

JavaScript

podstawy

v 1.5

PLAN

- [Wstęp](#)
- [Komentarze i zmienne](#)
- [Chrome developer tools](#)
- [Typy danych](#)
- [Instrukcje warunkowe](#)
- [Pętle](#)
- [Operatory](#)
- [Funkcje](#)
- [Tablice](#)
- [Tablice wielowymiarowe](#)
- [Obiekty](#)
- [Funkcje – tematy zaawansowane](#)
- [Funkcje czasu](#)
- [String – metody](#)
- [Tablice - metody](#)

Wstęp

JavaScript

- Dynamiczny język programowania powstały w **1995 roku**.
- **Używany przede wszystkim do interakcji z użytkownikiem.**
- Wykorzystuje silniki zaimportowane w przeglądarkach internetowych.
- Od kilku lat jest możliwy do korzystania jako **język serwerowy (NodeJS)**.
- Aktualnie (częściowo) implementowana jest wersja **ES6/2015** (trwają też prace nad ES7).
- Istnieją języki kompilowane do **JavaScript (CoffeeScript, TypeScript)**.

Pierwszy program w JavaScriptie

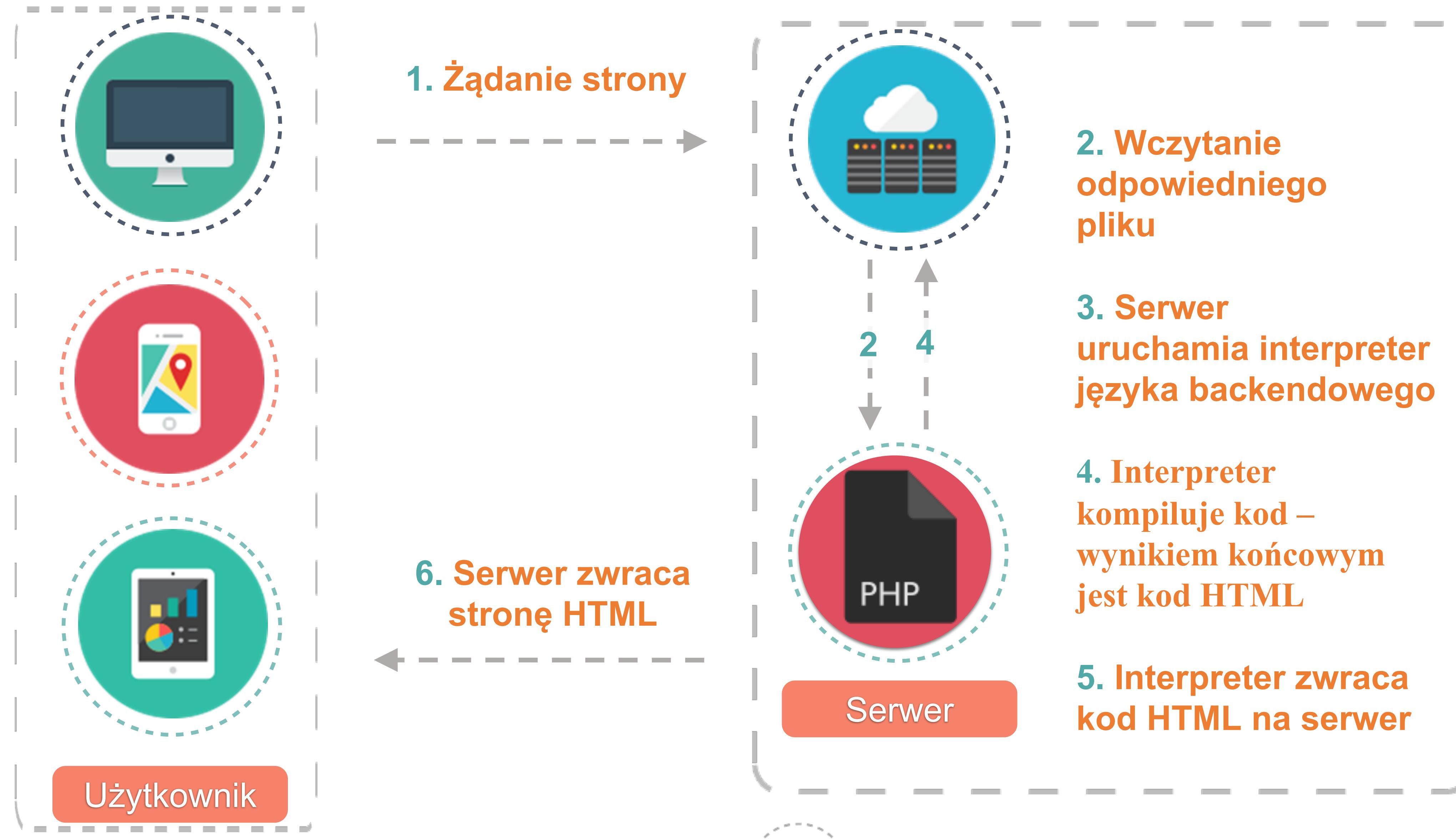
Plik HTML

```
<!doctype html>  
<html>  
  <head>  
    <title>Coders Lab</title>  
  </head>  
  <body>  
    <script src='app.js'></script>  
  </body>  
</html>
```

Plik JavaScript

```
console.log('Hello World!');
```

Jak działa serwer?



Gdzie jest w tym wszystkim JavaScript?

- JavaScript jest używany w dwóch celach:
 - jako język backendowy (np. nodejs),
 - jako język komplikowany po stronie przeglądarki.



Jak działa JavaScript?

- JavaScript działa dzięki silnikom wbudowanym w przeglądarki internetowe.
- Silniki te wczytują kod JS podpięty pod stronę HTML i uruchamiają go na komputerze użytkownika.
- Silnik przeglądarki może też nasłuchiwać odpowiednich czynności wykonanych przez użytkownika.
- Na przykład po ruchu myszą czy kliknięciu silnik uruchamia wskazany przez programistę kawałek kodu.

Komentarze i zmienne

Komentarze

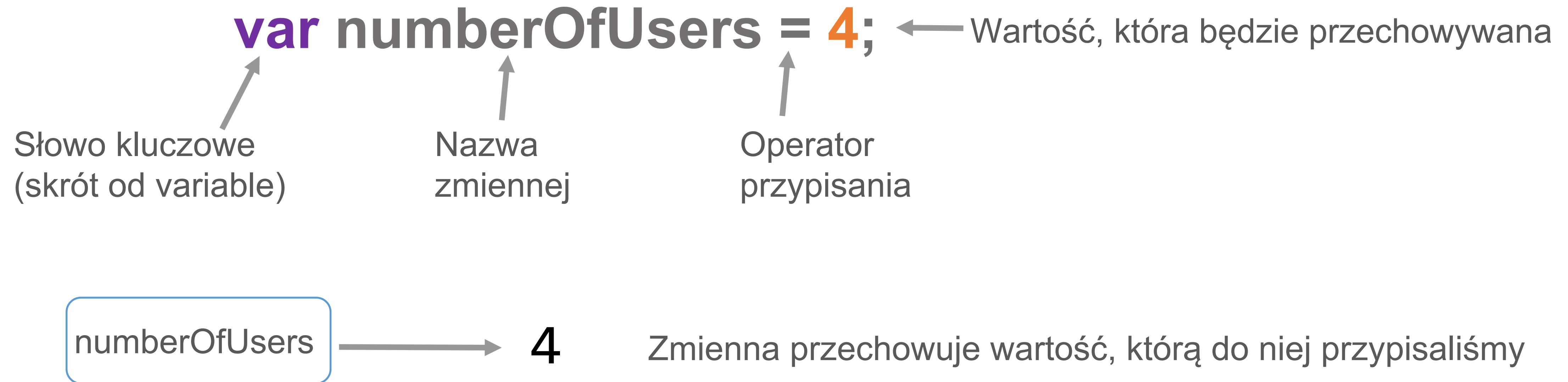
Komentarze

// Komentarz mieszczący się w jednej linii

```
/*
Komentarz, który wykorzystuje więcej
niż jedną linię
*/
```

Zmienne

Zmiennych używamy do przechowywania danych i zarządzania nimi.



var

- Jeżeli definiujemy zmienną, powinniśmy zawsze pamiętać o słowie kluczowym **var**.
- Dzięki temu nasz kod jest czytelny oraz unikamy nieprzewidywalnych zachowań silnika javascriptowego (zmiennych globalnych).





Chrome developer tools

Chrome developer tools

Uruchamiamy:

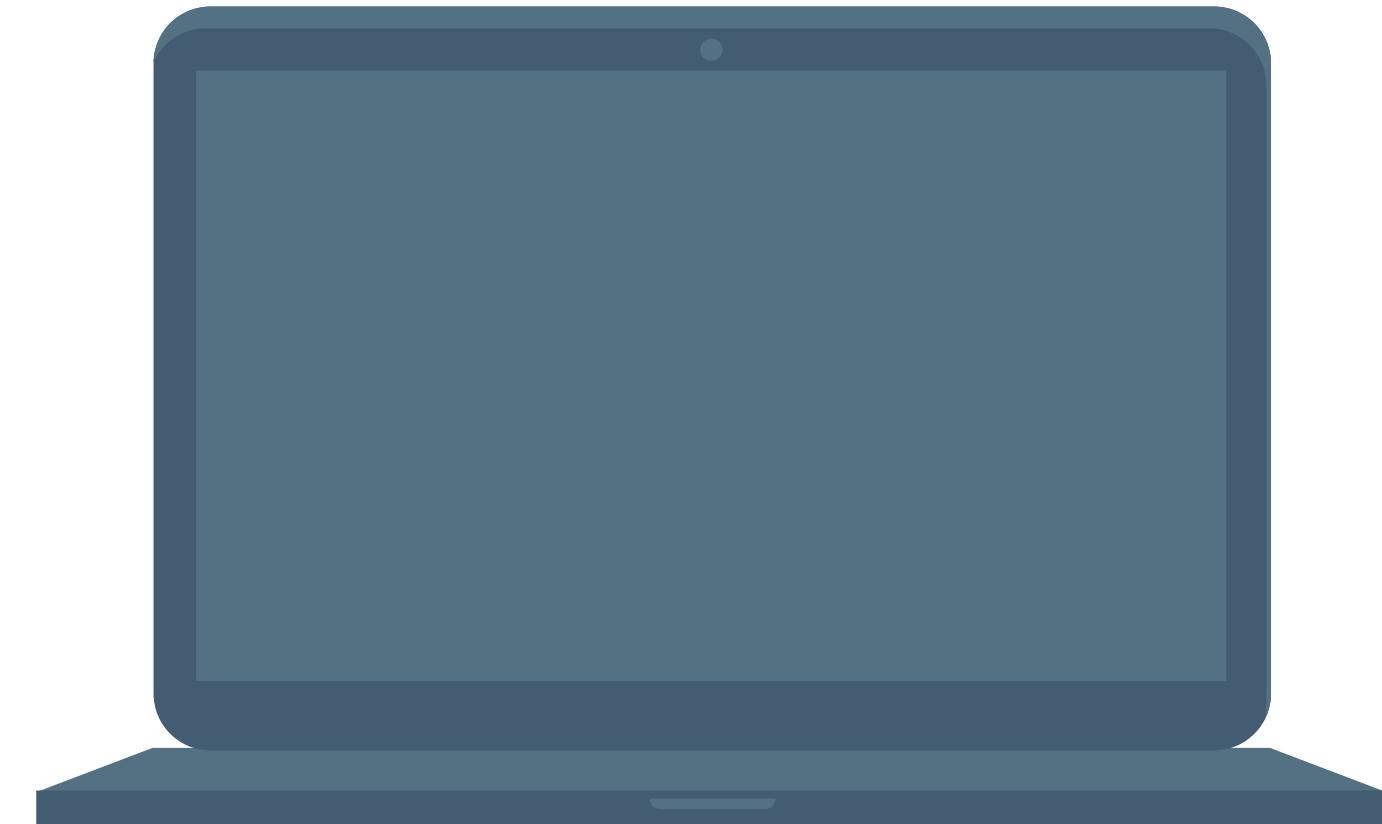
- **Na Windows:** Crtl + Shift + I
- **Na OS X:** Cmd + Opt + I

Narzędzie domyślnie zainstalowane w każdej przeglądarce Chrome.

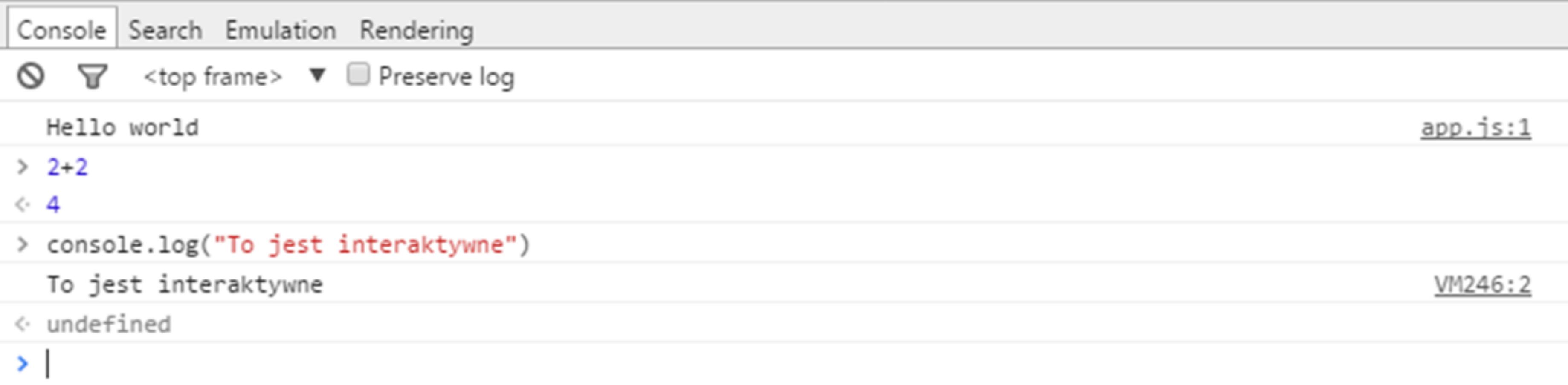
Więcej informacji o tym, jak używać tego narzędzia:

<http://developer.chrome.com/devtools>

Podobne narzędzie znajdziemy w przeglądarce Firefox i Safari



Chrome developer tools



The screenshot shows the Chrome Developer Tools interface with the 'Console' tab selected. The console output is as follows:

```
Console Search Emulation Rendering
  <top frame> ▾  Preserve log
Hello world                                         app.js:1
> 2+2
< 4
> console.log("To jest interaktywne")
To jest interaktywne                               VM246:2
< undefined
> |
```

A red dashed horizontal line is drawn below the VM246:2 entry.

Narzędzie ma interaktywną konsolę JavaScript.

Możemy w niej testować zachowanie naszych skryptów,
używać do debugowania i kontrolowania przepływu programu
(prostej interakcji z użytkownikiem).

Co w razie błędu?

- Jeśli w naszym skrypcie jest błąd składniowy, **skrypt będzie wykonywany**, aż natrafi na ten błąd!
- W przypadku błędu podane zostaną następujące informacje:
 - typ błędu,
 - plik, w którym ten błąd wystąpił,
 - linia zawierająca błąd.

Ta linia kodu
została wykonana

W tej linii jest błąd

```
> console.log("Hello world");
      unknown_function();
Hello world
VM84:2
✖ ➤ Uncaught ReferenceError: unknown_function is not defined(...)
VM84:3
>
```

Typy danych

Typy danych

Liczby(Number)

```
var liczba = 10;  
var liczba2 = 2.2;
```

Wartości logiczne (Boolean)

```
var prawda = true;  
var falsz = false;
```

Ciągi znaków (String)

```
var text = "Ala ma kota";  
var text2 = "2.2";
```

Specjalne

```
var foo = null;  
var bar = undefined;
```

Obiekty

```
var kot = {  
    imie: "Mruczek",  
    wiek: 3  
}
```

Tablice

```
var tab1 = [1, 2, "Ala"];  
var tab2 = [1, [2], 45];
```

Prymitywne typy danych

18

18

Sprawdzanie typu – typeof

Za pomocą **typeof** możemy sprawdzać typ danych.

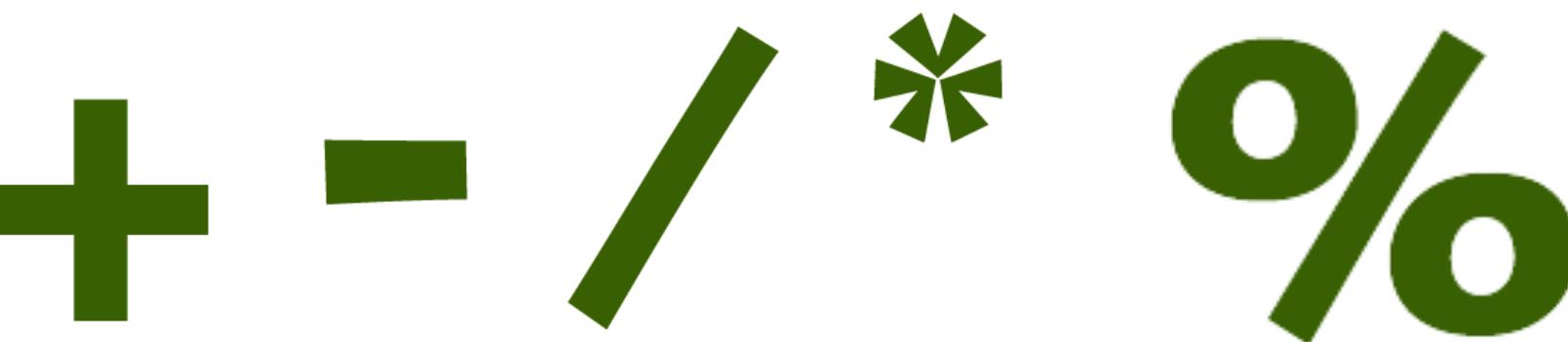
- **typeof null;** object 
- **typeof 2;** number
- **typeof "Ala ma kota";** string

Sprawdź inne rodzaje typów danych...

Liczby

- `var result = 9;`
- `var length = 102.52;`
- `var numberOfboxes = 20;`

Na liczbach możemy wykonywać działania matematyczne.



Modulo
(reszta z dzielenia)

Stringi

Stringi to ciągi znaków.

```
var text = "Ala ma kota";
```

```
var name = "John";
```

```
var text2 = "20";
```

Stringi możemy dodawać jest to tzw.
konkatenacja, czyli łączeniełańcuchów znaków.



```
name + " Travolta";
```

/* otrzymamy wynik */

```
"John Travolta"
```

Stringi – konwersja do liczb

parselnt

Za pomocą tej funkcji możemy konwertować stringi do liczb całkowitych (integer).

parselnt(string, system liczbowy 2–36)

NaN – zwracane kiedy wynik nie jest liczbą Not a Number

Przykład

```
var textVar = "9";  
var numberVar = parselnt(textVar, 10);  
  
parselnt("24px", 10); // 24 (2*10^1 + 4*10^0)  
parselnt("100", 2); // 4 (1*2^2 + 0*2^1 + 0*2^0)  
parselnt("546", 2); // NaN – nie ma takich liczb w systemie dwójkowym  
parselnt("Hello", 8); // NaN – to nie są cyfry
```

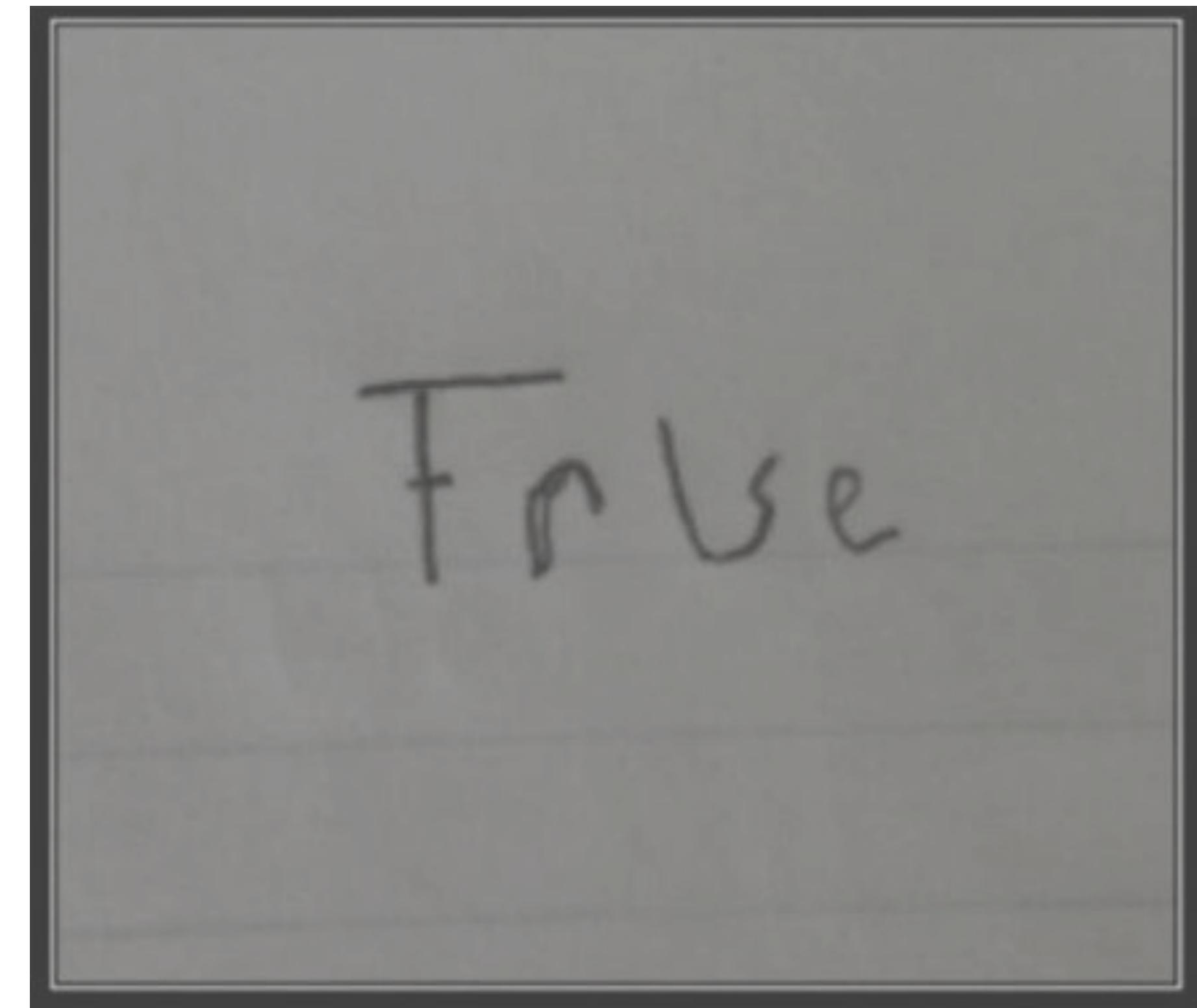
Wartości logiczne

Prawda lub fałsz

```
var isChecked = false;  
var isRendered = true;
```

Fałszem jest również:

- 0 (zero)
- "" (pusty string)
- null
- undefined
- NaN



Wartości specjalne

null

null reprezentuje pustą wartość.

Uwaga! null nie oznacza 0(zero).

```
var foo = null;
```

```
var bar = undefined;
```

undefined

undefined reprezentuje wartość niezdefiniowaną, czyli jeżeli stworzymy zmienną i nic do niej nie przypiszemy, wywołanie jej zwróci undefined.

```
var maxValue;  
maxValue;
```

Konsola wypisze undefined

Instrukcje warunkowe

if ... else

Więcej niż jeden warunek

if ... else umożliwia sprawdzanie warunku **if** i wykonanie „alternatywnego” kodu jeśli warunek nie jest spełniony **else**.

Istnieje możliwość sprawdzenia więcej niż jednego warunku za pomocą **else if**.

```
if (warunek) {  
    polecenia;  
} else if (warunek2) {  
    inne polecienia;  
} else {  
    jeszcze inne polecienia;  
}
```

switch

```
var expression = "John";  
  
switch (expression) {  
    case "Ala":  
        console.log("Jestem, wpadaj na kawę");  
        break;  
    case "John":  
        console.log("Jestem, ale zaraz idę");  
        break;  
    default:  
        console.log("Nie ma mnie w domu");  
}
```

Program szuka etykiety **case** pasującej do **expression**. Jeżeli napotka **break**, to kończy działanie instrukcji **switch**.

Jeżeli żadna etykieta **case** nie pasuje do **expression**, zostanie wykonana domyślna akcja z etykiety **default**.

Pętle

Pętla for

```
var result = 120;  
for (var i = 0; i < 3; i++) {  
    result = result + 1;  
}
```

Krok	i=0	result = 120	stop (i < 3)
1	0	121	i < 3
2	1	122	i < 3
3	2	123	i < 3



i jest równe 2, a warunek stopu mówi, że nie może być większe od 3, dlatego koniec pętli.

Pętla while

Dopóki **while** spełniony jest warunek, wykonuj pętlę.

```
var i = 0;  
while (i != 5) {  
    console.log("Pętle są fajne");  
    i = Math.floor(Math.random() * 10);  
}
```



Pętlę **while** wykorzystujemy, jeżeli nie wiemy, ile razy będziemy wykonywać jakieś instrukcje.

Obiekt **Math** jest wbudowany w JavaScript, dzięki jego metodom możemy korzystać z funkcji matematycznych, tutaj **floor** to zaokrąglenie w dół a **random** losowa liczba z przedziału 0-1

Np.:

Math.random() zwróci 0.2315634

$0.2315634 \times 10 = 2.315634$

$\text{Math.floor}(2.315634) = 2$

Pętla do ... while

Pętla **do ... while** działa tak samo jak pętla **while**, z tym że kod wykona się zawsze przynajmniej jeden raz, ponieważ kod z bloku **do** wykonuje się najpierw a dopiero sprawdzany jest warunek **while**.

```
var i = 0;  
do {  
    console.log("Pętle są fajne");  
    i = Math.floor(Math.random() * 10);  
}  
while (i != 5);
```

Break

Wszystkie pętle możemy zakończyć przed warunkiem stopu za pomocą polecenia **break**.

```
for (var i = 0; i < 3; i++) {  
    var result = Math.floor(Math.random() * 10);  
    if (result === 5) {  
        break;  
    }  
}
```

Operatory

Operatory arytmetyczne

```
var liczba1 = 2;  
var liczba2 = 4;
```

```
liczba1 + liczba2; // 6  
liczba1 - liczba2; // -2  
liczba1 / liczba2; // 0.5  
liczba1 * liczba2; // 8  
liczba1 % liczba2; // 2  
liczba1++; // 3  
liczba2--; // 3
```

Inkrementacja:

```
liczba1 = liczba1 + 1;
```

Dekrementacja:

```
liczba2 = liczba2 - 1;
```

```
var text1 = "2";  
var liczba2 = 4;
```

```
text1 + liczba2; // "24"  
text1 - liczba2; // -2  
text1 / liczba2; // 0.5  
text1 * liczba2; // 8  
text1 % liczba2; // 2
```

Oprócz dodawania JavaScript podczas wykonywania działań zamienia stringa na liczbę.

Ale tylko wtedy, jeżeli może!

Przykład:

"2ala" - 3 = NaN

Operatory porównania

Operatory porównania stosuje się w instrukcjach warunkowych:

```
var liczba1 = 1;  
var liczba2 = 77;
```

```
liczba1 == liczba2; // false  
liczba1 != liczba2; // true  
liczba1 === liczba2; // false  
liczba1 !== liczba2; // true  
liczba1 > liczba2; // false  
liczba1 < liczba2; // true  
liczba1 >= liczba2; // false  
liczba1 <= liczba2; // true
```

== luźna równość (loose equality)

=== ścisła równość identyczność (strict equality)

```
var text = "2";  
var liczba1 = 2;
```

```
text == liczba1 // true;  
text === liczba1 // false;
```

Podczas porównywania **==** JavaScript porównuje także **typ danych**, czyli w przypadku powyżej mamy porównanie nie tylko wartości, ale i typu, co daje w efekcie **false**.

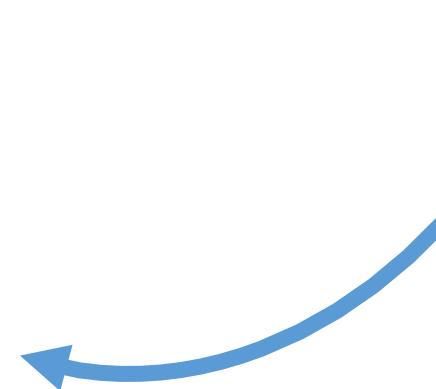
Operatory przypisania

```
var liczba3 = 1;  
var text = "Ala ma kota";
```

```
liczba3 += 2; // 3  
liczba3 -= 100; // -99  
liczba3 *= 25; // 25  
liczba3 /= 12; // 0.083333  
liczba3 %= 4; // 1
```

```
text += "a"; // Ala ma kotaa
```

W przypadku przypisywania do stringów
tylko konkatenacja ma sens.
Reszta operatorów zwróci NaN.



Operatory logiczne

AND, OR

Operatory logiczne stosuje się w instrukcjach warunkowych

var liczba3 = 23;

(liczba3 != 23) && (liczba3 > 10)

(liczba3 != 23) || (liczba3 > 10)

➤ && AND (logiczne i)

Jeżeli pierwszy warunek nie jest spełniony, dalsza część nie jest sprawdzana i zwracana jest wartość **false** ponieważ obydwa warunki muszą być spełnione.

➤ || OR (logiczne lub)

Wystarczy, że jeden z tych warunków będzie spełniony – zwracana jest wartość **true**.

Operatory logiczne

NOT i XOR

Operatory logiczne stosuje się w instrukcjach warunkowych

`var liczba3 = 23;`

`!(liczba3 > 22)`

`(liczba3 > 22) ^ (liczba3 != 23)`

➤ ! NOT (logiczne nie)

Jeżeli warunek jest prawdą, zwróci **false** i na odwroć.

➤ ^ XOR

Operator sprawdza, czy jeden z dwóch warunków jest spełniony, przy czym nie mogą być spełnione oba. Jeżeli jest spełniony jeden warunek, wtedy zwraca **true**, jeśli żaden lub dwa – **false**.

Czas na zadania



Wykonajcie zadania z działu:

1. Podstawy_JavaScript

Katalog

1. Podstawy

Funkcje

Deklaracja funkcji i wywołanie

```
function getName() {  
    console.log("Ala");  
}
```

} Deklaracja funkcji zostanie wykonana dopiero wtedy, kiedy ją wywołamy.

```
getName();  
} Wywołanie funkcji  
"Ala"
```

Funkcję możemy wywołać wcześniej ... :D

Wyrażenie funkcyjne i wywołanie

Wyrażenie funkcyjne

Funkcję przypisujemy do zmiennej.

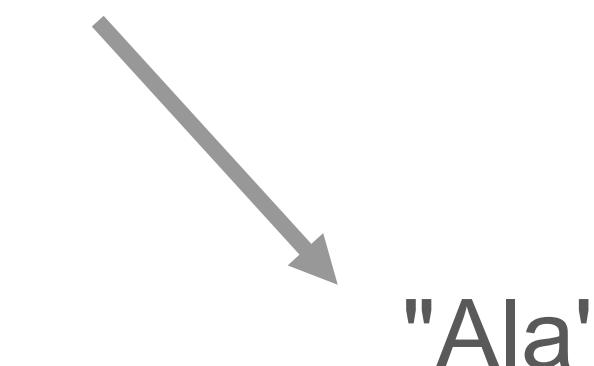
```
var foo = function getName() {  
    console.log("Ala");  
}  
  
foo();
```



Anonimowe wyrażenie funkcyjne

Funkcję przypisujemy do zmiennej i usuwamy jej nazwę.

```
var bar = function() {  
    console.log("Ala");  
}  
  
bar();
```



Argumenty funkcji

Argumenty funkcji to parametry, które możemy wrzucić do funkcji i przetworzyć.

```
function getFunnyName(name) {  
    return name + "yeah";  
}  
  
var result = getFunnyName("Ala");  
  
console.log(result);
```

Podstawienie

"Alayeah"

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");
```

Czas na zadania



Wykonajcie zadania z działu:

1. Podstawy_JavaScript

Katalog

2. Funkcje

Debugowanie

Debugowanie kodu JavaScript

Proces debugowania polega na znalezieniu miejsca występowania błędu w naszym kodzie.

Następnie dzięki temu możemy znaleźć przyczynę wystąpienia błędu oraz go poprawić.

Do debugowania będziemy używać narzędzi deweloperskiego dostępnego w każdej przeglądarce. W naszym przypadku użyjemy Google Chrome.



Debugowanie kodu JavaScript

Najprostszym a zarazem najmniej precyzyjnym sposobem debugowania kodu jest jego wykonanie i sprawdzenie w konsoli w jakim pliku i linii został wywołany błąd, następnie lokalizacja błędu i jego naprawienie.

Drugi sposób to dodanie w naszym kodzie `console.log()` w odpowiednich miejscach aby móc na konsoli śledzić jak przebiega wykonanie naszego skryptu.

```
function getName(name) {  
    console.log('Start function getName');  
    console.log('Get attr ' + name);  
  
    name = 'Hello ' + name;  
    console.log('Added greetings to name');  
  
    return name;  
}
```

Debugowanie kodu JavaScript

Jeśli jednak nasz kod jest bardziej skomplikowany niż kilka linii musimy skorzystać z bardziej zaawansowanych narzędzi.

Dzięki narzędziom debugowania możemy prześledzić aktualny stan podczas wykonywania skryptu w dosłownie każdym jego momencie.

```
var globalName = 'Tomek';

function sayMyName(name) {
    var greeting = 'Hello';
    name = greeting + ' ' + name;
    debugger;

    return name;
}

function sayGlobalName() {
    var greeting = 'Hi';
    var name = greeting + ' ' + globalName;

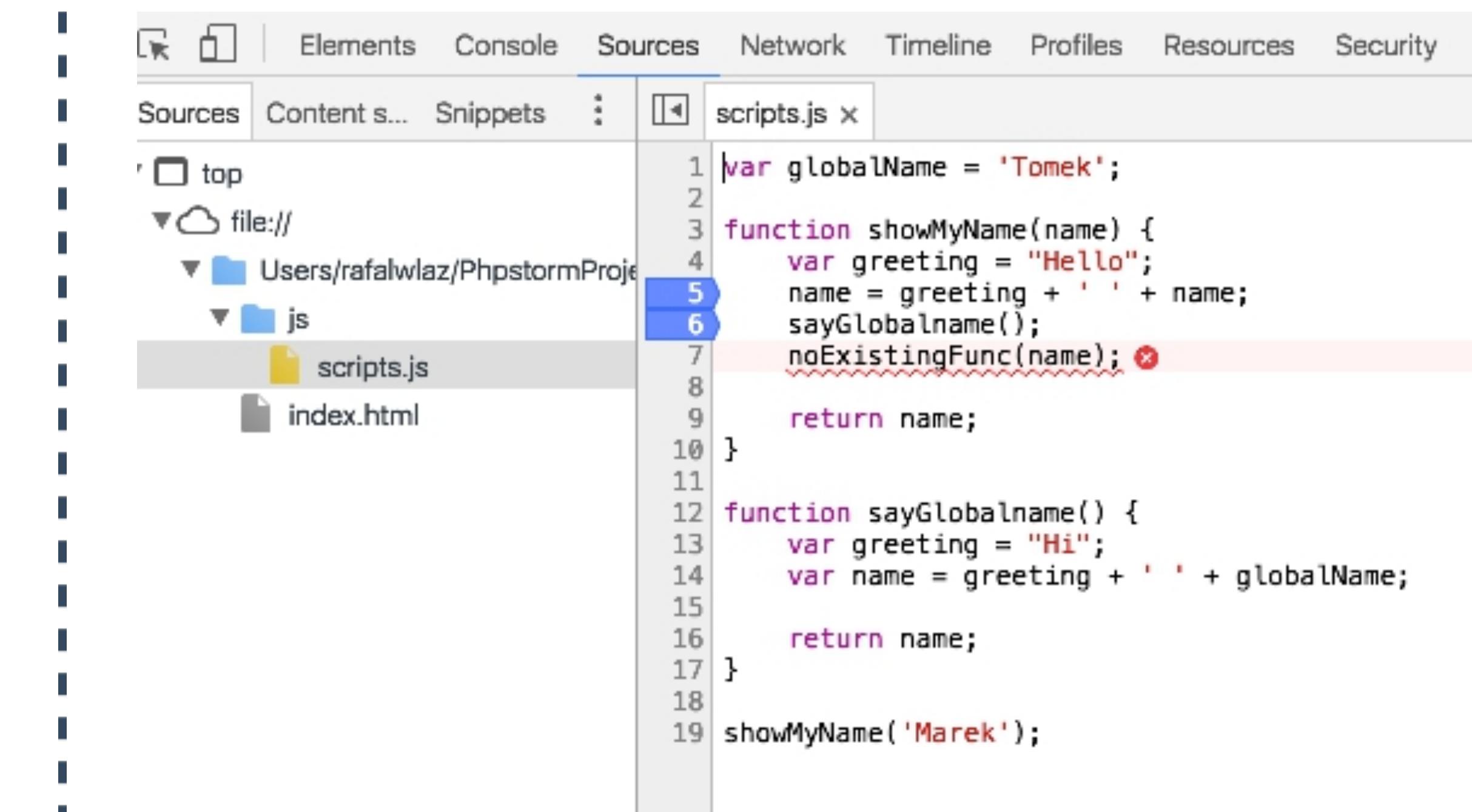
    return name;
}
```

Debugowanie kodu JavaScript

Aby rozpocząć proces debugowanie należy przejść do zakładki **Sources** a następnie wybrać plik JavaScript który chcemy debugować.

Klikając na numer linii kodu możemy oznaczyć tzw. breakpointy czyli miejsce gdzie wykonywanie naszego kodu się zatrzyma, możemy dodać ich dowolną ilość.

Breakpointy są zaznaczone na niebiesko.

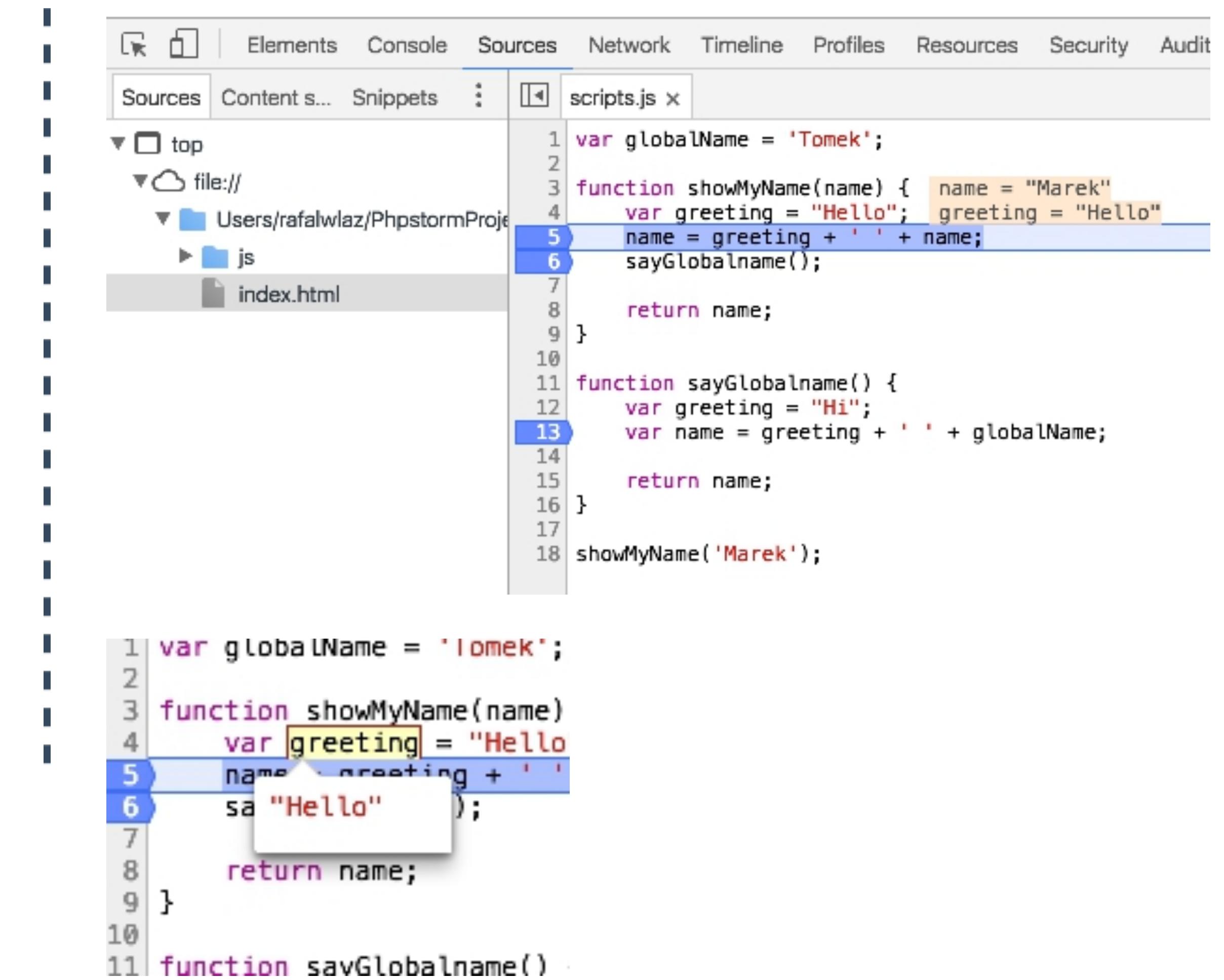


```
var globalName = 'Tomek';
function showMyName(name) {
  var greeting = "Hello";
  name = greeting + ' ' + name;
  sayGlobalname();
  noExistingFunc(name); noExistingFunc(name);
  return name;
}
function sayGlobalname() {
  var greeting = "Hi";
  var name = greeting + ' ' + globalName;
  return name;
}
showMyName('Marek');
```

Debugowanie kodu JavaScript

Po odświeżeniu strony skrypt rozpoczęcie pracę w trybie debugowania i zatrzyma się w miejscu pierwszego breakpointu a aktualna linia zostanie podświetlona.

Mogemy także najechać myszką na dowolną zmienną i w dymku pojawi się jej aktualna wartość.



The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. The left sidebar shows a file tree with 'scripts.js' selected. The main pane displays the code of 'scripts.js':

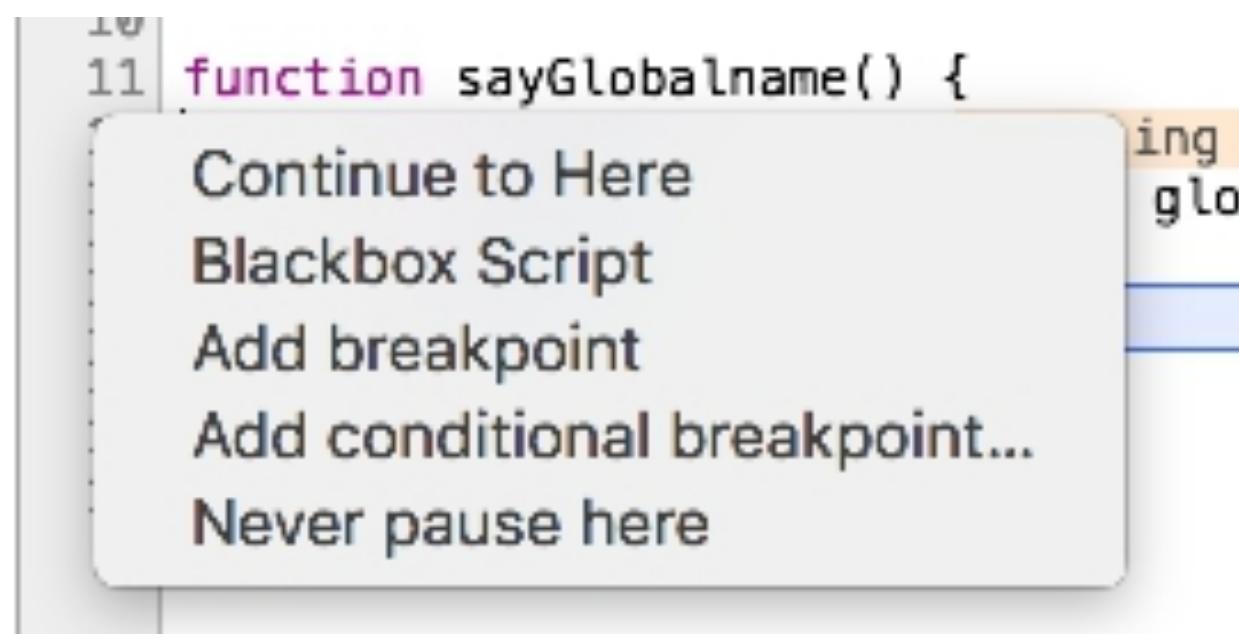
```
1 var globalName = 'Tomek';
2
3 function showMyName(name) { name = "Marek"
4   var greeting = "Hello"; greeting = "Hello"
5   name = greeting + ' ' + name;
6   sayGlobalname();
7
8   return name;
9 }
10
11 function sayGlobalname() {
12   var greeting = "Hi";
13   var name = greeting + ' ' + globalName;
14
15   return name;
16 }
17
18 showMyName('Marek');
```

A tooltip is shown over the line 'name = greeting + ' ' + name;' with the value 'Hello' displayed. The line number 5 is highlighted with a blue background.

Debugowanie kodu JavaScript

Nasze brakpointy mogą również zadziałać warunkowo tylko w określonej sytuacji.

Aby dodać warunkowy breakpoint klikamy myszką na numerze linii kodu i wybieramy opcję Add contitional breakpoint



Wpisujemy warunek pod jakim ma nastąpić zatrzymanie wykonywania skryptu.

```
12 | var greeting = "Hi"; greeting = "Hi"
| The breakpoint on line 12 will stop only if this expression is true:
| globalName='Tomek'
13 | var name = greeting + ' ' + globalName; name = "Hi Tomek"
```

A screenshot of a browser developer tools debugger interface. A conditional breakpoint is set on line 12. The condition is 'globalName='Tomek''. The code on line 12 is 'var greeting = "Hi"; greeting = "Hi"' and on line 13 is 'var name = greeting + ' ' + globalName; name = "Hi Tomek"'.

Linia, w której występuje breakpoint warunkowy podświetlona jest na żółto.

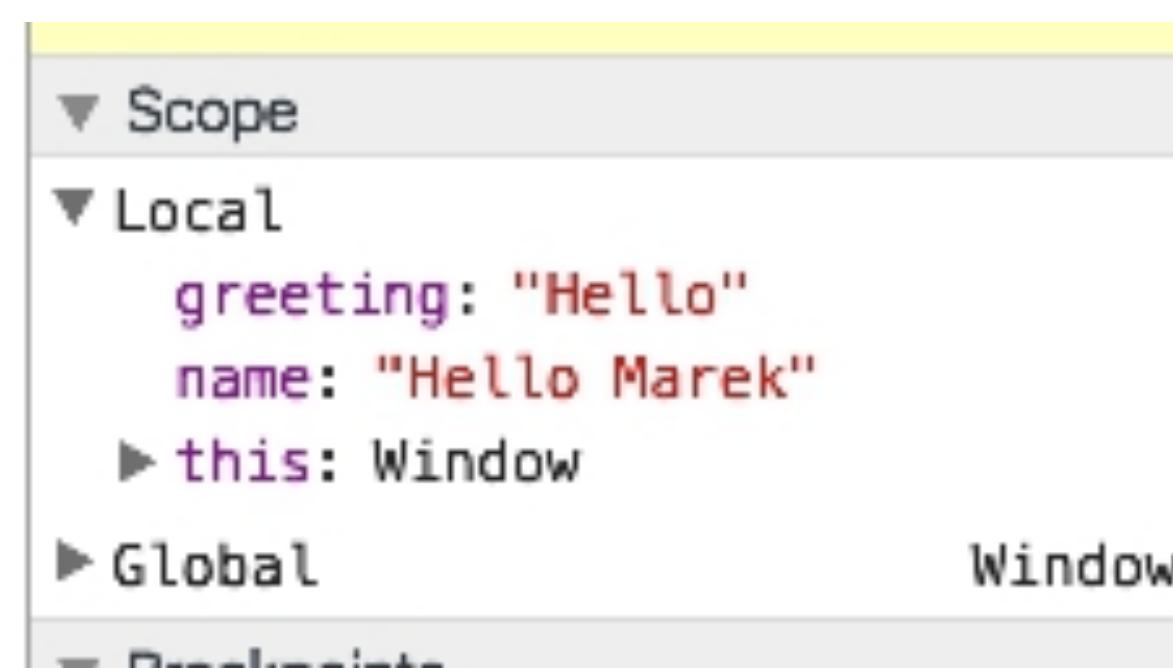
Mozemy dodać breakpoint również bezpośrednio w kodzie używając słowa kluczowego **debugger**

Debugowanie kodu JavaScript

Po prawej stronie okna narzędzi deweloperskich znajduje się menu sterowania przebiegu skryptu.



Dodatkowo znajdują się tam informacje o aktualnych wartościach zmiennych w naszym skrypcie w zakresie lokalnym np. funkcji oraz globalnym.



Wznowienie działania kodu po zatrzymaniu na breakpointie, przejście do kolejnego breakpointu – jeśli istnieje



Step over – pominięcie wejścia do funkcji jeśli znajduje się ona w kolejnej linii kodu. Jeśli w funkcji znajduje się breakpoint to debugger do niej "wejdzie"



Step into – wejście do funkcji i wykonywanie kodu z zatrzymaniem w każdej linii. Dzięki tej funkcji możemy prześledzić wywołanie kodu linia po linii.



Step out – wyjście z funkcji i przejście do kolejnej linii kodu.

Debugowanie kodu JavaScript

Pamiętajcie, że Debugger pozwala nam prześledzić sposób w jaki kod jest wykonywany.

Dodatkowo prześledzić jak zmieniają się zmienne oraz w jakiej kolejności kod jest wykonywany czyli tzw. Call Stack.

Breakpointy pozwalają zatrzymać wykonywanie skryptu w wybranym momencie aby prześledzić aktualny stan.

Szczegółowe informacje oraz instrukcje odnośnie debugowania w Google Chrome znajdziecie na stronie:

<https://developers.google.com/web/tools/chrome-devtools/debug/?hl=en>



Tablice

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



Wykonajcie zadania z działu:

1. Podstawy_JavaScript

Katalog

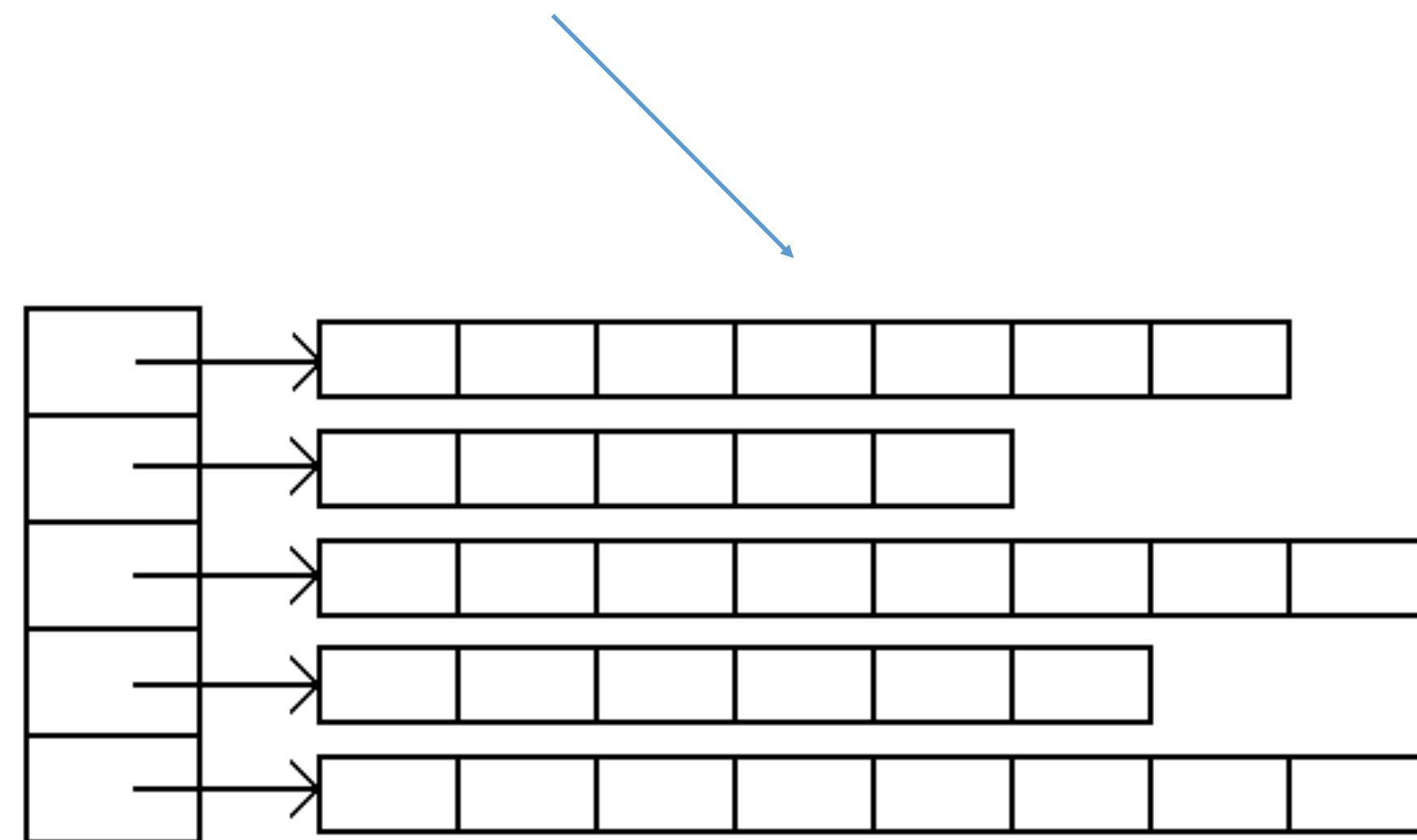
3. Tablice

Tablice wielowymiarowe

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],  
];
```



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] = [];
```

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



Wykonajcie zadania z działu:

1. Podstawy_JavaScript

Katalog

4. 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**.

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, czyli zbiór metod i atrybutów, 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.

Prototypy

```
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



Wykonajcie zadania z działu:

1. Podstawy_JavaScript

Katalog

5. Obiekty



Funkcje - tematy zaawansowane

Przenoszenie instrukcji

- Silnik JavaScript 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 zostały 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. **}**

Zmienne lokalne i globalne

W języku JavaScript 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 przeglądarki.
- Wyjście z naszej strony nie powoduje zniszczenia tych zmiennych!
- **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);  
}  
  
function sayLocalName() {  
    var name = "Janek"; // Zmienna lokalna  
    console.log(name);  
}  
  
var name = "Adam"; // Zmienna globalna name  
  
sayGlobalName(); // Adam  
sayLocalName(); // Janek  
console.log(name); // Adam
```

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

Wyświetlona zostanie zmienna **lokalna**.

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 sąbrane 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**

Funkcje wyższego rzędu

- 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.

Przykład

- 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.

Funkcje czasu

setTimeout

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

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

clearTimeout

- **clearTimeout** czyści **timeout** nastawiony 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 nastawiony przy pomocy **setInterval()**.
- Do tej funkcji trzeba podać **ID** interwału, który chcecie usunąć.

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

Czas na zadania



Wykonajcie zadania z działu:

1. Podstawy_JavaScript

Katalog

6. Zaawansowane funkcje

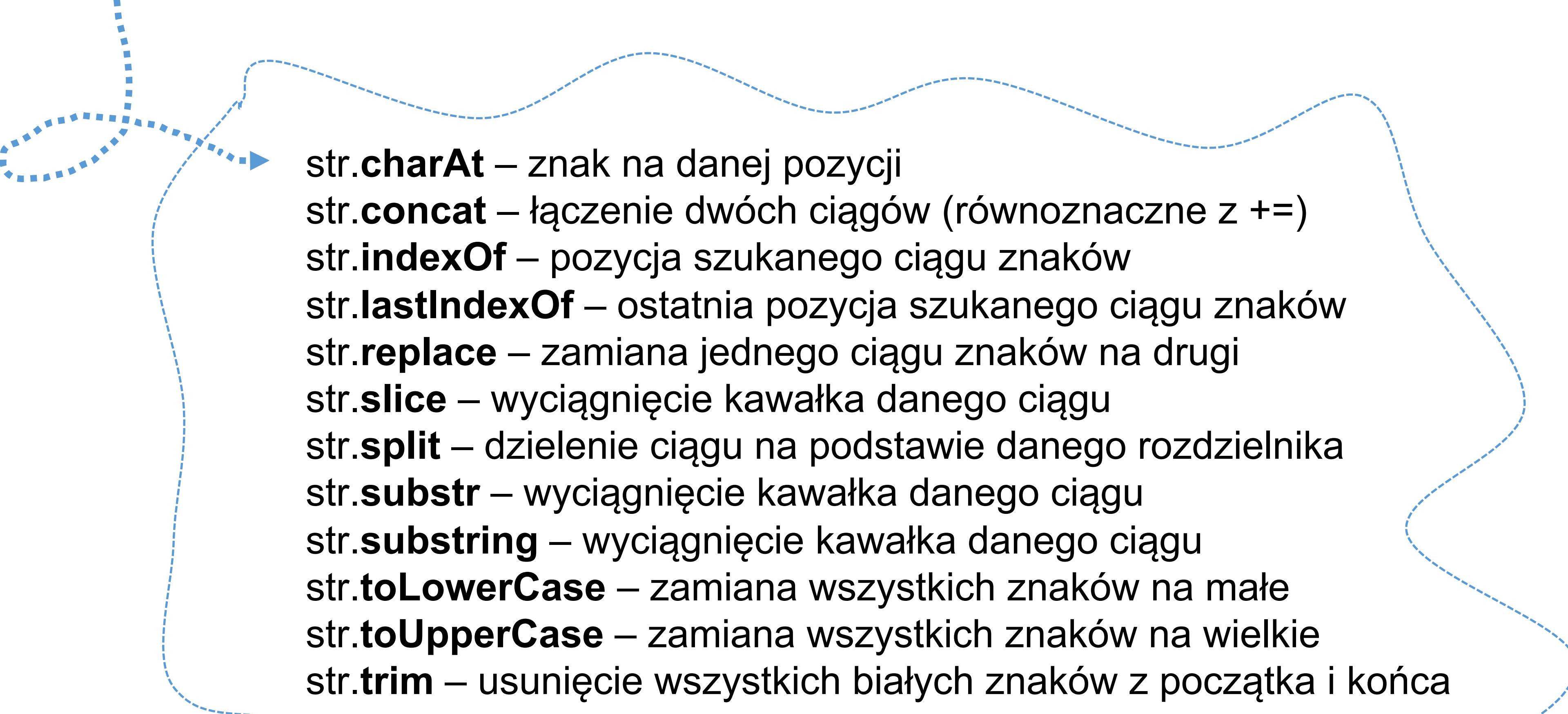


**Do przeczytania w
domu**

String metody

String metody

str to zmienna będąca ciągiem znaków

- 
- str.charAt** – znak na danej pozycji
 - str.concat** – łączenie dwóch ciągów (równoznaczne z `+=`)
 - str.indexOf** – pozycja szukanego ciągu znaków
 - str.lastIndexOf** – ostatnia pozycja szukanego ciągu znaków
 - str.replace** – zamiana jednego ciągu znaków na drugi
 - str.slice** – wyciągnięcie kawałka danego ciągu
 - str.split** – dzielenie ciągu na podstawie danego rozdzielnika
 - str.substr** – wyciągnięcie kawałka danego ciągu
 - str.substring** – wyciągnięcie kawałka danego ciągu
 - str.toLowerCase** – zamiana wszystkich znaków na małe
 - str.toUpperCase** – zamiana wszystkich znaków na wielkie
 - str.trim** – usunięcie wszystkich białych znaków z początku i końca

Tablice metody

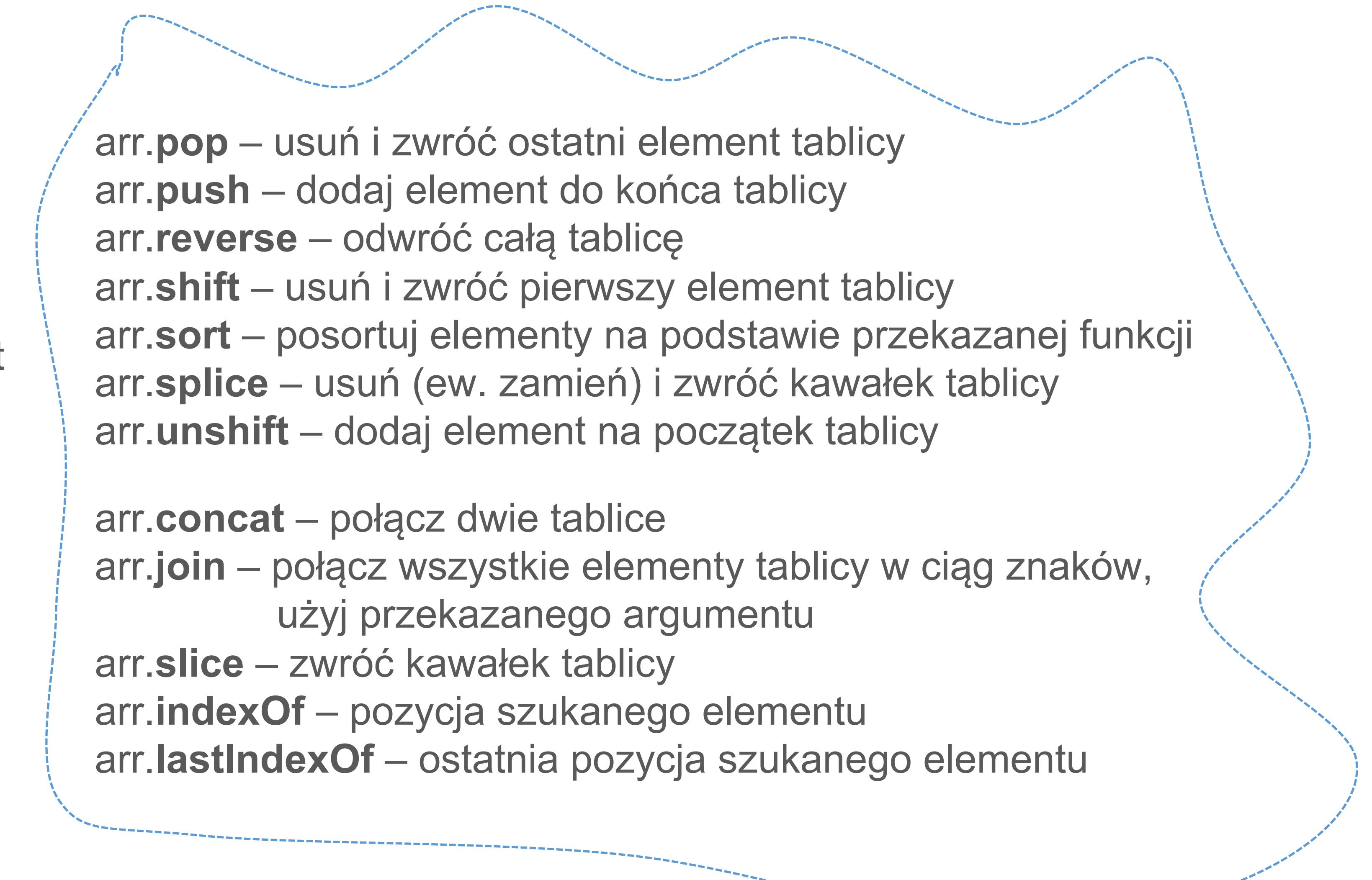
Tablice - metody

Mutacyjne

Modyfikujące oryginalną tablicę

`arr` – to zmienna, która jest tablicą

Dostępowe

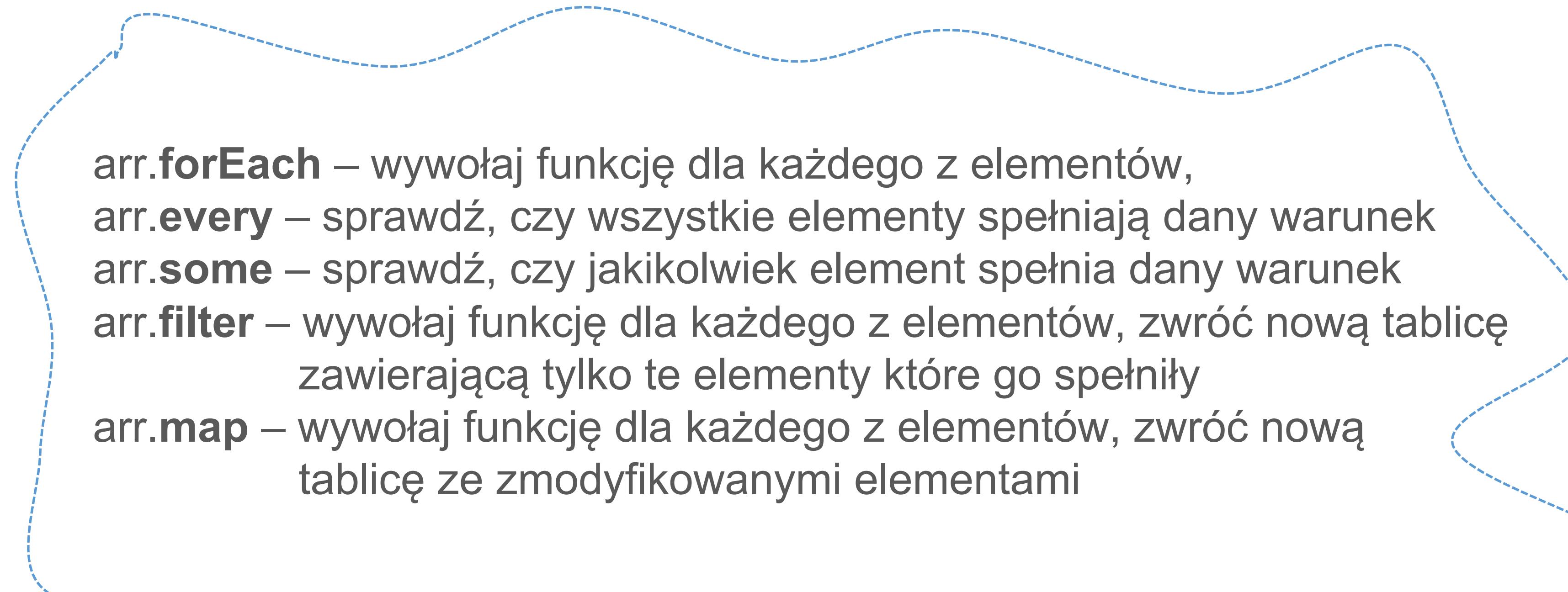
- 
- `arr.pop` – usuń i zwróć ostatni element tablicy
 - `arr.push` – dodaj element do końca tablicy
 - `arr.reverse` – odwróć całą tablicę
 - `arr.shift` – usuń i zwróć pierwszy element tablicy
 - `arr.sort` – posortuj elementy na podstawie przekazanej funkcji
 - `arr.splice` – usuń (ew. zamień) i zwróć 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]
```

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

Metody dostępowe

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

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

Metody dostępowe

slice()

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

Zwróć dwa elementy,
zacznij od indeksu 0.

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
```

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.