

JavaScript

podstawy

v 2.1

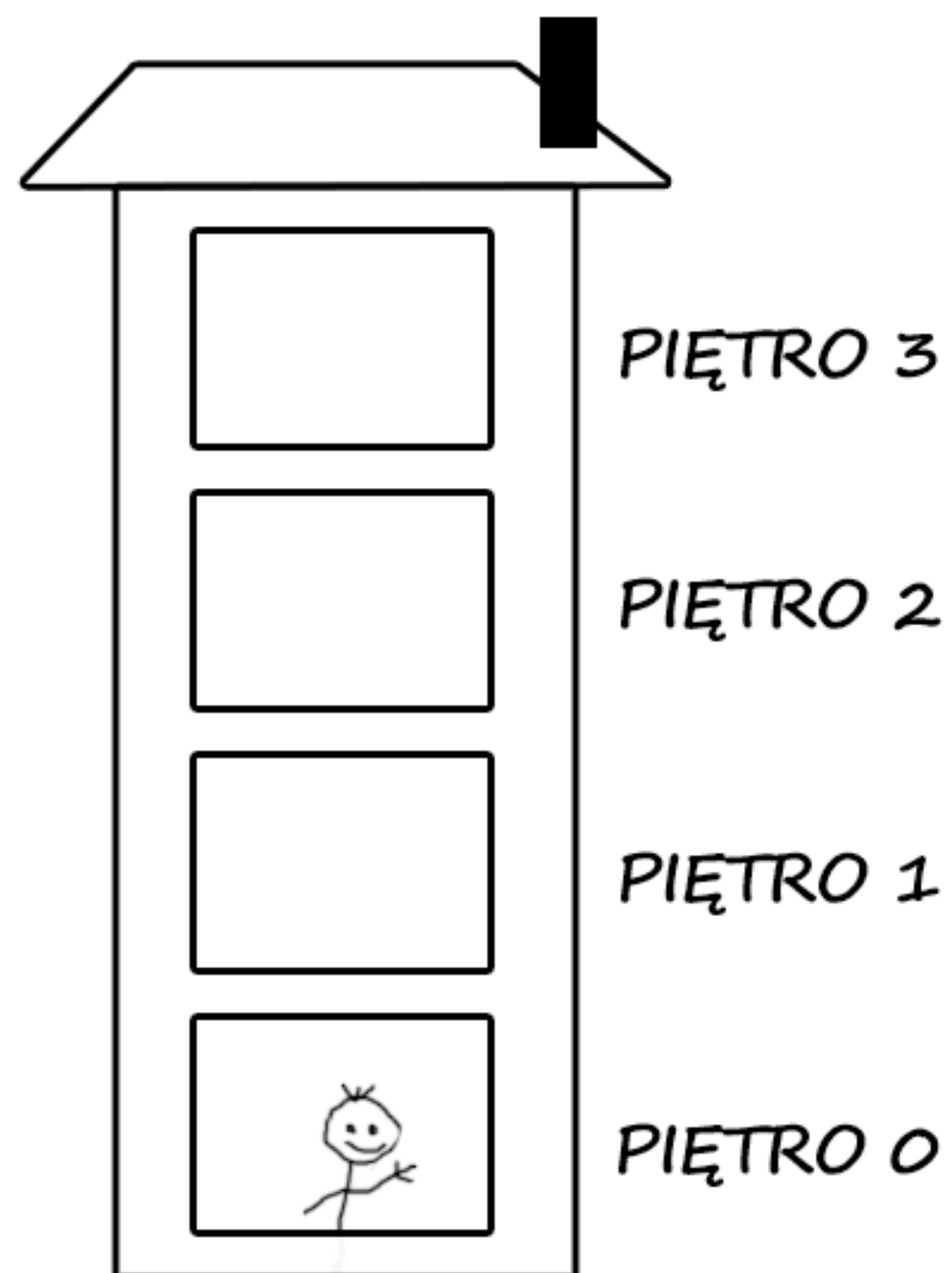
PLAN

- Tablice
- Tablice wielowymiarowe
- Obiekty
- Funkcje czasu
- Funkcje – tematy zaawansowane
- Tablice – metody

Tablice

Tablice

Jak dowiedzieć się, kto mieszka w tym bloku na drugim piętrze?



```
var users = ["Ala", "John"];
```

```
var prices = [23, 12, 9.40];
```

Elementy w tablicy umieszczamy w nawiasach kwadratowych i rozdzielamy przecinkiem.

Aby pobrać element tablicy, musimy podać jej nazwę oraz indeks.

```
users[0]; // wypisze: "Ala"
```

```
users[1]; // wypisze: "John"
```

```
users[2]; // wypisze: undefined
```

Tablice

Typy danych

Tablice mogą przechowywać różne typy danych:

- liczby,
- stringi,
- typy specjalne,
- wartości logiczne - boolean,
- obiekty,
- funkcje,
- inne tablice.

```
var mixTypes = ["Ala",  
                23,  
                true,  
                { name: "Ala"},  
                function() { return 2; },  
                null  
];
```

Indeksy (klucze) tablic rozpoczynają się od 0.

```
mixTypes[0]; // wypisze "Ala"  
mixTypes[1]; // wypisze 23  
mixTypes[2]; // wypisze true  
mixTypes[3].name; // wypisze "Ala"  
mixTypes[4](); // wypisze 2  
mixTypes[5]; // wypisze null
```

Aby pobrać wielkość tablicy, korzystamy z atrybutu **length**:

```
mixTypes.length; // 6
```

Na końcu prezentacji znajdziesz najpopularniejsze metody dla tablic. Zapoznaj się z nimi samodzielnie.

Tablice - metody



Czas na zadania

Wykonaj zadania z
części Tablice

Tablice wielowymiarowe

Tworzenie tablic wielowymiarowych

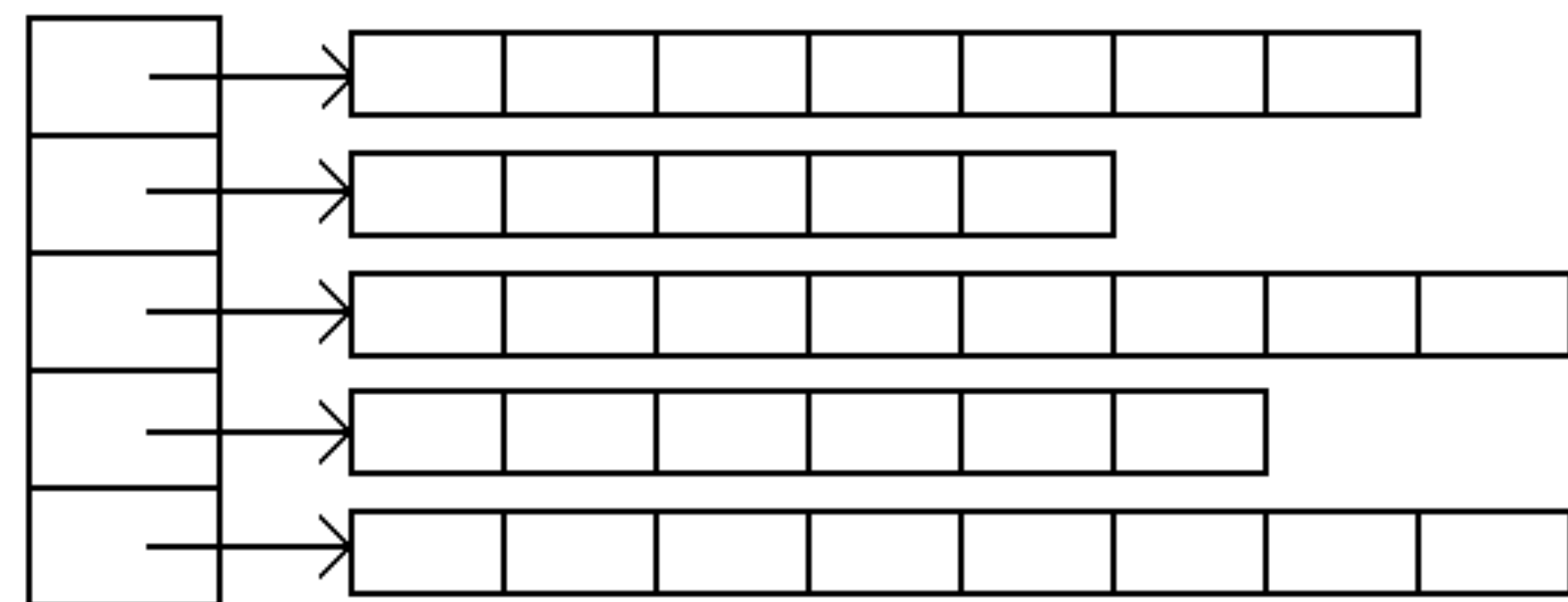
- Jeżeli chcemy stworzyć tablicę wielowymiarową, musimy najpierw stworzyć tablicę główną, a następnie w każdej z jej komórek umieścić pustą tablicę.
- Tablice wielowymiarowe z tego powodu nazywa się też **tablicami tablic**.

```
var array2D = [ ];  
array2D[0] = [ ];  
array2D[1] = [ ];  
array2D[2] = [ ];  
array2D[3] = [ ];
```

Tablice wielowymiarowe

- Nic nie stoi na przeszkodzie, żeby w komórce tablicy trzymać inną tablicę.
- Dostajemy wtedy **tablicę wielowymiarową**.

```
var array2D = [  
  [1, 2, 3, 4],  
  ["Ala", "Adam", "Kasia"],  
  [true, false],  
];
```



Korzystanie z tablic wielowymiarowych

- Jeżeli chcemy dostać się do jakiejś komórki z tablicy wielowymiarowej, musimy podać wszystkie wymiary tej komórki.
- Każdy wymiar musi znajdować się w osobnych nawiasach kwadratowych!

```
var array2Dnew = [];  
array2Dnew[0] = [1, 2, 3, 4, 5];  
array2Dnew[1] = ["Ala", "Adam"];  
array2Dnew[2] = ["Wojtek", "Kasia"];  
array2Dnew[3] = [3, 4, 5, 6];
```

```
array2Dnew[0][4]; // Zwróci 5  
array2Dnew[1][1]; // Zwróci "Adam"  
array2Dnew[2][0]; // Zwróci "Wojtek"  
array2Dnew[3][3]; // Zwróci 6  
array2Dnew[3][4]; // Zwróci undefined
```



Czas na zadania

Wykonaj zadania z
części Tablice
wielowymiarowe

Obiekty

Obiekty

Obiekty w informatyce mają jednak większe znaczenie. Dla ułatwienia pisania dużych programów używamy obiektów do **enkapsulacji**.

Enkapsulacja – oznacza silne oddzielenie wewnętrznego rozwiązania problemu od późniejszego użycia takiego rozwiązania.

Obiekty możemy stworzyć na dwa sposoby:

- używając nawiasów klamrowych (tworzymy nowy obiekt),
- używając słowa kluczowego **new** i **konstruktora**.

Obiekty

Czym są obiekty?

Obiekt ma własności np.:

- name,
- age.

Można do własności przypisywać wszystkie typy danych np.

- własność friends ma przypisaną **tablicę**,
- a własność isSleepy – **Boolean**.

Możemy również tworzyć obiekty w obiektach.

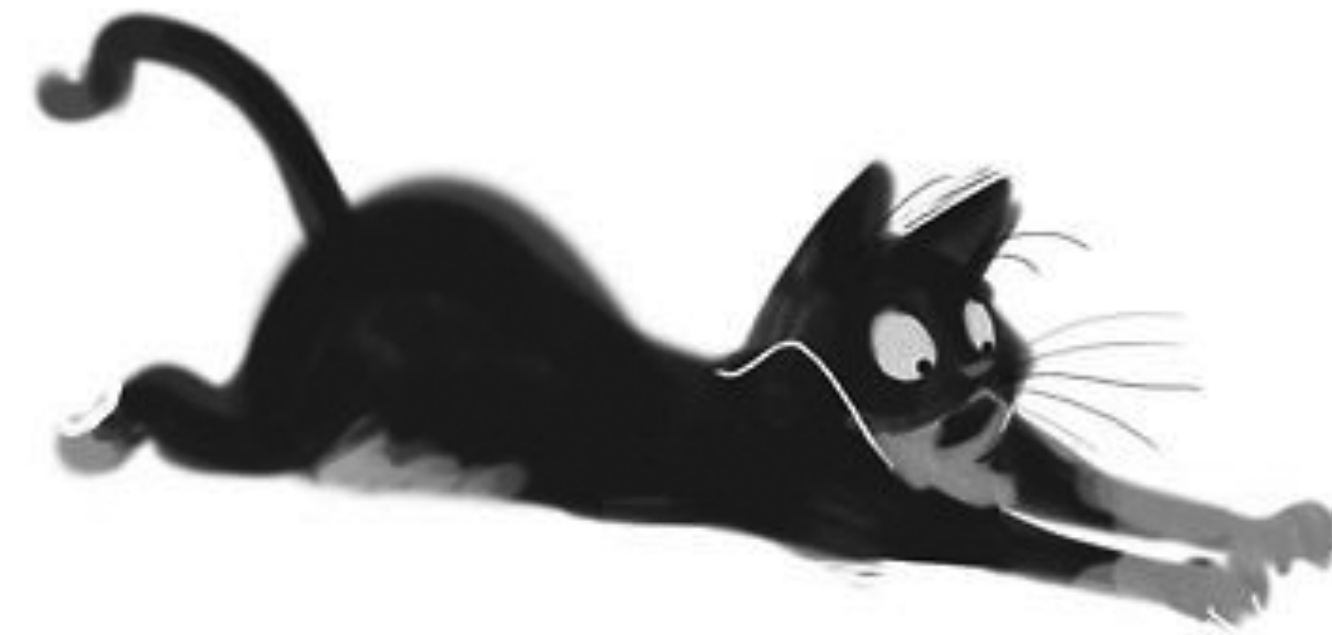
```
var cat = {  
  name: "Filemon",  
  age: 2,  
  friends: ["Mruczek", "Reksio"],  
  isSleepy: true  
}
```



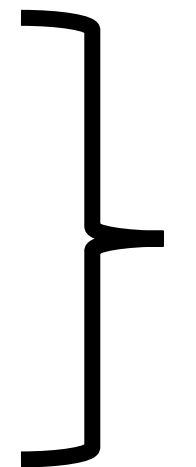
Obiekty

Jak się dostać do pól obiektu?

Aby dostać się do wartości w obiekcie używamy znaku kropki.



```
cat.name // wypisze: Filemon  
cat.age  // wypisze: 2  
cat.friends[1] // wypisze: Reksio  
cat.isSleepy // wypisze: true
```



Atrybuty

- Każdy obiekt ma swój wewnętrzny stan, w którym się znajduje. Stan ten opisany jest przez **atrybuty**.

```
var teacher = {  
  name: "Janusz",  
  surname: "Kowalski",  
  subject: "Programowanie JS",  
  teach: function() { ... }  
};
```

- Atrybuty to zmienne przypisane do obiektu. Możemy się do nich dostać w następujący sposób:

`nazwa_obiektu.nazwa_atrybutu`

`teacher.name` // "Janusz"

Metody

- Obiekt może też mieć w sobie funkcje, które nazywa **metodami obiektu**.
- Możemy je uruchomić używając składni:
`nazwa_obiektu.nazwa_metody()`

Na przykład:

`teacher.teach()`

Słowo kluczowe this

- W metodach mamy dostęp do naszego obiektu. Jest on reprezentowany przez specjalną zmienną **this**.
- Dzięki niej możemy odnieść się do stanu obiektu bez konieczności używania nazwy zmiennej, w której zapisany jest nasz obiekt.
- Jest to też bardzo przydatne, gdy mamy wiele obiektów mających tę samą metodę.

```
var teacher = {  
  name: "Janusz",  
  surname: "Kowalski",  
  subject: "Programowanie JS",  
  teach: function(){  
    console.log(this.name);  
    console.log(this.surname);  
    console.log(this.subject);  
  }  
};
```

Dodawanie metod i atrybutów

- Do obiektów możemy dodawać atrybuty i metody w czasie trwania programu.
- Dodajemy je przez przypisanie do danego obiektu (przez co obiekty mogą się od siebie bardzo różnić).

```
var teacher = {  
  name: "Janusz",  
  surname: "Kowalski",  
  subject: "Programowanie JS"  
};  
  
console.log(teacher.students); //undefined  
teacher.students = ["Ala", "Kasia", "Adam"];  
console.log(teacher.students); // Array [...]
```

Konstruktor

- Obiekt możemy też stworzyć dzięki tak zwanym **konstruktorom**.
- Są to specjalne funkcje służące do stworzenia obiektu i nastawienia mu początkowego stanu.
- Powinny to być funkcje, których nazwa zaczyna się dużą literą.
- Do nastawiania stanu w konstruktorze powinniśmy używać słowa kluczowego **this**.
- Aby potem stworzyć obiekt na bazie konstruktora, powinniśmy użyć słowa kluczowego **new**.

```
var Car = function(type, hp, color) {  
    this.type = type;  
    this.hp = hp;  
    this.color = color;  
};  
  
var fiat = new Car("fiat", 125, "blue");  
  
console.log(fiat.type);  
console.log(fiat.hp);  
console.log(fiat.color);
```

Prototypy

- W języku JavaScript obiektowość jest zaimplementowana dzięki zasadzie prototypów. W innych językach jest stosowana klasowość np. w PHP.
 - Idea ta mówi, że każdy obiekt ma swój prototyp, od którego dostaje wszystkie jego metody i atrybuty.
- Dzięki połączeniu prototypów i konstruktorów możemy łatwo tworzyć podobne do siebie obiekty.
 - Prototypy dopisujemy bezpośrednio do konstruktora.

Prototype

```
var Car = function(type, hp, color) {  
    this.hp = hp;  
    this.type = type;  
    this.color = color;  
    this.km = 0;  
};  
  
Car.prototype.drive = function(km){  
    console.log(this.color + " " + this.type + "  
    drives for " + km + " km");  
    this.km += km;  
}
```

```
var mercedes = new Car("Mercedes", 120,  
    "Czarny");  
var trabant = new Car("Trabant", 40, "Szary");  
  
trabant.drive(10);  
    // Wypisze: Szary Trabant drives for 10km.  
  
mercedes.drive(10);  
    // Wypisze: Czarny Mercedes drives for 10km.
```



Czas na zadania

Wykonaj zadania z
części Obiekty

Funkcje czasu

setTimeout

- Wywołuje podaną funkcję po podanym czasie (czas podajemy w milisekundach).
- Funkcja ta zwraca unikatowy numer identyfikujący ustawiony przez nas timer (ID).

```
var timeout = setTimeout(function () {  
    console.log('I will be invoke in 5s');  
}, 5000); // 5s
```

clearTimeout

- **clearTimeout** czyści **timeout** ustawiony przez funkcję **setTimeout()**.
- Do tej funkcji musicie podać ID timera, który chcecie usunąć.

```
var timeout = setTimeout(function () {  
    console.log('I will be invoke in 5s');  
}, 5000); // 5s  
  
clearTimeout(timeout);
```

setInterval


- Uruchamia podaną funkcję co podany przedział czasu (czas podajemy w milisekundach).
- Funkcja ta zwraca unikatowy numer identyfikujący nastawiony przez nas **interval**.

```
var interval = setInterval(function () {  
    console.log('I will be invoke every 5s');  
}, 5000); // 5s
```

clearInterval

- **clearInterval** czyści interval ustawiony przy pomocy **setInterval()**.
- Do tej funkcji trzeba podać **ID** intervala, który chcecie usunąć.

```
var interval = setInterval(function () {  
    console.log('I will be invoke every 5s');  
}, 5000); // 5s  
  
clearInterval(interval);
```



Funkcje - tematy zaawansowane

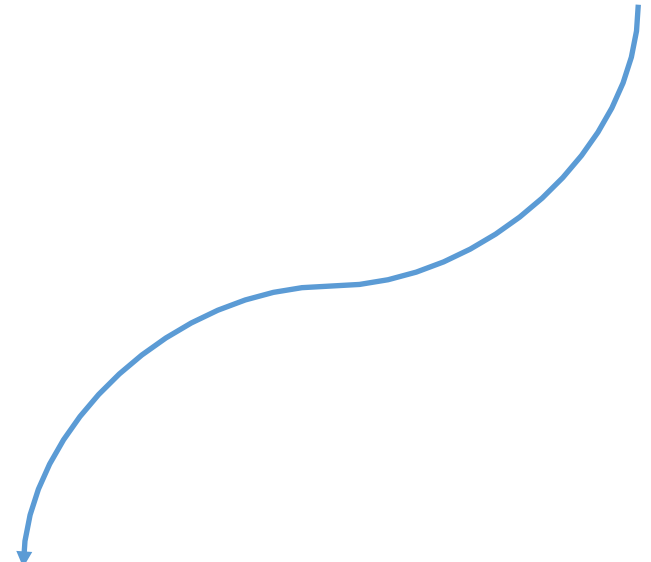
Argumenty funkcji

Do funkcji możemy przekazywać więcej niż jeden argument.

```
function sumSquare(a, b) {  
    return a*a + b*b;  
}  
sumSquare(2,3);
```

Jeżeli nie wiemy, ile argumentów będzie potrzebnych w funkcji, możemy użyć tablicy **arguments**.

```
function test() {  
    console.log(arguments);  
}  
test(2,3, "Ala");
```



Przenoszenie instrukcji

- Silnik JavaScriptu wykonuje instrukcję kod krok po kroku, zaczyna od pierwszej linii, kończy na ostatniej.
- Zdarza się jednak, że czasem przenosi pewne instrukcje na samą górę.



Deklaracja została przeniesiona, zanim program wystartował.

```
1. function zrobHerbate() { // ciało funkcji }  
2. zrobHerbate(); // ok  
3.  
4. function zrobHerbate() {  
5.   console.log("Nalej wodę do czajnika")  
6.   console.log("Wsyp do szklanki herbatę")  
7.   console.log("Zagotuj wodę")  
8.   console.log("Zalej herbatę" )  
9. }
```


Przenoszenie instrukcji

Wyrażenia funkcyjne nie są przenoszone, tylko zmienne, do których zostało przypisane.



W przypadku wyrażenia funkcyjnego musimy pamiętać, gdzie wywołujemy funkcję

1. **var** herbata;

2. herbata(); //błąd

3.

4. **var** herbata = **function** zrobHerbate()

5. console.log("Nalej wodę do czajnika");

6. console.log("Wsyp do szklanki herbatę");

7. console.log("Zagotuj wodę");

8. console.log("Zalej herbatę");

9. }

Funkcje wyższego rzędu

Przykład

- Bardzo często zdarza się, że do jakiejś funkcji przekazujemy drugą funkcję jako argument.
 - Takie funkcje nazywamy funkcjami wyższego rzędu (higher-order functions).
 - Choć często korzystamy z takich funkcji, to rzadko je piszemy.
- Mamy funkcję, która sortuje nam tablice od największego elementu do najmniejszego.
 - Musimy jednak używać innego sposobu porównywania do napisów, a innego do liczb.
 - Przykład ten możecie znaleźć w katalogu z ćwiczeniami.

Zmienne lokalne i globalne

W JavaScriptcie występują dwa typy zmiennych:

- **globalne** – są to zmienne, które nie są zadeklarowane w żadnym zakresie (na razie możecie uznać, że zakres to funkcja) albo są zadeklarowane **bez** słowa kluczowego **var**,
- **lokalne** – są to zmienne zadeklarowane w środku jakiejś funkcji przy pomocy słowa kluczowego **var**.

Zmienne globalne są bardzo niebezpieczne i nie powinno się ich używać w funkcjach!

- **Zmienne globalne** są widoczne w całym naszym programie i są niszczone dopiero podczas zamknięcia okna przeglądarki.
- **Zmienne lokalne** są widoczne tylko i wyłącznie w zakresie funkcji, w której zostały stworzone. Są niszczone w chwili, w której ta funkcja się kończy.

Zmienne lokalne i globalne

```
function sayGlobalName() {  
  console.log(name);  
}
```

W funkcji nie ma zmiennej **name**.
Brana jest pod uwagę zmienna globalna.

```
function sayLocalName() {  
  var name = "Janek"; // Zmienna lokalna  
  console.log(name);  
}
```

Wyświetlona zostanie zmienna lokalna.

```
var name = "Adam"; // Zmienna globalna name
```

```
sayGlobalName(); // Adam  
sayLocalName(); // Janek  
console.log(name); // Adam
```

Call stack

- Call stack to tak zwany „stos wywołań”. Trzyma on dokładne informacje na temat tego, w którym miejscu znajduje się nasz program w danej chwili.
- Funkcja, która jest właśnie uruchomiona, jest na samej górze tego stosu. Kiedy komputer natrafia na słowo kluczowe **return** usuwa najwyższą funkcję.
- Kiedy wywoływana jest jakaś funkcja, cały stan (tak zwany kontekst) zawierający wartości zmiennych musi zostać zapamiętany.
- Kiedy funkcja się kończy, wszystkie jej zmienne są zapominane, a program „wczytuje” ze stosu informacje potrzebne do przywrócenia programu w miejscu, w którym funkcja została uruchomiona.
- Proces ten zajmuje zarówno pamięć, jak i czas procesora, powinno się zatem unikać wielokrotnych zagnieżdżeń funkcji.

Call stack

```
function foo() {  
  var i = 0;  
  bar(i);  
  return i;  
}  
  
function bar(i) {  
  return i + 5; //check context  
}  
  
function basic() {  
  var j = foo();  
}  
  
basic();
```

Call stack:
JavaScript context
basic
foo
bar
bar
foo
basic
JavaScript context

Zakresy zagnieżdżone

- Zmienne lokalne mają zakres, w którym są widoczne (**variable scope**).
 - Do tej pory omówiliśmy zakres **globalny** (zmienne widoczne wszędzie) i **lokalny** (zmienne widoczne tylko w danej funkcji).
 - W języku JavaScript istnieje wiele poziomów zakresu lokalnego. Zjawisko to określa się poprzez termin zakresy zagnieżdżone (**nested scopes**).
- Zagnieżdżanie zakresów polega na tym, że jeżeli w jakiejś funkcji zdefiniujemy drugą funkcję, to wtedy tworzy ona własny zakres lokalny.
 - Funkcja zagnieżdżona ma jednak dostęp do wszystkich zmiennych lokalnych zakresów, w których jest osadzona.
 - Wartości tych zmiennych są brane na chwilę użycia funkcji zagnieżdżonej.

Zakresy zagnieżdżone

```
var fooOutside = function() {  
  var name = "Jacek";  
  
  var foo = function() {  
    var surname = "Kowalski";  
  
    console.log(hello + name + surname);  
  }  
  
  foo(); // Witaj Jacek Kowalski  
  name = "Wojtek";  
  foo(); // Witaj Wojtek Kowalski  
}
```

```
var hello = "Witaj"  
fooOutside();
```

Zakres Globalny
Dostęp do zmiennych:
globalnych (np. zmiennej **hello**)

Zakres lokalny 1 (dla funkcji **fooOutside**)
Dostęp do zmiennych:
globalnych i lokalnych dla **fooOutside**

Zakres lokalny 2 (dla funkcji **foo**)
Dostęp do zmiennych:
globalnych, lokalnych dla **fooOutside**
i lokalnych dla **foo**



Czas na zadania

Wykonaj zadania z
części Zaawansowane
funkcje

Tablice metody

Tablice - metody

Mutacyjne

Modyfikujące oryginalną tablicę

arr – to zmienna, która jest tablicą

Dostępowe

arr.pop – usuń i zwróć ostatni element tablicy

arr.push – dodaj element do końca tablicy

arr.reverse – odwróć całą tablicę

arr.shift – usuń i zwróć pierwszy element tablicy

arr.sort – posortuj elementy na podstawie przekazanej funkcji

arr.splice – usuń (ew. zamień) i zwróć kawałek tablicy

arr.unshift – dodaj element na początek tablicy

arr.concat – połącz dwie tablice

arr.join – połącz wszystkie elementy tablicy w ciąg znaków, użyj przekazanego argumentu

arr.slice – zwróć kawałek tablicy

arr.indexOf – pozycja szukanego elementu

arr.lastIndexOf – ostatnia pozycja szukanego elementu

Tablice – metody

Iteracyjne

Są to funkcje **wyższego rzędu**,
czyli przyjmujące inną funkcję jako argument.

arr.forEach – wywołaj funkcję dla każdego z elementów,
arr.every – sprawdź, czy wszystkie elementy spełniają dany warunek
arr.some – sprawdź, czy jakikolwiek element spełnia dany warunek
arr.filter – wywołaj funkcję dla każdego z elementów, zwróć nową tablicę zawierającą tylko te elementy które go spełniły
arr.map – wywołaj funkcję dla każdego z elementów, zwróć nową tablicę ze zmodyfikowanymi elementami

Metody mutacyjne

pop()

```
var foo = [1, 2, 3, 4];  
var lastElem = foo.pop();  
console.log(foo); // [ 1, 2, 3]
```

push()

```
var foo = [1, 2, 3];  
foo.push(12);  
console.log(foo); // [ 1, 2, 3, 12]
```

reverse()

```
var foo = [1, 2, 3];  
foo.reverse();  
console.log(foo); // [ 3, 2, 1]
```

shift()

```
var foo = [1, 2, 3, 12];  
var firstElem = foo.shift();  
console.log(foo); // [ 2, 3, 12]
```

unshift()

```
var foo = [2, 3, 12];  
foo.unshift(5);  
console.log(foo); // [ 5, 2, 3, 12]
```

Metody mutacyjne

splice([index początkowy], liczbaElementów, elementy do wstawienia)

```
var foo = [1, 2, 3];           //usuń pierwszy element  
foo.splice(0, 1);             //0 to indeks, 1 to ilość elem.  
console.log(foo);             // [2, 3]
```

```
var foo = [2, 3];              //usuń ostatni element  
foo.splice(-1);  
console.log(foo);             // [2]
```

```
var foo = [1, 2, 3, 4];  
foo.splice(2, 1, 24, "kot");  
console.log(foo);             // [1, 2, 24, "kot", 4]
```

Zaczniij od indeksu 2, usuń jeden element
i **wstaw** liczbę 24 oraz string "kot".

Metody dostępne

sort()

```
var foo = [23, 2, 3, 1, 34, 8];  
var baz = foo.sort();  
console.log(baz); // [1, 2, 23, 3, 34, 8]
```

Jak widzisz tablica jest posortowana, ale z wykorzystaniem porządku leksykograficznego tzn. cyfry są porządkowane jako ciągi Od ostatniej do pierwszej. Litery alfabetu występują po cyfrach.

Przykład:

[1, a, 5, 11, 7, c] taka tablica zostanie uporządkowana w następujący sposób:
[1, 11, 5, 7, a, c]

sort(func)

```
var foo = [23, 2, 3, 1, 34, 8];  
var baz = foo.sort(function(a, b) {  
    return a - b;  
});  
console.log(baz); // [1, 2, 23, 3, 34, 8]
```

Aby zaradzić temu problemowi wystarczy do funkcji sort przekazać parametr w postaci funkcji anonimowej, która sortuje cyfry za pomocą ich porównywania.

Metody dostępne

concat()

```
var foo = [1, 2, 3];  
var bar = [5, 6];  
var baz = foo.concat(bar);  
console.log(baz); // [ 1, 2, 3, 5, 6]
```

join()

```
var foo = ["wsiaść", "do", "pociągu"];  
var text = foo.join();  
console.log(text); // wsiaść,do,pociągu
```

```
var foo = ["wsiaść", "do", "pociągu"];  
var text = foo.join("+");  
console.log(text); // wsiaść+do+pociągu
```


Metody dostępne

slice()

```
var foo = [1, 2, 3];  
var restFoo = foo.slice(0, 2);  
console.log(restFoo); // [ 1, 2]
```

indexOf()

```
var foo = [1, 2, 3];  
var index = foo.indexOf(2);  
console.log(index); // 1
```

lastIndexOf()

```
var foo = [1, 2, 3, 1, 3, 3];  
var index = foo.lastIndexOf(1);  
console.log(index); // 3
```



Zwróć dwa elementy,
zacznij od indeksu 0.

Tablice – metody iteracyjne

forEach()

```
var foo = [1, 2, 3];  
foo.forEach(function(element, index, array) {  
    console.log("Element" + element);  
});
```

some()

```
var foo = [1, 2, 3];  
foo.some(function(element, index, array) {  
    return element % 2 !== 0;  
});
```

Sprawdź, czy **jakikolwiek** element jest nieparzysty, zwraca wartość boolean true lub false.

every()

```
var foo = [1, 2, 3];  
foo.every(function(element, index, array) {  
    return element % 2 === 0;  
});
```

Sprawdź, czy **wszystkie** elementy są parzyste, zwraca wartość boolean true lub false.

Tablice – metody iteracyjne

filter()

```
var foo = [1, 2, 3, 4];  
var bar = foo.filter(function(element, index,  
array) {  
    return element % 2 === 0;  
});  
console.log(bar); // [2, 4]
```

Znajdź **tylko** elementy parzyste.

map()

```
var foo = [1, 2, 3, 4];  
var bar = foo.map(function(element, index,  
array) {  
    return element * 2;  
});  
console.log(bar); // [2, 4, 6, 8]
```

Pomnóż elementy przez dwa.