**Higher Order Functions**

**Wstęp**

Szkolenie to stanowi uzupełnienie podstaw z JS. Z racji tego, że jest to wiedza w formie “bulletów” i nie wymaga tworzenia oddzielnych tematów, postanowiłem zawrzeć ją w jednym szkoleniu. Są to tematy, które zahaczają o podstawy, ale i również nawiązują już do bardziej zaawansowanych konceptów. Tak więc życzę Ci efektywnej pracy!

**Higher Order Functions**

Opisany w nagłówku tej sekcji rodzaj funkcji to taki, który operuje na innej funkcji - dzięki otrzymaniu jej jako argument lub zwrócenie funkcji (return). W kontekście JS skupimy się na wbudowanych funkcjach wyższego rzędu **Array.prototype.map**, **Array.prototype.filter** oraz **Array.prototype.reduce**.

**Array.prototype.map**

Funkcja ta służy do przekształcania tablicy elementów do innej postaci - tablicy nowych elementów (powstałych po przekształceniu starych).

Funkcja typu callback przesyłana do map akceptuje trzy argumenty: element, index, array.

Zanim przedstawię Ci przykładowe wykorzystanie funkcji map, bądź świadomy tego, iż w dużej części przypadków będziesz mógł wykorzystywać zarówno Higher Order Function jak i standardowy rodzaj funkcji przetwarzających dane. Jednak zaletą wynikającą z wykorzystania tego pierwszego rozwiązania jest to, że kod staje się o wiele krótszy i nie musimy wymyślać koła na nowo. Programista ma w rękach potężne rozwiązanie, które może optymalizować kod. Przewagę jednego podejścia nad drugim przedstawię Ci w poniższych przykładach.

**Syntax .map()**

Do listy argumentów funkcji .map() możemy przekazywać zmienne w różnej postaci (np. funkcje strzałkowe (arrow functions), callback functions czy inline callback functions).

| // Arrow function  map((element) => { ... })  map((element, index) => { ... })  map((element, index, array) => { ... })  // Callback function  map(callbackFn)  map(callbackFn, thisArg)  // Inline callback function  map(function callbackFn(element) { ... })  map(function callbackFn(element, index) { ... })  map(function callbackFn(element, index, array){ ... })  map(function callbackFn(element, index, array) { ... }, thisArg) |
| --- |

W swoich rozwiązaniach najczęściej wykorzystywać będzie arrow function oraz callback function. A to głównie dlatego, że w ten sposób zapis jest najkrótszy i oszczędzamy wiele miejsca.

**Zadanie**

Stwórz funkcjonalność, która umożliwi Ci potęgowanie każdego elementu z podanej listy: [1, 2, 3, 4, 10, 12, 20]

**Standardowe podejście**

| function powerElements(elems){  powers = []  for(elem of elems){  powers.push(elem \* elem);  }  return powers; }  console.log(powerElements([1, 2, 3, 4, 10, 12, 20])); // 1, 4, 9, 16, 100, 144, 400 |
| --- |

**Higher Order Function (array function)**

| function powerElements(elems){  return elems.map(elem => elem \* elem); }  console.log(powerElements([1, 2, 3, 4, 10, 12, 20])); // 1, 4, 9, 16, 100, 144, 400 |
| --- |

Zauważ, jak znacząco różni się drugie podejście od pierwszego. Jest o wiele krótsze od poprzedniego i wykorzystujemy w nim pierwszą poznaną higher order function - map().

Zwróć przede wszystkim uwagę na to, że .map() zwraca kopię nowo powstałego zbioru elementów. Fakt ten związany jest z jedną z cech higher order functions. Charakteryzują się one tym, że nie modyfikują one istniejących elementów, tylko na podstawie nich zwracają nowe wartości.

**Array.prototype.filter**

Filter() służy do przefiltrowywania iterowalnych struktur danych według określonego warunku. Syntax jest bardzo podobny do .map(), dlatego przejdziemy od razu do praktycznego zastosowania omawianej funkcji.

**Przykład nr 1**

| const countries = [  'Albania',  'Bolivia',  'Canada',  'Denmark',  'Ethiopia',  'Finland',  'Germany',  'Hungary',  'Ireland',  'Japan',  'Kenya', ]  const countriesContainingLand = countries.filter((country) => country.includes('land')); console.log(countriesContainingLand); |
| --- |

Wynik:

['Finland', 'Ireland']

**Przykład nr 2**

| const countriesHaveFiveLetters = countries.filter(  (country) => country.length === 5 ); console.log(countriesHaveFiveLetters); |
| --- |

**Array.prototype.reduce**

Reduce() to kolejny przykład funkcji “wyższego rzędu”. Funkcja ta służy do “redukowania” zawartości danej struktury. Czyli, np. może posłużyć do przekształcania listy wartości liczbowych w ich sumę.

Zanim przedstawię Ci syntax funkcji reduce, pochylmy się nad prostym przykładem wykorzystującym reduce().

| const numbers = [1, 2, 3, 4, 5] const sum = numbers.reduce((acc, cur) => acc + cur, 0);  console.log(sum); // 15 |
| --- |

Zapewne może zastanawiać Cię sens wykorzystania **acc** oraz **cur** w powyższym przykładzie. Otóż taki zapis wynika z faktu, iż reduce przyjmujący funkcję strzałkową, musi zawsze bazować na co najmniej dwóch argumentach - previousValue (acc) oraz currentValue (cur). Dzięki temu automatycznie mogą być przypisywane wartości do acc oraz cur, a następnie odpowiednio przetwarzane.

| // Arrow function  reduce((previousValue, currentValue) => { ... } )  reduce((previousValue, currentValue, currentIndex) => { ... } )  reduce((previousValue, currentValue, currentIndex, array) => { ... } )  reduce((previousValue, currentValue, currentIndex, array) => { ... }, initialValue)  // Callback function  reduce(callbackFn)  reduce(callbackFn, initialValue)  // Inline callback function  reduce(function callbackFn(previousValue, currentValue) { ... })  reduce(function callbackFn(previousValue, currentValue, currentIndex) { ... })  reduce(function callbackFn(previousValue, currentValue, currentIndex, array){ ... })  reduce(function callbackFn(previousValue, currentValue, currentIndex, array) { ... }, initialValue) |
| --- |

**Przykład nr 2**

Czasami dobrym rozwiązaniem może być również zdefiniowanie wartości początkowej, od której rozpocznie się naliczanie sumy redukowanej listy. Wówczas możemy skorzystać z poniższego zapisu...

reduce(callbackFn, initialValue)

... i oprócz przesłania callback funkcji, określić również wartość startową.

| const array1 = [1, 2, 3, 4]; const reducer = (previousValue, currentValue) => previousValue + currentValue;  // 5 + 1 + 2 + 3 + 4 console.log(array1.reduce(reducer, 5)); // 15 |
| --- |

**Array.prototype.every**

Every() jest przykładem funkcji, która sprawdza, czy każdy element danej struktury jest taki sam pod kątem sprecyzowanego aspektu. Every zwraca zawsze wartość logiczną true/false - w zależności od tego, czy elementy są wspólne dla danego warunku.

Poniżej przykład programu, który sprawdza, czy **każdy element** z listy jest stringiem.

**Przykład nr 1**

| const names = ['Asabeneh', 'Mathias', 'Elias', 'Brook'] const areAllStr = names.every((name) => typeof name === 'string');  console.log(arrAllStr); // true |
| --- |

**Przykład nr 2**

| const bools = [true, true, true, true] const areAllTrue = bools.every((b) => {  return b === true })  console.log(areAllTrue) // true |
| --- |

Standardowo poniżej umieszczam syntax funkcji every(). Aczkolwiek syntax ten jest na tyle podobny do wyżej opisanych funkcji, że do tworzenia every można podchodzić mocno intuicyjnie.

| // Arrow function  every((element) => { ... } )  every((element, index) => { ... } )  every((element, index, array) => { ... } )  // Callback function  every(callbackFn)  every(callbackFn, thisArg)  // Inline callback function  every(function callbackFn(element) { ... })  every(function callbackFn(element, index) { ... })  every(function callbackFn(element, index, array){ ... })  every(function callbackFn(element, index, array) { ... }, thisArg) |
| --- |

**Array.prototype.some**

I na koniec… czas na omówienie funkcji, która ma podobne działanie do every, z tą różnicą, że sprawdza, czy **co najmniej jeden element** spełnia określony warunek. Wówczas zwraca true, w przeciwnym razie false.

| const names = ['Asabeneh', 'Mathias', 'Elias', 13] const areAllStr = names.some((name) => typeof name === 'number'); console.log(areAllStr); // false |
| --- |