NIETYLKOPROGRAMISTA.PL

20 pytań na rozmowę techniczną dla Frontend Developera

NIE TYLKO
PROGRAMISTA

Czym jest higher-order function?

Funkcje w Javascript są ***first-class objects***. To znaczy, że mogą być przechowywane w zmiennych, przekazywane jako argument do funkcji i zwracane z funkcji. ***Higher-order function*** (HOF) to funkcja, która przyjmuje funkcję jako argument lub ją zwraca.

Przykłady HOF przyjmujących funkcję: *Array.prototype.map()*, *Array.prototype.filter()* ...

Przykładowy HOF zwracający funkcję:

```
function add(a) {
  return function(b) {
    return a + b;
  }
}

const add2 = add(2);
console.log(add2(5)); // 7
```

Jaka jest różnica między == i === ?

Porównanie przy pomocy **==** sprawdza, czy typy są takie same i jeśli się zgadzają, to porównuje wartości. W przeciwnym wypadku dokonuje koercji zgodnie z algorytmem opisanym w specyfikacji.

Porównanie przy pomocy **===** sprawdza czy typy są takie same i jeśli są to porównuje wartości. W przeciwnym wypadku zwraca *false*.

Czy {} === {} ?

```
console.log({} === {}); // false

const a = {};
console.log(a === a); // true
```

{} === {} zwróci *false*, ponieważ w przypadku typów złożonych takie porównanie odbywa się poprzez porównanie referencji. To znaczy, że sprawdzane jest czy oba te obiekty są w tym samym miejscu w pamięci.

a === a zwróci *true*, ponieważ jest to jeden i ten sam obiekt, znajdujący się w tym samym miejscu w pamięci.

Jak skopiować obiekt? (1/3)

Object.assign() i *...spread* operator

```
const objectToCopy = {a: 1, b: 2};
const copy1 = Object.assign({}, objectToCopy);
const copy2 = {...objectToCopy};
```

Dwie pierwsze propozycje to skorzystanie albo z **Object.assign**, albo ze **...spread operator**. W obu przypadkach kopiowanie jest płytkie. To znaczy, że jeśli jednym z pól obiektu jest jakiś typ złożony (obiekt, tablica, funkcja itp.) to zostanie skopiowana jego referencja a nie wartość.

```
const obj = { test: 1 };
const objectToCopy = { a: obj };
const copy1 = {...objectToCopy};
obj.test = 99;
copy1.a.test // 99
```

Jak skopiować obiekt? (2/3)

JSON.stringify(), JSON.parse()

```
const obj = { test: 1 };
const objectToCopy = { a: obj };
const copy1 = JSON.parse(JSON.stringify(objectToCopy));
obj.test = 99;
copy1.a.test // 1
```

Użycie *JSON.parse(JSON.stringify(obj))*, aby uzyskać kopiowanie głębokie **jest słabym pomysłem**. Nie warto używać tego sposobu dlatego, że:

- Jest to najmniej wydajny opcja
- Tracimy informację o funkcjach dostępnych dla obiektu (JSON nie wspiera serializacji funkcji)
- Wartości takie jak NaN czy Infinity zostaną zamienione na null (JSON nie wspiera serializacji takich wartości)
- Daty utracą swój typ i zostaną zamienione na stringi

Jak skopiować obiekt? (3/3)

lodash, ramda

Bardzo często projekt już posiada jakąś bibliotekę pomocniczą. Istnieje duża szansa, że taka biblioteka posiada funkcję do kopiowania. W lodashu jest to **_.cloneDeep()**, a w ramdzie **R.clone()**.

Własna funkcja

Na rozmowie kwalifikacyjnej możesz zostać poproszony o napisanie własnej funkcji głębokiego kopiowania, dlatego warto wiedzieć jak to zrobić.

```
function cloneDeep(obj) {
  return Object.entries(obj).reduce((acc, [key, value]) => {
    if (typeof value === 'object') {
      acc[key] = cloneDeep(value);
    } else {
      acc[key] = value;
    }

    return acc;
  }, {});
}
```

Czym jest *pure function*?

Czyste funkcja ma następujące cechy:

- Deterministyczna – wywołana z tymi samymi argumentami za każdym razem zwróci taki sam wynik (tak jak funkcja matematyczna)
- *Referential transparency* – wynik funkcji zależy tylko i wyłącznie od jej argumentów. Widząc w kodzie wywołanie czystej funkcji, moglibyśmy je zamienić na wartość, którą zwróci bez stworzenia jakichkolwiek błędów.
- Nie produkuje efektów ubocznych (*side effects*) takich jak manipulacja DOM, zapytania HTTP czy operacje I/O.

Takie funkcje są **łatwe do testowania**, ale aplikacja nie może się składać z samych czystych funkcji, ponieważ trzeba odczytać dane, pokazać je użytkownikowi i na końcu zapisać zmiany, które zrobił. To wszystko są efekty uboczne, których czyste funkcje zabraniają.

Funkcje realizujące logikę powinny być czyste i przetestowane. Możemy ich potem użyć w procedurach produkujących skutki uboczne. Dzięki temu, w wypadku pojawienia się jakiegoś *buga*, będziemy mogli szybciej sprawdzić czy błąd znajduje się w czystych funkcjach (testy), czy w procedurach.

Jak napisać funkcję przyjmująca nieokreśloną liczbę argumentów?

Aby napisać taką funkcję należy skorzystać z ***...rest operator***. W ten sposób zbierzemy wszystkie przekazane argumenty do tablicy.

```
function add(...numbers) {
  return numbers.reduce((total, number) => total + number);
}
```

Jako ciekawostkę można pokazać jeszcze jedno inne rozwiązanie. **Jest ono przestarzałe i nie zalecane**. Funkcje zadeklarowane przy pomocy słowa kluczowego *function* posiadają domyślną "tablicę" przekazanych argumentów: *arguments*.

arguments formalnie nie jest tablicą (posiada tylko właściwość length), dlatego konwertuję je do tablicy korzystając ze ***...spread operator***.

```
function add() {
  const numbers = [...arguments];
  return numbers.reduce((total, number) => total + number);
}
```

Jak działa this? (1/4)

Domyślne działanie w funkcji

this to kontekst funkcji. Domyślnie wskazuje na obiekt *window*. W *strict mode* ma wartość *undefined*.

```
function thisStrict() {
  "use strict"
  console.log(this); // undefined
}
function thisDefault() {
  console.log(this); // window
}
```

Jak działa this? (2/4)

Domyślne działanie w obiekcie

W przypadku, kiedy odwołujemy się do ***this*** w metodzie należącej do obiektu, wartością ***this*** staje się obiekt po lewej stronie od kropki poprzedzającej metodę.

```
const ala = {
  name: 'Ala',
  sayHi() {
    console.log(`Hello, I'm ${this.name}`);
  }
}
const john = {
  name: 'John',
  sayHi() {
    console.log(`Hello, I'm ${this.name}`);
  }
}
ala.sayHi() // Hello I'm Ala
john.sayHi() // Hello I'm John
```

Jak działa this? (3/4)

Ustawianie kontekstu funkcji w czasie wywołania

Kontekst funkcji możemy ustawić w momencie wywołania za pomocą metody ***call*** lub ***apply***. Obie z nich w ten sam sposób zmieniają *this*. Różni się jedynie sposób w jaki przekazują dodatkowe argumenty. Dzięki temu możemy zmodyfikować domyślne zachowanie brania obiektu przed kropką jako *this* ustawiając je jako dowolną wartość.

```
thisStrict.call("CALL"); // CALL
thisDefault.apply("APPLY"); // APPLY
john.sayHi.call(ala); // Hello I'm Ala
```

Jak działa this? (4/4)

Stworzenie nowej funkcji z ustawionym kontekstem

Ostatnim sposobem na zarządzanie kontekstem jest metoda ***bind***, która tworzy nową funkcję z *this* ustawionym na sztywno. Kontekst funkcji otrzymanej po użyciu ***bind*** nie może zostać zmodyfikowany przez *call* lub *apply*.

```
const bindFn = thisStrict.bind("BIND");
bindFn(); // BIND
bindFn.call("TEST?"); // BIND
```

bind jest szczególnie przydatny, kiedy przekazujemy funkcję jako *callback*. Dzięki skorzystaniu z ***bind***, możemy mieć pewność, że zostanie użyty odpowiedni kontekst.

```
setTimeout(ala.sayHi, 0); // Hello, I'm undefined
setTimeout(ala.sayHi.bind(ala), 0); // Hello, I'm Ala
```

Praca z liczbami w języku Javascript

W języku Javascript istnieje typ ***Number***. Przechowuje on tylko liczby zmiennoprzecinkowe i ma na to 64 bity.

To znaczy, że wszystkie obliczenia mają ograniczoną precyzję, więc w przypadku gdy wykonujemy operacje wymagające dużej precyzji (np. pieniądze), powinniśmy skorzystać z biblioteki, która ją zapewnia.

```
console.log(0.1 + 0.2); // 0.30000000000000004
console.log(0.1 + 0.2 === 0.3); // false
console.log(9007199254740991.123); // 9007199254740991
console.log(9007199254740.991123); // 9007199254740.99
```

Jeśli operujemy tylko na liczbach całkowitych, to możemy skorzystać z API ***BigInteger***. Liczbę oznaczamy jako BigInteger poprzez dodanie na końcu litery n.

```
const a = 9007199254740991n;
const b = 9007199254740991n;
console.log(a + b) // 18014398509481982n
```

CSS Specificity

Siła selektorów od najsilniejszego do najsłabszego:

1. Selektor id – #logo, #hamburger ...
2. Selektor klasy – .list, .list__item ...
3. Selektor elementu – h1, header, main, footer ...

Selektor zawierający dwie klasy jest silniejszy niż ten zawierający tylko jedną klasę, ale selektor po id i tak będzie od niego silniejszy.

W przypadku gdy oba selektory mają taką samą siłę, to style zostaną wzięte od tego selektora, który znajduję się najniżej (jest najdalej w pliku).

Jeśli chcesz dowiedzieć się w jaki sposób kalkulowana jest siła selektora to mogę Ci polecić ten wpis.

RWD, Mobile first

RWD - projektowanie stron w taki sposób, że wyglądają dobrze niezależnie od szerokości i wysokości ekranu na którym są wyświetlane. Jest to możliwe dzięki zastosowaniu *media queries*.

Mobile first - korzystamy z *media queries* w górę. To znaczy, że najpierw dodajemy takie style, aby strona wyglądała dobrze na telefonach. Następnie stopniowo zwiększamy szerokość strony. W momencie, kiedy strona przestaje dobrze wyglądać dodajemy *@media (min-width: X)*, gdzie X to szerokość ekranu w którym wygląd strony zaczął się psuć i poprawiamy niekorzystanie wyglądające elementy.

Osobiście często zamiast znajdować te miejsca samemu, korzystam z wartości breakpointów używanych przez popularne biblioteki (bootstrap, material-ui, tailwind.css czy bulma).

Co jest lepsze: Flexbox czy Grid?

Żadne z nich nie jest ani lepsze, ani gorsze. Jedne problemy łatwiej rozwiązać używając *flexboxa*, inne wykorzystując *grida,* a czasem nie ma różnicy które z nich wybierzemy, ponieważ oba sprawdzą się tak samo dobrze.

Grid daje kontrolę nad tym w jaki sposób rozmieszczamy elementy. Pozwala na dokładne określenie wierszy i kolumn. Łatwo określić przerwy między elementami dzięki atrybutowi *grid-gap*.

Flexbox z drugiej strony daje elastyczność i pozwala na wykorzystanie 100% dostępnej przestrzeni.

Podsumowując trzeba znać oba rozwiązania i wykorzystywać je w zależności od napotkanego problemu. Wes Bos stworzył świetne darmowe kursy o *flexboxie* ii *gridzie*. Pokazuje w nich kilka przykładowych zastosowań obu rozwiązań.

Pseudo-elementy, pseudo-klasy

Pseudo-elementy - wirtualne elementy pozwalające na dodatkowe stylowanie bez dodawania zbędnych elementów do DOM. Do najczęściej wykorzystywanych należą *::before* i *::after*. Listę wszystkich dostępnych pseudo-elementów możesz znaleźć w dokumentacji MDN.

Pseudo-klasy - pozwalają na stylowanie elementów na podstawie stanu w jakim się znajdują (np. *:hover*, *:disabled*, *:checked*, ...) lub pozycji w DOM (*:first-child*, *:last-child*, *:nth-child()*, *:nth-of-type()*, ...). Jedną z moich ulubionych pseudo-klas jest *:not()* umożliwiający wybranie elementów, które nie spełniają zadanego warunku. Można łączyć wiele pseudo-klas. Listę wszystkich dostępnych pseudo-klas znajdziesz w dokumentacji.

Przykładowe selektory korzystające z pseudo-klas:

- *.order__item:not(:first-child):not(:last-child)* - wszystkie elementy poza pierwszym i ostatnim
- *.order__item:nth-child(2n)* - co drugi element zaczynając od drugiego
- *.order__item:nth-child(2n + 1)* - co drugi element zaczynając od pierwszego

Architektura CSS

CSS bardzo szybko wymyka się spod kontroli. Receptą na walkę z powstającym chaosem jest wykorzystanie sprawdzonej architektury. Jednym z najpopularniejszych podejść jest BEM (block-element-modifier).

Zaletami BEM'a są:

- naturalność – patrząc na stronę, z reguły bez trudu możemy określić komponenty (bloki) z których się składa (np. nawigacja, formularz logowania, artykuł).
- prostota – aby go używać wystarczy stosować się do jasno określonych zasad nazywania klas
- modularność – strona składa się z komponentów. Bloki składają się z elementów. Zarówno bloki jak i elementy mogą być w różnym stanie w zależności od modyfikatora. Modularność o której tu wspomniałem polega na tym, że taki kawałek HTML'a (z odpowiednio nazwanymi klasami) wraz z dotyczącym go CSS możemy bez trudu wyciąć, wkleić do innego projektu i wszystko będzie działać.

Protokoły http/https to protokoły bezstanowe. Jakie są możliwości zachowania stanu pomiędzy wizytami użytkownika?

1. **local storage**
   - przechowuje dane bez okresu wygaśnięcia (dane mogą być usunięte przez użytkownika lub za pomocą js)
   - dostęp tylko po stronie przeglądarki
2. **session storage**
   - dane są przechowywane do momentu zamknięcia karty/przeglądarki
   - dane nie znikają pomiędzy przeładowaniami strony
   - dostęp tylko po stronie przeglądarki
3. **cookies**
   - możliwość ustawienia zarówno po stronie przeglądarki, jak i serwera
   - przechowują mała ilość danych (nie więcej niż 4KB)
   - mogą być ustawione z flagą *httpOnly*. Wtedy można je czytać tylko po stronie serwera
   - można zarządzać czasem przez który są dostępne za pomocą parametrów: expires lub max-age

Jakie narzędzie pozwala na używanie najnowszych funkcjonalności JS w starszych przeglądarkach?

Narzędziem pozwalającym na korzystanie z najnowszych funkcjonalności języka Javascript jest ***babel***. Umożliwia on zdefiniowanie przeglądarek, które chcemy wspierać i na tej podstawie babel dokona transpilacji kodu na taki, który będzie działał w każdej z nich.

Babel ma wiele pluginów pozwalających na korzystanie z funkcjonalności jeszcze niedostępnych w języku Javascript. Dzięki użyciu takich wtyczek możemy dać sobie możliwość korzystania z dekoratorów, *optional chaining*, operatora *pipe* czy prywatnych metod klasowych.

Jest to świetne narzędzie pozwalające programiście skupić się na rozwiązywanym problemie, jednocześnie dając mu możliwość korzystania z wszystkich możliwych udogodnień języka.

Testy

Unit testy (testy jednostkowe) – testują pojedynczą atomową funkcjonalność.
Testy integracyjne – testują interakcje pomiędzy modułami.
Testy e2e – testują cały system, najczęściej korzystając z graficznego interfejsu użytkownika.

TDD (*test driven development*) – najpierw piszemy test sprawdzający funkcjonalność systemu, a dopiero potem realizujemy ją za pomocą kodu. Pozwala to lepiej przemyśleć rozwiązanie i bardzo często skutkuje czystszym kodem wynikowym.

TDD odbywa się w cyklu:

1. Piszemy test
2. Piszemy kod
3. Refaktoryzujemy (po każdej zmianie odpalamy test, aby sprawdzić czy nic nie popsuliśmy)

Dependency injection (wstrzykiwanie zależności) – polega na przekazywaniu zależności jako parametr (zamiast używać *fetch* bezpośrednio w funkcji, przekazujemy do niej stworzony przez nas *apiClient*, który udostępnia metodę get wykorzystującą *fetch* pod spodem). Dzięki temu, w teście możemy przekazać inny obiekt, o takim samym interfejsie, ale nie robiący prawdziwego zapytania.

Dobre praktyki, które warto znać (1/2)

DRY (*Don't repeat yourself*) – wydzielanie powtarzających się czynności do funkcji

KISS (*Keep it simple, stupid*) – im prościej tym lepiej. Czasem lepiej napisać kod mniej optymalny, ale bardziej czytelny dla reszty programistów. Wyjątkiem od tej reguły są oczywiście wszelkiej maści gry czy systemy czasu rzeczywistego, gdzie optymalizacja jest szalenie ważna.

Law of Demeter – staramy się zachować jak najmniejszy *coupling* i jak największą kohezję. To prawo mówi o tym, że metoda danego obiektu może odwoływać się jedynie do metod należących do:

- tego samego obiektu
- parametrów przekazanych do metody
- obiektów stworzonych w tej metodzie
- do parametrów lub metod klasy w której się znajduje

Dobre praktyki, które warto znać (2/2)

SOLID

- ***S (single responsibility principle)*** – każda klasa czy funkcja powinna mieć tylko jedną odpowiedzialność.
- ***O (open/closed principle)*** – klasy powinny być zamknięte na modyfikacje, ale otwarte na rozszerzenia. To znaczy, że powinniśmy pisać takie klasy, które można rozszerzyć bez modyfikacji ich kodu źródłowego (poprzez dziedziczenie lub kompozycje).
- ***L (Liskov substitution principle)*** – metoda, która oczekuje obiektu klasy bazowej T, powinna działać tak samo dobrze, jeśli przekażemy do niej obiekt klasy E (gdzie <E extends T>). W języku Javascript nie ma statycznego typowania, jednak ta zasada jest realizowana za pomocą ***duck typing***. To znaczy, że jeśli do metody przekażemy obiekt o podobnym interfejsie (mający atrybuty i metody używane w danej funkcji), to kod się wykona bez żadnego błędu.
- ***I (interface segregation principle)*** – wiele specyficznych interfejsów jest lepsze niż jeden ogólny. W języku Javascript mówiąc interfejs mamy na myśli metody udostępnione przez obiekt. Ta zasada mówi o zachowaniu jak największej kohezji. Nie powinniśmy do obiektu *Calculator(add, subtract, multiply, divide)* nagle dodać metody *readFileFromPath*, ponieważ nie ma ona żadnego związku z resztą metod w tym obiekcie.
- ***D (dependency inversion principle)*** – zasada mówiąca o zachowaniu jak najmniejszego *couplingu.* Między modułami wysokopoziomowymi a niskopoziomowymi powinniśmy tworzyć abstrakcję, poprzez którą będzie odbywać się komunikacja. Komunikując się należy unikać odwoływania się do szczegółów implementacyjnych a korzystać z udostępnionych interfejsów.

Czym się różni *git fetch* od *git pull*?

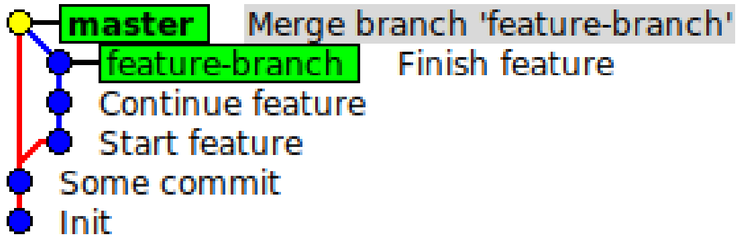
git fetch – pobiera dane ze zdalnego repozytorium. Jest to bezpieczna operacja, która nie dokonuje zmiany w plikach źródłowych.

git pull – pobiera dane ze zdalnego repozytorium i dokonuje synchronizacji ze zdalną gałęzią za pomocą *git merge*. Jest to operacja destrukcyjna, może wystąpić *merge conflict*.

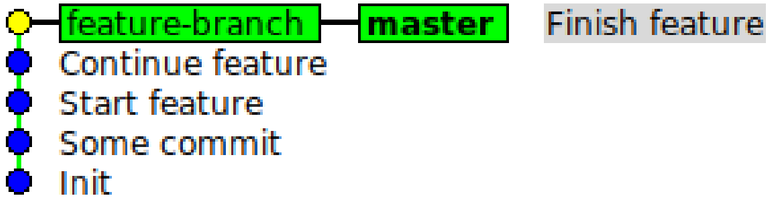
Czym się różni *git merge* od *git rebase*?

Obie komendy mają za zadanie zintegrować kod z dwóch gałęzi, robią to jednak w różny sposób. Mając *mastera* i *feature-branch* historia będzie wyglądać inaczej w zależności której z tych metod użyjemy.

git merge - tworzy dodatkowy *commit* tak zwany *merge commit*. Struktura historii wygląda wtedy następująco



git rebase - nie tworzy dodatkowego *commita*. Dla każdego *commita* z *feature-branch* tworzony jest nowy *commit* na *masterze*. To sprawia, że hasze *commitów* z *mastera* i z *feature-brancha* nie są takie same.



Czym się różni *git reset* od *git reset --hard*

git reset - usuwa zmiany ze *staging area*. Jest to całkowicie bezpieczna operacja. Nie usunie nic z naszego komputera. Jeśli podamy hasz *commita*, to przejdzie do tego miejsca w historii, ale wszystkie zmiany zostaną zachowane w *working area*.

git reset --hard - usuwa pliki z komputera. Jest to operacja destrukcyjna. Jeśli podamy hasz *commita*, to przejdzie do tego miejsca w historii pozbywając się wszystkich zmian.

Warto tutaj wspomnieć, że istnieje coś takiego jak ***git reflog***. Połączenie *git reflog* i *git reset --hard* pozwala na cofnięcie każdej zmiany (nawet destrukcyjnej).

Dzięki!

Jestem bardzo ciekawy, czy pytania które tu wymieniłem były dla Ciebie przydatne. Koniecznie napisz do mnie maila i podziel się swoją opinią! (adres znajdziesz poniżej)

Jeszcze raz Ci dziękuje i życzę miłego dnia.

nietylkoprogramista.pl

kontakt@nietylkoprogramista.pl

@darekmarkiewicz

Nie Tylko Programista

NIE TYLKO
PROGRAMISTA