bootcamp online

JAVASCRIPT

Witaj w drugim tygodniu Bootcampu

Rozpoczynamy drugi tydzień zmagań w ramach Bootcampu JavaScript. Mam nadzieję, że poprzedni tydzień był dla Ciebie ciekawy oraz zawierał wiele cennych informacji. W tym tygodniu kontynuujemy poznawanie **Obiektowego Modelu Dokumentu**, a także rozpoczynamy bardziej zaawansowane tematy, takie jak **Dziedziczenie Prototypowe** i **Ajax**.

Piotr Palarz

Piotr Palarz

eduweb.pl

--- Zadania na tydzień 2 ---

*Zadania domowe spakuj ZIPem i umieść na kanale **#prace_domowe** na Slacku.*
W przypadku zadań, gdzie niezbędny jest dodatkowy kod HTML i CSS, nie musisz się skupiać na tym, aby strona wyglądała pięknie. Jeśli masz czas i chęci, dopracuj swój projekt, ale najważniejszy pozostaje i tak kod JavaScript.

1. Pokazywanie ukrytego elementu

Stwórz projekt z ukrytym elementem HTML (np. `<div>` z `display:none` w CSS), a także przyciskiem (`<button>`). Przypisz na kliknięcie przycisku funkcję, która pokaże ukryty element, gdy jest niewidoczny i ukryje go, gdy jest widoczny. Podczas zmiany stanu widoczności tego elementu, zmień również tekst przycisku np. z "*Pokaż treść*" na "*Ukryj treść*" i na odwrót.

2. Walidator formularza

Stwórz prosty walidator formularza, który zawierał będzie pola `<input>` o typach "text", "email", "number" oraz element `<textarea>`. Dodaj do elementu `<form>` atrybut `novalidate`, aby wyłączyć domyślną walidację przeglądarki. Od Ciebie zależy, czy chcesz wyświetlać komunikaty o błędach czy tylko podświetlać niepoprawnie uzupełnione pola np. na czerwono. Za poprawnie uzupełnione pole `<input>` o typie "text" lub pole `<textarea>` uznajemy takie, które ma wpisany przynajmniej jeden znak. W przypadku pola o typie "email" sprawdź czy zawiera ono znak @, a w przypadku pola o typie "number" czy podana wartość jest liczbą (pamiętaj, że DOM zwróci Ci zawsze wartość o typie `String`, więc musisz znaleźć sposób, jak sprawdzić czy string ten zawiera wyłącznie liczbę).

Uwaga: Jeśli stworzysz pole `<input>` o typie "number" to w niektórych przeglądarkach nie będzie możliwości wpisania innych znaków niż liczby. Mimo to napisz logikę, która taką walidację wykona, a aby móc wprowadzić inne znaki niż tylko liczby, zmień na czas testów atrybut `type` pola `<input>` z "number" na "text".

3. Odliczanie od 10 do 0

Stwórz projekt, który po uruchomieniu odpowiedniej funkcji, pozwoli na odliczanie **od 10 do 0**. Wszystkie wartości powinny być wyświetlane na stronie, a czas pomiędzy zmianą wartości powinien wynosić **1 sekundę**. Choć cały Twój kod może być podzielony na wiele funkcji, jedna z nich powinna uruchamiać proces odliczania. Przy wywołaniu tej funkcji, daj możliwość przekazania innej funkcji jako argument. Przekazaną funkcję wywołaj, gdy licznik osiągnie wartość **0**.

Istotą przekazywania jednej funkcji do drugiej (w tym przypadku nazwalibyśmy ją funkcją **callback**) jest to, aby dać użytkownikowi naszego kodu pewną gotową funkcjonalność (odliczanie od 10 do 0), ale także możliwość dodania czegoś od siebie, tj. wykonania własnej funkcji po zakończeniu odliczania. W przekazanej funkcji możesz wpisać po prostu `console.log("Odliczanie zakończone!")`.

Podpowiedź: użyj funkcji `setTimeout` wywołując ją wielokrotnie zamiast `setInterval`, którą wykonałbyś wyłącznie raz.

4. Dziedziczenie z klasy EventEmitter

Przygotowany pod adresem http://pastebin.com/YEBncx0d kod zmodyfikuj tak, aby obiekty tworzone z klasy `Database` mogły korzystać z wszystkich metody klasy `EventEmitter`. Na chwilę obecną, podany kod wygeneruje błąd, gdyż klasa `Database` nie zawiera metody `on` oraz `emit`. Skorzystaj z dziedziczenia prototypowego, aby klasą nadrzędną dla `Database` stała się klasa `EventEmitter`. Zadanie to wymaga od Ciebie dopisania wyłącznie kilku linijek kodu.

5. Funkcja debounce

Stwórz funkcję o nazwie `debounce`, która przyjmie przy wywołaniu dwa argumenty. Pierwszy z nich to inna **funkcja** do późniejszego wywołania, a drugi to **czas** w milisekundach. Po takim wywołaniu, funkcja ta powinna zwrócić **nową funkcję**, którą można zapisać np. w zmiennej. Następnie tę nową funkcję będzie można wielokrotnie wywoływać (a ona powinna wywołać pierwotnie przekazaną funkcję), jednak jeśli czas pomiędzy poszczególnymi wywołaniami będzie krótszy, niż podany wcześniej (wspomniany

argument z czasem w milisekundach), to funkcja nie powinna nic zrobić, ale ustawić licznik na kolejne **Xms** i dopiero wtedy się wywołać.

Rozwiązania typu **debounce** stosuje się np. przy obsłudze zdarzenia `scroll` w przeglądarkach, które wywoływane jest bardzo wiele razy. Jeśli nie chcemy, aby nasza funkcja negatywnie wpływała na wydajność podczas scrollowania, a wystarczy, że wywoła się np. 100ms po zakończeniu scrollowania, wtedy stosujemy funkcję debounce. Innym przykładem jest np. zdarzenie `resize`, które też jest wielokrotnie wywoływane podczas skalowania okna przeglądarki, a nam może zależeć, aby wywołać pewien kod dopiero wtedy, gdy taka akcja się zakończy.

Całość przetestować możesz z użyciem tego kodu: https://pastebin.com/J0BJVqtR. Twoim zadaniem jest dopisanie funkcji `debounce`, aby ten kod zaczął działać.

Podpowiedź: będziesz potrzebował funkcji `setTimeout` i `clearTimeout`.

Obiektowy Model Dokumentu

Znasz już podstawy języka JavaScript i wiesz jak dodawać swoje skrypty do stron internetowych. Wówczas Twój kod wykonywany jest w środowisku przeglądarki internetowej. I nie byłoby w tym nic nadzwyczajnego, potrzebujemy przecież jakiegoś środowiska, aby kod źródłowy uruchomić. Istotą środowiska uruchomieniowego jest jednak to, że może ono udostępniać swoje interfejsy programistyczne (**API**), z których korzystać można za pomocą języka JavaScript.

Tak jest właśnie w środowisku przeglądarki internetowej. Znając już język JavaScript, możesz manipulować stronami internetowymi. Wszystko to umożliwia odpowiednie API przeglądarek, które dostępne jest z poziomu języka JavaScript.

W praktyce oznacza to tylko (i aż) tyle, że pisząc kod JavaScript na potrzeby sieci, przeglądarka w momencie jego uruchamiania "wstrzyknie" do globalnego zakresu zmiennych pewne obiekty, do których Ty możesz się w swoim kodzie odwołać. Obiekty te mogą zawierać przydatne informacje na temat aktualnej strony lub przeglądarki (np. tytuł strony lub szerokość okna przeglądarki), ale także wiele metod (czyli po prostu funkcji), po wywołaniu których uzyskasz określone rezultaty.

Jeśli zatem chciałbyś dynamicznie utworzyć element `<div>`, dodać do niego klasę CSS i ustawić jego wewnętrzny tekst, a następnie taki element wstawić na stronę - możesz tego dokonać pisząc kod JavaScript. Innym przykładem może być przypisanie możliwości kliknięcia w przycisk, które wywoła funkcję pobierającą dane z serwera bez przeładowywania witryny.

DOM API !== JavaScript

Zapamiętaj jedno: **DOM API !== JavaScript**. Oznacza to, że na pytanie "*Czy w JavaScripcie można tworzyć nowe elementy HTML?*" powinieneś odpowiedzieć "*Sam język JavaScript tego nie umożliwia, ale w środowisku przeglądarki internetowej istnieją metody, które z jego pomocą pomogą taki efekt osiągnąć*". Wiem, że to pokrętna odpowiedź, ale jedyna prawdziwa. Kiedy zatem mógłbyś powiedzieć, że coś można zrobić w samym języku JavaScript? Zadajmy kolejne pytanie: "*Czy w języku JavaScript można stworzyć funkcję dodającą dwie liczby?*". Tutaj odpowiedź to po prostu "*Tak.*".

Znajomość języka JavaScript to zatem jedno, a umiejętność jego wykorzystania w różnych środowiskach to drugie. Musisz poznać nie tylko sam język, ale także API np. przeglądarek internetowych, jeśli chcesz budować aplikacje webowe. Ważne jest jednak to, że współpraca pomiędzy takim API a samym językiem jest naprawdę niezauważalna. Nadal pracujemy w znanym nam języku, po prostu na chwilę otrzymuje on dodatkowe obiekty i predefiniowane zmienne, pod którymi drzemie prawdziwa moc. Sama współpraca polega natomiast na tym, że o ile sam język JavaScript "nie potrafi" odczytać szerokości okna przeglądarki, to kiedy odwołamy się do predefiniowanej zmiennej `window`, a następnie do właściwości tego obiektu o nazwie `innerWidth`, to przeglądarka zwróci "do naszego kodu" wartość liczbową, a więc znany nam z JavaScriptu typ `Number`, np. `1200`. Oznaczać to będzie, że szerokość okna przeglądarki w momencie wykonywania się tego kodu to 1200 pikseli. A to, że zwrócony został typ `Number` oznacza, że po pierwsze jest on poprawny dla języka JavaScript, a po drugie, że możemy na nim pracować w dokładnie taki sam sposób, jakbyśmy robili to ze zmienną przechowującą wartość `1200`.

Mam nadzieję, że teraz zaczynasz dostrzegać, że sam język JavaScript to tylko pewne reguły, zapis, zbiór typów danych, operatorów i konstrukcji, lecz aby za jego pomocą robić naprawdę fascynujące rzeczy, potrzebne jest nam odpowiednie API.

Programowanie obiektowe

Oprócz Obiektowego Modelu Dokumentu, w tym tygodniu poznasz również techniki **programowania obiektowego** w języku JavaScript. Skupiają się one wokół pewnego mechanizmu tego języka, nazywanego **Dziedziczeniem Prototypowym**. To rzecz niełatwa do wytłumaczenie w tekście, dlatego omówimy ją teraz wyłącznie pobieżnie. Wszystko co istotne, znajdziesz w materiałach wideo.

Na czym zatem polega samo programowanie obiektowe? Na koncepcji **klas** i ich **instancji**. Brzmi strasznie? Nie martw się, nie jest to takie skomplikowane. Klasą określimy pewien schemat, szablon. Na jego podstawie chcielibyśmy później produkować nowe obiekty - instancje tej klasy.

Wyobraź sobie, że definiując klasę możemy jej przypisać właściwości i metody. To dokładnie ta sama terminologia jak w przypadku obiektów w języku JavaScript. W innych językach programowania te konstrukcje mogą być inaczej nazywane, ale chodzi o

dokładnie to samo. Najpopularniejszym przykładem klasy jest zazwyczaj **Osoba**. Definiując taką klasę, przypisalibyśmy jej pewne właściwości, które powinny mieć wszystkie utworzone później obiekty - osoby. Mogą to być np. **imię** i **nazwisko**. Mamy zatem pewien schemat **Osoby**. Teraz na bazie tego schematu utworzyć możemy wiele obiektów - instancji. Każdy z nich przy tworzeniu otrzyma "swoje własne" imię i nazwisko.

To jest właśnie istotą programowania zorientowanego obiektowo, że na podstawie pewnego schematu tworzymy wiele obiektów, ale mają one niezależne od siebie wartości ustalonych właściwości, np. "Anna Nowak" i "Jan Kowalski". Co jednak może (i powinno) być wspólne dla nich wszystkich? **Metody**.

Metodą jest funkcja, która może operować na zapisanych w obiekcie danych (pod właściwościami). Wyobraź sobie zatem metodę o nazwie `przedstawSie`. Potrzebuje ona do wykonania zadania imienia oraz nazwiska. Metodę dodajemy zatem przy definiowaniu klasy, podobnie jak właściwości `imię` oraz `nazwisko`. Następnie tworzymy dowolną liczbę nowych obiektów (osób) na podstawie tej klasy, każdemu z nich podając unikalne imię i nazwisko.

Od tej pory każdy z obiektów ma również metodę (w JavaScripcie dostępną na zmiennej po kropce) o nazwie `przedstawSie`. Kiedy ją wywołamy (jak funkcję, z parą nawiasów okrągłych) to otrzymamy wynik np. w postaci "`Witaj, nazywam się Jan Kowalski`". Kiedy tę samą operację wykonamy na obiekcie, któremu podaliśmy inne dane, możemy otrzymać wynik "`Witaj, nazywam się Anna Nowak`".

I choć na początku może wydaje się to trudne do zrozumienia, kiedy zobaczysz te konstrukcje w akcji, wszystko stanie się jasne. Zapamiętaj tylko tyle, że klasa jest to pewien **szablon**, który definiuje dane (właściwości) i metody (funkcje), a instancje są to **obiekty**, utworzone na podstawie tej klasy, które mają swoje własne dane oraz dziedziczą metody, które na tych danych operują.

To tyle teorii, do dzieła!