadzania tekstu, jaką jest metoda wprowadzająca tekst do ciała dokumentu:

```
document.write(…)
```

Jak widzimy, jest to metoda udostępniana przez obiekt o nazwie `document`, będący predefiniowanym elementem środowiska.

Dla zilustrowania powyższych informacji – utwórzmy minimalny dokument HTML5 zawierający kod JS:

```
<!DOCTYPE html>

<html>

  <head>
```

[3] Ponadto pamiętajmy, że CSS i JavaScript stają się w tym przypadku dwoma elementami kodu, który zostanie zastosowany do przetworzenia dokumentu znacznikowego stanowiącego bieżącą stronę. Z tego punktu widzenia takie wyraziste wyakcentowanie kolejności przetwarzania przez rozmieszczenie odsyłaczy: CSS na początku JS pod koniec dokumentu, pozwoli uniknąć zakłóceń przetwarzania CSS przez przetwarzanie JS.

```
    <title>Hello!</title>
  </head>
  <body>
    <script>
      document.write("Hello, world!");
    </script>
  </body>
</html>
```

Drugim popularnym sposobem jest wykorzystanie okien dialogowych. Mają one charakter modalny i mogą służyć:

- Wyświetleniu powiadomienia wymagającego zatwierdzenia – metoda `window.alert()`
- Wyświetleniu komunikatu i uzyskaniu decyzji Tak/Nie (jako zwróconej wartości boolean) – metoda `window.confirm()`
- Wyświetlenie komunikatu i pobranie od użytkownika wartości tekstowej – metoda `window.prompt()`.

Oferujący powyższe metody obiekt `window` stanowi globalne środowisko działania skryptu, dlatego też jego nazwę można pominąć, przez co wywołania ww. metod wyglądają jak wołania tradycyjnych (nie obiektowych) funkcji.

Możemy także, zwłaszcza w ramach diagnozowania kodu, posłużyć się metodą log obiektu `console`:

```
console.log("Hello!");
```

W tym przypadku jednak rezultat zaobserwujemy tylko wtedy, gdy zadbamy o otwarcie w naszej przeglądarce okna konsoli.

Kontrola bieżąca	( )	*komentarz odpowiedzi*
Dla skryptu JS działającego w przeglądarce obiekt-korzeń reprezentujący dostępne środowisko znajdziemy pod nazwą:		
`console`		Nie. To tylko fragment środowiska. Możesz sprawdzić, jak to środowisko prezentuje web debugger…
`window`	x	Tak.
`env`		Nie. Możesz sprawdzić, jak to środowisko prezentuje web debugger (zob. nast. sekcja)
`document`		Nie. To tylko fragment środowiska. Możesz sprawdzić, jak to środowisko prezentuje web debugger (zob. nast. sekcja)
`machine`		Nie. Możesz sprawdzić, jak to środowisko prezentuje web debugger (zob. nast. sekcja)

Debugging kodu JavaScript

Powyższa wzmianka prowadzi nas ku ważnemu zagadnieniu, a mianowicie – diagnostyce kodu JS. Warto ten temat podjąć już teraz. Wprawdzie nasze ćwiczeniowe zadania nie będą nadmiernie złożone, ale z kolei warto skorzystać z narzędzi diagnostycznych także dla samego poznania szczegółów działania języka oraz jego środowiska.

Przykładem silnego narzędzia dewelopersko-diagnostycznego jest choćby Firebug (http://getfirebug.com/). Obok samej analizy działania kodu JS oferuje też rozbudowane narzędzia dla testowania CSS. Warto wreszcie dodać, że dość rozbudowane narzędzia wspierające programistę występują jako integralna część poszczególnych przeglądarek (warto, zależnie od używanej przeglądarki, sprawdzić zakres możliwości zestawu narzędzi występujących w sekcji o nazwie *Web developer*, *Programmer tools* lub podobnej).

<<SCREENCAST S6 – Użycie Firebug do analizy wykonana skryptu JavaScript: pokazanie globalnego środowiska oraz możliwości krokowego wykonania>>

Document Object Model

Specyfika programowania w środowisku strony WWW polega m.in. na tym, że skrypt nie działa w pustej przestrzeni, ale ma w zasięgu swojej widoczności liczne struktury danych reprezentujące okno, dokument i szereg jego właściwości. Tym samym wiele zadań sprowadza się nie do tworzenia nowych struktur danych, ale do nawigacji po oraz ewentualnego modyfikowania struktur zastanych.

Dlatego też, z punktu widzenia prostszych zadań programistycznych, od znajomości szczegółów konstrukcji języka i jego zaawansowanych właściwości istotniejsze jest poznanie interfejsu programistycznego do struktury dokumentu.

W początkowym okresie rozwoju JS interfejs ten nie był opisany standardem i oferował dostęp do wybranych tylko, uchodzących za najistotniejsze dla przetwarzania, składników dokumentu HTML.

Document Object Model (DOM) jest standardem konsorcjum W3C (www.w3.org). Mówiąc o tym interfejsie należy rozróżnić jego dwa odmienne komponenty. Otóż DOM zaprojektowano m.in. dla przetwarzania dowolnych dokumentów XML. W tym zastosowaniu interfejs musi mieć charakter generyczny, to jest być przygotowanym na przetwarzanie dowolnych hierarchii elementów o nazwach, których lista nie jest z góry znana i zamknięta. Stąd też ta część standardu operuje m.in. takimi pojęciami jak Węzeł (*Node*), Element (*Element*), Atrybut (*Attribute*), Dokument (*Document*) i pozwala nawigować po i modyfikować hierarchie obiektów o ww. typach.

Z kolei, interesujący nas w tym module, DOM HTML (zob. http://www.w3.org/TR/DOM-Level-2-HTML/), jak sama nazwa wskazuje jest interfejsem dla przetwarzania dokumentów HTML, to jest może skorzystać ze statycznej wiedzy na temat zdefiniowanych w tym języku elementów i atrybutów, oferując tym samym bardziej precyzyjne interfejsy (to jest, np. uwzględniając w definicji typu odpowiedniego obiektu, że obiekt reprezentujący element o nazwie `a` posiada atrybut o nazwie `href`).

Jak sugeruje powyższy opis, dysponując imperatywnym językiem programowania JS oraz interfejsem do struktury dokumentu DOM HTML, jesteśmy w stanie dowolnie przebudowywać załadowany już do okna przeglądarki dokument, modyfikować zawartość jego elementów i atrybutów, czy wręcz – zbudować nowy dokument od zera. Jednakże na takim przekształcaniu dokumentu znacznikowego możliwości DOM HTML się nie kończą. Obejmuje on dwa inne istotne elementy:

- Możliwość tworzenia i zmieniania ustawień stylów CSS zastosowanych do poszczególnych elementów;
- Możliwość uruchamiania kodu reagującego na poszczególne rodzaje predefiniowanych zdarzeń związanych z dokumentem. Przykładami takich zdarzeń są zdarzenia:
    - myszy: `onMouseOver`, `onMouseOut`, `onClick`, `onMouseDown`, `onMouseUp`;
    - formularzy: `onBlur`, `onFocus`, `onChange`, `onSelect`, `onSubmit`;
    - inne: `onLoad`, `onUnload`, `onError`, `onAbort`;

Obsługę zdarzeń dotyczących poszczególnych obiektów dokumentu definiujemy kojarząc z nimi odpowiednie funkcje obsługi. Można tego dokonać dwojako:

- Wołając na obiekcie reprezentującym wybrany element metodę `addEventListener` – pamiętajmy, że JS umożliwia przekazanie funkcji jako parametru wołania takiej metody;
- Umieszczając fragment kodu (zwykle stanowiący samo wywołanie operacji) w atrybucie on* (gdzie * oznacza nazwę zdarzenia) wybranego elementu HTML.

Warto dodać, że z poziomu funkcji obsługującej zdarzenie, generujący to zdarzenie element będzie dostępny jako obiekt, na rzecz którego wywołano tę operację, identyfikowany słowem kluczowym `this`.

Kontrola bieżąca	( )	*komentarz odpowiedzi*
Interfejs DOM **nie** pozwala skryptom JavaScript na:		
usuwanie istniejących elementów w dokumencie HTML		Nie. Ta funkcjonalność jest dostępna.
tworzenie nowych węzłów dokumentu		Nie. Ta funkcjonalność jest dostępna.
modyfikacje stylów CSS przypisanych do elementu		Nie. Ta funkcjonalność jest dostępna.
przeszukiwanie dokumentów HTML zgromadzonych w katalogu domowym użytkownika	x	Tak. Ta funkcjonalność wykracza poza dostępne JS środowisko.
obsługę zdarzeń myszy		Nie. Ta funkcjonalność jest dostępna.

Przykład – konstruktory, DOM, prototypy

Jak zatem widzimy, uruchomiony w kontekście strony kod, choć sam może nie powoływać żadnych obiektów, to działa w bogatym środowisku zawierającym m.in. drzewo dokumentu, którego obiekty implmenentują niekiedy bogatą funkcjonalność.

Poniższy krótki przykład, choć nieco sztuczny (w obliczu dostępności licznych bibliotek i frameworków) ilustruje m.in. możliwość wykorzystania mechanizmu prototypów do wzbogacenia funkcjonalności predefiniowanych typów obiektów.

(Przy okazji, jako przeciwieństwo składni użytej w początkowym przykładzie – tutaj zastosowaliśmy rygory składniowe specyficzne dla języka XHTML.)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head><title>Dialogi, zdarzenia, prototyp</title>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
    // <![CDATA[
```

```
      function message(par, type){
            if (type == "alert") alert(par);
            else if (type=="confirm") confirm(par);
            else prompt(par,"");
      }

      function getRadioValue(radioName){
            for(i=0;i<this[radioName].length;i++)
                  if (this[radioName][i].checked) return
this[radioName][i].value;
            return null;
      }

      HTMLFormElement.prototype.getRadioValue = getRadioValue;

      HTMLParagraphElement.prototype.alterStyle = function (x, v) {
              this.style[x] = v;
          }

      function podlacz() {
          ps = document.body.getElementsByTagName("p");
          for (i = 0; i < ps.length; i++) {
              ps[i].addEventListener("mouseover", function () {
this.alterStyle('color', 'red'); });
          }
          for (i = 0; i < ps.length; i++) {
              ps[i].addEventListener("mouseout", function () {
this.alterStyle('color', ''); });
          }
      }
          // ]]>
</script>
</head>
<body onload="podlacz()">
<h3>JavaScript</h3>
<p>naprowadz kursor myszy tu</p>
<p id="sample_par">albo tu</p>
<form id="form1" action="dialogi.htm">
<p><input type="radio" id="r_Alert" name="messageType" value="alert"
checked="checked"/> Alert </p>
<p><input type="radio" id="r_Confirm" name="messageType" value="confirm" />
Confirm </p>
<p><input type="radio" id="r_Prompt" name="messageType" value="prompt" />
Prompt </p>
<p><input type="text" id="komunikat" /></p>
<p>Tekst komunikatu: <input type="button" value="Pokaz"
onclick="message(document.getElementById('komunikat').value,
getElementById('form1').getRadioValue('messageType'));"  /></p>
</form>
</body>
</html>
```

Zwróćmy uwagę na następujące składniki rozwiązania:

- Podłączenie obsługi zdarzenia (`onload`) za pomocą atrybutu w elemencie HTML.
- Podłączenia zdarzeń za pomocą metody (`addEventListener`) w interfejsie DOM.
- Wykorzystanie JS do sprawdzenia wartości w elementach formularza.

- Wykorzystanie obiektów prototypu dla określonego typu obiektów DOM (reprezentujących odpowiednie typy elementów HTML) do wyposażenia elementów dokumentu w dodatkowe cechy.
- Dynamiczne modyfikowanie ustawień stylu wybranych elementów dokumentu.
- Zademonstrowanie działania okien dialogowych.

JQuery

Po zapoznaniu się z koncepcyjnie przejrzystymi, acz żmudnymi w stosowaniu elementarnymi funkcjonalnościami interfejsu DOM możemy obecnie poszukać sposobów podniesienia poziom abstrakcji, a wraz z nim produktywności.

JQuery jest biblioteką JavaScript ułatwiającą dostęp do oraz manipulację drzewem DOM. Zgodnie z tym, co sugeruje sama nazwa, pozwala ona zaoszczędzić szczególnie dużo nakładu przy działaniach związanych z wyszukaniem w strukturze dokumentu pożądanych elementów.

Chcąc korzystać z jej funkcji musimy zadbać o połączeniu ze stroną kodu stanowiącego tę bibliotekę. Może to być pobrany przez nas plik z kodem albo – jak w poniższym przykładzie, odsyłacz do zasobu dostępnego w ramach CDN (*Content Delivery Network*).

```
<body>
     <script
   src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"
   ></script>

  <script>
      ... tu nasz kod ...
  </script>
</body>
```

Dostarczenie kodu w ten sposób wiąże się z powstaniem nowej zależności zewnętrznej dla naszej witryny, ma jednak szereg zalet:

- zmniejszenie ruchu generowanego na naszym serwerze (kod biblioteki pobierany jest z innych lokalizacji),
- wysoka dostępność i przepustowość CDN,
- potencjalnie szybsze załadowanie strony związane ze zrównolegleniem żądań pomiędzy więcej niż jeden serwer (treści strony – nasz serwer; biblioteka – serwery CDN),
- potencjalna szansa na uniknięcie transferu – w sytuacji, gdy więcej niż jedna witryna odwiedzana przez użytkownika wykorzystuje ten sam zasób z CDN.

Głównym walorem JQuery jest ułatwienie eksploracji struktury dokumentu przez dostarczenie zwięźlejszych, deklaratywnych środków. Centralną rolę odgrywa tu wszechstronna funkcja o nazwie $, która może realizować m.in. zapytania do struktury dokumentu. Przykładem może tu być wykorzystanie selektorów CSS – np.

```
var akapity = $("p");
```

czy znacznie bardziej wyrafinowane złożenia selektorów – np.

```
$("#main > div.entry:visible a.comments:contains('Dodaj komentarz')");
```

Funkcji $ można użyć również do tworzenia nowych obiektów – np.

```
var $link = $("<a></a>");
```

Interfejs JQuery jest reprezentantem podejścia tzw. *fluent API* pozwalającego na łączenie wywołań metod w łańcuchy przybierające postać intuicyjnych dla człowieka wykazów sekwencji działań – np.

```
$("#main").find("p").not(".ohmy").addClass("new");
```

Zwróćmy uwagę że zapis ten pozwala uniknąć podawania złożonej, bardziej podatnej na błędy struktury selektorów oraz że poszczególne wywołania metod realizują implicite iterację po pośrednich rezultatach.

Biblioteka udostępnia też znacznie wygodniejszą niż w czystym DOM formę przyłączania zachowania do zdarzeń w interfejsie użytkownika – m.in.:

- `ready( fn )` – zdarzenie wywoływane po załadowaniu dokumentu
- `hover( fn_over, fn_out )` – odpowiednik onMouseOver i onMouseOut
- `blur( fn )` – element traci focus
- `click( fn )` – kliknięcie elementu
- `dblclick( fn )` – double click elementu
- `focus( fn )` – gdy element uzyskuje focus

Metody te możemy wywołać na rezultacie „zapytania” zrealizowanego funkcją $().

Według podobnego wzorca można modyfikować zawartość wyselekcjonowanych elementów – m.in. metodami:

- `html( )` – pobranie innerHTML pierwszego elementu, zwraca String
- `html( val )` – ustawienie innerHTML dopasowanych elementów
- `text( )` – pobranie tekstu wybranych elementów, zwraca String
- `text( val )` – ustawienie tekstu dopasowanych elementów
- `append( content )` – dokleja content do dopasowanych elementów
- `appendTo( selector )` – dokleja wybrane elementy do elementów wskazanych przez selektor
- `prepend( content )` – jw., na początku
- `prependTo( selector )` – jw., na początku

Biblioteka obejmuje też (nieprezentowane tutaj) metody do:

- Opakowywania czymś wybranych elementów
- Wstawiania przed i po wybranych elementach
- Zastępowania
- Usuwania elementów
- Klonowania elementów (w tym głębokiego)

Istotne w efektywnym modyfikowaniu wyglądu elementu może okazać się modyfikowanie jego klasy. Dysponujemy w tym celu następującymi metodami:

- `addClass( class )` – dodanie klasy
- `hasClass( class )` – sprawdzenie, czy obiekt ma klasę
- `removeClass( class )` – usunięcie klasy
- `toggleClass( class )` – „przełączenie” klasy
- `toggleClass( class, switch )` – dodanie, jeśli `switch=true`, usunięcie jeśli `false`

Interakcję z użytkownikiem można wesprzeć różnego rodzaju efektami zastosowanymi do wyselekcjonowanych elementów, w tym:

- `show( )` – pokazywanie
- `hide( )` – ukrywanie
- `show( speed, callback )` – pokazywanie z wywołaniem funkcji po zakończeniu animacji
- `hide( speed, callback )` – ukrywanie z wywołaniem funkcji po zakończeniu animacji
- `toggle( )`
- `toggle( switch )` – `switch = true` pokazuje, `switch = false` ukrywa
- `toggle( speed, callback )`
- `slideDown( speed, callback )` – pokazywanie
- `slideUp( speed, callback )` – ukrywanie
- `slideToggle( speed, callback )` – pokazanie lub ukrycie
- `fadeIn( speed, callback )` – podobnie jak wyżej, ale poprzez zmianę przezroczystości
- `fadeOut( speed, callback )`
- `fadeTo( speed, opacity, callback )`

Dostępne są również metody umożliwiające bezpośredni dostęp do i manipulację ustawieniami stylistycznymi poszczególnych elementów:

- `css( name )` – wartość własności `css` o nazwie `name` w pierwszym dopasowanym elemencie
- `css( properties )` – ustawienie własności css w dopasowanych elementach
- `css( name, value )` – ustawienie własności o nazwie `name` na wartość `value` w dopasowanych elementach

Można również m.in. odczytywać bieżący rozmiar elementu i bezpośrednio nim manipulować:

- `height( val )` – ustawienie wysokości elementów
- `width( val )` – ustawienie szerokości elementów

Dalsze informacje

Nie sposób wyczerpać tematu języka JS w pojedynczym module kurs. Z uwagi na obecną ważność i powszechność tego języka, zachęcamy do poszerzenia swojej wiedzy o zagadnieniu. Pośród dostępnych nieodpłatnie źródeł można wskazać m.in.:

- Tutoriale dotyczące DOM HTML w JS: https://dom-tutorials.appspot.com/static/index.html lub http://www.advanced-javascript-tutorial.com/the-html-document-object-model.cfm
- Omówienie DOM HTML na portalu W3Schools: http://www.w3schools.com/js/js_htmldom.asp
- Omówienie JQuery na portalu W3Schools: https://www.w3schools.com/jquery/default.asp

Zadanie

Sporządź formularz zgodny z tematyką Twojego projektu, zawierający kilka pól znakowych. W ramach strony zbuduj oparty na JavaScript mechanizm walidacji, który obejmie:

1) dla co najmniej jednego pola tekstowego: kontrolowanie, czy zawiera dane o odpowiedniej długości (np. z zakresu długości od 2 do 60 znaków)
2) dla pola przeznaczonego na wpisanie adresu e-mail – sprawdzenie (w tym celu warto zastosować wyrażenie regularne) czy wprowadzony łańcuch znaków odpowiada składni adresu e-mail.

Po wykryciu błędów walidacji powinny być pokazywane oznaczenia przy odpowiednich formatkach oraz wyświetlany (w sąsiedztwie przycisku wysyłania) tekst podsumowania błędów.
Sprawdzenie warunków powinno być wykonywane w reakcji na zdarzenie **submit**.
(jeśli zapewnimy, że wywoływany w odpowiedzi na to zdarzenie kod dostarczy przeglądarce rezultatu ewaluowanego jako false – zapobiegnie to wysłaniu formularza pod adres docelowy wskazany atrybutem action)