Contents

CSS

Wprowadzenie

CSS został stworzony w celu odseparowania struktury dokumentu od formy jego prezentacji. Separacja ta zwiększa zakres dostępności witryny, zmniejsza zawiłość dokumentu, ułatwia wprowadzanie zmian w strukturze dokumentu. CSS ułatwia także zmiany w renderowaniu strony w zależności od obsługiwanego medium (ekran, palmtop, dokument w druku, czytnik ekranowy). Stosowanie zewnętrznych arkuszy CSS daje możliwość zmiany wyglądu wielu stron naraz bez ingerowania w sam kod (X)HTML, ponieważ arkusze mogą być wspólne dla wielu dokumentów.

Rozmazany tekst

Hej, dziś krótko. Poniżej macie prosty tip na to jak przy pomocy CSS zrobić rozmazany tekst.

```
.class {
   color: transparent;
   text-shadow: 0 0 5px rgba(0,0,0,0.5);
}
```

Skalę rozmazania możecie zmienić manipulując wartością opactity kodu rgba, a także wartością rozmycia właściwości text-shadow. Proste. Dodatkowo mam dla was bardzo ciekawy tester prezentacji fontów dla urządzeń mobilnych. Tutaj w wygodny sposób możecie sprawdzić jak wybrany przez was font prezentuje się na ekranach telefonów i tabletów. Opcjonalnie można zmienić rozmiar testowanego kroju a także jego rozmieszczenie względem ekranu.

Clearfix

Prawdopodobnie najpolularniejszy trik jaki stosowano przy tworzeniu stron internetowych.

```
.clearfix:after {
     visibility: hidden;
     display: block;
     font-size: 0;
     content: " ";
     clear: both;
     height: 0;
     }
```

```
.clearfix { display: inline-block; }
/* start commented backslash hack \*/
* html .clearfix { height: 1%; }
.clearfix { display: block; }
/* close commented backslash hack */
```

Nicolas Gallagher opisał dokładnie "micro" clearfix, który jest nowszym podejściem do tematu.

```
.group:before,
.group:after {
    content: "";
    display: table;
}
.group:after {
    clear: both;
}
.group {
    zoom: 1; /* For IE 6/7 (trigger hasLayout) */
}
```

Aktaulna wersja, jeśli potrzebujesz wsparcia w IE8 lub nowszych przeglądarkach.

```
.group:after {
  content: "";
  display: table;
  clear: both;
}
```

Krótsze kody kolorów

Używanie skróconych kodów kolorów nie jest niczym nowym, ale wiele osób zwyczajnie, o tym zapomina. Niektórzy użytkownicy pamiętają jeszcze 216 bezpiecznych kolorów w przeglądarkach, choć dostępne spektrum dla 3 składowych to 256^3, czyli 16777216 kolorów.

Tutaj pojawia się pewien dylemat. Ile kolorów odróżnia ludzkie oko? To zależy indywidualnie od każdego z nas, ale w przypadku 16^3, otrzymujemy dokładne 4096 różnych kolorów. Czy to wystarczy? Oceńcie sami.

```
.white {
    color: #ffffff; /* use #fff */
}
.black {
```

```
    color: #000000; /* use #000 */
}
.test {
    color: #abc; /* use #aabbcc */
}
```

Powyższe przykłady są bardzo prozaiczne i zasadniczo możliwe jest zapisanie każdego z kolorów w formie 3 cyfrowego kodu. Kwestia jak duża będzie odległość pomiędzy odcieniami każdego koloru.

```
div {
    background: #c54218; /* use #c41 */
}
```

Potrzebowaliśmy koloru `#c54218`, a otrzymaliśmy `#cc4411`, czy całkiem bliski naszym oczekiwaniom.

'Uwaga:' Nie wszystkie kolory tak równie łatwo interpoluje się do wartości skróconych.

Dla wyjaśnienia ciemniejszy i najbliższy białemu kolor to `#eee`, który jest dość ciemny.

Połącz razem wiele skrótów i wartości definicji

Nieustanna optymalizacja naszych stylów, prowadzi czasami do trudnych w odnalezieniu błędów. Nadpisywanie właściwości skrótowych jest dobrym rozwiązanie, bo często zaoszczędzamy kilka znaków w naszych stylach.

Bardzo rzadko zdarza się, aby style wymagały czterech różnych kolorów obramowania, ale jeśli tak jest pozostaje tylko właściwe kodowanie.

```
div {
    border-top: 1px solid #f00;
    border-right: 1px solid #0f0;
    border-bottom: 1px solid #00f;
    border-left: 1px solid #aaa;
}
```

W tym wypadku mamy również szerokie możliwości optymalizacji. Przykład o tyle niefortunny, bo domyślne wartości obramowania to szerokości 1 piksela i linia ciągła.

```
div {
    border-color: #f00 #0f0 #00f #aaa;
}
```

Zatem wystarczyłoby samo zdefiniowanie samych kolorów. O ile mniej kodu. Szybko i czytelnie.

Jeśli nadpisujemy pojedyncze właściwości lub style, przeważnie lepsze jest zdefiniowanie tego samego raz i odpowiednie nadpisanie reguły.

Krótsze definicje właściwości

Arkusze stylów kaskadowych zostały pomyślane w ten sposób, aby programiści tworzyli kod jak najszybciej. Poszczególne właściwości zawierają szereg skrótów, które w jednej linii definiują wiele właściwości.

```
div {
    border: 1px solid #aaa;
}
```

Prosta definicja `border` umożliwia ustawienie zarówno wielkości obramowania, jego stylu jak i koloru. Tworzenie czterech reguł jest nieoptymalne z wielu powodów.

```
.global {
    border-top: 1px solid #aaa;
    border-right: 1px solid #aaa;
    border-bottom: 1px solid #aaa;
    border-left: 1px solid #aaa;
}
```

Po pierwsze waga pliku CSS jest większa. Dodatkowo analiza takich stylów bywa kłopotliwa, bo zmiana w jednym miejscu, bywa nadpisywana kolejną regułą w dalszej części.

Obrotowy link

HTML5 i CSS3 pojawiają się w aplikacjach i serwisach internetowych coraz częściej. Nie dziwi to ani trochę, skoro producenci przeglądarek wspierają coraz więcej standardów sieciowych w kolejnych wydaniach swoich produktów. Jak zwykle podczas poszukiwań inspiracji w sieci znalazłem ciekawy przykład wykorzystania nowych właściwości stylów kaskadowych.

Roll Link to przykład obrotowego linka, którego animacja opiera się na wykorzystaniu transformacji oraz atrybutów typu data. Serwis Webdesigner Depot prezentuje łącza w taki sposób w swoich artykułach.

Analiza rozwiązania

Struktura takiego rozwiązania jest nieskomplikowana choć na pierwszy rzut oka, element span wydaje się nadmiarowy.

```
<a href="some-url.html">
    <span data-title="Some text">Some text</span>
</a>
```

Nic bardziej mylnego. Wprawdzie do tego celu wykorzystać pseudo-elementy ::before i ::after, ale ich wsparcie jest dyskusyjne. Inna sprawa, że jak nie działa generated content to z dużą dozą prawdopodobieństwa nie działają także transformacje CSS.

Kształt całego rozwiązania stanowią odpowiednie style, podstawowe dla tagów a i span.

```
/* ROLL LINKS */
.roll-link {
    color: #DD4D42;
    font-weight: bold;
    text-decoration: none;
    word-wrap: break-word;

    display: inline-block;
    overflow: hidden;

    vertical-align: top;

    -webkit-perspective: 600px;
       -moz-perspective: 600px;
       -ms-perspective: 600px;
       perspective: 600px;

    -webkit-perspective-origin: 50% 50%;
       -moz-perspective-origin: 50% 50%;
       -ms-perspective-origin: 50% 50%;
       perspective-origin: 50% 50%;
}

.roll-link:hover {
    text-decoration:none;
}
```

Klasa .roll-link stanowi zasady dla naszych obrotowych linków. Poza podstawowymi i bardzo dobrze znanymi, jak kolor, grubość tekstu określono kilka

kilka innych. Sposób wyświetlania jako liniowo-blokowy oraz najważniejsze, perspective i perspective-origin.

Problem stanowi brak pełnego wsparcia dla tych właściwości CSS3, ale prefiksy działają. Poza tym obie właściwości wpierają tylko transformacje 3d, więc występują razem.

```
.roll-link span {
    display: block;
    position: relative;
    padding: 0 2px;

    -webkit-transition: all 400ms ease;
       -moz-transition: all 400ms ease;
       -ms-transition: all 400ms ease;
       transition: all 400ms ease;

    -webkit-transform-origin: 50% 0%;
       -moz-transform-origin: 50% 0%;
       -ms-transform-origin: 50% 0%;
       transform-origin: 50% 0%;

    -webkit-transform-style: preserve-3d;
       -moz-transform-style: preserve-3d;
       -ms-transform-style: preserve-3d;
       transform-style: preserve-3d;
}
.roll-link:hover span {
    background: #DD4D42;

    -webkit-transform: translate3d( 0px, 0px, -30px ) rotateX( 90deg );
       -moz-transform: translate3d( 0px, 0px, -30px ) rotateX( 90deg );
       -ms-transform: translate3d( 0px, 0px, -30px ) rotateX( 90deg );
       transform: translate3d( 0px, 0px, -30px ) rotateX( 90deg );
}
```

Element span jest dzieckiem znacznika a, a jednocześnie pojemnikiem dla generowanego automatycznie pseudo elementu :after. Przejście ease elementu span następuje w momencie pojawienia się kursora myszy nad linkiem. Wówczas po 400ms zmienia się także kolor tła elementu.

```
.roll-link span:after {
    content: attr(data-title);

    display: block;
    position: absolute;
```

```
    left: 0;
    top: 0;
    padding: 0 2px;

    color: #fff;
    background: #DD4D42;

    -webkit-transform-origin: 50% 0%;
       -moz-transform-origin: 50% 0%;
       -ms-transform-origin: 50% 0%;
       transform-origin: 50% 0%;

    -webkit-transform: translate3d( 0px, 105%, 0px ) rotateX( -90deg );
       -moz-transform: translate3d( 0px, 105%, 0px ) rotateX( -90deg );
       -ms-transform: translate3d( 0px, 105%, 0px ) rotateX( -90deg );
       transform: translate3d( 0px, 105%, 0px ) rotateX( -90deg );
}
```

Jeszcze większa magia dotyczy pseudo-elementu wstawianego po znaczniku span. Ten znowu pozycjonowany jest absolutnie względem rodzica, a jego zawartość wykorzystuje atrybut data-title.

Podsumowanie

Oto prosty sposób wykorzystania transformacji 3d w celu osiągnięcia interesującego efektu. Pewien problem stanowi jeszcze odpowiednie wsparcie przez nowoczesne, nie mówiąc o wszystkich przeglądarkach. Jeśli taki efekt nie stanowi kluczowego mechanizmu strony ani uniemożliwia korzystania ze strony w starszych będzie traktowany jako dodatkowy krok w kierunku przyszłości i ukłon w stronę użytkowników korzystających z nowoczesnych przeglądarek.

Skrótowe wartości właściwości

Używanie skróconych definicji niesie wiele korzyści. Jeśli w jednym momencie definiujemy zbiór wspólnych wartości zmniejszamy wielkość stylów. Dodatkowo dajemy sobie łatwą możliwość zmiany wyglądu w jednej linii, bez dopisywania kolejnych reguł.

```
p {
    margin: 10px;
}
```

Nasza strona jest bardzo prosta i prezentuje paragrafy z marginesem wielkości `10px`. Jedna linia ustawia od razu wszystkie wartości

Jednak nie zawsze spotykamy równie banalne przykłady. Choć CSS pozwala na jednoczesne ustawienie wartości skrajnych.

```
p {
    margin: 0 10px; /* padding: 0 10px 0 10px; */
}
```

Taki zapis odnosi się do zerowego marginesu w pionie i `10px` po bokach.

```
p {
    margin: 0 10px 25px; /* margin: 0 10px 25px 10px */
}
```

Powyższa reguła ustawia dolny margines paragrafu na `25px`, co jest krótsze niż zdefiniowanie lewego marginesu przez kolejną liczbę.

Istnieją przypadki, kiedy konieczne jest podanie wszystkich czterech wartości. Jednak nawet wtedy bedzie to znacznie krótszy zapis niż 4 osobne reguły.

```
p {
    margin: 10px 10px 25px 50px;
}
```

Analogicznie poza marginesem, ustawimy dopełnienie elementu, obramowanie.

VERIFY

JavaScript

Wprowadzenie

Najczęściej spotykanym zatosowaniem języka JavaScript są strony WWW. Skrypty służą najczęściej do zapewnienia interaktywności poprzez reagowanie na zdarzenia, sprawdzania poprawności formularzy lub budowania elementów nawigacyjnych. Podczas wzbogacania funkcjonalności strony internetowej istotne jest, aby żaden element serwisu nie stał się niedostępny po wyłączeniu obsługi JavaScriptu w przeglądarce. Skrypt JavaScriptu ma znacznie ograniczony dostęp do komputera użytkownika (o ile nie zostanie podpisany cyfrowo). Niektóre strony WWW zbudowane są z wykorzystaniem JavaScriptu po stronie serwera, jednakże znacznie częściej korzysta się w tym przypadku z innych języków.

Zmienne

Brak instrukcji `var` przed nazwą zmiennej tworzy w JavaScript zmienną o zasięgu globalnym. Nie zawsze tego chcemy.

```
var name     = "John";
var fullname = "Doe";
```

Jednak wielokrotne użyci instrukcji `var` wcale nie jest najlepszym sposobem na tworzenie zmiennych o mniejszym zasięgu. Możliwe jest zadeklarowanie wszystkich zmiennych, których potrzebujemy w danym momencie.

```
var name = "John", fullname = "Doe";
```

Co lepsze. Dostępna jest także inicjalizacja zmiennych.

TODO compose with hoisting...

Domyślne wartości zmiennych

Nie zawsze wiesz czy szukana zmienna istnieje lub została zainicjalizowana. Ponowne przypisanie wartości zatrze informacje o dotychczasowej zmiennej. Istnieje łatwe sprzwdzenie czy istnieje dana zmienna lub przypisanie jej wartości domyślnej.

```
var app = app || {};
```

Przykładowe użycie...

Średniki i nowe linie

JavaScript nie wymaga średników w pewnych miejscach naszego kodu. Jednak dla większej czytelności oraz pewności, że dostaniemy oczekiwany wynik stosujmy pewne zasady. Oczywiście poniższy kod jest w pełni poprawny składniowo.

```
function getValue() {
    var a = 10;
    return
        a;
}
```

Zastanawiasz się czy ten kod jest błędny albo dostaniesz to czego oczekujesz. Nasza funkcja zwróci `undefined`, a nie spodziewanej liczby `10`.

TODO przyklad dla srednikow... ## Referencje do obiektów DOM

Doskonale wiemy, że większość jeśli nie wszystkie operacje DOM są wolne. Zatem jeśli planujemy odwoływanie się do DOM więcej niż jeden raz, stwórzmy odpowiednią referecję.

```
document.getElementById('nav').className = 'test';
document.getElementById('nav').setAttribute('display', 'block');
```

Nasz dodatkowy obiekt pozwoli na bezpośdernie odwołanie do elementu, bez konieczności szukania go w drzewie DOM.

```
var obj = document.getElementById('nav');
    obj.className = 'test';
    obj.setAttribute('display', 'block');
```

Dodatkowa zmienna i określone instrukcje będą zawsze szybsze niż tuzin wywołań `getElementById`.

Pętla `for`

Zawsze inicializujemy zmienne, których użyjemy.

```
var text = "Hello, world!";
for (var i = 0; i < text.length; i++) {
    console.log(text.charAt(i));
}
```

Istnieją dwie możliwości na efektywniejszą pętlę.

```
var len = text.length;
for (var i = 0; i < len; i++) { }
```

Teraz nasza pętla nie pobiera długości zmiennej `text` przy każdej iteracji, a tylko porównuje wartość `i` z ustaloną długością ciągu, przechowywaną w osobnej zmiennej `len`.

Zróbmy to jeszcze lepiej. Istnieją przypadki, kiedy określone zmienne są potrzebne tylko weenątrz pętli, zatem zdefiniujemy je w zasięgu pętli.

```
var len = text.length;
for (var i = 0, len = text.length; i < len; i++) { }
```

Wszystko zależy od sytacji, ale nawet najprostsza instrukcje obniża wydajność kiedy powtarza się wielokrotnie. ## Combine control conditions and control variable changes when using loops

Whenever talking about performance, work avoidance in loops is a hot topic because, quite simply, loops run over and over again. So if there are any performance gains to be had, you will most likely see the largest boosts within your loops. One way to take advantage of this is to combine your control condition and control variable when you define your loop. Here's an example that doesn't combine these controls:

```
var time = new Date();
var idx = 0;

for ( var x = 0; x < 1000000; x++ ) { idx++; };

console.log(idx);
console.log((new Date() - time) + 'ms');
```

Before we add anything at all to this loop, there are a couple operations that will occur every iteration. The Javascript engine must #1 test if x exists, #2 test if x $<$ 0 and #3 add the increment x++. However if you're just iterating over some items in an array, you can cut out one of these operations by flipping this iteration around and using a while loop:

```
var x = 999999;
var time = new Date();

var idx = 0;
do { idx++; } while( x-- );

console.log(idx);
console.log((new Date() - time) + 'ms');
```

If you want to take loop performance to the next level, Zakas also provides a more advanced loop optimization technique, which runs through the loop asynchronously (so cool!).

Opóźnienie

Nie każdy wie, że pierwszy argument funkcji `setTimeout`, ale także `setInterval` nie musi być ciągiem znakowym. Taka sytuacja pozwala na wiele elastyczniejsze użycie funkcji.

```
setTimeout('loop()', 1000);
```

Jednak zamiast wywołania funkcji przekażemy jej ciało.

```
setTimeout(loop, 1000);
```

Wyobraź sobie konieczność przesłania argumentu do naszej metody, czasem nawet wielokrotnego przesłania. Takie zmienne tracą zasięg (poza zmiennymi globalnymi).

```
setTimeout('loop(counter)', 1000);
```

Istnieje dobry sposób rozwiązania tego problemu, czyli funkcja anonimowa.

```
setTimeout(function() { loop(counter); }, 1000);
```

Liczenie elementów DOM

Optymalizacja kodu lub poszukiwanie błędów wymaga od programistów sprawnego poruszania się w modelu obiektowym. Odnalezienie elementu od danym identyfikatorze czy klasie nie jest wcale trudne, ale tak prozaiczne czynności często nie wystarczają do rozwiązania problemu.

Ile elementów zawiera nasza strona?

```
var all = document.getElementsByTagName('*');
var script = document.getElementsByTagName('script');
var ids = 0, classes = 0, srcs = 0, async = 0;
for (var i = all.length; i--;) {
    if (all[i].id !== '') {
        ids++;
    }
```

```
    if (all[i].className !== '') {
        classes++;
    }
}
for (var i = script.length; i--;) {
    if (script[i].src !== '') {
        srcs++;
    }
    if (script[i].async) {
        async++;
    }
}
console.log('All elements:    ' + all.length);
console.log('  ids & classes: ' + ids + ', ' + classes);
console.log('All scripts:     ' + script.length);
console.log('  src & async:   ' + srcs + ', ' + async);
```

Liczenie elementów DOM według typu

Skomplikowanie stron internetowych bywa różne. Brak optymalizacji często widzimy gołym okiem. Szybkość ładowania strony zależy od wielu czynników. Złożona struktura dokumentu czy nadmiarowe znaczniki mają wpływ na generowanie kodu HTML. Jedyna kwestia to ocena naszej strony pod kątem wykorzystania semantycznych znaczników.

```
var all = document.getElementsByTagName('*');
var types = {};
for (var i = all.length; i--;) {
    if (all[i].tagName in types) {
        types[all[i].tagName]++;
    } else {
        types[all[i].tagName] = 1;
    }
}
console.log('*: ' + all.length);
```

Określenie liczby wszystkich elementów naszej strony to proste zadanie. Niewiele trudniej uzyskamy inforamcje o liczbie różnych znaczników, a od tegu już prosta droga do optymalizacji.

```
var sortable = [];
for (var tag in types) {
    sortable.push([tag, types[tag]]);
}
```

```
sortable.sort(function(a, b) {return a[1] - b[1]});

var i = sortable.length-1;
do {
    console.log(sortable[i][0] + ': ' + sortable[i][1]);
} while(i--);
```

Teraz dokładnie wiemy ile znaczników `<div>` lub `<span>` zawiera badana strona. Sami wyciągamy wnioski czy każdy z nich jest konieczny.

Wzorzec modułu

Brak klas w JavaScript stanowi spory problem dla początkujących programistów. Często

Tworzymy

```
var aya = aya || {};

aya.framework = function() {

    // private property
    var version = '0.0.2';

    // private method
    var getVersion = function() {
        return version;
    }

    // all returned is a public
    return {

        // initialization
        init: function() {
            console.log('init successful...');
        },

        printModuleVersion: function() {
            console.log('Version: ' + getVersion());
        }
    }
}();
```

Przygotowany w ten sposób obiekt przechowuje wewnątrz pewne zmienne jako prywatne. Natomiast kod zwracany jest traktowany jako publiczny. Ten prosty sposób pozwala na ukrycie części implementacji i dostęp spoza samego obiektu.

```
aya.framework.init();
```

Jak się spodziewamy w konsoli wyświetli się informacja o inicjalizacji obiektu.

```
aya.framework.printModuleVersion();
```

Podobnie będzie, kiedy sprawdzimy wersję naszego obiektu, ale możliwe będzie to jedynie poprzez publiczną metodę.

```
aya.framework.getVersion();
```

Próba uzyskania wersji przy użyciu metody prywatnej poza obiektem, kończy się błędem, czyli dokładnie jak powinno.

```
> TypeError: Object #<Object> has no method 'getVersion'
```

Rzutowanie zmiennych

Zachowaujemy ostrożność podczas rzutowanie zmiennych poprzez funkcję `parseInt`. Funkcja nie wymaga podstawy systemu liczbowego na jaki przetwarzamy pierwszy argument. Dla naszej wygody stara się o określenie podstawy systemu na podstawie przetwarzanego ciągu.

```
parseInt('010');
```

Pozostaje tylko pytanie, czym jest dla nas `'010'`, które rzutujemy na liczbę całkowitą. W systemie dziesiętnym odpowiedź jest bardzo prosta, ale dla mnie ta liczba wygląda jak system ósemkowy.

```
parseInt('010');
>>> 10
```

Jak się okazuje konsola Google Chrome przedstawia wynika jako `10`, jeśli nie określimy podstawy systemu liczbowego.

```
parseInt('010', 10);
>>> 10
parseInt('010', 8);
>>> 8
```

Spodziewane wyniki dostaniemy podczas rzutowania z określonymi podstawami systemu.

```
parseInt(010);
>>> 8
```

Jednak najciekawszy wynik mamy prze agumenci `010`, który w rzeczywiści jest wartością w systemie ósemkowym. Nasza sytuacja komplikuje się bardziej, gdyż `010` nie jest ciągiem znakowym, zatem także prawidłowym argumentem funkcji.

`javascript > typeof 010 "number"` ## Warunkowe logowanie

Dobrze wiemy, że JavaScript to bardzo elastyczny język. Nieraz potrzebujemy warunkowego wykonania instrukcji. Dla łatwiejszego zrozumienia przykładu tworzymy prosty obiekt

```
var user = {
    name: 'Luke',
    showName: function() {
        return 'Name: ' + this.name;
    }
};
```

Obiekt użytkownika zawiera nazwę i funkcję. Nasz cel to wyświetlenie `user.name`, jeśli istnieje obiekt `user`.

```
console.log(user && user.name);
```

W ten sposób wyświetlisz w konsoli właśność `name`, jeśli istnieje obiekt `user`. Możliwości tego triku nie kończą się na przekazaniu właśności obiektu. A może funkcja?

```
console.log(user && user.showName());
```

Zastowowanie tej sztuczki w naszych projektach nie sprawia wiele trudności.

Tak lub nie

???

`javascript if (Math.round(Math.random())) { // do if true } else { // do if false }` Bardzo prosty sposób na symulowanie działania naszego skryptu.