**Szkolenie X - JavaScript**

**Złożone struktury danych.**

**Wstęp**

W jednym z pierwszych szkoleń wspomniałem Ci o typach zmiennych. Poznałeś wtedy takie rodzaje danych jak: int, string, double itd. Napomknąłem wtedy również o, tzw. **złożonych strukturach danych**. Noszą one taką, a nie inną nazwę, z racji iż możemy w nich przechowywać wiele wartości jednocześnie. Do grona takich struktur zaliczamy m.in **Array**, **Map**, **Set**. Występuje jeszcze wiele innych złożonych typów jak: WeakMap, WeakSet, Int8Array, jednak z racji rzadszego wykorzystania, nie będę skupiał się na ich dokładnym omówieniu.

**ablice (ang. arrays)**

Zacznijmy od omówienia pierwszego złożonego typu danych - **tablicy** (nazywanej też **listą**). Tablice służą do przechowywania wielu wartości, które mogą być dowolnego typu. Każdy z tych elementów jest indeksowany, a więc ma przypisany określony numer, przez który możemy się do danej wartości odwołać. Co jest istotnie ważne w przypadku tablic to:

|  |
| --- |
| Indeksowanie zaczynamy od wartości 0. |

Powyższa zasada jest niezwykle ważna i stanowi jedną z głównych reguł języków programowania. Zgodnie z tym, chcąc odwołać się do pierwszego elementu, wykorzystamy indeks równy 0, do drugiego: 1, trzeciego: 2 itd.

Aby utworzyć listę elementów, należy wykorzystać nawiasy kwadratowe [] i przypisać je do dowolnej zmiennej, która to właśnie będzie reprezentowała omawianą strukturę. Mają już tak utworzoną tablicę będziemy mogli dodawać i usuwać poszczególne elementy, odczytywać niektóre z nich itd.

**Powszechne operacje na tablicach**

**Utworzenie tablicy**

|  |
| --- |
| let names = ['Kacper', 'Jan', 'Anna'] // utworzenie tablicy z trzema elementami let surnames = [] // utworzenie pustej tablicy (elementy będziemy dodawali później) |

**Odczytanie długości tablicy**

W celu sprawdzenia, z ilu elementów składa się tablica, wykorzystamy zapis **nazwaTablicy.length**.

|  |
| --- |
| console.log(names.length); // 3 |

**Pobieranie określonych elementów**

|  |
| --- |
| console.log(names[0]); // 'Kacper' console.log(names[names.length - 1]); // 'Anna' |

O indeksowaniu elementów wspomniałem już w poprzedniej sekcji, dlatego nie powinno Cię zaskoczyć wykorzystanie wartości 0 przy odwołaniu do pierwszego elementu.

Zwróć jednak szczególną uwagę na linię, w której pobraliśmy wyraz ‘Anna’. Indeks **names.length - 1** umożliwił nam uniwersalne odwołanie się do ostatniego elementu tablicy. Powiesz, że równie dobrze mogliśmy zapisać names[2] i wtedy otrzymalibyśmy ten sam efekt. Zgadza się, jednak przedstawione przeze mnie rozwiązanie jest o wiele bardziej uniwersalne i sprawdzi się w momencie, gdy programista nie wie, jakiej długości jest dana tablica elementów, a chciałby odwołać się do jej, np. ostatniego elementu.

**Dodanie nowego elementu do końca tablicy**

|  |
| --- |
| let newLength = names.push("Krzysztof"); // 4 |

**Push()** po dodaniu nowego elementu do wskazanej tablicy, jako rezultat zwraca jej nową długość - dlatego tę wartość zapamiętaliśmy pod zmienną **newLength**.

**Usunięcie elementu z końca tablicy**

|  |
| --- |
| let removedLast = names.pop(); // "Krzysztof"  // ["Kacper", "Jan", "Anna] |

Funkcja **pop()** po usunięciu ostatniego elementu z tablicy, zwraca jego wartość.

**Usunięcie elementu z początku tablicy**

|  |
| --- |
| let removedFirst = names.shift(); // "Kacper"  // ["Jan", "Anna"] |

**Umieszczenie nowego elementu na początku tablicy**

|  |
| --- |
| let newLength = names.unshift("Grzegorz"); // 3  // ["Grzegorz", "Jan", "Anna"] |

**Wyszukanie pozycji elementu**

|  |
| --- |
| let pos = names.indexOf("Jan"); // 1 |

**UWAGA:**

Jeśli na liście znajduje się kilka takich samych wartości, to **indexOf** zwróci indeks elementu stojącego **najbardziej “z lewej”**, np.

|  |
| --- |
| let fruits = ["Apple", "Banana", "Apple", "Banana", "Orange"] let pos = fruits.indexOf("Banana"); // 1 |

**Pobranie ostatniego indeksu**

Wyżej poznana metoda indexOf() umożliwia nam wyszukanie pierwszego elementu na liście o zadanej wartości. Jeżeli natomiast wykorzystamy metodę **lastIndexOf()**, będziemy mogli odnaleźć ostatni element o podanej wartości (zakładając, że mamy do czynienia z duplikatami).

|  |
| --- |
| let fruits = ["Apple", "Banana", "Apple", "Banana", "Orange"] let pos = fruits.lasIndexOf("Banana"); // 3 |

**Sprawdzenie, czy element znajduje się w liście**

Chcąc zweryfikować, czy dane element zawiera się w liście, wykorzystamy metodę **includes()**. Może ona również przyjmować indeks tablicy, od którego chcemy rozpocząć poszukiwanie (jeżeli nie podamy go, to metoda domyślnie rozpocznie przeszukiwanie od początku listy). Jako rezultat omawiana metoda zwraca wartość typu **bool** (true, gdy dany element został znaleziony, false w przeciwnym razie).

|  |
| --- |
| let fruits = ["Apple", "Banana", "Apple", "Banana", "Orange"] alert(fruits.includes("Orange")); // true alert(fruits.includes("Pineapple")); // false alert(fruits.includes(“Banana”, 4)); // false, bo poszukiwanie rozpoczynamy od 5-tego elementu |

**Usunięcie elementu po jego indeksie**

|  |
| --- |
| // names = ["Grzegorz", "Jan", "Anna"] let pos = 1; let removedElem = names.splice(pos, 1); // "Jan"  // ["Grzegorz", "Anna"] |

**UWAGA:**

**Splice** umożliwia również usuwanie wielu elementów od zadanej pozycji w tablicy. Wystarczy dostosować jego drugi argument, np. **names.splice(pos, 2)** spowoduje usunięcie dwóch elementów, wyliczając od zadanej pozycji (pos).

**Konkatenacja (łączenie) dwóch tablic**

Aby scalić ze sobą dowolną ilość tablic, wykorzystamy metodę **concat(arg1, arg2, ...)**. Przyjmuje ona dowolną ilość tablic i jako rezultat zwraca jedną dużą tablicę - będącą połączeniem przesłanych wartości.

|  |
| --- |
| let arr = [1, 2, 3, 4]; alert(arr.concat([5, 10])); // [1, 2, 3, 4, 5, 10] alert(arr.concat([1], 2, 3, 4)) // [1, 2, 3, 4, 1, 2, 3, 4] |

**Pobieranie części tablicy**

Chcąc wycinać z tablicy jej określone “skrawki”, wykorzystamy metodę **slice()**. Umożliwia ona pobranie określonej części tablicy. Załóżmy, że mamy następującą postać tablicy imion: names = [“Jan”, “Anna”, “Kacper”, “Krzysztof”]. Chcielibyśmy pobrać z niej “skrawek” składający się z środkowych elementów - “Anna” oraz “Kacper”.

|  |
| --- |
| let midPart = names.slice(1, 3); // 1 to start, 3 to end // ["Anna", "Kacper"] |

**UWAGA:**Stop to indeks określający do którego elementu **wyłącznie** mamy pobierać zawartość danej tablicy.

**Kopiowanie całej tablicy**

Chcąc skopiować zawartość całej tablicy, również wykorzystamy poznaną metodę **slice()**. Do slice nie będziemy jednak przesyłali żadnych argumentów. Powinieneś bowiem wiedzieć, że gdy nie definiujemy argumentów start i stop, slice automatycznie przyjmuje, że pobierany “skrawek” listy będzie zaczynał się na jej początku, a kończył na końcu (w wyniku czego kopiowana jest cała lista).

|  |
| --- |
| let orig = ["Apple", "Orange", "Strawberry"] let copy = orig.slice(); // ["Apple", "Orange", "Strawberry"] |

**Przechodzenie po tablicy przy użyciu pętli**

Załóżmy, że chcemy w programie przejść po całej tablicy elementów i wyświetlić z niej tylko i wyłącznie liczby parzyste. Aby to zrealizować, wykorzystamy pętlę for oraz instrukcje warunkowe. Tak naprawdę JavaScript umożliwia nam rozwiązanie tego problemu na wiele sposób, poniżej zaprezentuję Ci kilka z nich.

Najbardziej powszechne rozwiązanie:

|  |
| --- |
| let nums = [1, 2, 3, 10, 15, 20, 24, 28] for(let idx = 0; idx < nums.length; idx++){  if(!(nums[idx] % 2)){ // to samo co names[idx] % 2 == 0  console.log(nums[idx]);  } } |

Powyższy kod nie wymaga raczej wielu wyjaśnień, jednak bądź świadomy, iż tę funkcjonalność możemy zapisać w o wiele krótszy sposób.

**Alternatywny zapis fora**

|  |
| --- |
| let nums = [1, 2, 3, 10, 15, 20, 24, 28]  for(let num of nums){  if(!(num % 2)){  console.log(num);  } } |

**UWAGA:**Powyższy zapis działa tylko na nowszych wersjach przeglądarki, dlatego miej to na uwadze, gdy będziesz pracował na różnym oprogramowaniu.

**Wykorzystanie forEach**

Wyżej przedstawiłem Ci dwa sposoby na iterowanie po elementach tablicy. W tym podpunkcie chciałbym zaprezentować Ci trzeci sposób na iterację po tablicy i wykonywanie określonych działań. Do tego celu wykorzystamy metodę forEach, do której przekazywać będziemy funkcję realizującą określone działania.

Syntax:

|  |
| --- |
| array.forEach(function(item, index, array){  // ciało  }); |

Przykład wyświetlania tylko i wyłącznie liczb parzystych: **DOPYTAJ**

|  |
| --- |
| let nums = [1, 2, 3, 10, 15, 20, 24, 28]  nums.forEach(function(item, index, array){  if(!(item % 2)){  console.log(item);  } }); |

Z forEach możemy wykorzystać również wbudowane funkcje.

|  |
| --- |
| let nums = [1, 2, 3, 10, 15, 20, 24, 28]  nums.forEach(alert); // alertowanie każdego pojedynczego elementu |

**Inne rodzaje tablic**

Wyżej wymienione funkcjonalności są najczęściej wykorzystywane w odniesieniu do tablic. Niektóre z nich są mniej oczywiste i wymagają większego oswojenia, jednak trening czyni mistrza - będziesz miał okazję zaznajomić się z wyżej opisanymi metodami przy pisaniu własnego kodu, tudzież rozwiązywaniu naszych zadań.

Zanim przejdziemy do kolejnych, równie bardzo popularnych struktur danych (Map oraz Set), to chciałbym wspomnieć słowem o innych rodzajach tablicy, które są dostępne w JS. Możemy m.in wyróżnić: Int8Array, Uint8Array, Int16Array, Uint16Array itd.   
  
Różnica między poszczególnymi tablicami jest subtelna - różni się jedynie rozmiar elementów w niej zawartych. Domyślnie bowiem, typ Number (nazywany też int) w JS ma rozmiar równy około 9 B. Tak więc standardowa tablica złożona z 5-ciu liczb będzie zajmowała około 5 \* 9B = 45 B pamięci. Nie będzie to najefektywniejszy sposób przechowywania danych, gdy specyfika projektu wymagać będzie od programisty wykorzystania ograniczonych zasobów.   
  
Wtedy potencjalnym rozwiązaniem może być właśnie użycie tablic, które przechowują liczby “o mniejszym rozmiarze” (bądź jednak świadomy tego, iż mniejszy rozmiar typu liczbowego związany jest też z mniejszym zakresem wartości, jaki może on przechowywać).   
  
I tak przykładowo, gdybyśmy wykorzystali tablicę Int8Array, pojedynczy typ Number w niej zawarty, miałby wielkość równą zaledwie 8 bitów (1 B). Wtedy, gdybyśmy chcieli przechowywać 5 liczb, potrzebowalibyśmy zaledwie 5 B. Oczywiście w takiej sytuacji, tak jak już wspomniałem, musimy również mieć na uwadze, że mniejszy rozmiar zmiennej pociąga za sobą mniejszy zakres wartości, jaki możemy w niej umieścić (dostępny przedział dla int8 to od -127 do 127).

**Obiekt typu Map**

Mapa w JS służy do przechowywania par zmiennych, tj. **kluczy** i przypisanych do nich **wartości**. Tak naprawdę wszystkie niemutowalne zmienne dowolnego typu mogą być wykorzystywane jako klucze/wartości wewnątrz danej mapy. Bądź świadomy, że obok tablic jest to najczęściej wykorzystywany typ złożonych struktur danych. Umożliwia on wygodne przechowywanie skorelowanych ze sobą danych.

**Kiedy używać Map?**

Praktycznym przykładem wykorzystania Map może być sytuacja, kiedy to chcemy połączyć imiona użytkowników systemu z, np. ich datą urodzenia. Wówczas program efektywnie wykorzystujący mapę mógłby prezentować się następująco:

|  |
| --- |
| people = new Map(); // 1 people.set("Kacper", "01/06/1980"); // 2 people.get(“Kacper”); // 3 people.set("Jan", "01/05/2000") // 4  .set("Anna", "21/09/1965")   .set("Dominik", "20/01/1999"); people.delete("Kacper"); // 5 alert(people.size); // 6 |

1. Aby utworzyć mapę, wykorzystaliśmy zapis **new Map()**.
2. W celu dodania do mapy nowej pary klucz - wartość, zapisaliśmy: **map.set(key, value)**
3. Aby pobrać z mapy wartość znajdującą się pod danym kluczem (jeżeli klucz nie znajduje się w mapie, zostanie zwrócona wartość undefined), zapisaliśmy: **map.get(key)**. W rezultacie powyższy kod zwróci wartość równą **“01/06/1980”**.
4. JavaScript umożliwia również łańcuchowe wywołanie niektórych poleceń. Takie wywołanie umożliwia równoczesne dodanie do mapy trzech nowych par klucz-wartość.
5. Usunięcie pary na podstawie klucza (brak efektu, jeśli klucz nieobecny).
6. **map.size** - pobranie rozmiaru listy (ilości par), w powyższym przykładzie taka linia spowoduje wyświetlenie wartości: 3

**Inicjalizacja mapy wraz z wartościami**

Z powyższego przykładu przekonałeś się, jak utworzyć pusty obiekt typu Map, a dopiero później dodać do niego pary klucz-wartość (przy użyciu metody set). Jak się okazuje możliwe jest również zainicjalizowanie mapy wraz z konkretnymi wartościami - tak, aby wraz z utworzeniem, mogła przechowywać pewne wartości. Chcąc stworzyć mapę, która od razu będzie przechowywała dwie pary klucz - wartość, zapiszemy w następujący sposób:

|  |
| --- |
| let names = new Map([['Kacper', '01/06/1980'], ['Jan', '01/01/1979']]); |

Dzięki utworzeniu obiektu w powyższy sposób, otrzymujemy mapę, która zawiera dwie pary: ‘Kacper’ - ‘01/06/1980’ oraz ‘Jan’ - ‘01/01/1979’.

**Iteracja po mapie**

Chcąc przejść po kolejnych elementach mapy, musimy najpierw rozważyć, czy zależy nam na iteracji po jej kluczach, wartościach, a może wszystkich parach.

Aby zacząć iterować po określonych elementach, musimy dowiedzieć się jak pobierać:

* klucze z mapy: **map.keys()**
* wartości z mapy: **map.values()**
* pary elementów: **map.entries()**

**Przykład**

|  |
| --- |
| let names = new Map([['Kacper', '01/06/1980'], ['Jan', '01/01/1979'], ['Magdalena', '20/04/1960']]);  for(let name of names.keys()){  alert(name); // Kacper, Jan, Magdalena }  for(let date of names.values()){  alert(date); // '01/06/1980',' 01/01/1979', '20/04/1960' }  for(let entry of names.entries()){  alert(entry);   /\* Kacper,01/06/1980  Jan,01/01/1979  Magdalena,20/04/1960  \*/ } |

**ForEach na mapie**

Podobnie jak typ array, mapa dostarcza nam możliwość wywołania na niej forEach i przeprowadzenia określonych operacji.

|  |
| --- |
| let names = new Map([['Kacper', '01/06/1980'], ['Jan', '01/01/1979'], ['Magdalena', '20/04/1960']]);  names.forEach(function(value, key, map){  alert(`${key} ${value}`); }); |

**Set**

W szkoleniu tym, jako ostatni typ złożonych struktur, chciałbym przedstawić Ci **set** (zbiór). Jego zasadniczą różnicą w stosunku do tablic jest to, iż elementy się w nim znajdujące mogą **pojawiać się tylko raz**. Tak więc zbiór służy do trzymania unikalnych wartości, co jak sam zobaczysz, daje dużo możliwości i rozwiązań.

Aby zrozumieć workflow pracy z tym typem struktury, zapoznaj się z poniższym kodem (który omówimy w poniższych podpunktach):

|  |
| --- |
| let pesels = new Set(["00321521927", "62012716982", "81032186861"]); // 1 pesels.add("82120439465"); // 2 pesels.add("82120439465"); // 3 pesels.delete("00321521927"); // 4 pesels.has("81032186861"); // 5 console.log(pesels.size); // 6 pesels.clear(); // 7 |

1. Sposób na utworzenie zbioru wraz z początkowymi wartościami. Elementy, które mają zostać dodane przy inicjalizacji do struktury pesels umieszczamy wewnątrz tablicy.
2. Dodanie nowego elementu do zbioru - za pomocą metody **add()**
3. Dodanie kolejnego elementu do zbioru (próbujemy zrobić duplikat elementu). Z racji, iż w zbiorze znajduje się już taki sam pesel, rozpatrywana linia kodu nie będzie miała żadnego efektu (duplikat nie doda się do zbioru).
4. Wykorzystanie metody **delete()** i usunięcie elementu ze zbioru
5. Sprawdzenie, czy zbiór zawiera określony element - metoda has() zwraca wartość logiczną true (jeśli element znajduje się na liście) lub false (jeżeli jest nieobecny)
6. Pole **size** służy do odczytania rozmiaru zbioru
7. **Clear()** umożliwia wyczyszczenie zbioru ze wszystkich elementów - po wykonaniu tej linii, rozmiar zbioru będzie równy 0.

**Praktyczne wykorzystanie?**

Zastanawiasz się zapewne, gdzie zbiór mógłby znaleźć swoje zastosowanie. Otóż wykorzystanie tego typu danych pozwala przede wszystkim zabezpieczyć się przed nieumyślnym zduplikowaniem elementów, które w domyśle powinny być unikalne. Tak jak zauważyłeś na poprzednim przykładzie - przy próbie dodanie elementów, które już występują w zbiorze, taka operacja jest po prostu ignorowana. Dzięki temu w zbiorze możemy, np przechowywać informacje o peselach użytkowników (wiemy, że pesel jest unikalny dla każdej osoby), czy informacje o identyfikatorach sesji zalogowanych użytkowników (które też są unikalne dla każdego z nich).

**Iteracja po zbiorze**

W razie potrzeby, możemy również bez problemu iterować po elementach danego zbioru. W tym celu możemy wykorzystać znaną Ci już funkcję **forEach** lub **for … of.** Musisz jednak pamiętać o dość kluczowej rzeczy przy pracy ze zbiorami. Otóż jego elementy **nie są indeksowane**. W związku z tym nie jest możliwe odwołanie się do danych elementów zbioru w standardowy sposób - za pomocą indeksów. JavaScript udostępnia kilka sposób na ominięcie tego problemu. Jednym z nich może być zamiana zbioru na tablicę i wówczas pobranie odpowiedniego elementu, np.

|  |
| --- |
| let names = new Set(['Kacper', 'Jan', 'Adrian']); // alert(names[0]); - undefined! alert(Array.from(names)[0]); |

**Wykorzystanie for … of w celu iterowania po elementach zbioru:**

|  |
| --- |
| let names = new Set(['Kacper', 'Jan', 'Adrian']); for(let name of names){  alert(name + "ek");  } // Kacperek, Janek, Adrianek |

**Wykorzystanie forEach**

**UWAGA!**

Funkcja forEach, jak się sam za chwilę przekonasz, w odniesieniu do zbiorów przyjmuje dość nietypowe parametry. Jest to **value**, **valueAgain** oraz **set**. Tę nietypowość najbardziej tworzy parametr **valueAgain** - który jest tak naprawdę duplikatem value. Twórcy JS zdecydowali się na takie rozwiązanie, aby zapewnić możliwość łatwej zamiany zbioru na mapę i vice versa (w przypadku mapy potrzebujemy bowiem dwóch zmiennych - klucz i wartość).

|  |
| --- |
| let names = new Set(['Kacper', 'Jan', 'Adrian']); names.forEach(function(value, valueAgain, set){  alert(value); }); |

**Zamiana listy na zbiór**

Częstą operacją, której wykonanie umożliwi Ci zbiór, jest otrzymanie unikalnych wartości z pewnej listy. Załóżmy, że masz do wykonania następujące zadanie:

Zlicz, ile różnych imion występuje w podanej tablicy:

[“Kacper”, “Jeremiasz”, “Mateusz”, “Kacper”, “Krzysztof”, “Grzegorz”, “Kinga”, “Magdalena”, “Kinga”]

Odpowiedzią dla powyższego zadania dla 9-cio elementowej tablicy, jest wartość równa: 7. Dzieje się tak, ponieważ imiona takie jak: “Kacper” oraz “Kinga” występują dwukrotnie.

Pierwsze rozwiązanie, jakie przychodzi do głowy, to iteracja po każdym elemencie w tablicy i sprawdzanie, czy dana zmienna pojawiła się jak dotąd na liście.   
  
Mogę Ci jednak powiedzieć już teraz, że nie jest to najlepsze rozwiązanie. O wiele lepiej będzie zamienić taką tablicę na zbiór elementów. Wystarczy dokonać konwersji, a duplikaty zostaną automatycznie usunięte.

|  |
| --- |
| let names = ["Kacper", "Jeremiasz", "Mateusz", "Kacper", "Krzysztof", "Grzegorz", "Kinga", "Magdalena, "Kinga”] let filtered = new Set(names); |

**Podsumowanie**

Wyżej przedstawiłem Ci najważniejsze możliwości, jakie dostarczają złożone struktury danych. Nieraz będziesz potrzebował wykroczyć poza obszar przedstawionej tutaj wiedzy i zaprogramować szersze funkcjonalności. Wówczas przydatne mogą okazać się zewnętrzne źródła jak oficjalna dokumentacja czy StackOverflow:

* [Więcej informacji o Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map)
* [Więcej informacji o Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)