11. OOP - magia podejścia obiektowego

Wyzwania:

- poznasz popularne podejścia do programowania;
- odkryjesz techniki programowania obiektowego;
- poznasz tajemnice obiektowego JavaScriptu;
- dowiesz się, czym jest kanban i stworzysz swój własny przy użyciu technik obiektowych.

11.1. Najpopularniejsze paradygmaty programowania w pigułce

To o czym piszemy w tym submodule, może wydawać ci się kompletną abstrakcją. Nie przejmuj się tym. Z każdym kolejnym przykładem czy zadaniem filozofia programowania obiektowego będzie dla ciebie coraz bardziej zrozumiała :)

Nie jest to dla Ciebie wiedza konieczna, jednak warto wiedzieć, o co tutaj chodzi, by lepiej zrozumieć filozofię programowania i dzięki temu zrozumieć różnice między poszczególnymi podejściami. Przyda ci się to podczas nauki kolejnych języków :)

Paradygmat w programowaniu to pewien wzór, sposób myślenia, według którego należy programować komputery.

W parze z rozwojem komputerów szły nowe pomysły na jeszcze prostsze sposoby ich kontrolowania. Programiści wymyślali coraz to lepsze sposoby rozumowania progra komputerowego. Można powiedzieć, że paradygmat każe nam w pewien sposób patrzeć na program. Zmusza nas do myślenia pewnymi kategoriami.

Oto kilka najpopularniejszych przykładów paradygmatów:

Proceduralny

Polega on na tym, że program dzielimy na tzw. procedury. Procedura to zestaw komunikatów, które wykonują pewne działania, aby osiągnąć zadany cel.

W programowaniu proceduralnym komunikacja w programach polega na przekazywaniu do procedur danych. JavaScript ma w sobie coś z języka proceduralnego. Jak się pewnie domyślasz, są to funkcje.

Wejściem funkcji są jej parametry, natomiast wyjście jest wskazywane za pomocą instrukcji `return`.

Strukturalny

Programowanie strukturalne wywodzi się z proceduralnego. Do paradygmatu strukturalnego dodano użycie prostych struktur kontrolnych, które ułatwiają pisanie programu.

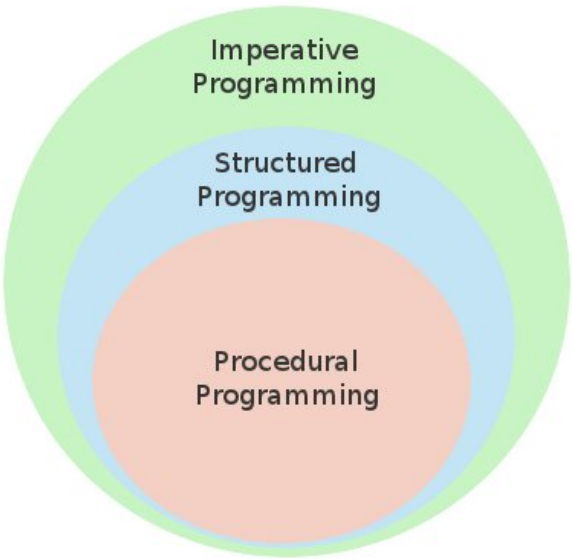
Przykładami struktur są:

- **Sekwencja** - czyli nic innego jak wykonanie pewnych instrukcji jedna po drugiej. Kod wykonywany jest linijka po linijce.
- **Wybór** - jest pewnym rozwidleniem w kodzie. Jeśli mamy pytanie, na które można odpowiedzieć wyłącznie prawda lub fałsz, to możemy użyć instrukcji wyboru. Najczęściej spotyka się konstrukcję if/else.
- **Pętla** - struktura dzięki której pewien fragment kodu może być wykonywany wiele razy z rzędu.

Imperatywny

Jest to najbardziej popularny i naturalny dla człowieka sposób pisania programu.

W tym paradygmacie nakazujemy, **w jaki sposób** coś ma być zrobione. Wcześniej wymienione paradygmaty proceduralne i strukturalne są podzbiorami paradygmatu imperatywnego. Tutaj opisujemy sposób, kolejne kroki, według których komputer ma dojść do celu.



Powyższy schemat idealnie pokazuje zależności, które występują pomiędzy paradygmatami imperatywnym, strukturalnym i proceduralnym.

Deklaratywny

Przeciwieństwem paradygmatu imperatywnego jest programowanie deklaratywne. Tutaj z kolei **opisujemy** sposób działania bez ścisłego sterowania komputerem.

Wiele języków stara się odejść od opisywania, w jaki sposób ma być coś zrobione i stara się opisać komputerowi problem. Całą resztą zajmuje się język programowania, w którym ukryta jest implementacja rozwiązująca opisywany problem.

W paradygmacie imperatywnym problem to algorytm, w którym należy opisać kolejne kroki działania. W programowaniu deklaratywnym opisujemy, co ma zostać wykonane.

Trochę jak praktyka kontra teoria albo inżynier praktyk kontra wykładowca. Na przykład w modelowaniu za pomocą równań matematycznych opisujemy pewien fragment rzeczywistości.

Przykładem takiego języka jest HTML. Zastanów się, w jakim trybie operujesz, pisząc strukturę HTML. Czy rozkazujesz, w jaki sposób ma się coś zrobić, czy raczej opisujesz, co ma się pojawić?

Funkcyjny

Struktura programu napisanego za pomocą paradygmatu funkcyjnego składa się z opisów matematycznych. W tym paradygmacie tworzymy coraz bardziej skomplikowane funkcje i za ich pomocą opisujemy, co komputer ma zrobić.

Początkujących programistów wprowadza się częściej w imperatywne języki, tak więc jeżeli chodzi o paradygmat funkcyjny, to na razie wszystko co musisz wiedzieć na jego temat.

Obiektowy

Najpopularniejszy spośród wszystkich paradygmatów jest zdecydowanie paradygmat obiektowy.

To podejście opisuje program jako zbiór obiektów, które komunikują się ze sobą. Każdy obiekt można przedstawić za pomocą pól (danych) oraz metod (pewnych funkcjonalności), które mogą na tych danych operować.

Miejsce JavaScriptu

JavaScript łączy w sobie wszystkie te podejścia - jest tzw. językiem wieloparadygmatowym (trudne słowo na dziś).

Można w nim zaobserwować wszystkie cechy poszczególnych paradygmatów. Jest **imperatywny**, ponieważ skrypty można pisać na zasadzie rozkazywania komputerowi (a w zasadzie przeglądarce), co ma wykonać. Można też użyć specjalnych konstrukcji, które są charakterystyczne dla programowania **funkcyjnego**, jednak najczęściej stosowanym jest paradygmat **obiektowy**. Między innymi również dlatego, że w JavaScripcie w praktyce wszystko jest obiektem.

Jeszcze nie raz się o tym przekonasz :)

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✔Zapoznałe(a)m się!

11.2. Dlaczego podejście obiektowe?

Podejście obiektowe zupełnie zmieniło sposób patrzenia programistów na oprogramowanie komputerowe. W tym rozdziale przekonasz się, dlaczego warto znać i korzystać z tego paradygmatu.

Prawdziwy świat jest obiektowy

Cały otaczający nas świat to obiekty. Programowanie obiektowe stara się nałożyć pewną abstrakcję, dzięki której możemy za pomocą języka programowania opisać obiekt, tzn. nadać mu pewne cechy czy napisać jakie czynności może wykonać.

Zależności między obiektami

Między obiektami mogą istnieć pewne zależności. Cechy jednego obiektu mogą się zmieniać, gdy na obiekt oddziałujemy innym obiektem.

Przykładem może być gra komputerowa, w której istnieją dwa obiekty **player** (gracz) i **enemy** (przeciwnik). Obiekt gracza może np. zaatakować obiekt przeciwnika, odbierając mu tym samym kilka punktów życia.

Można oczywiście napisać to za pomocą języka proceduralnego, ale język obiektowy wydaje się być w tym wypadku naturalniejszy.

Łatwiejsze czytanie kodu

Już we wcześniejszych rozdziałach, w których poznawaliście DOM, mogliście zauważyć, że posługujemy się tak naprawdę obiektami. Przykładowo:

```
document.querySelectorAll('p')
```

Czytamy jako: weź wszystkie paragrafy na stronie (obiekty)

Innym przykładem może być obiekt `console` wbudowany w przeglądarkę. Za pomocą metody tego obiektu możemy np. wylogować pewne informacje w konsoli, korzystając z poniższej metody:

```
console.log('hello');
```

Znowu - przeczytanie takiego fragmentu kodu nie powinno sprawiać nam żadnego kłopotu. W kontekście pewnego obiektu wykonaj pewną czynność. Tutaj można to przeczytać następująco: w kontekście konsoli wykonaj metodę log lub - prościej - konsolo, wykonaj metodę `log`.

W poprzedniej sekcji wprowadziliśmy obiekt gracza i wroga. Kod odpowiadający za atak gracza na przeciwnika mógłby wyglądać w następujący sposób:

```
player.attack(enemy);
```

Co przeczytalibyśmy jako "graczu zaatakuj wroga".

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✔Zapoznałe(a)m się!

11.3. Klasa a obiekt

W tym podrozdziale wyjaśnimy, na czym polega różnica między klasą, a obiektem. Początkującym programistom rozróżnienie tych dwóch pojęć sprawia wiele problemów, ale sprawa jest bardzo prosta.

Klasa

Klasę można w krótki sposób przedstawić jako szablon do tworzenia obiektów. Przy tworzeniu kolejnych obiektów danego rodzaju wykorzystuje się klasę, czyli szablon.

Bez klasy nie ma pojęcia obiekt i bez obiektu nie ma pojęcia klasa.

Możemy mieć obiekt samochodu, ale bez klasy (jego planu) nie moglibyśmy stworzyć tego samochodu.

Poniżej przykład definiowania klasy:

```
function Car(model, manufacturer) { //class properties are given in t
   //we assign values from function parameters to properties
   this.model = model;
   this.manufacturer = manufacturer;
}
Car.prototype.logModel = function(){
    console.log("This car's model is called " + this.model + ".");
};
```

W ten sposób tworzymy klasę Car z właściwościami `model` oraz `manufacturer`. Dodaliśmy także metodę `logModel` dla tej klasy, która poda nazwę modelu dla każdego obiektu tej klasy.

Obiekt

Obiekt jest tworem, który powstał na podstawie klasy. Jeśli będziemy nadal porównywać klasę do planu, to możemy powiedzieć, że inżynier wziął plan i skonstruował to, co na tym planie było opisane. Stworzył tzw. instancję danej klasy, czyli obiekt. Teraz inżynier może zabrać się za stworzenie kolejnej, zupełnie niezależnej instancji, korzystając z tego samego planu (klasy).

Instancja to pojedyncze wystąpienie obiektu stworzone na podstawie klasy, którą zdefiniowaliśmy wyżej.

```
var fordMustang = new Car("Mustang", "Ford");

fordMustang.logModel();
//console: "This car's model is called Mustang."
```

Podsumowując: jeśli obiekt jest ciastkiem, to klasa jest foremką, dzięki której uformowaliśmy to ciastko.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✔Zapoznałe(a)m się!

11.4. Programowanie obiektowe w JavaScripcie

Poznaliśmy już podstawowe koncepcje, stojące za paradygmatem obiektowym. Pora na odzwierciedlenie tych wszystkich pojęć w kodzie.

Prymitywy, literały i konstruktor

Prymitywy

W rozdziale na temat podstaw JavaScriptu poznaliśmy już kilka typów danych. Przykładowo: `string`, `number`, `boolean`, `null`, `undefined`.

W kwestii rozróżnienia, typy wartości:

string, *number* i *boolean*, a także *null* i *undefined* to tzw. prymitywy.
Prymitywy są reprezentowane przez samą wartość, nie mają żadnych metod ani właściwości.

Przykładowo:

1. "Hello World" // String
2. 123 // Number
3. false // Boolean

Literały

Przypisując wartość do zmiennej, korzystaliśmy z tzw. literałów.

Wyglądało to w następujący sposób:

```
var name = "Jan Nowak"; // "Jan Nowak" is a string literal
```

Teraz już wiesz, jak nazywała się metoda, za pomocą której tworzyliśmy wcześniej nasze zmienne :)

Czym różni się literał od prymitywu?

Prymityw określa wartość i sposób przechowywania danych w pamięci, natomiast literał to sposób deklarowania zmiennej zawierającej prymityw.

Tworząc zmienną za pomocą literału, tworzymy tak naprawdę opakowanie dla naszego prymitywu.

Konstruktory

Dotychczas tworzyliśmy zmienne za pomocą literału, ale jest jeszcze jedna metoda, bardziej obiektowa — konstruktor.

Czym jest konstruktor? Jest to specjalna metoda w klasie, wykonywana za każdym razem, kiedy tworzymy nową instancję klasy za pomocą instrukcji `new`. Możemy w niej na przykład nadać obiektowi właściwości przekazane z zewnątrz; innymi słowy: przygotować obiekt do pracy.

JavaScript zawiera również wbudowane konstruktory dla różnych typów danych, a także obiektów — tym zajmiemy się za chwilę. Mówiąc prościej, konstruktor klonuje nowe instancje klas i nadaje ich właściwościom wartości opisane w swoim parametrze.

Korzystając z poniższych konstrukcji, osiągniemy ten sam efekt:

```
var name = new String("Jan Nowak");
```

albo bardziej ogólnie

```
var name = new Object("Jan Nowak");
```

Pojawiło się nowe słowo kluczowe: `new`. Służy ono do tworzenia nowych instancji pewnej klasy. `var name = new String("Jan Nowak")` można odczytać jako: stwórz nowy obiekt klasy String

W obu tych przypadkach zostanie utworzony obiekt typu **string**. Razem mamy trzy sposoby na stworzenie tego samego. Rodzi się zatem pytanie...

Którą metodę stosować?

Dla prostych wartości prawie zawsze stosuje się literał. Porady dotyczące pisania dobrego kodu, które można znaleźć we wszystkich podręcznikach, odradzają tworzenie nowych obiektów, gdy można zastosować literał.

Dlatego tworzenie nowej wartości string za pomocą `new String` lub `new Object`, czyli metod obiektowych, nie jest poprawne — to samo dotyczy pozostałych literałów!

Czym różni się string od object?

W JavaScripcie prawie wszystko jest obiektem. Jak już zauważyliście, nawet pod tekstem kryje się pewne opakowanie (obiekt), które pozwala nam korzystać z takich dobrodziejstw jak `.toLowerCase`, `.length` i innych.

Literał i konstruktory dla obiektu oraz tablicy

Dla obiektu

Oprócz wymienionych literałów (string, number, boolean), można też tworzyć nowe obiekty za pomocą prostej konstrukcji `{}`. Jest to tzw. literał obiektu.

Obiekt, podobnie jak string, można utworzyć na dwa sposoby:

```
var person = {} // literal
```

```
var person = new Object() // constructor
```

Dla tablicy

Istnieje również prostszy zapis odnoszący się do tworzenia tablicy, także literał, który wygląda tak: `[]`. Znowu, podobnie do poprzednich przykładów, obiekt tablicy można stworzyć na dwa sposoby:

```
var people = [] // literal
```

```
var people = new Array() // constructor
```

Literał oczywiście jest znacznie lepszą opcją, o czym już wspominaliśmy!

Więcej o obiektach

Brawo! Przebrnęliśmy przez naprawdę ciężki temat. Teraz dowiemy się nieco więcej o obiektach.

Właściwości obiektu

Mówiliśmy już sobie o tym, że obiekty w JavaScripcie mają właściwości i metody.

Wspomnieliśmy także, że właściwość ma pewną wartość, która opisuje dany obiekt.

Dla obiektu typu string taka właściwość to np. `length`. Omawialiśmy to już w poprzednich rozdziałach, ale powtórzmy, w JavaScripcie dodajemy właściwości do obiektu w następujący sposób (przy użyciu literału):

```
var person = {
    name: "Jan Nowak",
    age: 30,
    isMarried: false
};
```

Należy pamiętać, że przy deklaracji obiektu każda kolejna właściwość musi być oddzielona przecinkiem, a nie średnikiem! Co ważne, ostatnia właściwość nie ma przecinka!

Metody obiektu

Wiemy już, że obiekt może posiadać nie tylko atrybuty, ale też metody, czyli czynności, które potrafi wykonywać. Inaczej, funkcje zawarte w obiekcie.

Przykładem takiej metody dla obiektu `person` może być:

```
var person = {
    name: "Jan Nowak",
    age: 30,
    isMarried: false,
    sayHello: function() {
        console.log("Hi, my name is " + this.name);
    }
};
```

Jak widzisz powyżej, funkcja `sayHello` jest naszą metodą, którą wywołujemy w następujący sposób:

```
person.sayHello(); // "Hi, my name is Jan Nowak"
```

Waszą uwagę na pewno przykuło słowo kluczowe `this`. To słowo reprezentuje kontekst obiektu, do którego się odnosimy. Tak więc, jeśli wywołujemy metodę `sayHello()` w kontekście obiektu `person`, to odwołując się do `this`, mamy na myśli właśnie obiekt `person`.

Na temat `this` dowiesz się więcej w późniejszym submodule :)

Obiekty tworzone za pomocą konstruktora

JavaScript jest językiem obiektowym, ale nie ma klas. To zdanie może być lekkim szokiem, szczególnie że cały czas posługujemy się terminem klasy i wiemy, że jest to swego rodzaju szablon do tworzenia nowych obiektów.

Klas co prawda nie ma, ale język zapewnia funkcje konstruujące, za pomocą których tworzymy nowe obiekty.

Poznaliśmy już wbudowane funkcje konstruujące `Object()` i `String()`, oprócz nich są też: `Number()`, `Date()`, `Boolean()` i wiele innych.

Prócz istniejących funkcji konstruujących możemy też tworzyć własne tego typu funkcje, które pełnią rolę klas, o czym dokładnie przeczytasz nieco niżej :)

Własne funkcje konstruujące

Poza literałami i wbudowanymi konstruktorami możemy tworzyć obiekty za pomocą własnych funkcji konstruujących — w naszym projekcie może być przecież więcej niż jeden obiekt osoby. Możemy chcieć utworzyć kilka obiektów tego samego rodzaju, więc przyda nam się "forma" do tworzenia większej ilości obiektów tego typu, czyli po prostu własna funkcja konstruująca na podstawie naszej formy.

Funkcja konstruująca (klasa):

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello = function() {
        console.log("Hi, my name is " + this.name);
    };
};
```

Przykład użycia "klasy" do stworzenia nowych obiektów:

```
var person1 = new Person("Jan Nowak", 35)
var person2 = new Person("Adam", 32)
var person3 = new Person("Ewa", 18)
```

Po stworzeniu nowych osób (Jan, Adam, Ewa) możemy wykorzystać metodę `sayHello` dla tych obiektów:

```
person3.sayHello() // will display: Hi, my name is Ewa
person1.sayHello() // will display: Hi, my name is Jan Nowak
```

W JavaScripcie nie ma klas. `Person` to po prostu funkcja konstruująca, która posiada cechy klasy, stąd potocznie nazywamy ją właśnie klasą.

Prototypy

Wcześniej poznaliśmy sposób, który pozwalał na dodawanie metod bezpośrednio do wzoru, z którego tworzyliśmy nowy obiekt. Problem w tym podejściu polega na tym, że przy dodawaniu w ten sposób metod do obiektów tworzymy mnóstwo kopii tej samej metody, tyle że w różnych obiektach.

Aby to sobie zobrazować, przyjmijmy, że mamy dwa obiekty — iPhone6 i iPhone5. Gdybyśmy do wzoru (klasy), z którego tworzymy te obiekty, dodali metodę, która wyświetla w konsoli model telefonu, wyglądałaby ona dokładnie tak samo.

```
this.logModel = function() {
    console.log(this.model);
}
```

Różnica pomiędzy tymi obiektami (telefonami) w odniesieniu do metody `logModel` leży tylko i wyłącznie w wartości `this.model` — poza tym, te metody są identyczne dla każdego obiektu wytworzonego z tego samego wzoru (klasy).

Zgodnie z poznaną przez nas wcześniej zasadą DRY (Don't Repeat Yourself), powinniśmy zapobiegać bezsensownemu powtarzaniu kodu. W tym momencie z pomocą przychodzą nam prototypy.

Prototypy to nic innego, jak metody, które przypisujemy do klasy poza nią samą, aby nie replikować tego samego kodu przy tworzeniu nowego obiektu na podstawie klasy. Są one dostępne dla każdego obiektu tworzonego z danej klasy, ale nie są one powtarzane wewnątrz poszczególnych instancji.

Oznacza to tyle, że prototypu możemy użyć na każdym obiekcie utworzonym z klasy, do której jest przypisany dany prototyp.

Przykładowy prototyp wygląda tak:

```
Smartphone.prototype.logModel = function(){
    console.log("This phone's model is called " + this.model + ".");
};
```

Rozłóżmy ten przykład na części pierwsze. Smartphone ilustruje komputerowi, że ten prototyp ma się odnosić do instancji klasy Smartphone. Prototype to po prostu część składni symbolizująca przeglądarkę, że ta funkcja ma być prototypem. logModel to nic innego jak nazwa metody, którą tworzymy :)

Gdy deklarujemy prototyp, to z metody korzystamy zwyczajnie, tak jak z metod deklarowanych w środku klasy.

Przykład użycia metody pochodzącej z prototypu:

```
iPhone6.logModel(); //the result is "This phone's model is called iPh
```

Przykład użycia metody oraz prototypu

Poniżej znajduje się przykład z użyciem metody wewnątrz klasy oraz prototypu.

Przykład metody zadeklarowanej wewnątrz klasy:

```
function Smartphone(brand, model) {
    this.brand = brand;
    this.model = model;
    this.logModel = function() {
        console.log(this.model);
    }
}

iPhone6.logModel(); //call method for iPhone6 object
```

Przykład metody zadeklarowanej przy użyciu prototypu:

```
function Smartphone(brand, model) {
    this.brand = brand;
    this.model = model;
}
Smartphone.prototype.logModel = function() {
    console.log(this.model);
}

iPhone6.logModel();
```

Jak widzisz, metodę zadeklarowaną poprzez prototyp wywołujemy dokładnie w ten sam sposób — jedyna różnica to mniej powtarzania metod w pamięci RAM (a co za tym idzie, "lżejsza" strona), oraz inne miejsce deklarowania metody.

Podsumowanie

- Nie ma klas, za to są funkcje konstruujące, dzięki którym możemy tworzyć nowe obiekty.
- `this` wskazuje na kontekst, w którym operujemy.
- Funkcje w JavaScripcie są również obiektami.
- Każda funkcja zawiera w sobie pole `prototype`, do którego przypisujemy części wspólne, które powtarzają się pomiędzy instancjami.

Zanim przejdziemy do zadań, przedstawimy ostatni przykład będący podsumowaniem tego podrozdziału:

```
function Car(color, brand) { //construction functions are capitalized
    this.color = color; //creating a new object, we can specify its i
    this.brand = brand || 'ford'; //if no brand is given at this poir
}
```

Zadanie: Pierwsza klasa

Pobawimy się trochę klasami i obiektami. Zrobimy to na podstawie telefonów, czyli obiektów, bez których wiele osób nie wyobraża sobie życia :)

Zadanie, które wykonamy, można sobie wyobrazić jak tworzenie fabryki. Zrobimy odpowiednią klasę, która będzie naszą formą do tworzenia nowych modeli.

Potem skorzystamy z prototypu i w ten sposób dodamy do naszych telefonów funkcje.

A na końcu stworzymy kilka egzemplarzy :)

Całość wykonaj lokalnie i wyślij na GitHuba, pamiętaj o dostępie do projektu dla Twojego mentora.

Konstrukcja

Zaczniemy od stworzenia funkcji konstruującej nasze telefony (klasy) - jako argumenty podaj markę, cenę oraz kolor. Aby stworzyć taką funkcję, deklarujemy:

```
function Phone(brand, price, color) {
    //Here we put interior of the constructing function.
}
```

Twoja "klasa" (czyli nasza forma do tworzenia nowych telefonów) powinna posiadać następujące parametry: `brand`, `price`, `color`. Aby dodać te parametry, musimy skorzystać z `this` w następujący sposób:

```
function Phone(brand, price, color) {
    this.brand = brand;
    //by using this, the "brand" property of the object we create wil
    this.price = price;
    this.color = color;
}
```

Funkcje telefonu

Teraz pora na utworzenie metody - korzystając z prototypu, utwórz dla klasy Phone metodę o nazwie *printInfo* opisującą telefon za pomocą `console.log()`.

Zdanie skonstruowane przez tę metodę powinno wyglądać mniej więcej tak - "The phone brand is (marka), color is (kolor) and the price is (cena).".

Aby to zrobić, tuż pod swoją funkcją konstruującą zadeklaruj następującą funkcję prototypową:

```
Phone.prototype.printInfo = function() {
        console.log("The phone brand is " + this.brand + ", color
}
```

Produkcja telefonów

Klasa gotowa - efekt naszej dotychczasowej pracy powinien wyglądać następująco:

```
function Phone(brand, price, color) {
    this.brand = brand;
    this.price = price;
    this.color = color;
}
Phone.prototype.printInfo = function() {
    console.log("The phone brand is " + this.brand + ", color is " +
}
```

Teraz na jej podstawie utwórz obiekty, które będą przedstawiać następujące modele telefonów:

- Samsung Galaxy S6
- iPhone 6s
- OnePlus One.

Przykładowo utworzenie obiektu dla iPhone 6s na podstawie naszej funkcji konstruującej będzie wyglądało następująco:

```
var iPhone6S = new Phone("Apple", 2250, "silver");
```

Teraz przedstawimy w konsoli informacje na temat każdego utworzonego przed chwilą obiektu za pomocą metody printInfo.

Dla naszego nowo utworzonego obiektu *iPhone6s*, wywołanie metody przedstawiającej telefon dla naszego nowo utworzonego obiektu iPhone6s będzie wyglądało następująco:

```
iPhone6S.printInfo();
```

Gotowe! :)

Rozbuduj swoje telefony o dowolne właściwości. W ramach ćwiczenia możesz stworzyć im też dodatkowe funkcjonalności za pomocą prototypów.

Podgląd zadania

https://github.com/0na/

Wyślij link ✔

11.5. This, self... pułapki Javascriptu

Słowo kluczowe *this* poznaliśmy przy tworzeniu pierwszej klasy. *This* wskazywało na kontekst, w którym operowała nasza funkcja. Przyjrzyjmy się nieco bardziej temu słowu kluczowemu.

This - tutaj czai się kontekst

Pojęcie kontekstu w JavaScripcie sprawia początkującym spore problemy. Na podstawie kilku przykładów postaramy się więc wyjaśnić, o co chodzi.

Czym jest kontekst?

Zanim to zrobimy, rozwińmy słowo **kontekst**. W JavaScripcie kontekst jest trzymany przez właściciela, do którego przynależy funkcja. Początkowo może się to wydawać nieco skomplikowane, szczególnie że są przecież funkcje, które zdawać by się mogło nie mają żadnego właściciela!

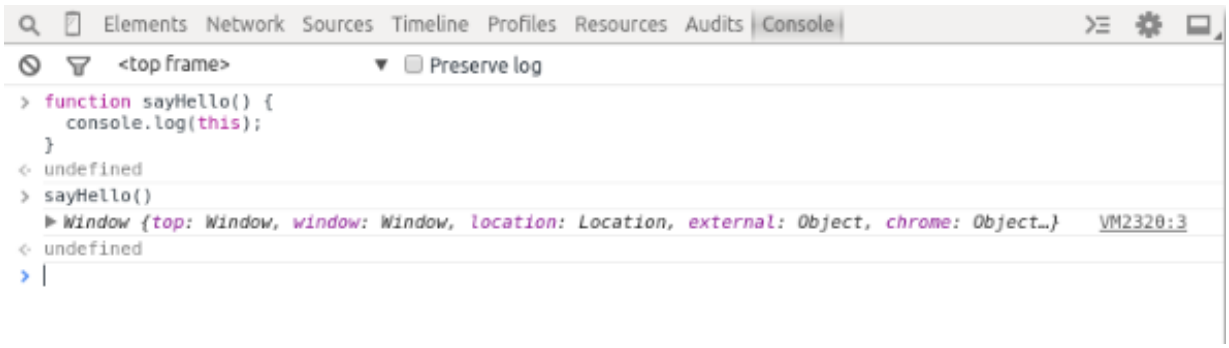
Na przykład:

```
function sayHello() {
    console.log(this);
}
```

Funkcja ta nie ma właściciela w postaci widocznego obiektu. Widoczne tutaj słowo jest kluczem do całej zagadki! Jeśli wywołamy tę funkcję:

```
sayHello();
```

w konsoli zobaczymy taki obraz:



Aha! Czyli funkcja ma jednak jakiegoś właściciela. Okazuje się, że kontekstem trzymanym przez funkcję `sayHello` jest obiekt *Window*.

Tip: *Obiekt Window jest obiektem globalnym i reprezentuje okno przeglądarki. Obiekt ten trzyma informację między innymi na temat drzewa DOM i właściwości okna, ale również wszystkich wbudowanych w przeglądarkę funkcji konstruujących (klas takich jak Number, String itp.).*

Nadawanie kontekstu

Jeśli chcemy, aby funkcja wykonywała się w konkretnym kontekście, musimy nadać jej właściciela, np. stworzyć ją jako metodę obiektu:

```
var person = {
    name: "Jan",
    sayHello: function() {
        console.log("Hello "  + this.name + "!");
    }
}
```

Co by się stało, gdybyśmy pozbyli się słowa *this*? Funkcja próbowałaby odwołać się do obiektu globalnego, czyli… *Window*! Czy w `Window.name` trzymamy jakąś wartość? Nie, dlatego na ekranie pokaże nam się: "Witaj !" bez imienia.

Innym przykładem nadania kontekstu jest tworzenie nowych obiektów za pomocą funkcji konstruującej (klasy).

Zmodyfikujmy nieco poprzedni przykład:

```
function Person(name) {
    this.name = name;
    this.sayHello = function() {
        console.log("Hello " + this.name + "!");
    };
}
var p1 = new Person("Jan");
var p2 = new Person("Zbigniew");
```

Metoda `sayHello` jest wywoływana dla każdego z tych obiektów w ich własnym kontekście.

```
p1.sayHello() // Hello Jan!
p2.sayHello() // Hello Zbigniew!
```

Niestety w JavaScripcie istnieje pewien problem polegający na utracie powiązania (ang. *binding*) ze swoim właścicielem. Dowiesz się o nim w kolejnym submodule.

Self - kiedy kontekst jest nieco dalej

Problem gubienia kontekstu spędza sen z powiek niejednemu programiście JavaScript. Oczywiście mając świadomość utraty powiązania, jesteśmy w stanie poradzić sobie z tym problemem całkiem sprawnie.

Spróbujmy pokazać, o co chodzi i zasymulować utratę kontekstu:

```
var person = {
    name: "Jan",
    sayHello: function() {
        console.log("Hello "  + this.name + "!");
    }
}
var hello = person.sayHello;
```

Stworzyliśmy sobie nową zmienną, która jest skrótem (tzw. *alias*) do metody `person.sayHello`.

Wywołanie `hello` odwoła się więc do implementacji, która znajduje się w `person.sayHello`. Jest tutaj jednak pewien problem. Gubimy kontekst! Kiedy wywołamy funkcję `hello()`, this wskaże na obiekt *Window*, który nie ma właściwości *name*.

Możesz wkleić do konsoli powyższy kod i wywołać w niej `hello()`, żeby przekonać się o efekcie.

Jak zawczasu zidentyfikować utratę kontekstu?

Utrata kontekstu pojawia się najczęściej w momencie przekazywania funkcji jako parametru innej funkcji.

Na przykład:

```
var person = {
    name: 'Jan',
    sayHello: function() {
        setTimeout(function() {
            console.log('Hello '  + this.name + '!');
        }, 1000);
    }
};
person.sayHello(); // Hello !
```

Teraz przekazaliśmy do funkcji `setTimeout` inną funkcję jako parametr. Pozbyliśmy się tym samym kontekstu. `this` wskazuje teraz na obiekt domyślny jakim jest *Window*.

Jedna z metod zapobiegania temu niepożądanemu zjawisku wygląda następująco:

```
var self = this;
```

W kodzie poniżej znajdziesz jej zastosowanie:

```
var person = {
    name: 'Jan',
    sayHello: function() {
        var self = this;
        setTimeout(function() {
            console.log('Hello '  + self.name + '!');
        }, 1000)
    }
};
person.sayHello() // Hello Jan!
```

Co się stało? W funkcji `sayHello` tworzymy zmienną, która będzie trzymać prawidłowy kontekst, na który wskazuje `this` (obiekt `person`). W efekcie za każdym razem, kiedy odwołujemy się do zmiennej `self`, mamy dostęp do właściwego kontekstu i wszystko działa tak, jak powinno.

Zadanie: Użycie "self"

Naszym zadaniem będzie skorzystanie ze zmiennej `self`. Zasymulujemy sobie tę sytuację, tworząc klasę `Button`. Klasa będzie miała za zadanie tworzyć nowe przyciski na naszej stronie.

Zadanie wykonaj lokalnie i udostępnij mentorowi za pomocą GitHuba.

Przejdźmy od razu do implementacji:

Krok 1

Napiszmy teraz funkcję konstruującą `Button`, która będzie tworzyła nowe przyciski - robimy to w następujący sposób:

```
function Button() {

}
```

Krok 2

Kolejnym krokiem będzie uzupełnienie konstruktora o parametr `text`, za pomocą którego ustawimy początkową wartość dla `text` w nowych obiektach.

```
function Button(text) {
    this.text = text;
}
```

Krok 3

Dodajmy możliwość tworzenia obiektu bez podania argumentu, w takim przypadku domyślną wartością właściwości `text` naszych przycisków będzie `'Hello'`.

```
function Button(text) {
    this.text = text || 'Hello';
}
```

Krok 4

Nasz przycisk można już co prawda stworzyć, ale nie mamy jeszcze metod obsługujących tworzenie przycisku na stronie - dodajmy je jako **prototyp**.

Jeśli nie pamiętacie, dlaczego warto korzystać z prototypów, to odsyłam do submodułu o własnych funkcjach konstruujących.

Stwórzmy sobie podstawową metodę `create` tworzącą przycisk na stronie.

```
Button.prototype = {
    create: function() {
        ...
    }
}
```

Ta metoda ma za zadanie tworzyć nowe przyciski na stronie

```
Button.prototype = {
    create: function() {
        this.element = document.createElement('button');
    }
}
```

W polu `this.element` będziemy trzymać nowo utworzony element przy wykorzystaniu API DOM.

Następnie ustawmy tekst na przycisku za pomocą pola `innerText`:

```
Button.prototype = {
    create: function() {
        this.element = document.createElement('button');
        this.element.innerText = this.text;
    }
}
```

Kolejnym krokiem będzie napisanie metody, która po kliknięciu w przycisk wyświetli tekst na ekranie komputera. W środku funkcji `create` należy więc dopisać:

```
this.element.addEventListener('click', function() {
    alert(self.text);
});
```

Widzicie użycie `self`? Mówiliśmy sobie, że w momencie, kiedy funkcja jest parametrem innej funkcji, nastąpi utrata kontekstu, dlatego musimy wskazać ten kontekst za pomocą zmiennej `self`. Ustawmy zmienną `self` na samej górze metody `create`. Metoda `create` powinna obecnie wyglądać następująco:

```
create: function() {
    var self = this;
    this.element = document.createElement('button');
    this.element.innerText = this.text;
    this.element.addEventListener('click', function() {
        alert(self.text);
    });
}
```

Metodzie `create` brakuje jeszcze jednej funkcjonalności - do tej pory stworzyliśmy przycisk, uzupełniliśmy go tekstem, przypięliśmy nasłuchiwanie na kliknięcie. Na koniec należy umieścić element w drzewie DOM!

Ostatnią linijką, którą należy dopisać w metodzie `create` jest:

```
document.body.appendChild(this.element);
```

Krok 5

Ok, klasa gotowa. Przejdźmy do tworzenia pierwszej instancji:

```
var btn1 = new Button('Hello!');
```

Krok 6

Następnie wywołajmy metodę `create` w celu stworzenia elementu:

```
btn1.create();
```

Element będzie miał w sobie taki tekst, jaki mu przypisaliśmy oraz przypięte nasłuchiwanie zdarzenia `click`.

Krok 7

Po stworzeniu instancji oraz użyciu metody `create` na stronie powinniśmy zobaczyć przycisk z treścią `'Hello!'`. Ok, to teraz kliknij w przycisk i zobacz, co się stanie. Czy na ekranie pojawił się alert z nazwą przycisku? Jeśli tak, to gratulacje. Wiesz już, jak poradzić sobie z zachowaniem kontekstu :)

Podgląd zadania

https://github.com/0na/

Wyślij link ✔

11.6. Własne Trello

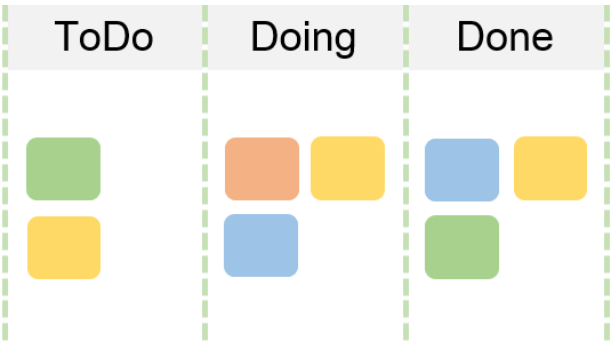
Czym jest Kanban?

Kanban w bardzo ogólnym znaczeniu jest jedną z wielu technik zarządzania projektami. Pierwotnie została ona stworzona na potrzeby zarządzania produkcją w latach 50. w Japonii. Została jednak zaadaptowana dla potrzeb wytwarzania oprogramowania.

Tablica kanban

Metodę tę w projektach programistycznych najlepiej obrazuje tzw. tablica kanban przedstawiająca proces wytwarzania za pomocą kart, które przyczepia się w odpowiednie miejsca na tablicy.

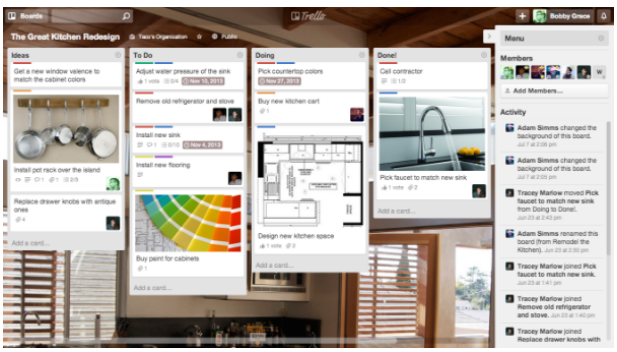
Karty te reprezentują zadania, które należy wykonać, natomiast sama tablica jest podzielona na kilka kolumn. Liczba kolumn, w zależności od zespołu i projektu, może być różna. Jednak najbardziej klasycznym przykładem podziału tablicy jest ten, który przedstawiliśmy na grafice poniżej:



W swojej pracy na pewno spotkasz się z podobną tablicą. Jak się pewnie domyślasz, cały “myk" polega na tym, aby wykonać zadania, które znajdują się po lewej stronie tablicy (*ToDo*), przenosząc je tym samym na prawą część (*Done*). Oczywiście nie da się tego zrobić od razu, dlatego w środku jest kolumna (*Doing*), która pokazuje nad czym obecnie pracujemy.

Kolory karteczek mogą mieć różne znaczenie. Mogą one symbolizować priorytet zadania albo to, czy zadanie jest dla *front-endu* czy *back-endu*.

Jednym z przykładowych narzędzi do zarządzania projektem, które wykorzystuje tablicę kanban, jest Trello



Przygotowanie do implementacji tablicy

W poprzednich podrozdziałach omówiliśmy podstawowe zagadnienia dotyczące obiektowości. W tym momencie nad Twoją głową powinna pojawić się lampka! Tak, tablicę kanban można przedstawić za pomocą obiektów!

Spróbujmy zatem rozpoznać obiekty, które tworzą tablicę kanban i zastanowić się, jak je opisać:

Tablica

Obiekt najwyższy w hierarchii. Tablica może posiadać takie atrybuty jak nazwa tablicy oraz element DOM.

Przykładowo:

```
var table = {
  name: 'project',
  element: <Node element>
};
```

Do metod można zaliczyć dodanie nowej kolumny.

Kolumna

Jest częścią tablicy i to w niej znajdują się karteczki. Kolumna pełni rolę informacyjną na temat statusu danego zadania (z karty). Parametrami, które opisują kolumnę, mogą być nazwa kolumny oraz - podobnie jak w przypadku tablicy - element DOM.

```
var column = {
  id: 12j82da20k,
  name: 'todo',
  element: <Node element> // for example document.createElement('div'
};
```

Metody kolumny to usunięcie kolumny oraz stworzenie nowej karteczki w danej kolumnie.

Karteczka

Stanowi najprostszy element tablicy. Do właściwości, które opisują karteczkę, mogą należeć: opis zadania, kolor oraz element DOM karteczki.

```
var card = {
  id: '2kd8s958ka',
  description: 'Create Kanban app',
  color: 'green',
  element: <Node element>
};
```

Czynnościami (metodami), które karteczka może wykonać, są usunięcie oraz przeniesienie jej do innej kolumny.

Jak widzicie, tablica składa się z kolumn, a kolumny składają się z karteczek.

Powyżej przedstawiono tylko obiekty, które zawierają w sobie parametry. Nie pokazano jeszcze metod, które przykładowy obiekt mógłby zawierać.

Dodatkowe założenia

Zanim zaczniemy pisanie jakichkolwiek klas, zajmiemy się uproszczeniem założeń naszego projektu.

Po pierwsze: tablica będzie jedna i będzie miała stałe wartości niektórych parametrów. Nie musimy zatem tworzyć klasy (szablonu), która będzie tworzyła nowe obiekty. Literał obiektu (pojedynczy obiekt) w zupełności wystarczy.

Po drugie: jak zauważyliście, do każdego obiektu dodaliśmy parametr-element wskazujący na obiekt drzewa DOM. Jest to kolejne uproszczenie, ale mało eleganckie.

Po trzecie: użyjemy gotowych bibliotek, które ułatwiają nam pracę. Mustache pomoże nam w zarządzaniu szablonami, a wtyczka Sortable dodaje możliwość przenoszenia kartek z jednej listy na drugą.

Implementacja tablicy

Dosyć gadania - bierzemy się do pracy!

Podstawowy HTML

Do rozpoczęcia pracy potrzebujesz podstawowej struktury HTML oraz bibliotek, o których wspominaliśmy wyżej. Jako że skupiamy się na JS i potrafisz zarówno stworzyć HTML, jak i podpiąć biblioteki, mamy dla ciebie gotowy kod na start.

Poniższy kod wklej do pliku *index.html*:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Kanban board</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
    <div id="board" class="board">
    <h1>Kanban board</h1>
    <button class="create-column">Add a column</button>
    <div class="column-container"></div>
    </div>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/mustache.js/2.3
  <script src="https://cdnjs.cloudflare.com/ajax/libs/Sortable/1.6.0/
  <script src="script.js"></script>
</body>
</html>
```

Co my tu mamy?

W elemencie <head> umieściliśmy podlinkowania do stylów natomiast skrypty, które będą potrzebne w trakcie tworzenia aplikacji, podpięliśmy pod koniec dokumentu. Dlaczego na koniec? Przeglądarka tworzy drzewo DOM, ładując HTML z góry do dołu. Chcemy jak najszybciej zaprezentować coś użytkownikowi naszej strony, więc przeniesienie ładowania skryptów na koniec jest rozsądnym rozwiązaniem.

W elemencie <body> znajduje się prosta struktura zawierająca jedną tablicę bez kolumn i kartek!

Przechodzimy do JS

Jak można zauważyć, w HTML nie mamy żadnych kolumn ani kartek. Będziemy tworzyć i dodawać je dynamicznie po załadowaniu drzewa DOM i na kliknięcie w odpowiednie przyciski akcji, np. *Add a column* czy też *Add a card*.

Otwórz plik *script.js* i dodaj odpowiednią instrukcję, która sprawi, że kod naszej aplikacji zacznie się wykonywać dopiero po załadowaniu całego drzewa DOM. Całą resztę naszego kodu JS będziemy dodawać wewnątrz.

```
document.addEventListener('DOMContentLoaded', function() {
    // here we will put the code of our application
});
```

Generowanie ID

Każda kolumna oraz karteczka musi być unikalnym obiektem, żeby zapobiec tworzeniu duplikatów. Systemy mają spory problem z duplikatami i w przypadku ich istnienia pojawia się masa komplikacji w stylu robimy coś z elementem A i jednocześnie z elementem A` (duplikatem), chociaż akcja miała dotyczyć tylko tego pierwszego.

Funkcja `randomString()` generuje id, które składa się z ciągu 10 losowo wybranych znaków.

Wygląda następująco:

```
function randomString() {
    var chars = '0123456789abcdefghiklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
    var str = '';
    for (var i = 0; i < 10; i++) {
        str += chars[Math.floor(Math.random() * chars.length)];
    }
    return str;
}
```

Podsumowując: funkcja losuje 10 elementów z tablicy znaków *chars* i składa je w jeden string. O prawdopodobieństwo wystąpienia duplikatu w identyfikatorze nie musimy się martwić :) Kombinacji jest zbyt wiele.

Generowanie templatek

W tym zadaniu skorzystamy z wcześniej poznanego systemu szablonów - Mustache.js Pomoże nam to w separacji kodu javascript od HTML. Naszym zadaniem będzie stworzenie funkcji, która będzie pobierała templatkę HTML z pliku index.html, parsowała, renderowała z użyciem naszej biblioteki szablonów, a następnie zwracała gotowy rezultat

Wygląda następująco:

```
function generateTemplate(name, data, basicElement) {
  var template = document.getElementById(name).innerHTML;
  var element = document.createElement(basicElement || 'div');

  Mustache.parse(template);
  element.innerHTML = Mustache.render(template, data);

  return element;
}
```

Parametry funkcji `generateTemplate(name, data, basicElement)` pozawalają na podanie nazwy templatki znajdującej się w pliku index.html, danych, które mogą zostać podstawione do szablonu Mustache oraz ostatni opcjonalny parametr, dający nam możliwość zdecydowania w jaki element zostanie opakowany nasz szablon. Opakowanie jest nam potrzebne, ponieważ Mustache.js zwraca w funkcji render string'a z zawartością naszego szablonu, a do kolejnych operacji w naszym projekcie potrzebujemy mieć dostęp do DOM API. Z tego też powodu, wytwarzamy element drzewa DOM - `document.createElement(basicElement || 'div');` jednocześnie dając możliwość - albo elementem tym będzie opcja `basicElement` podana jako argument, albo użyjemy domyślnej wartości `basicElement || 'div'` - div.

Podstawowe klasy i obiekt tablicy

Zacznijmy od stworzenia klas *Column* i *Card* oraz obiektu *board* (nie ma potrzeby tworzyć klasy dla board, ponieważ tablica będzie tylko jedna).

Przypominamy, że nazwy klas przyjęło się pisać wielką literą, natomiast obiektów małą.

Klasa *Column*

Na początku powinna ona wyglądać następująco:

```
function Column(name) {
  var self = this;

  this.id = randomString();
  this.name = name;
  this.element = generateTemplate('column-template', { name: this.nam
}
```

W przykładzie widzisz kilka standardowych elementów, jak `name`, czy `element`. Natomiast `id` generujemy za pomocą wcześniej utworzonej funkcji `randomString(`
Przy tworzeniu nowej instancji klasy `Column` tworzony jest też element *Node*. Dzieje si

to po wywołaniu funkcji `generateTemplate()`.

Szablon kolumy

W pliku index.html umieścimy szablon kolumny, o `id` identycznym, jak to, które przekazaliśmy do funkcji `generateTemplate()`.

Ten segment docelowo będzie wyglądał tak:

```
<script id="column-template" type="x-tmpl-mustache">
    <div class="column">
        <h2 class="column-title">{{ name }}</h2>
        <ul id={{ id }} class="column-card-list"></ul>
        <button class="btn-delete">X</button>
        <button class="add-card">Add a card</button>
    </div>
</script>
```

Ok, mamy stworzone wszystkie potrzebne klocki, z których powinna być zbudowana kolumna. Co dalej?

Podpinanie zdarzeń

Zależy nam też na tym, aby po kliknięciu w odpowiednie elementy można było usunąć kolumnę lub dodać nową kartę do kolumny. Więc zanim połączymy odpowiednio elementy (klocki, z których jest zbudowana kolumna), dodajemy nasłuchiwania zdarzeń.

Kod który powstanie w wyniku tej części trafia do konstruktora naszej klasy `Column`.

Kasowanie kolumny po kliknięciu w przycisk:

```
this.element.querySelector('.column').addEventListener('click', funct
  if (event.target.classList.contains('btn-delete')) {
    self.removeColumn();
  }

  if (event.target.classList.contains('add-card')) {
    self.addCard(new Card(prompt("Enter the name of the card")));
  }
});
```

Zwróć uwagę na odniesienie do `self`! Musieliśmy zastosować tę konstrukcję, gdyż mamy do czynienia z funkcją, która jest przekazana jako parametr metody `addEventListener()`. Te metody (`removeColumn()` oraz `addCard()`) jeszcze nie istnieją, ale dodamy je w późniejszym etapie do prototypu `Column`.

Inną rzeczą, która rzuca się w oczy, jest obecność funkcji `prompt()` - za jej pomocą pobieramy po prostu od użytkownika nazwę kolumny, którą chce stworzyć.

Spójrzmy jeszcze raz, jak wygląda końcowa implementacja klasy `Column`:

```
function Column(name) {
  var self = this;

  this.id = randomString();
  this.name = name;
  this.element = generateTemplate('column-template', { name: this.nam

  this.element.querySelector('.column').addEventListener('click', fun
    if (event.target.classList.contains('btn-delete')) {
      self.removeColumn();
    }

    if (event.target.classList.contains('add-card')) {
      self.addCard(new Card(prompt("Enter the name of the card")));
    }
  });
}
```

Metody dla klasy *Column*

Ostatnim krokiem uzupełniającym implementację klasy `Column` jest dodanie dwóch metod do jego prototypu: usuwanie kolumny i dodanie karty.

Kod umieść pod klasą `Column`.

Będzie to wyglądało następująco:

```
Column.prototype = {
    addCard: function(card) {
      this.element.querySelector('ul').appendChild(card.element);
    },
    removeColumn: function() {
      this.element.parentNode.removeChild(this.element);
    }
};
```

Metoda addCard()

Przyjmuje jako parametr kartę, którą chcemy dodać do kolumny. Dodajemy ją, wybierając element kolumny (stąd `this.element`). Przypominam, że struktura kolumny, którą sobie utworzyliśmy, będzie wyglądała tak:

```
<div class="column">
  <h2 class="column-title">Column name</h2>
  <ul id="random" class="column-card-list"></ul>
    <button class="btn-delete">x</button>


    <button class="add-card">Add a card</button>
</div>
```

`this.element` wskazuje na `div.column`

My chcemy dodawać kolejne karty do `div.column ul`, więc pobieramy element `ul` kolumny. Tak więc za pomocą `this.element.querySelector('ul')` dostaliśmy się do właściwej listy. Teraz możemy podpiąć do niej kartę za pomocą `appendChild(card.element)`.

Dlaczego `card.element`? Naszą kartę będziemy konstruować w sposób analogiczny do tego, w jaki konstruowaliśmy kolumnę. Tak więc obiekt karty będzie posiadał w sobie właściwość `element`, gdzie będzie trzymany węzeł DOM.

Metoda removeColumn

Po prostu kasuje kolumnę :) Nie stoi za tym żadna skomplikowana logika - za pomocą metody `removeChild()` usuwamy po prostu element ze strony. `this.element.parentNode.removeChild(this.element);` usunie naszą kolumnę w momencie, w którym przyciśniemy przycisk “x".

Klasa Card

Ok, kolumnę już mamy. Teraz zajmijmy się tworzeniem funkcji konstruującej klasę `Card`:

```
function Card(description) {
  var self = this;

  this.id = randomString();
  this.description = description;
  this.element = generateTemplate('card-template', { description: thi
}
```

Jeśli porównasz to z sekcją klasa `Column`, to zauważysz pewne podobieństwo. Klasy są skonstruowane niemal identycznie. Właściwość element (analogicznie do przykładu z kolumną) przechowuje element DOM, który tworzy nam ta funkcja.

Szablon kolumy

W pliku index.html umieścimy szablon kolumny, o id identycznym, jak to, które przekazaliśmy do funkcji `generateTemplate()`.

Ten segment docelowo będzie wyglądał tak:

```
<script id="card-template" type="x-tmpl-mustache">
  <div class="card">
    <p class="card-description">{{ description }}</p>
    <button class="btn-delete">x</button>
  </div
</script>
```

Ok, mamy stworzone wszystkie potrzebne klocki, z których powinna być zbudowana kolumna. Co dalej?

Podpinanie zdarzeń

Tutaj mamy do podpięcia tylko jedno zdarzenie - usunięcie karty.

Ten kod również umieszczamy wewnątrz funkcji konstruującej kartę `Card()`.

Usuwanie karty powinno prezentować się następująco:

```
this.element.querySelector('.card').addEventListener('click', functio
  event.stopPropagation();

  if (event.target.classList.contains('btn-delete')) {
    self.removeCard();
  }
});
```

Na kliknięcie wywołujemy (jeszcze nie istniejącą) metodę `removeCard()`. Metodę zaimplementujemy sobie troszkę później. Najpierw poskładajmy w jedną całość klocki, z których zbudowana jest karta.

Ok, jeśli postępowaliście zgodnie z instrukcją, to całość funkcji Card powinna wyglądać następująco:

```
function Card(description) {
  var self = this;

  this.id = randomString();
  this.description = description;
  this.element = generateTemplate('card-template', { description: thi

  this.element.querySelector('.card').addEventListener('click', funct
    event.stopPropagation();

    if (event.target.classList.contains('btn-delete')) {
      self.removeCard();
    }
  });
}
```

Metoda dla klasy `Card`

Jedyną metodą, którą musimy zaimplementować w przypadku karty, jest jej usunięcie. Dodajmy więc do prototypu metodę `removeCard()`:

```
Card.prototype = {
    removeCard: function() {
        this.element.parentNode.removeChild(this.element);
    }
}
```

Obiekt tablicy

Do tej pory stworzyliśmy dwie klasy: `Column` i `Card`. Jeśli spojrzysz na początkowe założenia projektu, to przypomnisz sobie, że uznaliśmy, że nie ma sensu tworzenie klasy dla tablicy, gdyż na naszej stronie będzie tylko jeden taki element.

Stworzyliśmy już nawet w pliku index.html jego podstawową strukturę:

```
<div id="board" class="board">
      <h1>Kanban board</h1>
      <button class="create-column">Add a column</button>
      <div class="column-container"></div>
</div>
```

Tablica to `div` o klasie `board`, który ma wewnątrz tytuł, przycisk do dodawania nowej kolumny oraz kontener, w którym będziemy przechowywali dodawane kolumny.

Musimy stworzyć obiekt tablicy w JavaScripcie i przypiąć odpowiednie nasłuchiwania zdarzeń. Robimy to w następujący sposób:

```
var board = {
    name: 'Kanban Board',
    addColumn: function(column) {
      this.element.appendChild(column.element);
      initSortable(column.id); //About this feature we will tell late
    },
    element: document.querySelector('#board .column-container')
};
```

Nie ma potrzeby tworzenia żadnego prototypu, bo nie ma klasy. Metoda, której będziemy używali do tworzenia nowej kolumny, jest podpięta bezpośrednio do obiektu `board`.

```
addColumn: function(column) {
  this.element.appendChild(column.element);
  initSortable(column.id); //About this feature we will tell later
}
```

Metoda ta ma za zadanie stworzyć kolumnę dzięki przypięciu jej elementu do elementu tablicy. Ok, to co w takim razie kryje się pod `this.element`?

`this.element` wskazuje na `board.element`. W tej właściwości trzymany jest poprawny element kontenera tablicy. Zgodnie ze strukturą HTML znajdziemy go, używając następującego selektora:

```
document.querySelector('#board .column-container')
```

Funkcja: initSortable()

Na początku, w ustaleniach, założyliśmy, że tablica kanban będzie miała funkcjonalność przenoszenia kart z jednego miejsca do innego (np. do innej kolumny albo do innej pozycji w tej samej kolumnie). W tym celu dodaliśmy sobie bibliotekę Sortable. Dzięki niej możemy korzystać z opcji drag'n'drop, która pozwala nam przenosić elementy na stronie. Dodatkowo biblioteka udostępnia opcję, dzięki której możemy sortować elementy listy za pomocą metody przeciągnij i upuść. Implementacja tej funkcji wygląda następująco:

```
function initSortable(id) {
  var el = document.getElementById(id);
  var sortable = Sortable.create(el, {
    group: 'kanban',
    sort: true
  });
}
```

Istotnym elementem, na który trzeba zwrócić uwagę to parametr, który przyjmuje ta funkcja. Wcześniej dodaliśmy do naszego szablonu kolumny id, które teraz pozwala nam jednoznacznie zidentyfikować kolumnę i włączyć dla każdej nowo utworzonej funkcjonalność naszej biblioteki Sortable. Dodatkowo opcja `group` musi być ustawiona na tę samą wartość we wszystkich elementach, pomiędzy którymi ma być dostępna funkcjonalność przenoszenia kart.

Nasza tablica ma w sobie przycisk służący do dodawania kolejnych kolumn. Trzeba podpiąć na ten element zdarzenie kliknięcia, aby obsługiwało wrzucanie nowej kolumny do tablicy.

```
document.querySelector('#board .create-column').addEventListener('cli
    var name = prompt('Enter a column name');
    var column = new Column(name);
    board.addColumn(column);
});
```

1. Wybieramy przycisk.
2. Na kliknięcie ma się wykonać funkcja, która zapyta nas o nazwę kolumny, którą chcemy stworzyć.
3. Funkcja utworzy też nową instancję, która z kolei ustawi tytuł kolumny.
4. Następnie funkcja utworzy na tablicy nową kolumnę.

Dla chętnych

Spróbujcie zastanowić się, jak przerobić obiekt `board` na klasę `Board` - tak aby można było tworzyć więcej niż jedną tablicę.

Tworzenie domyślnych kolumn i kart

Ok, mamy już wszystko, czego nam trzeba do tworzenia nowych obiektów. Do dzieła!

Na końcu skryptu dodaj następujące fragmenty kodu odpowiadające za stworzenie podstawowych elementów w Twoim kanbanie:

```
// CREATING COLUMNS
var todoColumn = new Column('To do');
var doingColumn = new Column('Doing');
var doneColumn = new Column('Done');

// ADDING COLUMNS TO THE BOARD
board.addColumn(todoColumn);
board.addColumn(doingColumn);
board.addColumn(doneColumn);

// CREATING CARDS
var card1 = new Card('New task');
var card2 = new Card('Create kanban boards');

// ADDING CARDS TO COLUMNS
todoColumn.addCard(card1);
doingColumn.addCard(card2);
```

Zadanie: Budujemy nasz mały Kanban

Twoim zadaniem jest zbudowanie własnego małego kanbana. Wykonaj wszystkie instrukcje z tego submodułu na swoim lokalnym środowisku i udostępnij wyniki pracy swojemu mentorowi za pośrednictwem GitHuba :)

Jeszcze raz przypomnijmy sobie kroki postępowania przy tworzeniu programu za pomocą obiektowego paradygmatu JavaScript.

1. Zastanawiamy się nad logiką naszej aplikacji. Jakie obiekty mogą mieć cechy i jakie czynności mogą wykonywać.
2. Implementujemy pierwsze funkcje konstruujące.
3. Dodajemy do klas atrybuty.
4. Dodajemy metody do prototypów pamiętając o zjawisku utraty kontekstu.
5. Tworzymy instancje obiektów za pomocą słowa kluczowego *new*.
6. Wykonujemy operacje na obiektach :)

Pamiętajcie o wrzuceniu kodu na repozytorium! W kwestii ostylowania wykażcie się swoją inwencją twórczą! Zobaczymy, kto z was zrobi najlepszy design. :)

Podgląd zadania

https://github.com/0na/

Wyślij link ✔