

# Wstęp do programowania

v.1.4

# Plan

- [Trochę teorii](#)
- [Schematy blokowe](#)
- [Pierwszy skrypt](#)
- [Typy danych](#)
- [Trochę więcej o liczbach – Obiekt Math](#)
- [Trochę więcej o stringach](#)

- [Operatory](#)
- [Kontrola przepływu programu](#)
- [Tablice](#)
- [Funkcje](#)
- [Debugowanie](#)
- [Stringi - metody](#)

# Trochę teorii - przypomnienie

# Trochę teorii - przypomnienie

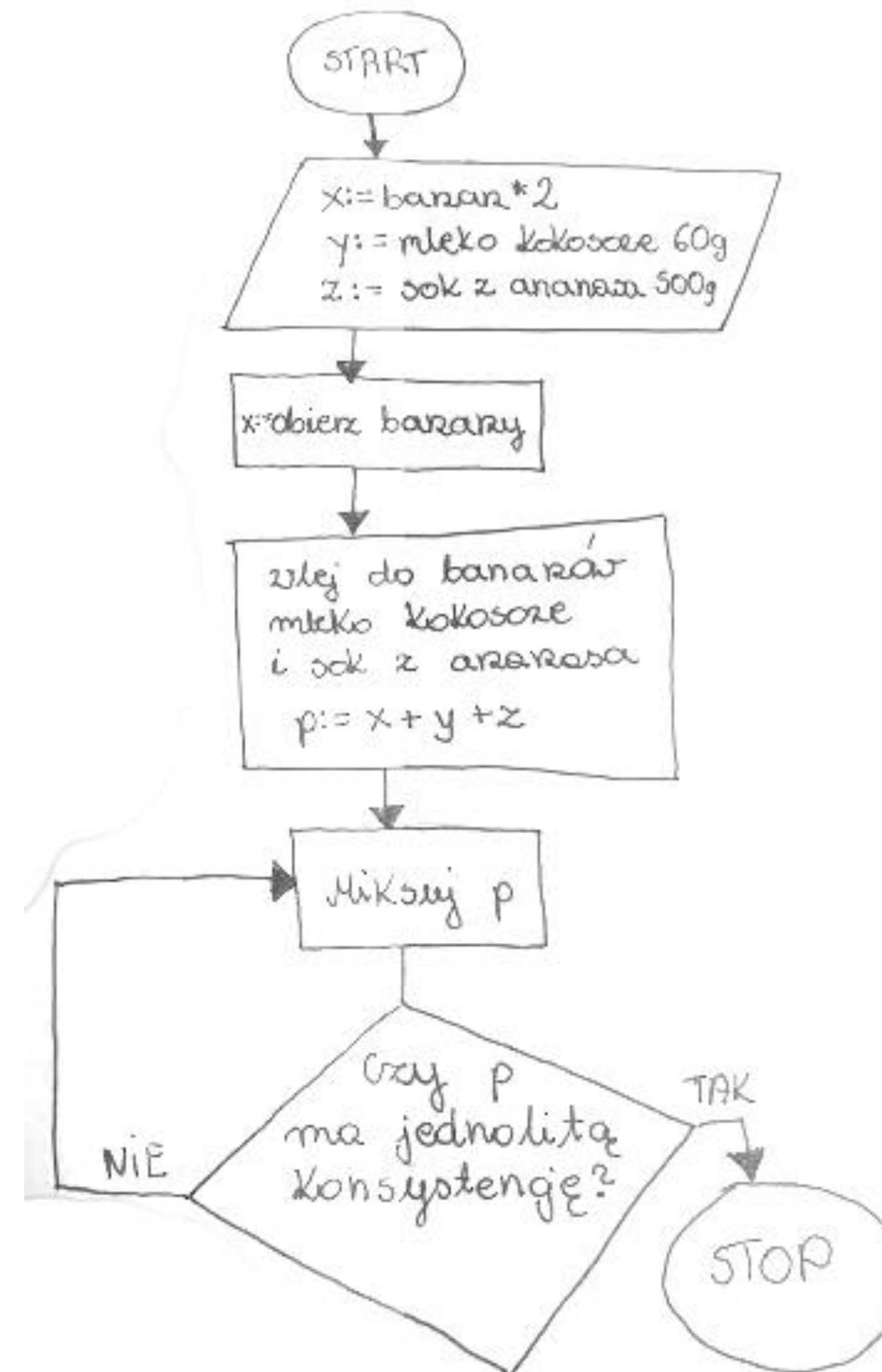
- Czym jest algorytm?
- Czym jest pseudokod?
- Jaka jest różnica między językiem programowania o kodem źródłowym?
- Czym jest program?
- Język programowania, a język znaczników. Jaka jest różnica?

# Schematy blokowe

# Schemat blokowy (flowchart)

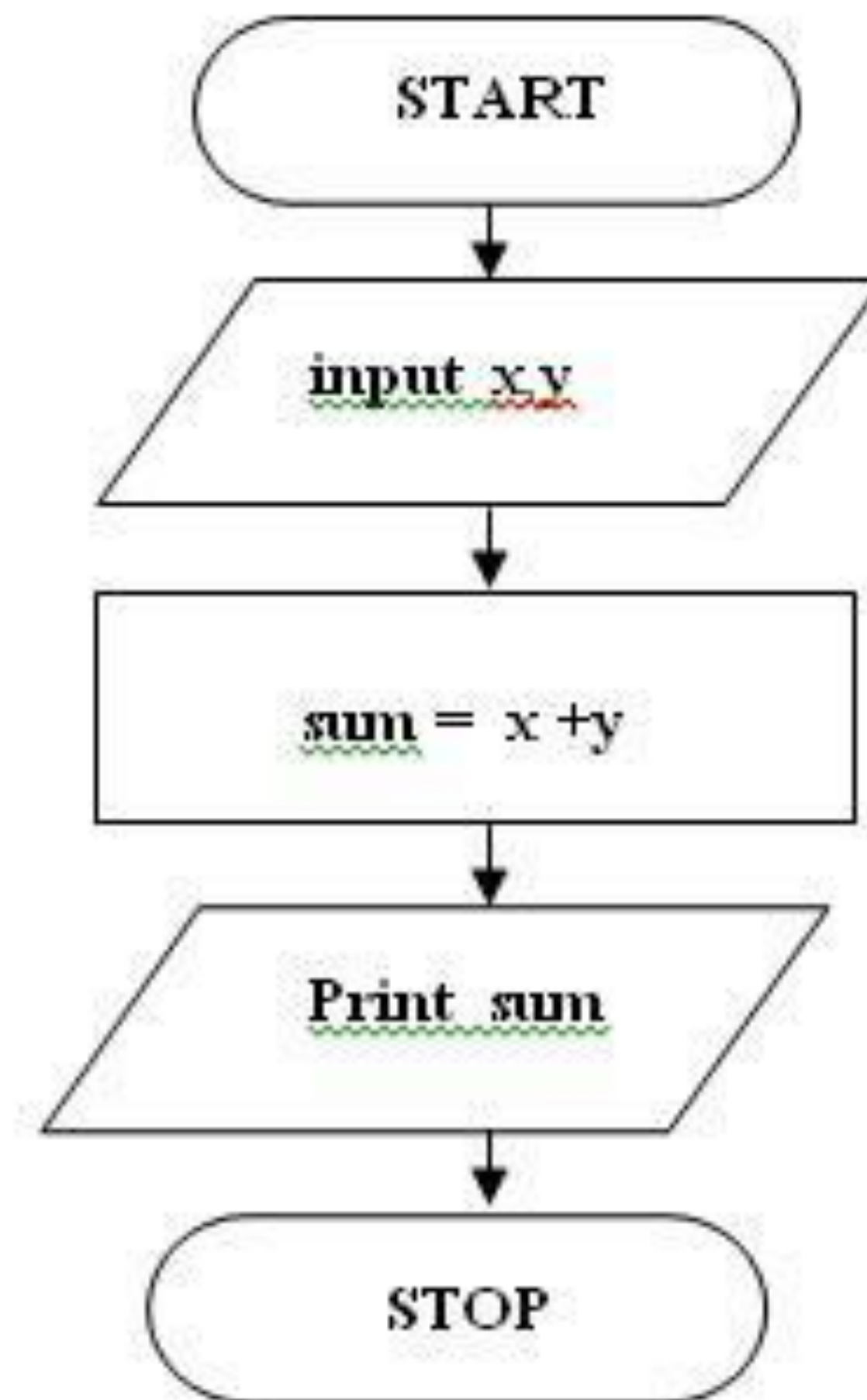
To narzędzie za pomocą, którego możemy pokazać czynności w algorytmie.

Występuje najczęściej w postaci **diagramu**.



# Schemat blokowy (flowchart)

- **Strzałka** – wskazuje jednoznacznie powiązania i ich kierunek.
- **Prostokąt** – zawiera wszystkie operacje z wyjątkiem instrukcji wyboru.
- **Równoległobok** – wejście/wyjście danych.
- **Romb** – wpisujemy wyłącznie instrukcje wyboru,
- **Owal/Okrąg** – oznacza początek bądź koniec schematu.



# Algorytmy są niezależne od języka

- Algorytm to idea działania programu.
- Algorytm idealny powinien być zapisany w pseudokodzie.
- Pseudokod algorytmu można przełożyć na praktycznie każdy język programowania.





Czas na zadania

Wykonaj zadania z  
części schematy  
blokowe

**Piszemy skrypt  
- przypomnienie**

# Pierwszy program w JavaScriptcie

## Plik HTML

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

## Plik JavaScript

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

# Chrome developer tools

Jak włączyć?

➤ **Na Windows:** *Crtl + Shift + I* lub *F12*

➤ **Na OS X:** *Cmd + Opt + I*

Narzędzie domyślnie zainstalowane w każdej przeglądarce Chrome.

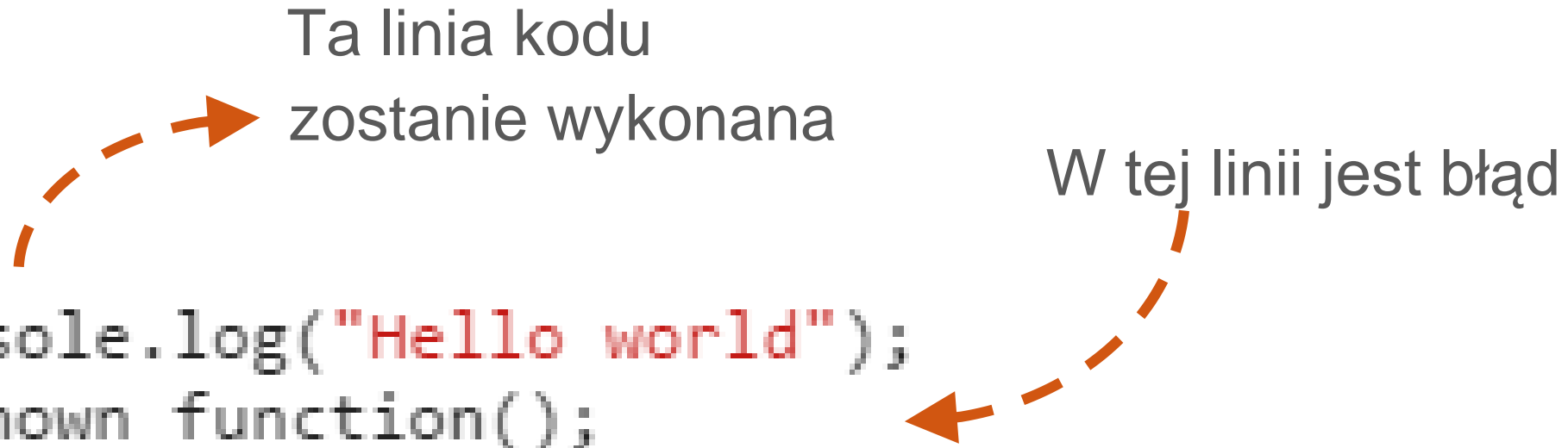
Więcej informacji o tym, jak używać tego narzędzia:

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



# 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.



```
> console.log("Hello world");  
unknown_function();
```

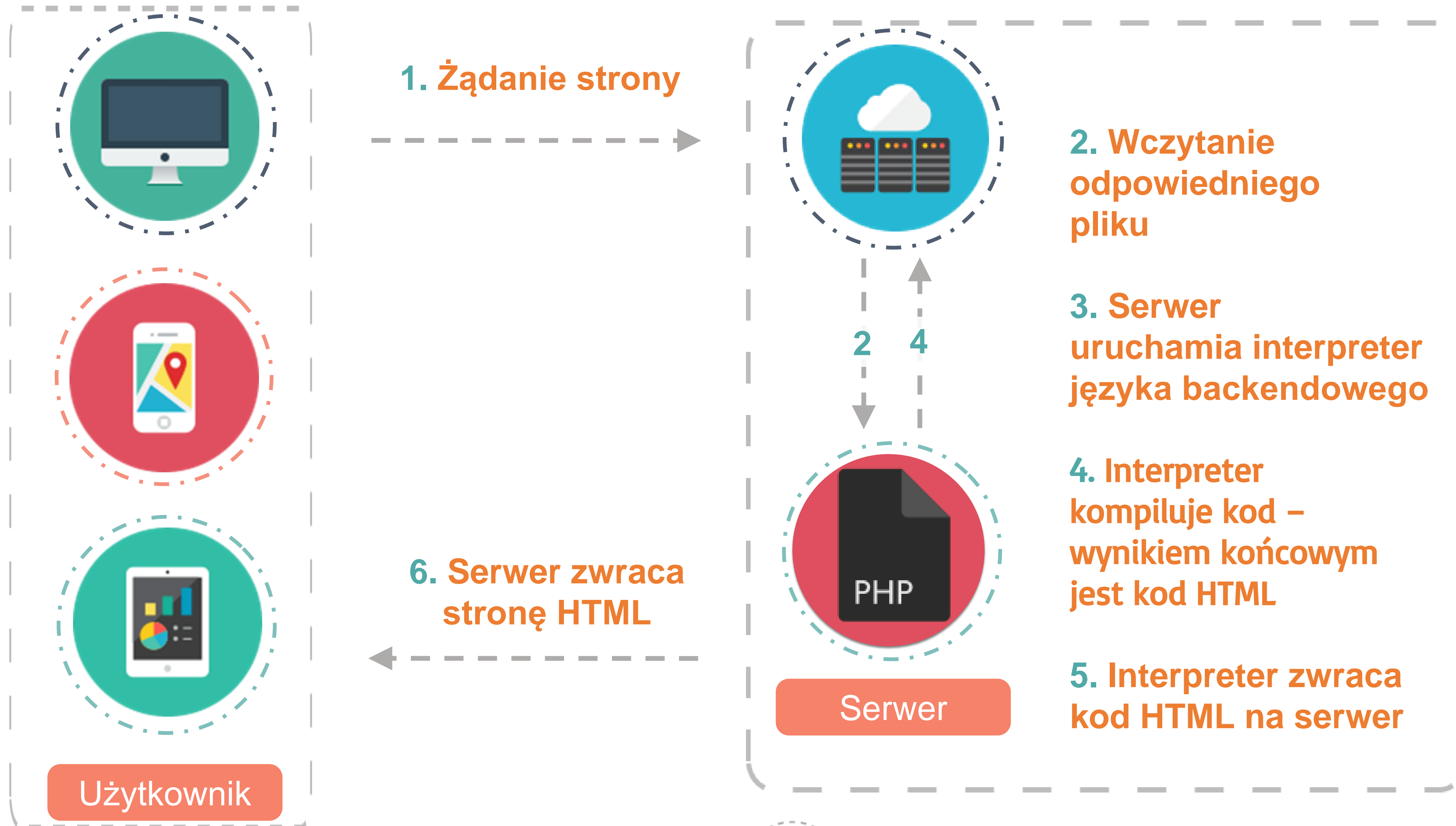
---

Hello world VM84:2

✖ ▶ Uncaught ReferenceError: unknown\_function is not defined(...) VM84:3

>

# Jak działa serwer?



# Gdzie jest w tym wszystkim JavaScript?

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



# Jak działa JavaScript?

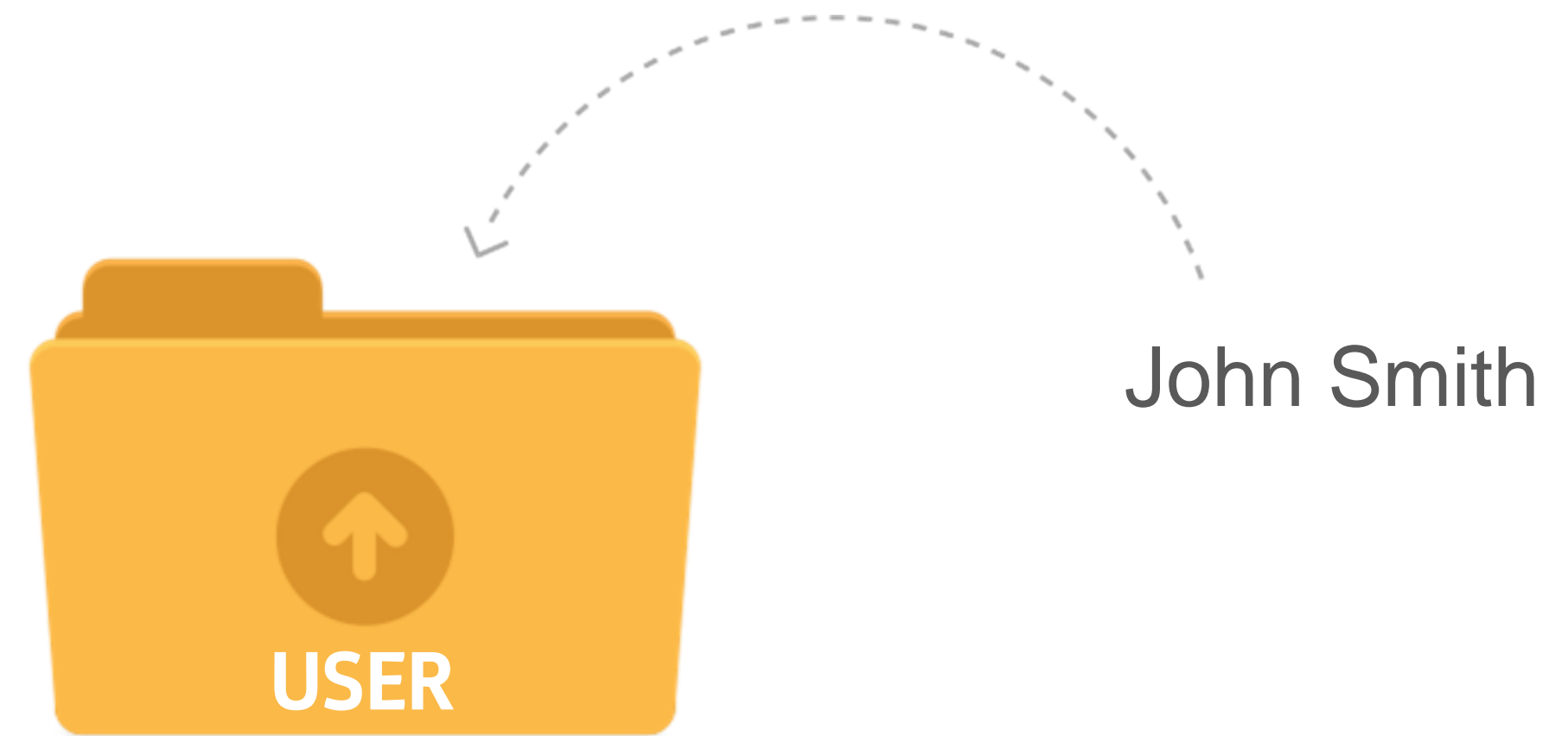
- JavaScript działa dzięki silnikom wbudowanym w przeglądarki internetowe.
- Silniki te wczytują kod JS podpięty pod stronę HTML i uruchamiają go na komputerze użytkownika.
- Silnik przeglądarki może też nasłuchiwać odpowiednich czynności wykonanych przez użytkownika.
- Na przykład po ruchu myszą czy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



# Typy danych

# Przypomnienie

- Czym jest zmienna?
- Co można przechowywać w zmiennych?
- Czy nazwa zmiennej `var 28numbers;` jest ok?
- Co oznacza słówko `var` i kiedy należy go używać, a kiedy nie?
- Jakie znasz typy danych?



# Typy danych - przypomnienie

## Liczby(Number)

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

## Ciągi znaków (String)

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

## Obiekty

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

## Wartości logiczne (Boolean)

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

## Specjalne

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


## Tablice

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

## Prymitywne typy danych

# Sprawdzanie typu – typeof

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

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

**Sprawdź inne rodzaje typów danych...**

Zauważ, że **typeof** zwraca informację o typie, ale w formie stringu. Czyli jeśli zapiszesz:

```
if (typeof null === object) {  
    // tutaj inny kod  
}
```

to nigdy nie będzie to prawdą, powinno się zatem porównywać w następujący sposób:

```
if (typeof null === "object") {  
    // tutaj inny kod  
}
```

# **Trochę więcej o liczbach - Obiekt Math**

# Obiekt Math

## Inne operacje matematyczne

W JavaScript mamy do dyspozycji specjalny obiekt Math. Dzięki jego metodom możemy wykonywać różnorodne operacje matematyczne takie jak na przykład:

- pierwiastkowanie
- potęgowanie
- zaokrąglanie
- inne

**Math.abs()** – liczba absolutna

**Math.ceil()** – zaokrąglenie w górę

**Math.floor()** – zaokrąglenie w dół

**Math.max()** – wartość maksymalna ze zbioru liczb

**Math.min()** – wartość minimalna ze zbioru liczb

**Math.pow()** – potęgowanie

**Math.random()** – losowa liczba z przedziału 0–1

**Math.round()** – zaokrąglanie

**Math.sqrt()** – pierwiastkowanie

## Przykład:

```
var foo = 2.8;  
Math.ceil(foo); // 3
```

```
var foo = 2.8;  
Math.floor(foo); // 2
```

# Trochę więcej o Stringach

# Stringi - przypomnienie

2 ≠ "2"

**Zapamiętaj!**

**Stringi to nie to samo co liczby**





# Jak zamienić stringa na liczbę?

## parseInt

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

`parseInt(string, system liczbowy 2–36)`

**NaN** – zwracane kiedy wynik nie jest liczbą **Not a Number**

## Przykład

```
var textVar = "9";
```

```
var numberVar = parseInt(textVar, 10);
```

```
parseInt("24px", 10); // 24 ( $2 \cdot 10^1 + 4 \cdot 10^0$ )
```

```
parseInt("100", 2); // 4 ( $1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$ )
```

```
parseInt("546", 2); // NaN – nie ma takich liczb  
w systemie dwójkowym
```

```
parseInt("Hello", 8); // NaN – to nie są cyfry
```

# Stringi - metody

## Co możemy robić ze stringami?

W JavaScript mamy do dyspozycji również specjalne metody, które pozwalają nam robić różne ciekawe rzeczy ze stringami 😊 np.:

- Wyciąganie ciągu znaków z całego napisu (stringa)
- Usuwanie białych znaków
- Zamiana znaków na duże litery .. lub na małe
- I inne

## Przykład

```
var text = "u mnie działa";  
text.toUpperCase();
```



"U MNIE DZIAŁA"

Na końcu prezentacji znajdziesz najpopularniejsze metody dla stringów. Zapoznaj się z nimi samodzielnie [Stringi - metody](#)



Czas na zadania

Wykonaj zadania z  
części Typy danych

# Operator

# Operatory arytmetyczne - przypomnienie

```
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 - przypomnienie

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 połączone

## Przypomnienie

W preworku poznaliśmy operatory przypisania były to:

- `" = "` - przypisz do zmiennej wartość
- `" ++ "` - inkrementacja
- `" -- "` - dekrementacja

## Przypomnienie

```
var liczba3 = 1; // 1
liczba++; // 2
liczba--; // 1
```

Obok znajdziesz połączenie operatorów przypisania razem z operatorami arytmetycznymi nazywamy je inaczej operatorami połączonymi

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

```
liczba3 += 2; // 3 - inaczej liczba3 = liczba3 + 2
liczba3 -= 100;
// -97 - inaczej liczba3 = liczba3 - 100, pamiętaj, że
// przed chwilą była równa 3 więc 3 - 100 = -97
```

```
liczba3 *= 2; // -194 inaczej liczba3 = liczba3 * 2
liczba3 /= 12; // -16.666 inaczej liczba3 = liczba3 / 12
liczba3 %= 2; // -0.16 inaczej liczba3 = liczba3 % 2
```

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



W przypadku przypisywania do stringów tylko konkatencja 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**.
- **|| 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 odwrót.
- ^ **XOR**  
Operator sprawdza, czy jeden z dwóch warunków jest spełniony, przy czym nie mogą być spełnione oba. Jeśli jest spełniony jeden warunek, wtedy zwraca **true**, jeśli żaden lub dwa – **false**.

# Operator warunkowy

Operatory logiczne stosuje się w instrukcjach warunkowych.

```
var liczba = 23;
```

```
var wynik = (liczba > 22) ? "okey" : "cos nie tak";
```

**3**

Podstaw to,  
co zwróci  
operator warunkowy  
pod zmienną wynik

**1**

Sprawdź warunek

**2a**

Jeżeli jest prawdą  
zwróć string "okey"

**2b**

Jeżeli nie jest prawdą  
zwróć string "cos nie tak"

# Kontrola przepływu programu

# Instrukcje warunkowe - przypomnienie

## if

```
var weather = "deszcz"; // tą wartość można zmienić
```

```
if (weather === "deszcz") {  
    console.log("Weź parasol");  
} else if (weather === "śnieg") {  
    console.log("Weź czapkę");  
} else {  
    console.log("Weź okulary słoneczne");  
}
```

## switch

```
var weather = "deszcz"; // tą wartość można zmienić
```

```
switch(weather) {  
    case "deszcz": {  
        console.log("Weź parasol");  
        break;  
    }  
    case "śnieg": {  
        console.log(" Weź czapkę");  
        break;  
    }  
    default : {  
        console.log(" Weź okulary słoneczne ");  
    }  
}
```

Oba te skrypty wykonują to samo. Pamiętaj, że problemy możemy rozwiązywać na różne sposoby



Czas na zadania

Wykonaj zadania z  
instrukcji  
warunkowych

# Pętla for i pętla while

## for

```
for(var i=0; i<=10; i=i+1) {  
    console.log(i);  
}
```

Wynik: 0 1 2 3 4 5 6 7 8 9 10

Pętle for wykonujemy jeśli wiemy na jakim zbiorze będziemy działać. W przykładzie powyżej wiemy, że to będą liczby od 0 do 10.

## while

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

Pętle while wykonujemy zazwyczaj jeśli nie wiemy ile razy mamy wykonać daną czynność. W przykładzie powyżej nie wiemy ile razy wykona się pętla, bo liczba, której używamy jako warunku stopu jest zawsze losowa.

# Pętle for podwójna (zależna i niezależna)

## Pętla zależna

```
for(var i=0; i<10; i++) {  
  for(var j=i; j<10; j++) {  
    console.log("i=" + i + ", j=" + j);  
  }  
}
```

Pętla zależna dlatego ponieważ jak widzisz druga zaczyna się w każdej iteracji od trochę dalszego numeru.

Pętla wewnętrzna jest zależna od pętli zewnętrznej.

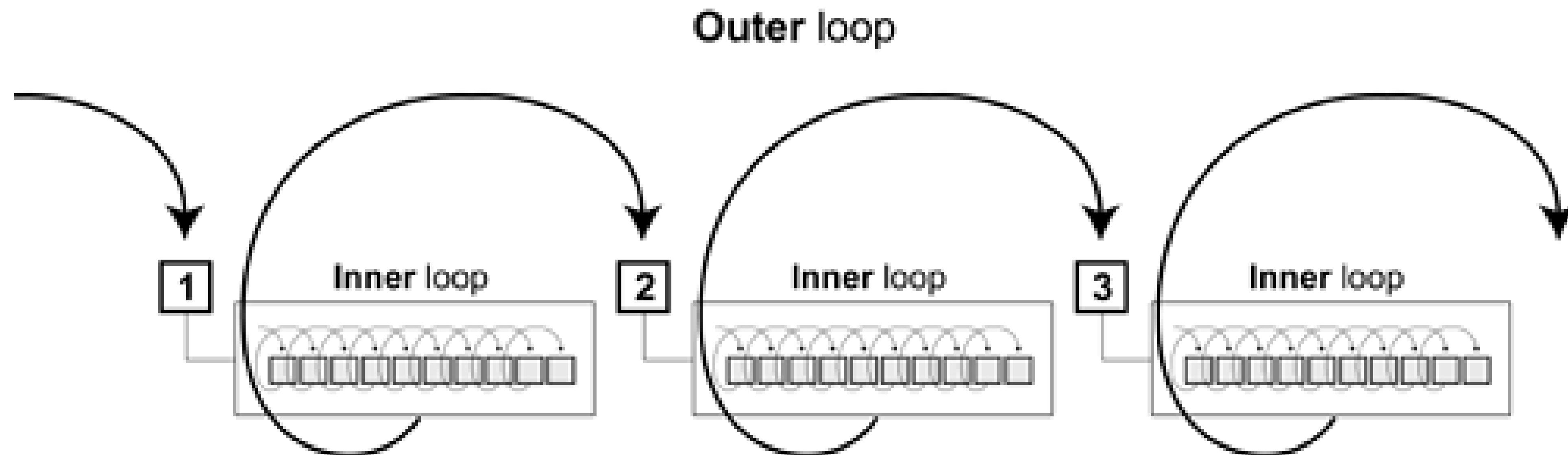
## Pętla niezależna

```
for(var i=0; i<10; i++) {  
  for(var j=0; j<10; j++) {  
    console.log("i=" + i + ", j=" + j);  
  }  
}
```

Tutaj obie pętle są niezależne..

# Pętla for (podwójna)

- Pętle możemy w sobie zagnieżdżać.
- Dzięki temu w każdej iteracji pętli zewnętrznej będzie wykonywane wiele iteracji pętli wewnętrznej.







Czas na zadania

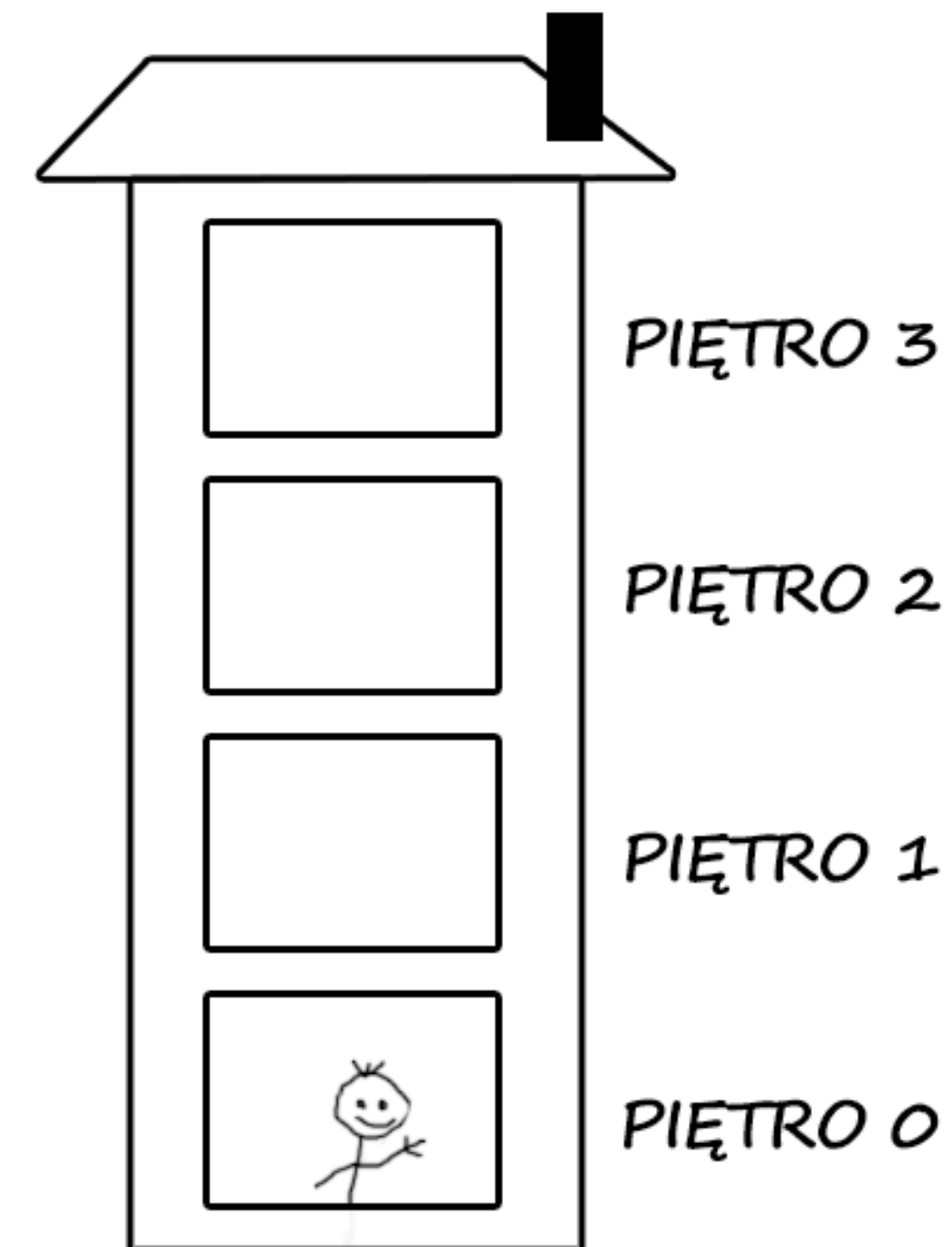
Wykonaj zadania z  
części Pętla

# Tablice

# Tablice - przypomnienie

- Czym jest tablica?
- Jak indeksujemy tablice?
- Jakie typy danych może zawierać tablica?
- Jak wyświetlić tablicę?
- Jak sprawdzić ile elementów ma tablica?

[1, 2, 3, "ala"]





Czas na zadania

Wykonaj zadania z  
części Tablice

# Funkcje

# Funkcje

- Czym jest funkcja?
- Jak ją stworzyć?
- Można ją stworzyć na kilka sposobów? Jak?
- Co to są argumenty?
- Jak to funkcja może coś zwracać? Którędy?
- Po co mi funkcja?

**Spróbujmy na kolejnych slajdach  
odpowiedzieć na te pytania.**

# Czym jest funkcja?

Funkcja wykonuje określone czynności, które możemy powtarzać kiedy chcemy



Funkcje to **odseparowany kawałek kodu** wykonujący jakąś czynność.

# Jak stworzyć funkcję?

## Kilka wskazówek

- Aby stworzyć funkcję potrzebujemy słowa kluczowego **function**.
- Następnie tworzymy **nazwę funkcji** (najlepiej taką, która nam pomoże po samej nazwie stwierdzić co funkcja ma robić).
- Pomiedzy nawiasami klamrowymi umieszczamy tak zwane **ciało funkcji**, czyli wszystkie czynności, które mają zostać wykonane.
- Aby funkcja mogła się wykonać, musimy ją **wywołać**. Robimy to wpisując jej nazwę z nawiasami okrągłymi – oczywiście **POZA JEJ CIAŁEM**.

## Przykładowa funkcja

```
function sayHello() {  
    console.log("Hello" );  
}
```

Definicja funkcji

`sayHello();`  Hello

Funkcja mogłaby nie mieć nazwy, bo nie jest to obowiązkowe, ale ciężko by ją było wywołać, prawda?

Niemniej jednak będziemy używać funkcji bez nazwy, mówimy wtedy o **funkcji anonimowej**. Porozmawiamy o tym później 😊



# Tworzenie funkcji inne sposoby 😊

## Wystarczą dwa sposoby

Funkcje w języku JavaScript możemy stworzyć na różne sposoby. Wszystko zależy od tego co chcemy osiągnąć.

**Na razie w swoim kodzie używaj definicji funkcji!** Kiedy omówimy zagadnienia zaawansowane stanie się jasne, kiedy używać której metody.

Definicję funkcji już znasz:

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

Inny sposób to wyrażenie funkcyjne:

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

Po co nam nazwa getName i tak jej nie używamy, usuńmy ją i mamy **anonimowe wyrażenie funkcyjne**:

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

# Argumenty funkcji - wejście

Aby zrozumieć na czym polega przekazywanie argumentów, musisz sobie wyobrazić, że funkcja to tak jakby **wydzielony teren**, który posiada **wejście i wyjście**.

Do wejścia wrzucamy tak zwane **argumenty** lub inaczej **parametry** funkcji. Argumentami mogą być wszystkie typy danych jakie znasz.

Na przykładzie obok przekazujemy do funkcji stringi.

Ale moglibyśmy przekazać również liczbę.

```
function getName(name) {  
    console.log(name + "Yeahh");  
}
```

Podstawienie

```
getName("Ala");  
getName("Jan");  
getName("Marek");  
getName("Karol");
```

Po wywołaniu funkcji, zostaną wyświetlone w konsoli imiona z doklejonym stringiem

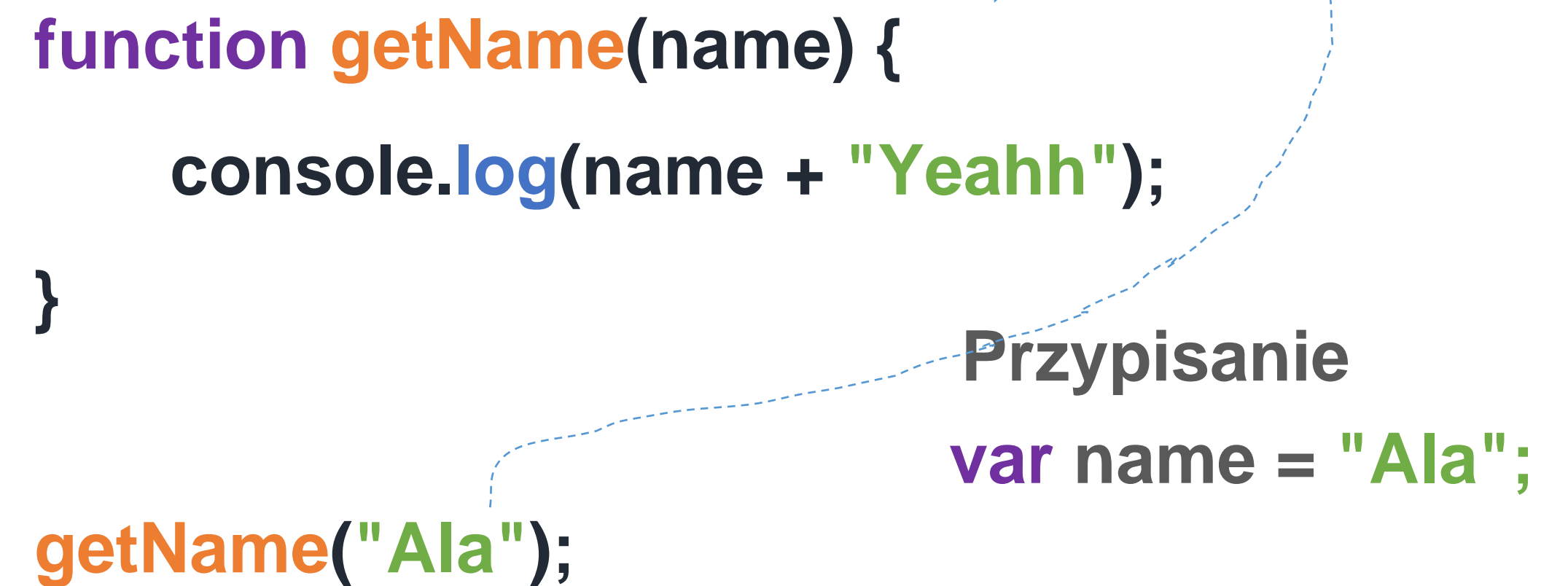
```
getName(23);
```

Po wywołaniu w tej linii co zostanie wyświetlone?

# Argumenty funkcji – wejście (przypisanie)

Podczas przekazywania argumentów trzeba sobie zdawać sprawę z bardzo ważnej rzeczy, a mianowicie z tego, że mamy do czynienia z **przypisaniem wartości do zmiennej**.

Nie jest to oczywiste, dlatego postaraj się to szczególnie zapamiętać.



The diagram illustrates the concept of variable assignment and its use in a function call. It features two code snippets separated by a vertical dashed line. On the right, a variable `name` is assigned the value `"Ala"` using the `var` keyword. A blue dashed arrow points from this assignment to the `name` parameter in a function definition `getName(name)` on the left. Another blue dashed arrow points from the `"Ala"` argument in a function call `getName("Ala")` on the left to the same `name` parameter in the function definition. The word "Przypisanie" (Assignment) is placed between the two snippets, with a blue dashed arrow pointing to the assignment statement.

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

Przypisanie  
`var name = "Ala";`

# Wiele argumentów funkcji

Do funkcji możesz przekazywać wiele argumentów.

Wymieniamy je zarówno przy definicji jak i przy wywołaniu funkcji, po przecinku.

```
function showInfo(name, age, sex) {  
    console.log(name);  
    console.log(age);  
    console.log(sex);  
}  
showInfo("Ala" , "99" , "female" );
```

# return – wyjście z funkcji

## Return kończy działanie funkcji

```
function getName(name) {  
    return name + " & Leo";  
}
```

Do zmiennej name dodajemy string.  
Następnie zwracamy połączone stringi  
(NIE WYŚWIETLAMY TUTAJ NIC!)

`getName("Kate");` → Nic się nie wypisze w konsoli po takim wywołaniu

//dopiero tak;  
`console.log(getName());` → "Kate & Leo"

## PAMIĘTAJ!

```
function getName(name) {  
    return name;  
    var name = "John";  
}
```

Instrukcje występujące po słowie **return** nigdy się nie wykonają.  
Tego typu zapis to duży błąd.

`console.log(getName("Kate"));` → "Kate"

# Po co tworzymy funkcje?

## Podsumowanie

- Funkcje tworzymy głównie po to, aby móc ten sam kod wykonywać kilka razy. **Nie powtarzamy się** – co jest jedną z podstawowych zasad programowania.
- Dodatkowo zyskujemy na **przejrzystości** naszego kodu, gdyż wszystkie czynności mamy posegregowane pod odpowiednimi blokami kodu.
- Funkcje możemy **parametryzować**, tzn. przekazywać do niej różne dane i wykonywać na nich te same czynności.

- Funkcja może zwracać różne wartości.
- Używaj funkcji!



Poprawne tworzenie funkcji  
i ponowne ich używanie  
jest jedną z **ważniejszych**  
rzeczy w programowaniu!





Czas na zadania

Wykonaj zadania z  
części 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 narzędzia 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;
}
```

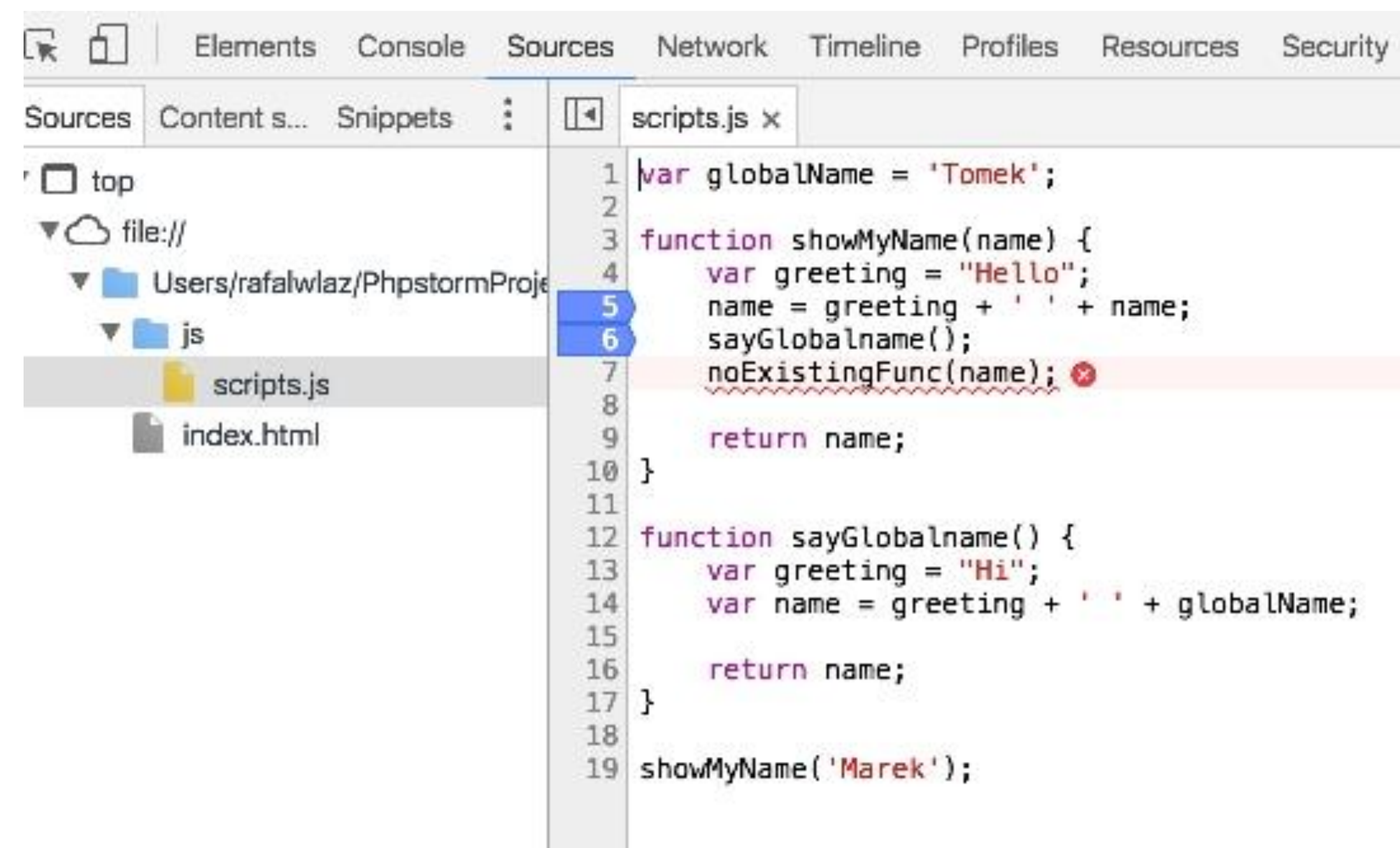


# Debugowanie kodu JavaScript

Aby rozpocząć proces debugowania 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.

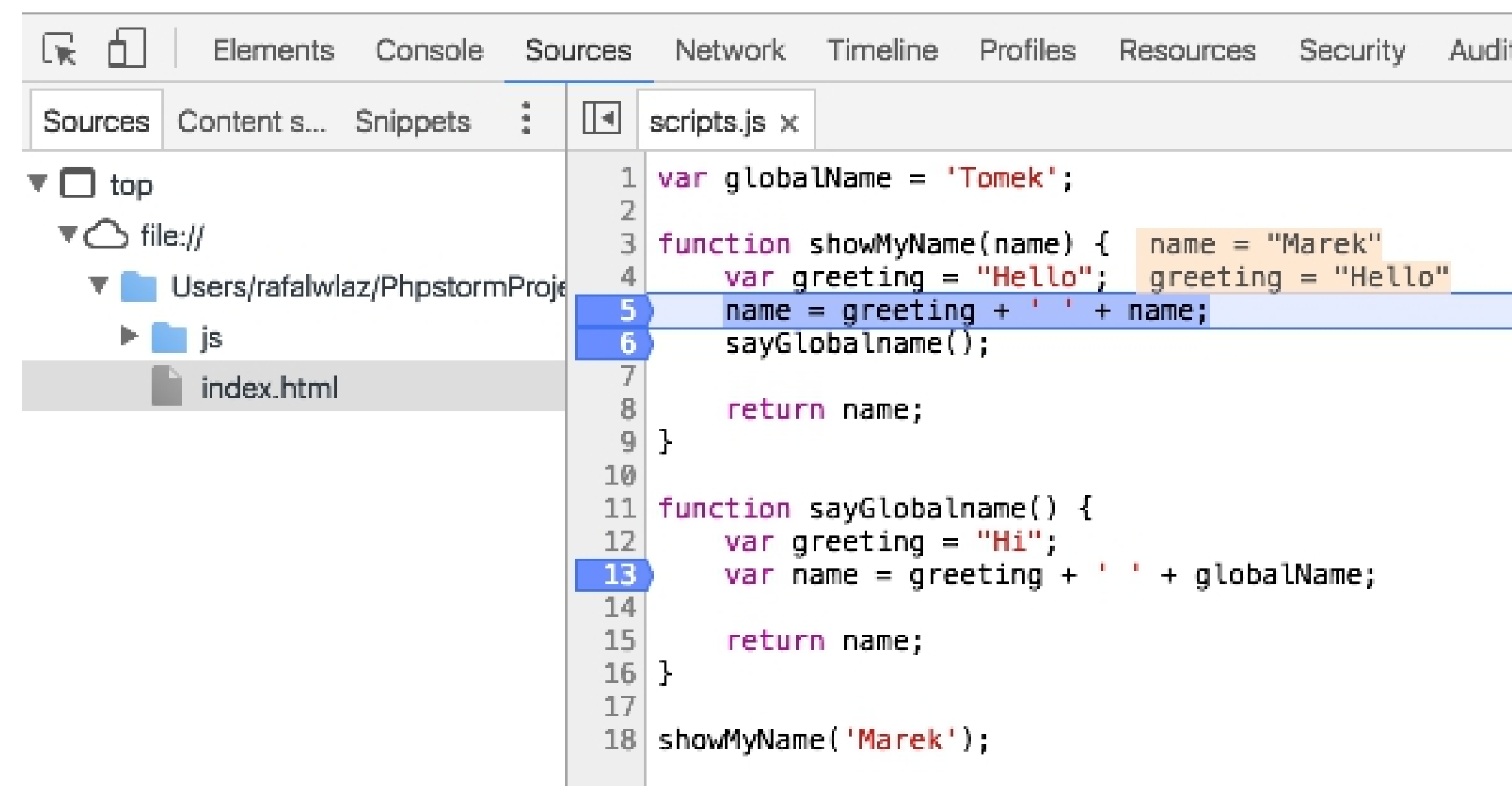
Breakpointy są zaznaczone na niebiesko.



# Debugowanie kodu JavaScript

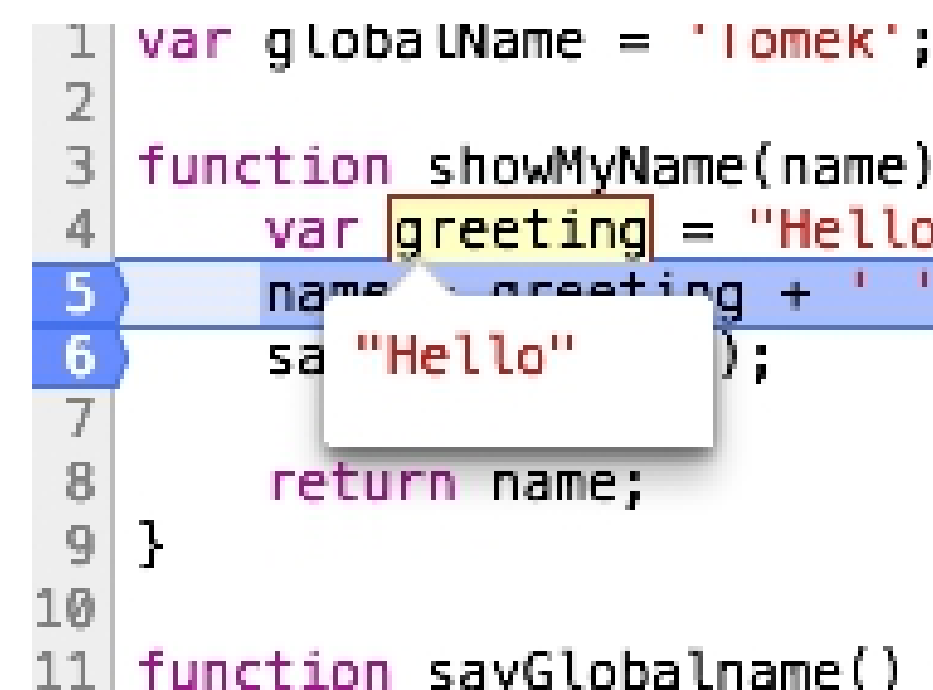
Po odświeżeniu strony skrypt rozpocznie pracę w trybie debugowania i zatrzyma się w miejscu pierwszego breakpointu a aktualna linia zostanie podświetlona.

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



The screenshot shows the Chrome DevTools 'Sources' panel. The file 'scripts.js' is open, and a breakpoint is set at line 5. The code is as follows:

```
1 var globalName = 'Tomek';
2
3 function showMyName(name) {
4   var 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');
```



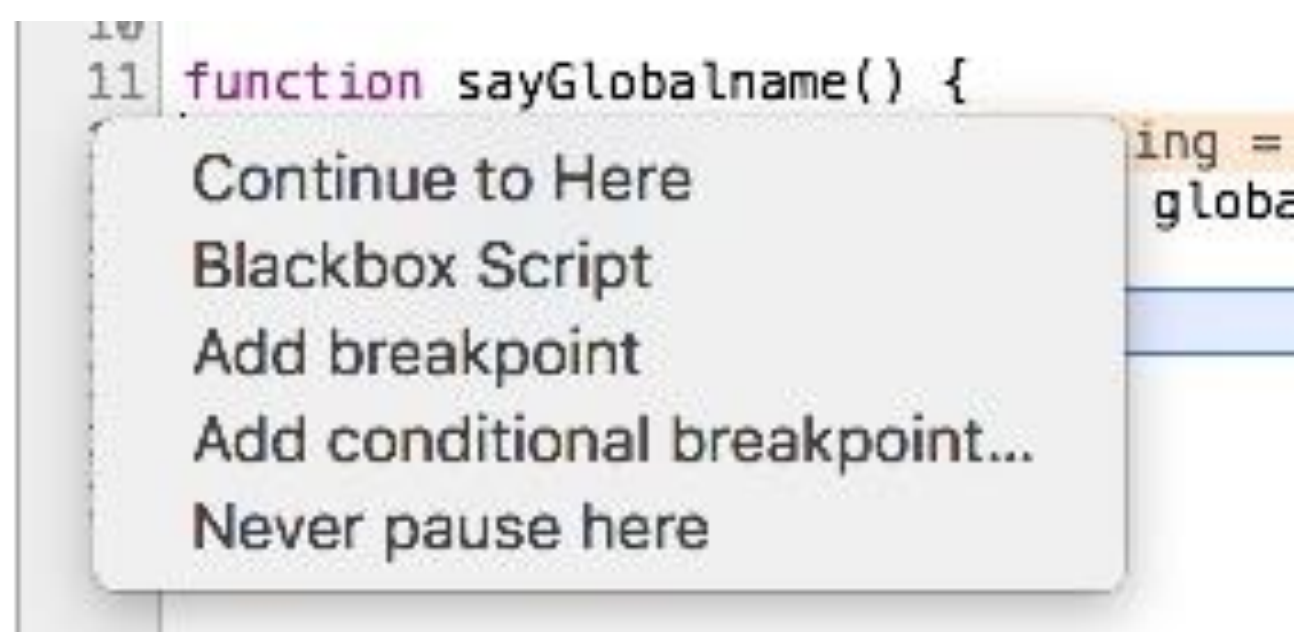
This close-up shows the code with a tooltip hovering over the 'greeting' variable on line 5. The tooltip displays the value 'Hello'.

```
1 var globalName = 'Tomek';
2
3 function showMyName(name)
4   var greeting = "Hello"
5   name = greeting + ' '
6   sa "Hello" );
7
8   return name;
9 }
10
11 function savGlobalname()
```

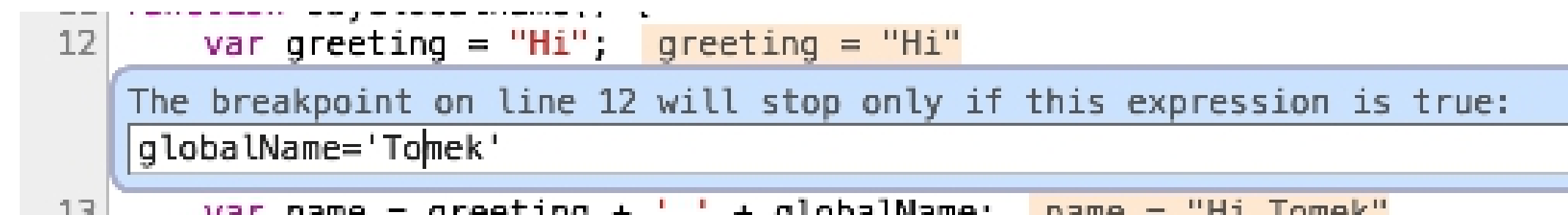
# Debugowanie kodu JavaScript

Nasze breakpoints mogą również zadziałać warunkowo tylko w określonej sytuacji.

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



Wpisujemy warunek pod jakim ma nastąpić zatrzymanie wykonywania skryptu.



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

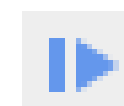
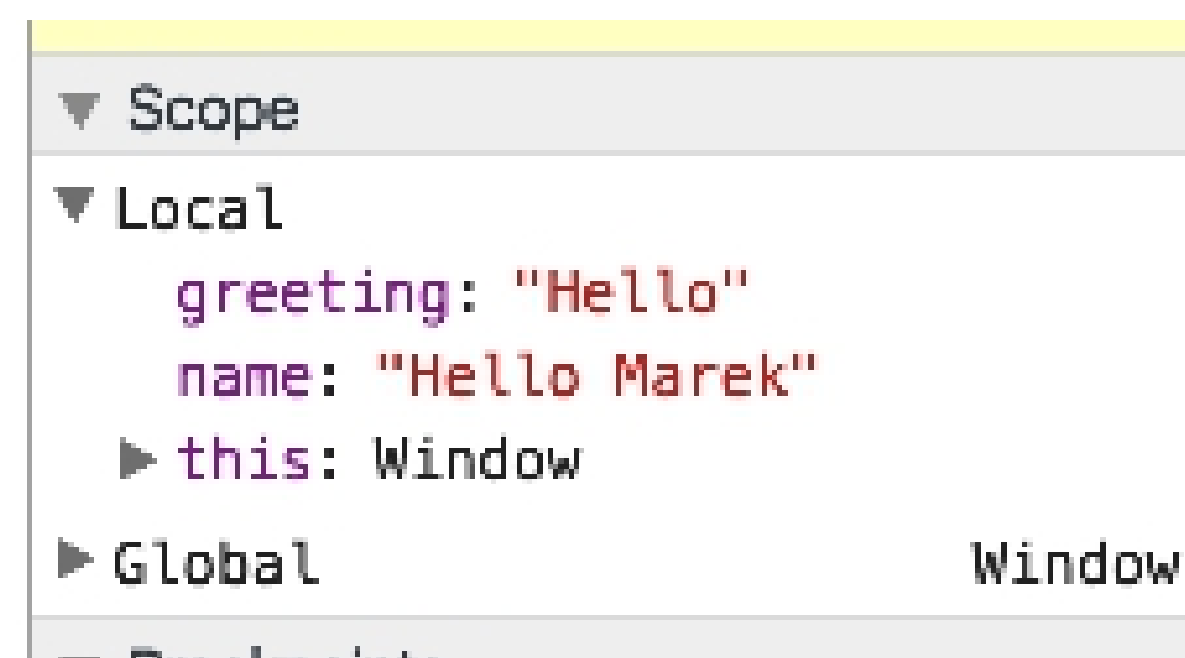
Możemy dodać breakpoint również bezpośrednio w kodzie używając słowa kluczowego **debugger**

# 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 debuggowania w Google Chrome znajdziecie na stronie:

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







Czas na zadania

Wykonaj zadania z  
części Debugowanie

# Stringi metody

# Stringi - 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.

# Stringi metody

## charAt

Metoda zwracająca znak, który znajduje się w danym indeksie.

```
var text = "bigos";  
text.charAt(2);
```

"g"

## concat

Ta metoda łączy stringi, tak jak operator += .

```
var text = "bigos";  
var text2 = "z charakterem";  
var text3 = text.concat(text2)
```

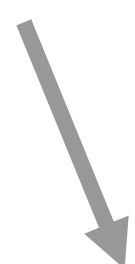
"bigosz charakterem"

# Stringi metody

## indexOf

Metoda zwraca pierwszą pasującą pozycję wzorca w stringu.

```
var text = "Zupa była za słona";  
text.indexOf("za");
```



10

## lastIndexOf

Zwraca ostatnią pasującą pozycję wzorca w stringu.

```
var text = "Zupa była za słona, ale była za free ";  
text.lastIndexOf("była");
```



24

# Stringi metody

## replace

Zamiana jednego ciągu znaków na drugi.

```
var kolacja = "Kanapka z serem";  
kolacja.replace("serem", "szynką");
```

"Kanapka z szynką"

## slice

Wyciągnięcie kawałka danego ciągu

slice(indeks początkowy, indeks końcowy)

```
var text = "Myśl pozytywnie";  
text.slice(0,4);
```

"Myśl"

# Stringi metody

## substr

Wyciągnięcie kawałka danego ciągu

`substr(indeksPoczątkowy, długość)`

```
var text = "Człowiek, który nie robi błędów,  
zwykle nie robi niczego.";  
text.substr(25, 6);
```

"błędów"

## substring

Wyciągnięcie kawałka danego ciągu

`substring(indeksPoczątkowy, indeksKońcowy)`

```
var text = "Człowiek, który nie robi błędów,  
zwykle nie robi niczego. ";  
text.substring(25, 31);
```

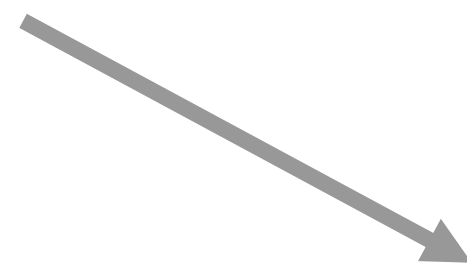
"błędów"

# Stringi metody

## split

Dzielenie ciągu na podstawie danego rozdzielnika.

```
var text = "Ka Boom! Bazinga";  
text.split(" ");
```

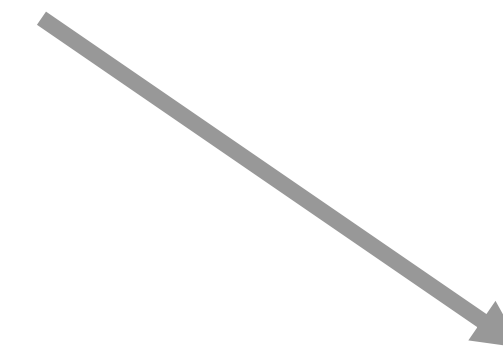


["Ka", "Boom!", "Bazinga"]

## trim

Usunięcie wszystkich białych znaków z początku i końca ciągu znaków.

```
var text = "  Lorem ipsum.  ";  
text.trim();
```



"Lorem ipsum"

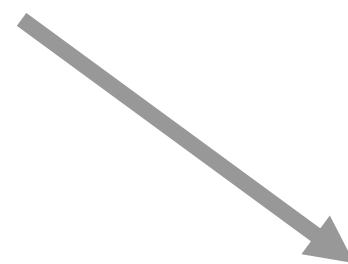


# Stringi metody

## toUpperCase

Zamiana wszystkich znaków na duże.

```
var text = "u mnie działa.";
text.toUpperCase();
```

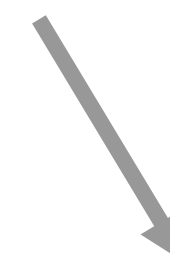


"U MNIE DZIAŁA"

## toLowerCase

Zamiana wszystkich znaków na małe.

```
var text = "CZAS START.";
text.toLowerCase();
```



"czas start"