

Wstęp do programowania

v3.1

Plan

1. Trochę teorii – przypomnienie
2. Schematy blokowe
3. Pierwszy skrypt
4. Typy danych
5. Trochę więcej o liczbach – Obiekt Math
6. Trochę więcej o stringach
7. Operatory
8. Kontrola przepływu programu
9. Tablice
10. Funkcje
11. Debugowanie
12. Stringi – metody

Trochę teorii – przypomnienie

Trochę teorii – przypomnienie

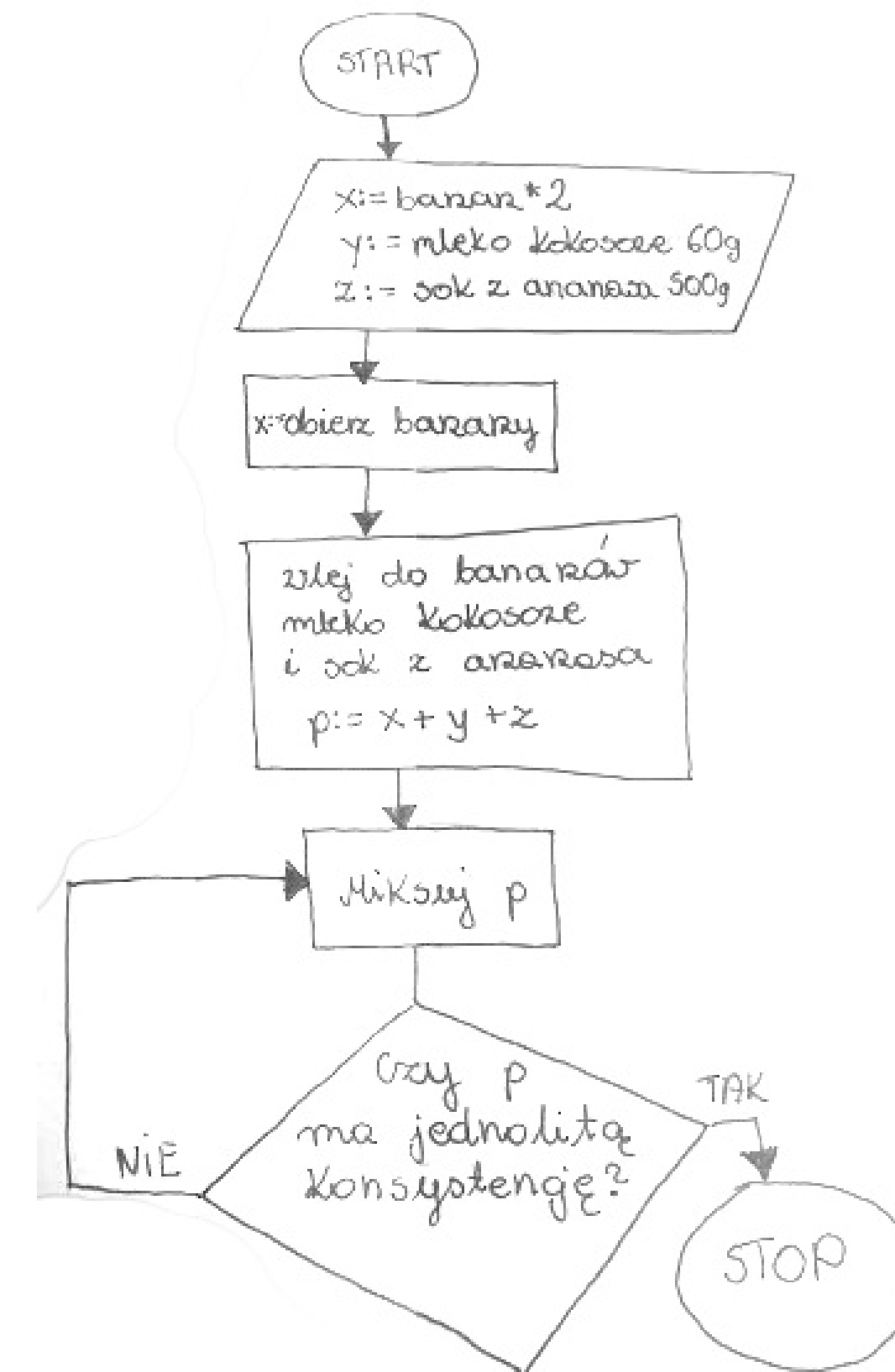
- Czym jest **algorytm**?
- Czym jest **pseudokod**?
- Jaka jest różnica między **językiem programowania** a **kodem źródłowym**?
- Czym jest **program**?
- **Język programowania**, a **język znaczników**. Jaka jest różnica?

Schematy blokowe

Schemat blokowy (flowchart)

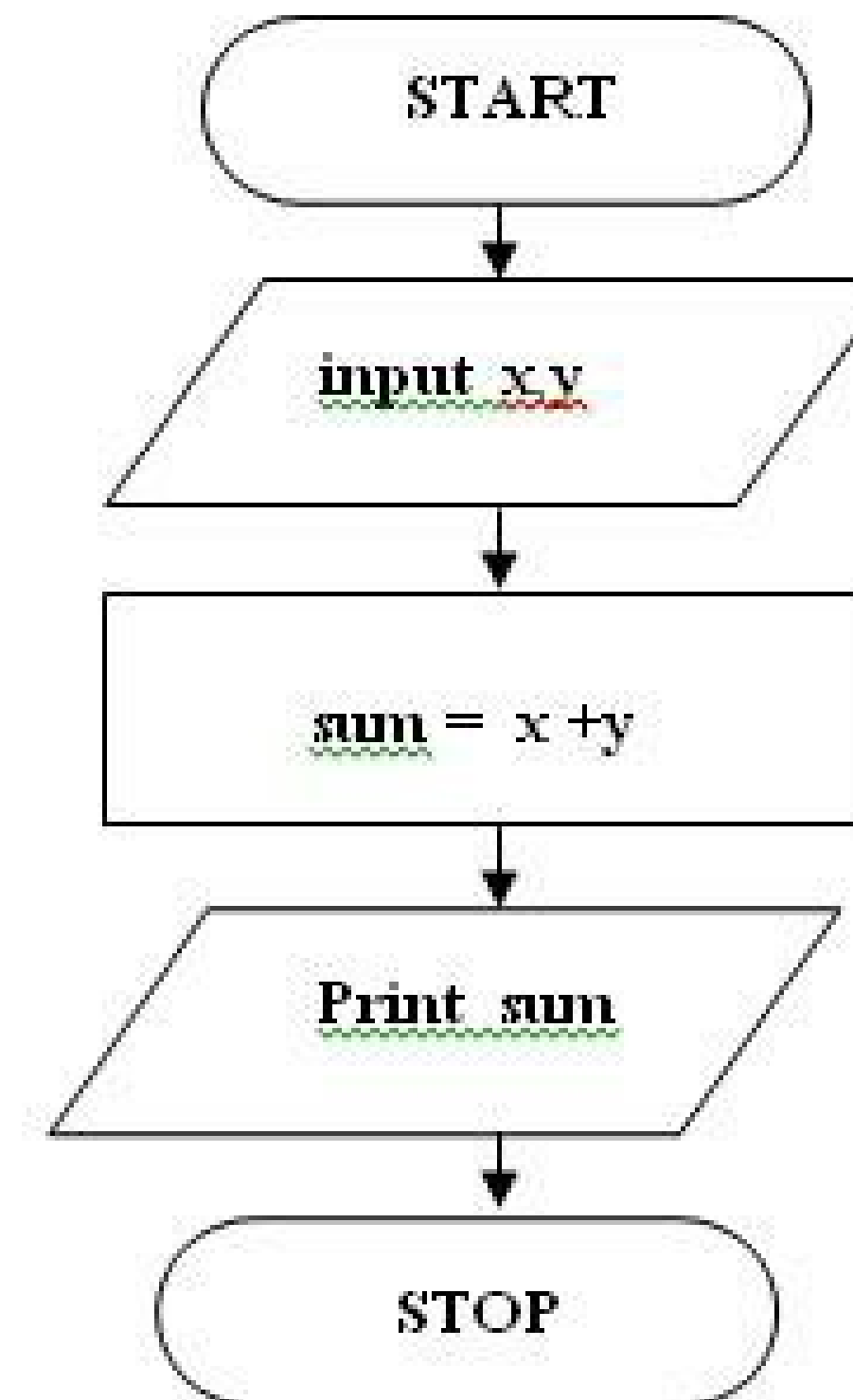
To narzędzie, za pomocą którego możemy pokazać czynności w **algorytmie**.

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

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:** Ctrl + 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

**VM143:2 Uncaught ReferenceError:
unknown_function is not defined**

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

**VM143:2 Uncaught ReferenceError:
unknown_function is not defined**

Ta linia kodu została wykonana.

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.

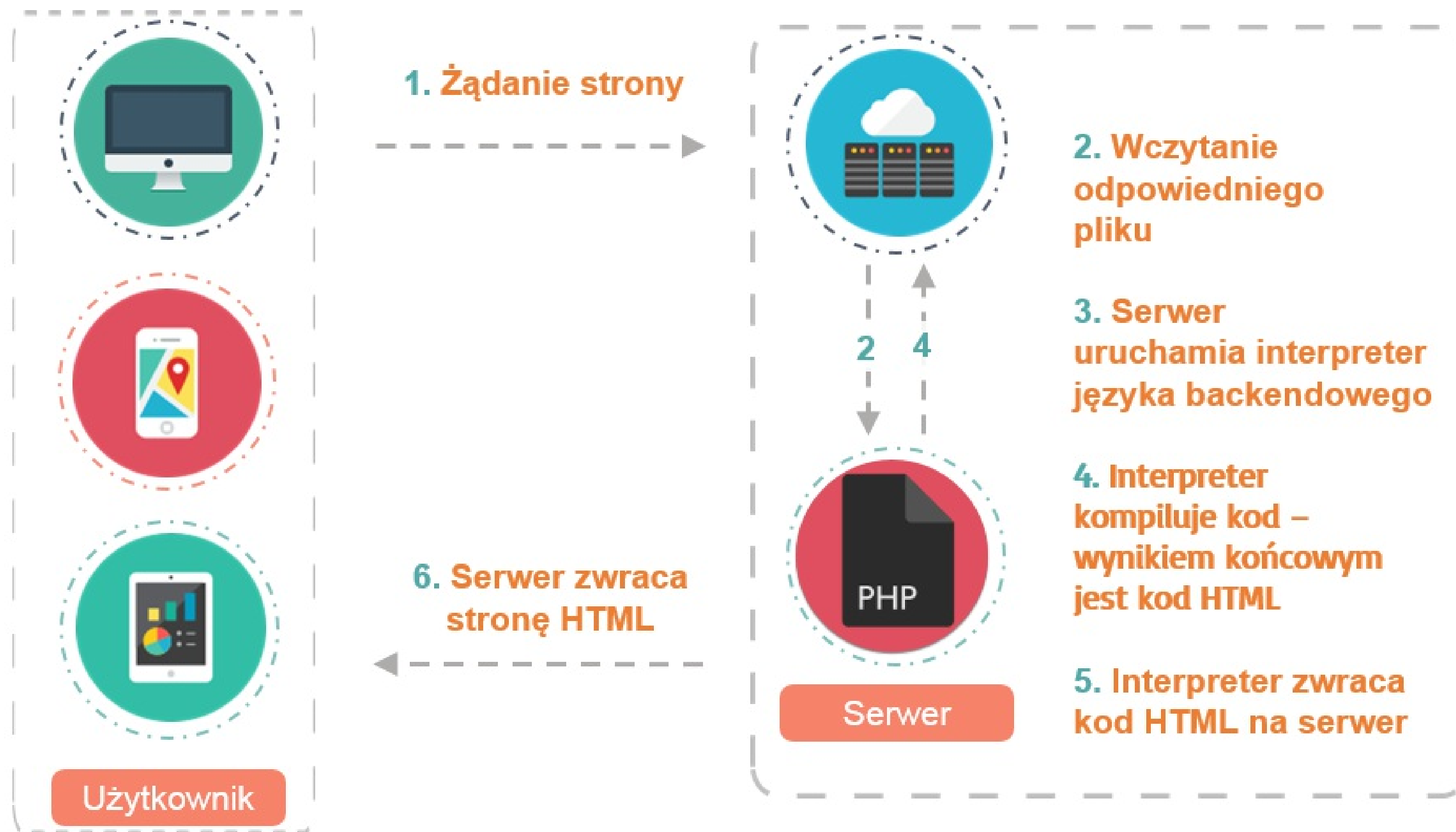
```
console.log("Hello world");  
unknown_function();
```

Hello world

**VM143:2 Uncaught ReferenceError:
unknown_function is not defined**

W tej linii jest błąd!

Jak działa serwer?



Gdzie jest w tym wszystkim JavaScript?

JavaScript jest używany w dwóch celach:

- jako język backendowy (np. Node.js),
- jako język kompilowany po stronie przeglądarki.



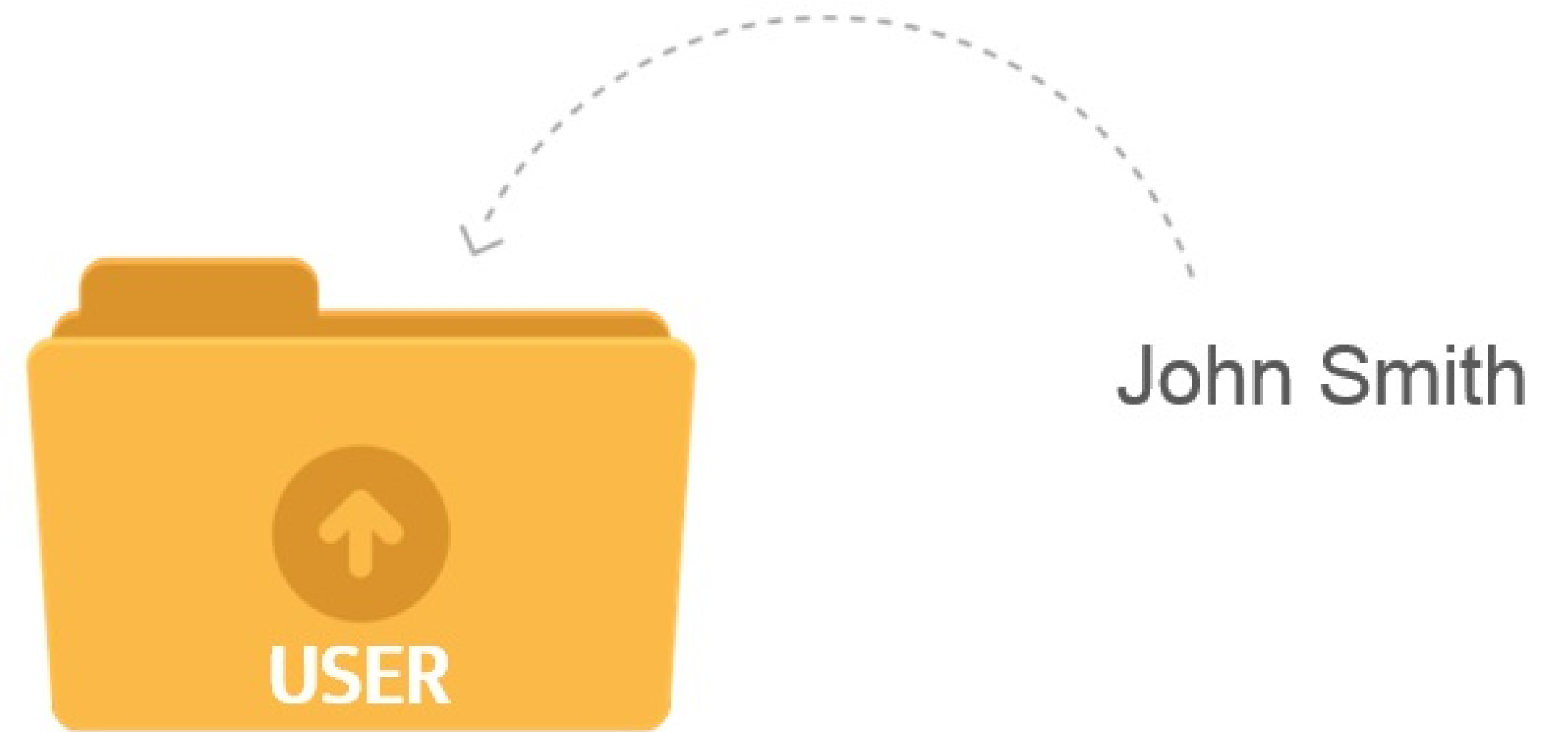
Jak działa JavaScript?

- 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

Typy danych – przypomnienie

- Czym jest zmienna?
- Co można przechowywać w zmiennych?
- Czy ta nazwa zmiennej jest prawidłowa: **28numbers**?
- Co oznacza słowo **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";
```

Wartości logiczne (Boolean)

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

Specjalne

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

Prymitywne typy danych

Obiekty

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

Tablice

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

Sprawdzanie typu – typeof

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

```
typeof null; /* "object" */
typeof 2;    /* "number" */
typeof "Ała 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

Trochę więcej o liczbach

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

- pierwiastkowanie,
- potęgowanie,
- zaokrąglanie.

```
var foo = 2.8;  
Math.ceil(foo); /* 3 */  
var foo = 2.8;  
Math.floor(foo); /* 2 */
```

Math.abs() – liczba absolutna

Math.ceil() – zaokrąglenie w górę

Math.floor() – zaokrąglenie w dół

Math.max() – wartość maksymalna ze zbioru

Math.min() – wartość minimalna ze zbioru

Math.pow() – potęgowanie

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

Math.round() – zaokrąglanie

Math.sqrt() – pierwiastkowanie

Trochę więcej o stringach

Stringi – przypomnienie

2 ≠ "2"

Zapamiętaj!

Stringi to nie to samo co liczby

Jak zamienić stringa na liczbę?

Number

Ta funkcja konwertuje stringi (i nie tylko) do liczb zmiennoprzecinkowych (float).

Number(string / wartość logiczna / obiekt)

parseInt

Tej funkcji używamy, gdy chcemy zamienić stringa na liczbę lub wyciągnąć ze stringa dane liczbowe. Nastąpi konwersja do liczb całkowitych (integer).

parseInt(string, system liczbowy 2–36)

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

Jak zamienić stringa na liczbę?

Przykład

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

Stringi – metody

Co możemy robić ze stringami?

W JavaScriptcie mamy do dyspozycji również specjalne metody, które pozwalają różnie modyfikować stringi, np.:

- wyciąganie ciągu znaków z całego napisu (stringa),
- usuwanie białych znaków,
- zamiana znaków na wielkie litery lub małe.

Przykład

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

Na końcu prezentacji znajdziesz najpopularniejsze metody dla stringów. Zapoznaj się z nimi samodzielnie
Stringi – metody.

Stringi – metody

Co możemy robić ze stringami?

W JavaScriptcie mamy do dyspozycji również specjalne metody, które pozwalają różnie modyfikować stringi, np.:

- wyciąganie ciągu znaków z całego napisu (stringa),
- usuwanie białych znaków,
- zamiana znaków na wielkie litery lub małe.

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

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 */
```

```
var text1 = "2";  
var liczba2 = 4;  
text1 + liczba2;      /* "24" */  
text1 - liczba2;      /* -2 */  
text1 / liczba2;      /* 0.5 */  
text1 * liczba2;      /* 8 */  
text1 % liczba2;      /* 2 */
```

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;**

```
var text1 = "2";  
var liczba2 = 4;  
text1 + liczba2;    /* "24" */  
text1 - liczba2;    /* -2 */  
text1 / liczba2;    /* 0.5 */  
text1 * liczba2;    /* 8 */  
text1 % liczba2;    /* 2 */
```

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 */
```

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 ciąg znaków na liczbę. Ale tylko wtedy, gdy może!

Przykład:

"2a1a" - 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 */
```

```
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 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)

```
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 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 (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:

- `=` – przypisz do zmiennej wartość,
- `++` – inkrementacja,
- `--` – dekrementacja.

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

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

Operatory połączone

```
var liczba3 = 1;
var text = "Ala ma kota";
liczba3 += 2;    /* 3 - inaczej liczba3 = liczba3 + 2 */
liczba3 -= 100; /* -97 - inaczej liczba3 = liczba3 - 100,
                  przed chwilą była równa 3, więc 3 - 100 = 97 */
liczba3 *= 2;    /* -194 inaczej liczba3 = liczba3 */
liczba3 /= 12;   /* -16.666 inaczej liczba3 = liczba3 / 12 */
liczba3 %= 2;    /* -0.16 inaczej liczba3 = liczba3 % 2 */
text += "a";     /* Ala ma kotaa */
```

Operatory połączone

```
var liczba3 = 1;
var text = "Ala ma kota";
liczba3 += 2;    /* 3 - inaczej liczba3 = liczba3 + 2 */
liczba3 -= 100; /* -97 - inaczej liczba3 = liczba3 - 100,
                  przed chwilą była równa 3, więc 3 - 100 = 97 */
liczba3 *= 2;    /* -194 inaczej liczba3 = liczba3 */
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, XOR

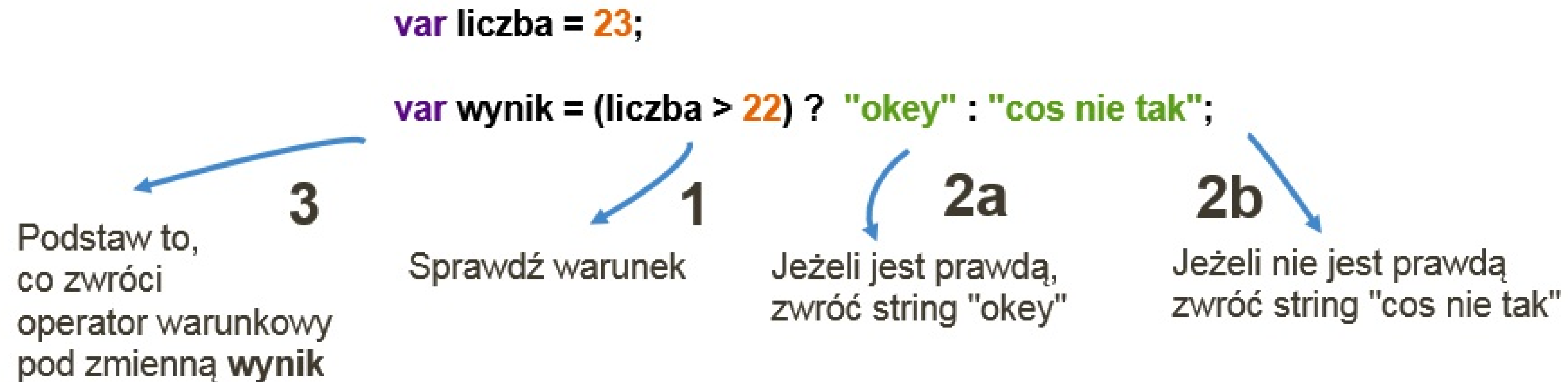
Operatory logiczne stosuje się w instrukcjach warunkowych:

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

- **! NOT** (logiczne **nie**)
Jeżeli warunek jest prawdą, zwróci **false** i na odwrót.
- **^ XOR** (logiczne **albo**)
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.



Kontrola przepływu programu

Instrukcje warunkowe – przypomnienie

If

```
var weather = "deszcz"; /* tę wartość można zmienić */
if (weather === "deszcz") {
    console.log("Weź parasol");
} else if (weather === "śnieg") {
    console.log("Weź czapkę");
} else {
    console.log("Weź okulary przeciwsłoneczne");
}
```

Instrukcje warunkowe – przypomnienie

switch

```
var weather = "deszcz"; /* tę wartość można zmienić */
switch(weather) {
  case "deszcz": {
    console.log("Weź parasol");
    break;
  }
  case "śnieg": {
    console.log("Weź czapkę");
    break;
  }
  default: {
    console.log("Weź okulary przeciwsłoneczne");
  }
}
```

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

Czas na zadania

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

Pętlę **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("Pętla są fajne");  
  i = Math.floor(Math.random() * 10);  
}
```

Pętlę **while** zazwyczaj wykonujemy, gdy 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.

Pętla 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);  
  }  
}
```

Druga pętla zaczyna się w każdej iteracji od innego, większego numeru. Pętla wewnętrzna jest zależna od pętli zewnętrznej.

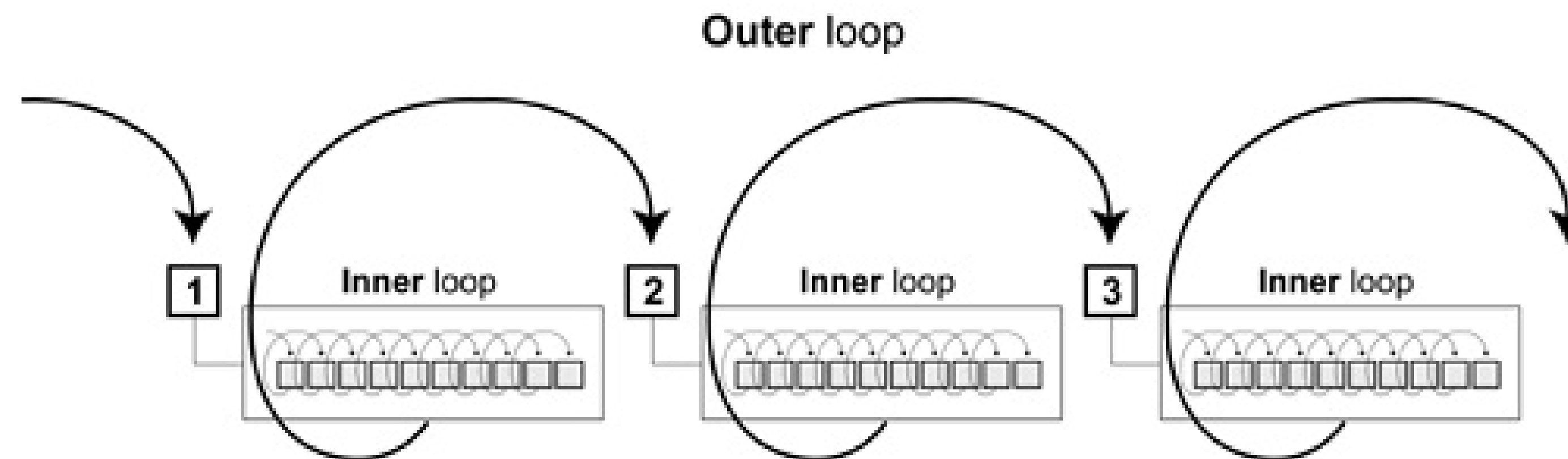
Pętla niezależna

```
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 od siebie.

Pętla for (podwójna)

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



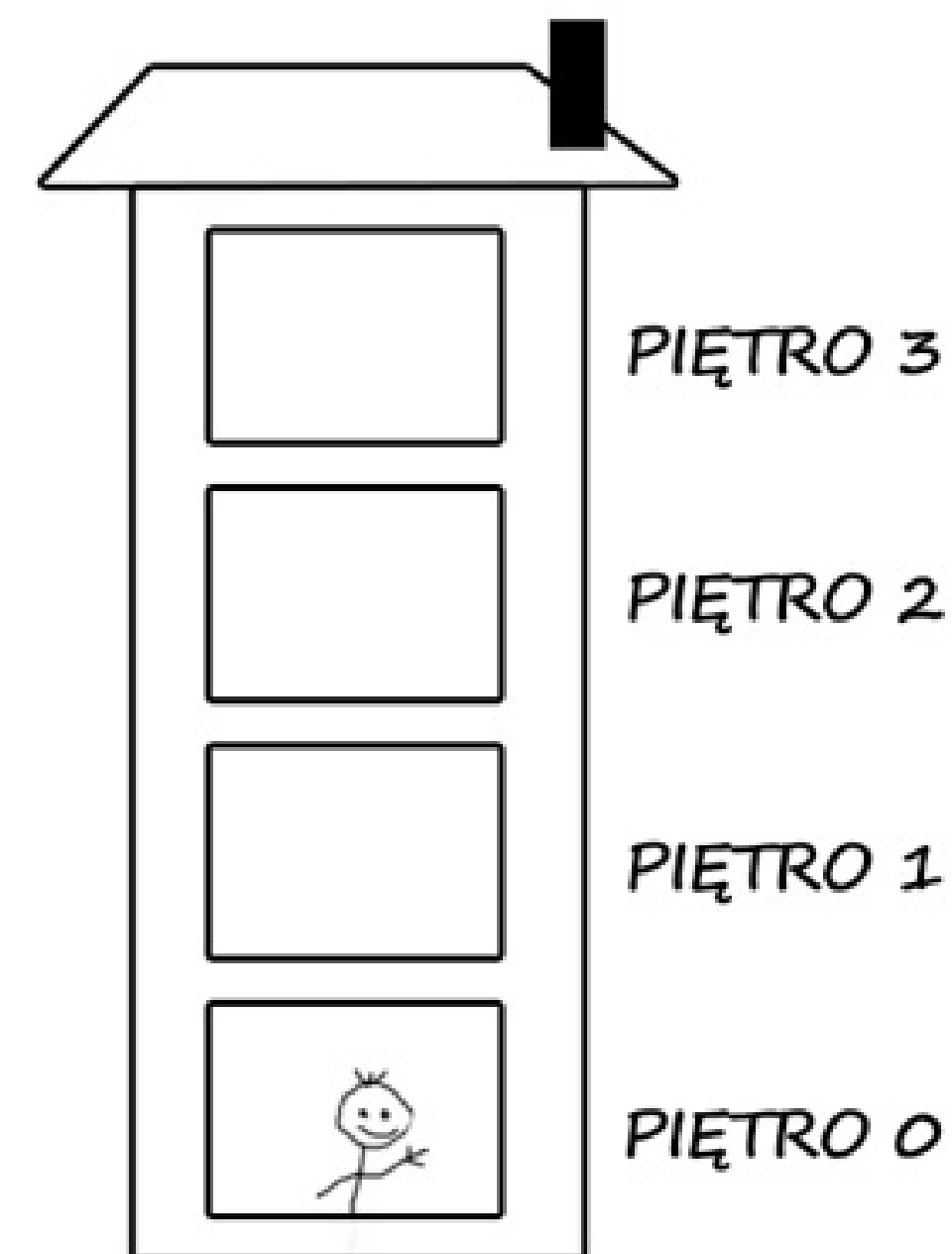
Czas na zadania

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

Funkcje

Funkcje

- Czym jest funkcja?
- Jak ją stworzyć?
- Jakie są sposoby na stworzenie funkcji?
- 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.

```
function obliczRownanie() {  
    //ciało funkcji  
}
```

```
function odpalRakiete() {  
    //ciało funkcji  
}
```

```
function zrobKanapke() {  
    //ciało funkcji  
}
```

Funkcja to odseparowany kawałek kodu wykonujący jakąś czynność.

function – słowo kluczowe.

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 **ciało funkcji**, czyli wszystkie czynności, które mają być 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");  
}  
sayHello();
```

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

Będziemy jednak używać funkcji bez nazwy tzw. **funkcji anonimowej**.

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 **ciało funkcji**, czyli wszystkie czynności, które mają być 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");  
}  
sayHello();
```

Definicja funkcji.

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

Będziemy jednak używać funkcji bez nazwy tzw. **funkcji anonimowej**.

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 **ciało funkcji**, czyli wszystkie czynności, które mają być 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");  
}  
sayHello();
```

Definicja funkcji.

Hello

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

Będziemy jednak używać funkcji bez nazwy tzw. **funkcji anonimowej**.

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**? Nie używamy jej. Usuńmy ją i mamy **anonimowe wyrażenie funkcyjne**:

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

Argumenty funkcji – wejście

Jak zrozumieć, na czym polega przekazywanie argumentów? Wyobraź sobie, że funkcja to wydzielony teren z wejściem i wyjściem.

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

W przykładzie obok przekazujemy do funkcji ciąg znaków.

Moglibyśmy przekazać również liczbę.

```
function getName(name) {  
  console.log(name + " Yeah");  
}  
getName("Ała");  
getName("Jan");  
getName("Marek");  
getName("Karol");
```

Po wywołaniu tej linii, co zostanie wyświetlone?

```
getName(23);
```


Argumenty funkcji – wejście

Jak zrozumieć, na czym polega przekazywanie argumentów? Wyobraź sobie, że funkcja to wydzielony teren z wejściem i wyjściem.

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

W przykładzie obok przekazujemy do funkcji ciąg znaków.

Moglibyśmy przekazać również liczbę.

```
function getName(name) {  
  console.log(name + " Yeah");  
}  
getName("Ała");  
getName("Jan");  
getName("Marek");  
getName("Karol");
```

Po wywołaniu funkcji, zostaną wyświetlone w konsoli imiona z doklejonym ciągiem znaków.

Po wywołaniu tej linii, co zostanie wyświetlone?

```
getName(23);
```

Argumenty funkcji – wejście (przypisanie)

Podczas przekazywania argumentów trzeba sobie zdawać sprawę z bardzo ważnej rzeczy. Mamy do czynienia z **przypisaniem wartości do zmiennej**.

Nie jest to oczywiste. Postaraj się to szczególnie zapamiętać.

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

Argumenty funkcji – wejście (przypisanie)

Podczas przekazywania argumentów trzeba sobie zdawać sprawę z bardzo ważnej rzeczy. Mamy do czynienia z **przypisaniem wartości do zmiennej**.

Nie jest to oczywiste. Postaraj się to szczególnie zapamiętać.

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

Wewnątrz funkcji
następuje przypisanie
var name = "Ała";

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

Pamiętaj

```
function getName(name) {  
    return name;  
    var name = "John";  
}  
console.log(getName("Kate"));
```

return – wyjście z funkcji

Return kończy działanie funkcji

```
function getName(name) {  
    return name + " & Leo";  
}  
getName("Kate");  
console.log(getName("Kate"));
```

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

Pamiętaj

```
function getName(name) {  
    return name;  
    var name = "John";  
}  
console.log(getName("Kate"));
```

return – wyjście z funkcji

Return kończy działanie funkcji

```
function getName(name) {  
    return name + " & Leo";  
}  
getName("Kate");  
console.log(getName("Kate"));
```

Nic się nie wypisze w konsoli po takim wywołaniu.

Pamiętaj

```
function getName(name) {  
    return name;  
    var name = "John";  
}  
console.log(getName("Kate"));
```

return – wyjście z funkcji

Return kończy działanie funkcji

```
function getName(name) {  
    return name + " & Leo";  
}  
getName("Kate");  
console.log(getName("Kate"));
```

"Kate & Leo"

Pamiętaj

```
function getName(name) {  
    return name;  
    var name = "John";  
}  
console.log(getName("Kate"));
```


return – wyjście z funkcji

Return kończy działanie funkcji

```
function getName(name) {  
    return name + " & Leo";  
}  
getName("Kate");  
console.log(getName("Kate"));
```

Pamiętaj

```
function getName(name) {  
    return name;  
    var name = "John";  
}  
console.log(getName("Kate"));
```

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

return – wyjście z funkcji

Return kończy działanie funkcji

```
function getName(name) {  
    return name + " & Leo";  
}  
getName("Kate");  
console.log(getName("Kate"));
```

Pamiętaj

```
function getName(name) {  
    return name;  
    var name = "John";  
}  
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** – jest to jedna z podstawowych zasad programowania.
- Zyskujemy na **przejrzystości** naszego kodu, gdyż wszystkie czynności mamy posegregowane pod odpowiednimi blokami kodu.
- Funkcje możemy **parametryzować**, tzn. przekazywać do nich różne dane i wykonywać na nich te same czynności.

- Funkcja może zwracać różne wartości.
- Używaj funkcji!



Poprawne tworzenie funkcji
i ponowne ich używanie
jest bardzo ważne
w programowaniu!

Czas na zadania

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 następuje lokalizacja błędu i jego naprawienie.

Drugi sposób to dodanie w naszym kodzie **console.log()** w odpowiednich miejscach, aby móc w 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 nasz kod jest dłuższy niż kilka linii, to musimy skorzystać z bardziej zaawansowanych narzędzi.

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

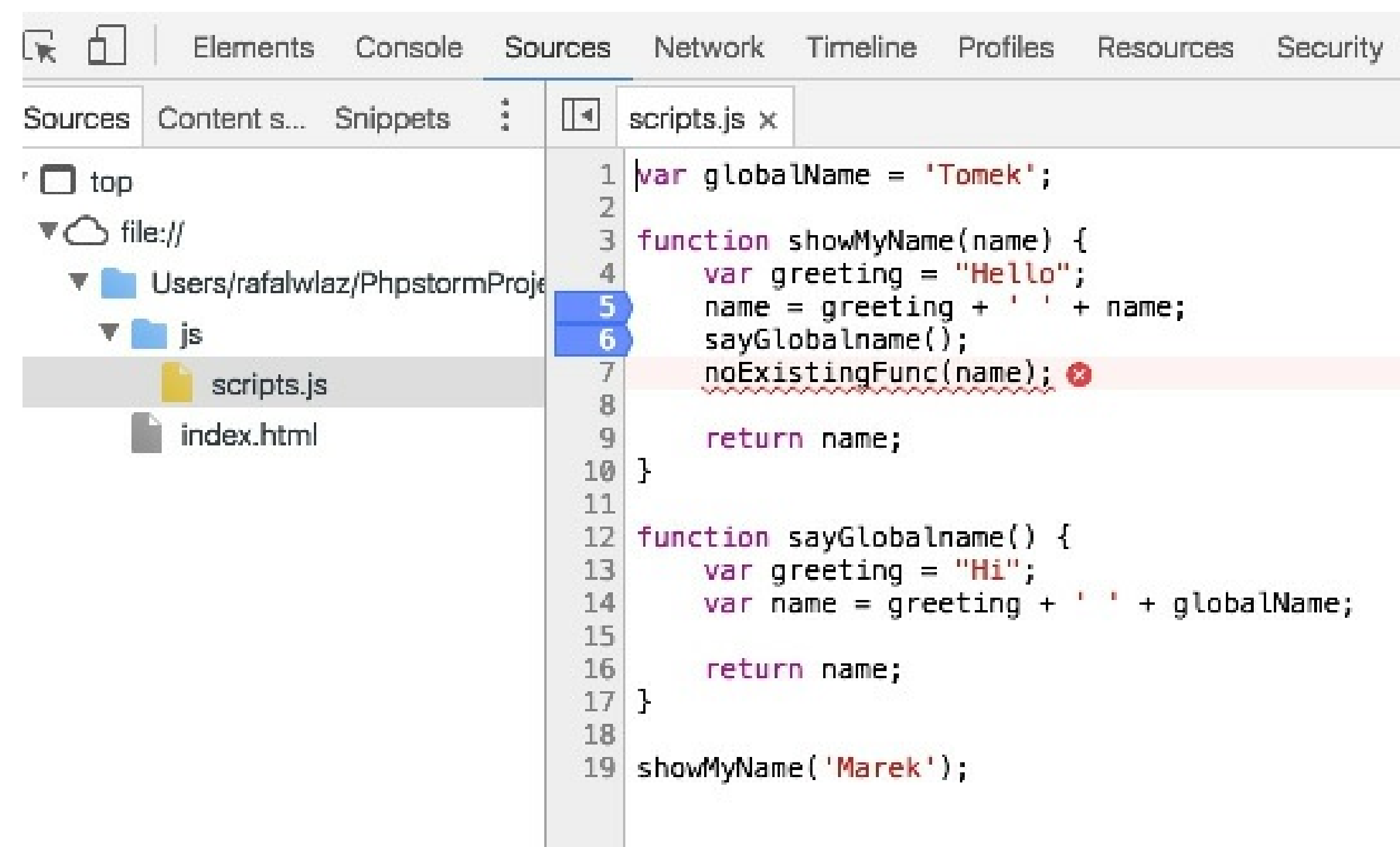
```
var globalName = 'Tomek';
function sayMyName(name) {
  var greeting = 'Hello';
  name = greeting + ' ' + name;
  debugger;
  return name;
}
function sayGlobalName() {
  var greeting = 'Hi';
  var name = greeting + ' ' + globalName;
  return name;
}
```


Debugowanie kodu JavaScript

Jak 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 miejsca, gdzie wykonywanie naszego kodu się zatrzyma. Możemy dodać ich dowolną liczbę.

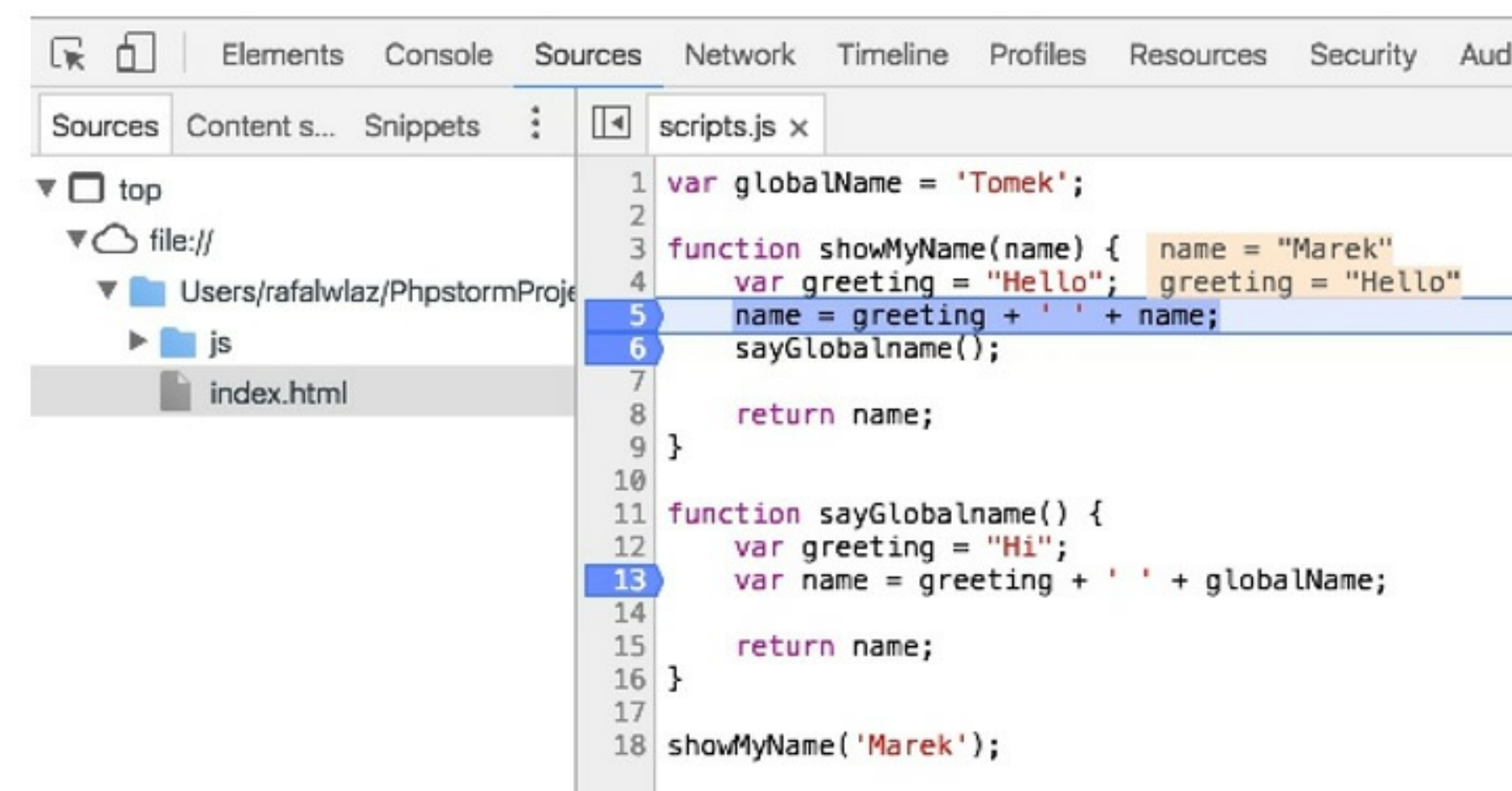
Breakpointy są zaznaczone na niebiesko.



Debugowanie kodu JavaScript

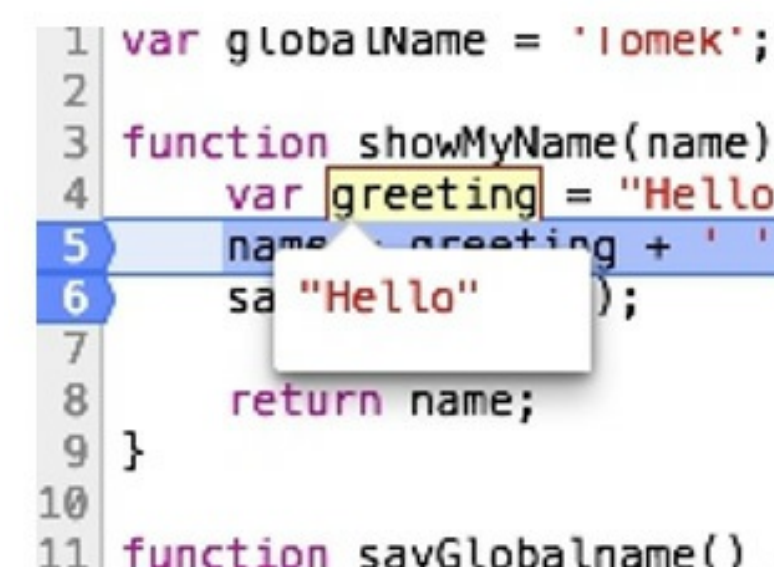
Po odświeżeniu strony, skrypt rozpocznie pracę w trybie debugowania i zatrzyma się w miejscu pierwszego breakpointu a aktualna linia zostanie podświetlona.

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



The screenshot shows the Chrome DevTools 'Sources' panel. On the left, a file tree shows 'index.html' selected. The main pane displays the contents of 'scripts.js'. A blue line indicates a breakpoint is set at line 5: `name = greeting + ' ' + name;`. The code includes a `showMyName` function and a `sayGlobalname` function. The current execution context is 'file:///Users/rafalwiaz/PhpstormProj....

```
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 breakpoint at line 5. A mouse cursor is hovering over the `greeting` variable in the expression `greeting + ' ' + name`. A tooltip bubble appears, displaying the current value of `greeting`, which is `"Hello"`.

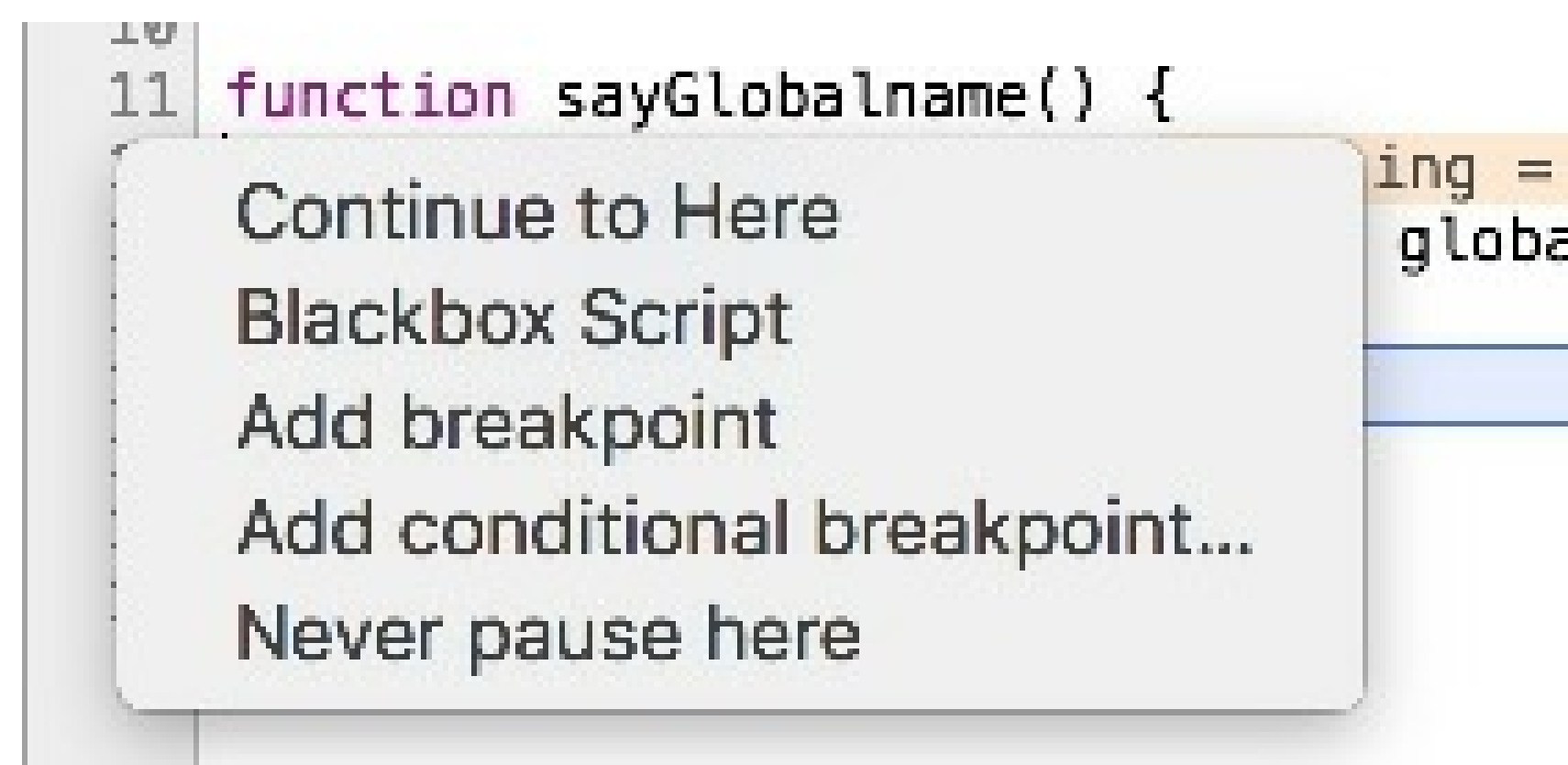
```
1 var globalName = 'Tomek';
2
3 function showMyName(name)
4   var greeting = "Hello"
5   name = greeting + ' '
6   sa "Hello" );
7
8   return name;
9 }
10
11 function savGlobalname()
```

Debugowanie kodu JavaScript

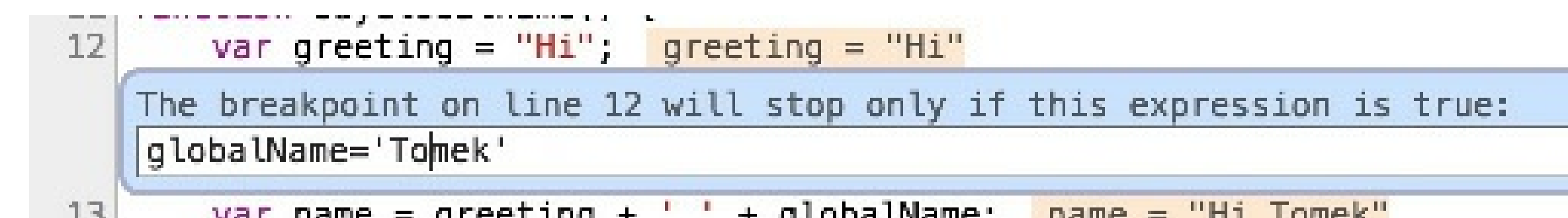
Nasze breakpointy mogą również zadziałać warunkowo tylko w określonej sytuacji.

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.

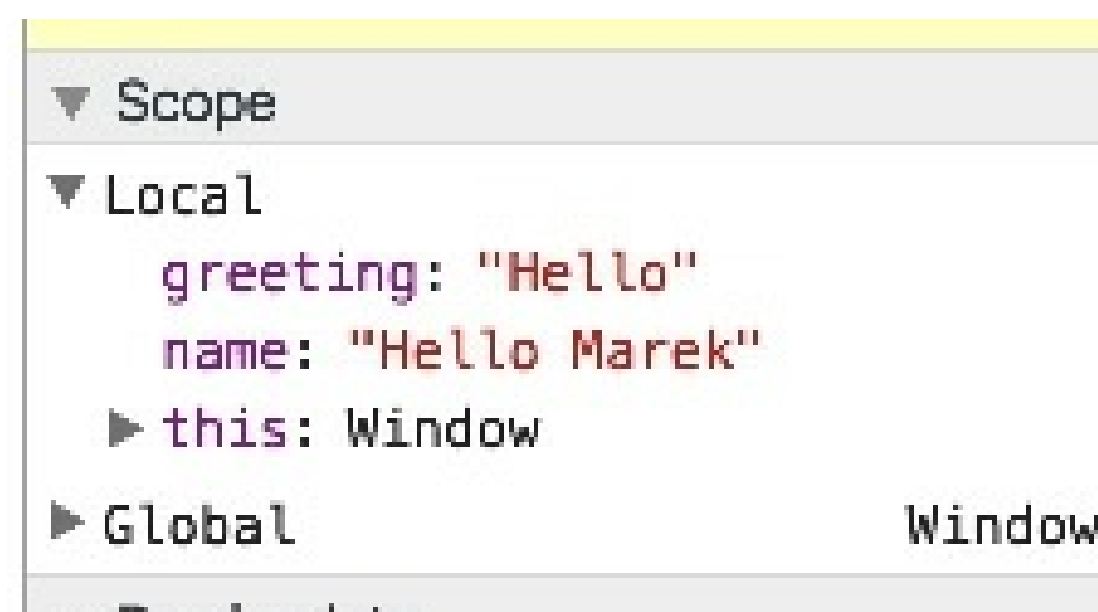
Możemy dodać breakpoint również bezpośrednio w kodzie, używając słowa kluczowego **debugger**.





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, zarówno w zakresie lokalnym (np. funkcji), jak i 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. Możemy więc 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.

Możemy też prześledzić, jak zmieniają się zmienne oraz sprawdzić, w jakiej kolejności kod jest wykonywany. Jest to 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

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.

Stringi – metody

- **`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);
```

concat()

Ta metoda łączy stringi, tak jak operator `+=`.

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

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);
```

stringi – metody

charAt()

Metoda zwracająca znak, który znajduje się w danym indeksie.

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

concat()

Ta metoda łączy stringi, tak jak operator `+=`.

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

"bigos z charakterem"

stringi – metody

indexOf()

Metoda zwracająca pierwszą pasującą pozycję wzorca w stringu.

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

lastIndexOf()

Zwraca ostatnią pasującą pozycję wzorca w stringu.

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

stringi – metody

indexOf()

Metoda zwracająca 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");
```

stringi – metody

indexOf()

Metoda zwracająca pierwszą pasującą pozycję wzorca w stringu.

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

lastIndexOf()

Zwraca ostatnią pasującą pozycję wzorca w stringu.

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

stringi – metody

replace()

Zamiana jednego ciągu znaków na drugi.

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

slice()

Wyciągnięcie kawałka danego ciągu.

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


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.

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

stringi – metody

replace()

Zamiana jednego ciągu znaków na drugi.

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

slice()

Wyciągnięcie kawałka danego ciągu.

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

"Myśl"

stringi – metody

substr()

Wyciągnięcie kawałka danego ciągu. Idź do znaku o numerze **indeksPoczątkowy** i weź następnie **długość** znaków, licząc od **indeksPoczątkowy**.

substr(indeksPoczątkowy, długość)

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

substring()

Wyciągnięcie kawałka danego ciągu. Idź do znaku o numerze **indeksPoczątkowy** i weź wszystkie znaki do **indeksKońcowy** (bez tego znaku).

**substring(indeksPoczątkowy,
indeksKońcowy)**

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

stringi – metody

substr()

Wyciągnięcie kawałka danego ciągu. Idź do znaku o numerze **indeksPoczątkowy** i weź następnie **długość** znaków, licząc od **indeksPoczątkowy**.

substr(indeksPoczątkowy, długość)

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

"błędów"

substring()

Wyciągnięcie kawałka danego ciągu. Idź do znaku o numerze **indeksPoczątkowy** i weź wszystkie znaki do **indeksKońcowy** (bez tego znaku).

**substring(indeksPoczątkowy,
indeksKońcowy)**

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

stringi – metody

substr()

Wyciągnięcie kawałka danego ciągu. Idź do znaku o numerze **indeksPoczątkowy** i weź następnie **długość** znaków, licząc od **indeksPoczątkowy**.

substr(indeksPoczątkowy, długość)

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

substring()

Wyciągnięcie kawałka danego ciągu. Idź do znaku o numerze **indeksPoczątkowy** i weź wszystkie znaki do **indeksKońcowy** (bez tego znaku).

**substring(indeksPoczątkowy,
indeksKońcowy)**

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

"błędów"

stringi – metody

split()

Dzielenie ciągu na podstawie danego rozdzielnika.

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

trim()

Usunięcie wszystkich białych znaków z początku i końca ciągu znaków.

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

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();
```

stringi – metody

split()

Dzielenie ciągu na podstawie danego rozdzielnika.

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

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 wielkie.

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

toLowerCase()

Zamiana wszystkich znaków na małe.

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

stringi – metody

toUpperCase()

Zamiana wszystkich znaków na wielkie.

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

"U MNIE DZIAŁA."

toLowerCase()

Zamiana wszystkich znaków na małe.

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

stringi – metody

toUpperCase()

Zamiana wszystkich znaków na wielkie.

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

toLowerCase()

Zamiana wszystkich znaków na małe.

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

"czas start."