

**WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA**
z siedzibą w Rzeszowie



Programowanie

Język JavaScript

Dr inż. Barbara Fryc

WPROWADZENIE

- ▶ Jest to język wysokiego poziomu (czyli programowanie polega na operowaniu wyższą abstrakcją a nie manipulacją w pamięci i bezpośrednio na procesorze).
- ▶ Kod wykonywany jest domyślnie w przeglądarce (czyli po stronie klienta).
- ▶ JavaScript to język interpretowany (program przed uruchomieniem nie musi być kompilowany) a jego specyfikacją jest ECMAScript.
- ▶ JavaScript wspiera *wieloparadygmatowość* – nie wymusza konkretnego stylu programowania i pozwala programować w sposób obiektowy i proceduralny.



WSTAWIAMY SKRYPT NA STRONĘ

▶ Znacznik `<script>`

- ▶ znacznik `<script>` możemy umieszczać wielokrotnie w dowolnym miejscu pliku HTML (najczęściej przed zamknięciem `</body>`)

```
<html>
<body>
Umieszczamy skrypt JS w kodzie strony
<script>
document.write("<B>Ten tekst został napisany dzięki JavaScript</B>")
</script>
</body>
</html>
```

- ▶ kod źródłowy skryptu można umieścić w osobnym pliku z rozszerzeniem `.js`

```
<script src=„skrypt.js"> </script>
```

- ▶ **Uwaga: skrypt wykonuje się od razu po załadowaniu dokumentu!**



ATRYBUTY ASYNC I DEFER

```
<script src="plik.js" async></script>
```

```
<script src="plik.js" defer></script>
```

- ▶ **Różnica** między tymi **atrybutami** jest też taka, że skrypty z **atrybutem defer** będą odpalane w kolejności w jakiej zostały wstawione do dokumentu. W przypadku **async** skrypty będą odpalane w kolejności "kto pierwszy ten lepszy", czyli który skrypt wczyta się wcześniej, ten zostanie wcześniej odpalony.



DEBUGOWANIE KODU W JAVASCRIPT

- ▶ Debugger (czyli narzędzie, które pokazuje błędy w kodzie) to konsola w przeglądarce internetowej, którą w Google Chrome uruchamiamy klawiszem F12.

```
console.log("Witaj w debuggerze");
```



DEBUGOWANIE KODU W JAVASCRIPT

- ▶ Instrukcja `console.log()` jest bardzo przydatna w debugowaniu. Słowo `console` to obiekt a `log` to metoda tego obiektu. Istnieją jeszcze metody `error` i `warn`:

```
console.error("To jest błąd");
```

- ▶ która pokaże nam napis wyróżniony na czerwono, oraz:

```
console.warn("To jest ostrzeżenie");
```


- ▶ która pokaże nam napis wyróżniony na żółto. Obie instrukcje podadzą także numer linii kodu w której zostały wywołane.




DOBRA RADA

- ▶ Starsze przeglądarki mogą sobie nie radzić z JS - dlatego stosuje się sztuczkę
 - ▶ stara przeglądarka nie rozpozna znacznika `<script>` ale zobaczy komentarze `<!--`, więc zignoruje cały kod między tymi znacznikami.
 - ▶ nowa przeglądarka zobaczy znak komentarza `<!--` ale dla JS znak ten oznacza komentarz tylko w 1 linii, więc tę linię zignoruje, ale wykona kod JS będący w liniach niżej.
 - ▶ Zamykający znacznik komentarza `-->` jest dodatkowo komentarzem JS (`//`), bo inaczej byłby błędem dla JS, a musi on być po to, aby stara przeglądarka rozpoznała gdzie kończy się

```
<script>
<!--
var zmienna="Java Script";
alert("Witaj"+zmienna);
//-->
</script>
```



Podstawowe elementy języka JavaScript



ZMIENNE

- ▶ nie wymagają definiowania ani przydzielania konkretnego typu
- ▶ notacja camelCase czyli mojaPierwszaZmienna
- ▶ zasady:
 - ▶ nazwa zmiennej nie może zaczynać się od cyfry,
 - ▶ nazwa zmiennej nie może zawierać spacji (można zamiast spacji używać podkreślenia),
 - ▶ nazwą zmiennej nie może być słowem kluczowym zarezerwowanym przez JavaScript



DEKLAROWANIE ZMIENNYCH I STAŁYCH

▶ var, let

```
var mojaZmienna = „moja pierwsza zmienna”;
```

▶ const

```
const mojaStała = „moja pierwsza stała”;
```

▶ var vs const

```
mojaZmienna = „moja druga zmienna”;  
mojaStała = „moja druga stała” //błąd - nie można zmieniać stałej
```



LOKALNE VS GLOBALNE

```
function getArea(width, height) {  
  var area = width * height;  
  return area;  
}
```

```
var wallSize = getArea(2, 3);  
Document.write(wallSize);
```

- ▶ zakres lokalny (na poziomie funkcji)
- ▶ zakres globalny



Operatory

Operator	Działanie operatora
+	dodawanie liczb oraz klejenie łańcuchów (konkatenacja)
-	odejmowanie liczb
*	mnożenie liczb
/	dzielenie liczb
++	inkrementacja (zwiększenie liczby o dokładnie 1)
--	dekrementacja (zmniejszenie liczby o dokładnie 1)
%	reszta z dzielenia (tzw. modulo), na przykład: $27 \% 6$ jest równe 3, gdyż cztery szóstki mieszczą się w 27 (co daje 24 i reszty zostaje 3)
=	przypisanie wartości do zmiennej
!	zaprzeczenie (negacja)
==	porównanie wartości

Operatory

Operator	Działanie operatora
===	porównanie wartości i typu zmiennych
<	mniejsze od
<=	mniejsze lub równe od
>	większe od
>=	większe lub równe od
&&	logiczne "i" (and)
	logiczne "lub" (or)
+=	skrótowy zapis, np. <code>a += b</code> odpowiada: <code>a = a + b</code>
-=	skrótowy zapis, np. <code>a -= b</code> odpowiada: <code>a = a - b</code>
*=	skrótowy zapis, np. <code>a *= b</code> odpowiada: <code>a = a * b</code>
/=	skrótowy zapis, np. <code>a /= b</code> odpowiada: <code>a = a / b</code>

OPERATORY BITOWE

► Operator &

```
const a = 5;           // 00000000000000000000000000000101
const b = 3;           // 00000000000000000000000000000011
console.log(a & b);    // 00000000000000000000000000000001
// Wynik: 1
```

► Operator |

```
const a = 5;           // 00000000000000000000000000000101
const b = 3;           // 00000000000000000000000000000011
console.log(a | b);    // 00000000000000000000000000000111
// Wynik: 7
```

► Operator ^ (XOR)

```
const a = 5;           // 00000000000000000000000000000101
const b = 3;           // 00000000000000000000000000000011
console.log(a ^ b);    // 00000000000000000000000000000110
// Wynik: 6
```

► Operator ~

```
const a = 5;           // 00000000000000000000000000000101
const b = -3;          // 11111111111111111111111111111101
console.log(~a);       // 11111111111111111111111111111010 // Wynik: -6
console.log(~b);       // 00000000000000000000000000000010 // Wynik: 2
```

INSTRUKCJE WARUNKOWE

```
<script>
const mojaStala = 8;
if (mojaStala === 10) {
    //ten kawałek kodu się nie
    wykona
}

if (mojaStala <= 10) {
    //ten kawałek kodu się
    wykona
}

if (mojaStala != 8) {
    //ten kawałek kodu się nie
    wykona
}
</script>
```

```
<script>
const mojaZmienna1 = 5;
const mojaZmienna2 = 15;

if (mojaZmienna1 === 5 && mojaZmienna2 === 10) {
    //ten kawałek kodu się nie wykona bo mamy
    "i", oba muszą być spełnione
}

if (mojaZmienna1 === 5 || mojaZmienna2 === 10) {
    //ten kawałek się wykona bo mamy "lub"
}

if (mojaZmienna1 === 5 ^ mojaZmienna2 === 15) {
    //ten kawałek się nie wykona bo mamy "xor",
    a oba są spełnione
}

if (!(mojaZmienna1 === 5)) {
    //ten kawałek się nie wykona, bo mamy
    negację! i mojaZmienna1 !== 5
}

if ((mojaZmienna1 === 2) || 1) {
    //ten kawałek zawsze się wykona, bo
    mojaZmienna1 nie jest 2, a 1 jest true
}</script>
```



SWITCH

```
godz=(prompt("ile godzin spędzasz na  
czacie?"));  
switch(godz)  
{  
    case 0:  napis="Nie wiesz co to jest?";  
break;  
    case 1:  napis="To w zupełności  
wystarczy.";    break;  
    case 2:  napis="Jesteś maniakiem.";  
break;  
    default: napis="Powinieneś się leczyć";  
break;  
}
```



PETLE

```
for (var i = 0; i < 10; i++) {  
    document.write(i);  
}
```

```
while (i < 10) {  
    document.write(i);  
    i++;  
}
```

```
do {  
    document.write(i);  
    i++;  
} while (i < 10)
```



OKNA DIALOGOWE: ALERT, PROMPT, CONFIRM

Alert

- ▶ informacja dla użytkownika, wyposażona jedynie w przycisk “OK”, służy do wyprowadzania danych (instrukcja wyjścia).

```
alert("Wyskakujące okienko!");
```

Confirm

- ▶ (ang. "potwierdzenie") – to okno dialogowe wyposażone jest w dwa przyciski: (“OK” oraz “Anuluj”), co w kombinacji z instrukcją warunkową if może pozwolić nam poprosić użytkownika o podjęcie decyzji

```
if (confirm("Podejmij decyzję!")) {  
    alert("Wybrano opcję: OK");  
} else {  
    alert("Wybrano opcję: Anuluj");  
}
```

OKNA DIALOGOWE: ALERT, PROMPT, CONFIRM

Prompt

- ▶ okno, które może posłużyć do wprowadzania danych (instrukcja wejścia) – oczywiście w praktyce lepiej jest użyć pól edycyjnych, ale istnieje możliwość pobrania wartości także w oknie dialogowym. Drugi argument funkcji **prompt()** (w przykładzie poniżej słowo “Adam”) to tzw. placeholder, czyli wartość domyślna (od razu będzie ona zaznaczona, tak aby rozpoczęcie pisania usunęło ją z okna dialogowego).

```
var imie = prompt("Podaj imię", „Barbara");  
document.write("Twoje imię: "+imie);
```

Zadania

Wykorzystując instrukcję **prompt**:

- ▶ Napisz skrypt, w którym wczytasz dwie liczby i wyświetlisz ich sumę.
- ▶ Napisz skrypt, w którym wczytasz liczbę i sprawdzisz, czy jest parzysta, czy nieparzysta.
- ▶ Napisz skrypt, w którym wczytasz dwie liczby i operator. Dla operatorów $+$, $-$, $*$, $/$ wyświetlisz prawidłowy wynik, dla pozostałych informację o błędnym operatorze.



Typy danych

Typy prymitywne

- ▶ Number
- ▶ String
- ▶ BigInt
- ▶ Boolean
- ▶ Undefined
- ▶ Null
- ▶ Symbol

Typy złożone (referencyjne)

- ▶ Object (w tym Array, Map i Set)

Wszystko co nie jest typem prymitywnym jest obiektem.



Typy prymitywne

```
// Typy liczbowe
let a = 10;
let b = 3.14;
let c = -100;

// Typ logiczny
let d = true;
let e = false;

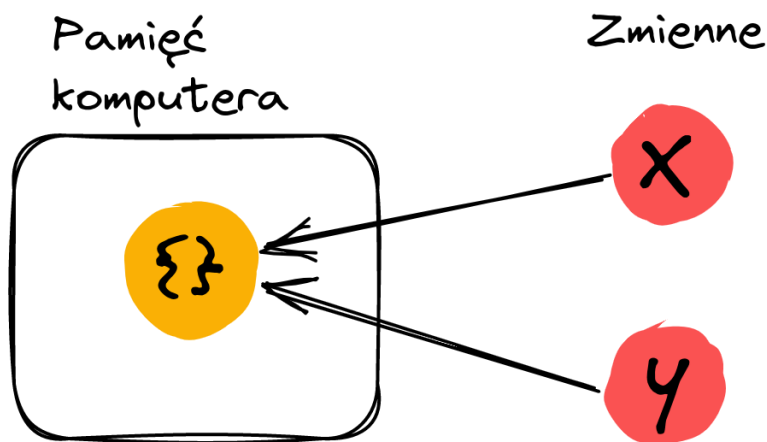
// Typy tekstowe
let f = "hello world";
let g = "JavaScript";

// Typy specjalne
let h = null;
let i; // undefined

console.log(typeof a); // "number"
console.log(typeof b); // "number"
console.log(typeof c); // "number"
console.log(typeof d); // "boolean"
console.log(typeof e); // "boolean"
console.log(typeof f); // "string"
console.log(typeof g); // "string"
console.log(typeof h); // "object"
console.log(typeof i); // "undefined"
```

Typy złożone

- ▶ **Typy złożone to takie, które są przechowywane w pamięci komputera w sposób pośredni.** Oznacza to, że zmienna zawierająca typ złożony **nie przechowuje wartości bezpośrednio**, ale przechowuje informację o miejscu, w którym ta wartość jest przechowywana w pamięci komputera. Dzięki temu możemy mieć **wiele różnych zmiennych, które wskazują na tę samą wartość**.



Zmienne wskazują na ten sam obiekt

Typy złożone

- ▶ **Typy referencyjne są bardzo przydatne** w języku JavaScript, ponieważ pozwalają na przechowywanie dużych i złożonych danych oraz na współdzielenie tych danych pomiędzy różnymi zmiennymi i funkcjami. Dzięki temu możemy uniknąć tworzenia wielu kopii tych samych danych, co znacznie zwiększa wydajność naszej aplikacji.

```
let x = [1, 2, 3];  
let y = x;  
  
console.log(x); // [1, 2, 3]  
console.log(y); // [1, 2, 3]
```

```
x.push(4);  
  
console.log(x); // [1, 2, 3, 4]  
console.log(y); // [1, 2, 3, 4]
```



Typy mutowalne

- ▶ Typy referencyjne **to również typy mutowalne**, co oznacza, że możemy zmieniać ich wartości w **dowolnym momencie**. Na przykład jeśli mamy zmienną `x` zawierającą tablicę, to możemy w dowolnym momencie **dodawać nowe elementy** do tej tablicy:

```
let x = [1, 2, 3];  
console.log(x); // [1, 2, 3]  
  
x.push(4);  
console.log(x); // [1, 2, 3, 4]  
  
x.push(5);  
console.log(x); // [1, 2, 3, 4, 5]
```



Obiekty

- ▶ **Typy referencyjne to także obiekty**, które są szczególnym rodzajem typów referencyjnych w języku JavaScript. **Obiekty są złożonymi strukturami danych**, które pozwalają na przechowywanie wielu różnych wartości w strukturze klucz-wartość. Na przykład możemy utworzyć obiekt zawierający informacje o osobie:

```
let person = {  
  firstName: "Mateusz",  
  lastName: "Morawski",  
  age: 50  
};  
  
console.log(person.firstName); // Mateusz  
console.log(person.lastName); // Morawski  
console.log(person.age); // 50
```

- ▶ Możemy również zmieniać wartości w obiekcie w dowolnym momencie:

```
person.age = 51;  
console.log(person.age); // 51
```

Typowanie dynamiczne

- ▶ Typowanie dynamiczne oznacza to, że **przy deklarowaniu zmiennej nie musimy określać jej typu, a w przeciągu działania aplikacji może się on kilkukrotnie zmienić.** Działanie takie najczęściej nie jest pożądane, ponieważ może prowadzić do nieprzewidzianych błędów.

```
// Deklaracja zmiennej bez określenia jej typu
let dynamicValue;

// Przypisanie liczby do zmiennej (typ number)
dynamicValue = 10;

// Wyświetlenie typu zmiennej
console.log(typeof dynamicValue); // "number"

// Przypisanie ciągu znaków do zmiennej (typ string)
dynamicValue = "Hello, world!";

// Wyświetlenie typu zmiennej
console.log(typeof dynamicValue); // "string"
```

Liczby w JavaScript

- ▶ W przypadku Javascript do zapisu liczb w pamięci wykorzystywany jest używany powszechnie standard IEEE_754, który charakteryzuje się tym, że przy operacjach na liczbach o bardzo małej precyzji, zaokrąglane są one do najbliższej możliwej do zapisu w systemie dwójkowym liczby. Obliczenia mogą być niedokładne:

```
console.log( 0.1 + 0.2 ); //0.30000000000000004
```

- ▶ **Jak zapisać liczbę ze stałą liczbą miejsc po przecinku?**

```
(1.2).toFixed();      // "1"  
(1.5).toFixed();      // "2"  
(1.25).toFixed(1);    // "1.3"  
(1.2).toFixed(3);     // "1.200"  
(-12.3).toFixed(3);   // "-12.300"  
NaN.toFixed();        // "NaN"  
Infinity.toFixed();    // "Infinity"
```



Liczby w JavaScript

► Duże liczby

```
const nr1 = 1e6; //1 * 1000000  
const nr2 = 2e5; //2 * 100000
```

► Małe liczby

```
const nr1 = 1e-5; //5 zer na lewo od liczby => 0.00001  
const nr2 = 2e-3; //3 zera na lewo 0.002
```

► Liczby w różnych formatach

```
//szesnastkowy - znany z kolorów CSS
```

```
console.log(0xFF); //255
```

```
console.log(0x66); //102
```

```
//ósemkowy
```

```
const nr1 = 0o377; //255
```

```
//dwójkowy
```

```
const nr2 = 0b11111111; //255
```

```
console.log(0xFF === 0o377, 0o377 === 0b11111111, 0xFF === 0b11111111); //true, true, true
```

Funkcja Math

Math.abs(liczba)	- zwraca wartość bezwzględną (absolutną) liczby
Math.ceil(liczba)	- zwraca najmniejszą liczbę całkowitą, większą lub równą podanej liczbie
Math.floor(liczba)	- zwraca największą liczbę całkowitą mniejszą lub równą podanej liczbie
Math.max(liczba1, liczba2, liczba3...)	- zwraca największą z przekazanych wartości
Math.min(liczba1, liczba2, liczba3...)	- zwraca najmniejszą z przekazanych wartości
Math.pow(liczba1, liczba2)	- zwraca wartość liczby1 podniesionej do potęgi liczby2
Math.random()	- zwraca wartość pseudolosową z przedziału 0 - 1
Math.round(liczba)	- zwraca zaokrąglenie danej liczby do najbliższej liczby całkowitej
Math.sqrt(liczba)	- zwraca pierwiastek kwadratowy liczby



Funkcja Math

```
const var1 = 56.5;
const var2 = 74.3;

Math.min(var1, var2) //56.5
Math.max(var1, var2) //74.3
Math.max(1,3,6,2) //6

Math.abs(-1) //1

Math.round(var1) //57
Math.round(20.52) //21
Math.round(-10.21) //-10
Math.round(-11.82) //-12

Math.floor(var1) //56
Math.floor(20.52) //20
Math.floor(-10.21) //-11
Math.floor(-11.82) //-12

Math.ceil(var1) //57
Math.ceil(20.52) //21
```



Losowa liczba z przedziału

```
const min = 3;  
const max = 7;  
  
const result = Math.floor(Math.random() * (max-min+1) + min);
```



Stringi

► Definiowanie tekstu

```
const text = "Ala ma kota, a kot ma Ale."; //podwójne cudzysłowy  
  
//lub  
  
const text = 'Ala ma kota'; //pojedyncze apostrofy
```

► Długość tekstu

```
const text = "Ala";  
text.length; //3  
  
"Koty i psy są fajne".length //19
```



Poruszanie się po tekście

```
1  const txt = "abecadłŃo";
2
3  for (let i=0; i<txt.length; i++) {
4      console.log(txt[i]);
5  }
6
7  for (const el of txt) {
8      console.log(el);
9  }
```



Stringi

- ▶ Typy proste (w tym String) nie są mutowalne, stąd nie możemy w nich bezpośrednio podmieniać liter.

```
const txt = "Ala ma kota";  
txt[0] = "E";  
console.log(txt); //"Ala ma kota"
```

- ▶ Żeby podmienić literę w tekście, musimy wykorzystać dowolny sposób, który zwróci nam nowy tekst. Może to być zastosowanie **replace()**, **substring()**, czy inny sposób:

```
const txt = "Ala ma kota";  
const tab = [...txt];  
tab[0] = "E";  
const newText = tab.join("");  
console.log(newText); //"Ela ma kota"
```



Funkcja replace()

```
let text = "Ala ma kota"
let nowy = [...text]
console.log(nowy)
nowy[0] = 'E'
console.log(text)
nowy = nowy.join("")
console.log(nowy)
let nowy2 = text.replace(/a/g, "e")
console.log(nowy2)
```

// /a/g to wyrażenie regularne, które jest używane do wyszukiwania
//wszystkich wystąpień litery "a" w danym ciągu znaków.



Łamanie tekstu

//poprzez operator przypisania

```
let text = "Stoi na stacji lokomotywa,<br>";  
text += "Ciężka, ogromna i pot z niej spływa:<br>";  
text += "Tłusta oliwa.";
```

//poprzez dodawanie części tekstu

```
let text = "Stoi na stacji lokomotywa,<br>"  
+ "Ciężka, ogromna i pot z niej spływa:<br>"  
+ "Tłusta oliwa."
```

//Poprzez zastosowanie znaku backslash na końcu linii

```
let text = "Stoi na stacji lokomotywa,<br>\  
Ciężka, ogromna i pot z niej spływa:<br>\  
Tłusta oliwa.\  
"
```

//Najlepsza metoda - użycie template strings

```
let text = `Stoi na stacji lokomotywa,<br>  
Ciężka, ogromna i pot z niej spływa:<br>  
Tłusta oliwa.  
`
```



Dodawanie elementu do tekstu

```
const age = 10;
```

```
const text = "Ten pies ma " + age + " lat";
```

```
const text = 'Ten kot ma ' + age + ' lat';
```

```
const text = `Ten chomik ma ${age} lat`;
```

```
const a = 112;
```

```
const b = 120;
```

```
const text = "Cena produktu A to " + a + "zł, cena produktu B to " + b + "zł, a suma to "
```

```
const text = `Cena produktu A to ${a}zł, cena produktu B to ${b}zł, a suma to ${a+b}zł`;
```



Funkcje na tekście

► Pobieranie znaku na danej pozycji

```
const text = "Ala ma kota, a kot ma Ale";

console.log(text.charAt(0)); //A
console.log(text.charAt(4)); //m

console.log(text[0]); //A
console.log(text[4]); //m

console.log(text.charAt(text.length-1)); //e
console.log(text[text.length-1]); //e
```



Funkcje na tekście

- ▶ Funkcje **toUpperCase()** i **toLowerCase()** służą odpowiednio do zamieniania tekstu na duże i małe litery.

```
1  const text = "Ala ma kota";  
2  
3  console.log(text.toUpperCase()); //"ALA MA KOTA"  
4  console.log(text.toLowerCase()); //"ala ma kota"
```

- ▶ Funkcja **indexOf** zwraca pozycję szukanego fragmentu w tekście (ale także w tablicy, bo funkcja ta dostępna jest dla stringów i tablic). Wynik **-1** oznacza, że szukanego fragmentu nie znaleziono:

```
1  "Ala ma kota".indexOf("kot"); //7
```



Funkcje na tekście

- ▶ funkcja **lastIndexOf**, podaje ona jednak numer **ostatniego** wystąpienia podtekstu

```
1 "Ala ma kota i tak już jest".lastIndexOf("a"); //15
```

- ▶ Funkcje `startsWith()` i `endsWith()` służą do sprawdzenia czy dany tekst zaczyna się i kończy od wskazanego fragmentu:

```
1 const text = "Ala ma kota";  
2  
3 text.startsWith("Ala"); //true  
4 text.startsWith("Ola"); //false  
5  
6 text.endsWith("kota"); //true  
7 text.endsWith("psa"); //false
```

Funkcje na tekście

- ▶ Funkcja **substr(start, lng)** służy do zwracania kawałka tekstu. Pierwszym jej parametrem jest początek pobieranego kawałka tekstu, a drugi opcjonalny wskazuje długość pobieranego tekstu. Jeżeli drugi parametr nie zostanie podany, wówczas pobierany kawałek będzie pobierany do końca tekstu.

```
1  const text = "Ala ma kota";  
2  
3  console.log(text.substr(2)); //"a ma kota"  
4  console.log(text.substr(0, 3)); //"Ala"
```

- ▶ Funkcja **substring()** zamiast długości wyznacza miejsce końca pobieranego kawałka. Jeżeli drugi parametr nie zostanie podany, wtedy kawałek będzie pobierany do końca tekstu. Jeżeli zostaną podane oba parametry, ale drugi będzie mniejszy od pierwszego, wtedy automatycznie zostaną one zamienione miejscami.

```
1  const text = "Ala ma kota";  
2  
3  console.log(text.substring(0, 3)); //"Ala"  
4  console.log(text.substring(4)); //"ma kota"  
5  console.log("Ala ma kota".substring(6, 4)); //"ma"
```

Funkcje na tekście

- ▶ Funkcja **slice(start, stop)**, która zwraca nam kawałek tekstu. Jej działanie jest praktycznie identyczne do działania funkcji **substring()**, jednak występują małe różnice. Jeżeli drugi argument będzie mniejszy od pierwszego, wtedy w przeciwieństwie do **substring()** argumenty nie zostaną zamienione miejscami.

```
1  const txt = "Ala ma kota";
2
3  const txt2 = txt.slice(0,3);
4  console.log(txt2); //"Ala"
```

- ▶ Funkcja **split(znak, długość)** zwraca tablicę, która składa się z podzielonych fragmentów tekstu. Miejsce podziału jest podawane w parametrze **znak**, a maksymalna ilość zwracanych elementów w parametrze **długość**:

```
1  const text = "Ala ma kota, a kot ma Alę, Ala go kocha, a Kot ją wcale ;("
2  const parts = text.split(", ");
3
4  parts.forEach(function(letter) {
5      console.log(letter.toUpperCase());
6  });
```

Tablice

► Tworzenie nowej tablicy

```
const TAB = ["ALA", 123, "PIES", "KOT"];
```

TAB				
0	1	2	3	4
"ALA"	234	"PIES"	"KOT"	{...}
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]

```
const tab1 = []; //pusta tablica
```

```
const tab2 = [12, 21, 13, 41];
```

```
const tab3 = ["Ola", "Ania", "Monika"];
```



Tablice

▶ Odwołania do elementów tablicy

```
const tab = ["Ala", "ma", "rudego", "kota"]  
var tekst = tab[0] + tab.at(1) + tab.at(-1)  
console.log(tekst)    //Alamakota
```

- ▶ Pierwsza wartość w tablicy ma indeks 0, druga 1, trzecia 2 i tak dalej, aż do ostatniego indeksu który wynosi długość tablicy - 1 czyli tab.length-1.



Tablice

- Iteracja po tablicach - przejście przez wszystkie elementach zbioru

```
18      //indeks:      0      1      2
19      const tab = ["Jabłko", "Banan", "Śliwka"];
20
21      console.log( tab.length ); //3
22      console.log( tab[length-1]); //Śliwka
23
24      for (let i=0; i<tab.length; i++) {
25      |   console.log(tab[i]);
26      }
```



Tablice

- Enumeracja po tablicach - przejście przez wszystkie elementach zbioru bez używania indeksów

```
const tab = ["Marcin", "Ania", "Agnieszka"];

for (const el of tab) { //el to nazwa zmiennej wymyślona przez nas
  | console.log(el); //"Marcin", "Ania", Agnieszka
}
```



Pętla for of

- ▶ Pętla **for...of** używana jest do iterowania po obiektach iterowalnych jak string, tablice, struktury danych jak Map czy Set.

```
const numbers = [1, 2, 3, 4, 5]
for (const number of numbers) {
  console.log(number)
}
```

```
const countries = ['Finland', 'Sweden', 'Norway', 'Denmark', 'Iceland']
for (const country of countries) {
  console.log(country.toUpperCase())
}
```



Tablice

► Enumeracja - pętla **forEach**

```
const numbers = [1, 2, 3, 4, 5]
numbers.forEach((number, i) => {
  console.log(number, i)
})
```

```
const countries = ['Finland', 'Sweden', 'Norway', 'Denmark', 'Iceland']
countries.forEach((country, i, arr) => {
  console.log(i, country.toUpperCase())
})
```



Tablice

► Enumeracja - pętla **forEach**

```
30 function forEach(array, operation) {  
31   for (let i = 0; i < array.length; i++) {  
32     const element = array[i];  
33     operation(element);  
34   }  
35 }  
36  
37 const tab = ["Jabłko", "Banan", "Śliwka"];  
38 forEach(tab, function(item) {  
39   console.log(item);  
40 });  
41 // Wypisze:  
42 // Jabłko  
43 // Banan  
44 // Śliwka
```



Operacje na tablicach

- ▶ Dodawanie elementów do tablicy
- ▶ Czy tablica jest tablicą?
- ▶ Metody `push()` i `pop()`
- ▶ Metody `unshift()` i `shift()`
- ▶ Funkcja `join()`
- ▶ Zamiana tekstu na tablicę
- ▶ Funkcja `reverse()`
- ▶ Metody `indexOf()`, `lastIndexOf()` i `includes()`
- ▶ Funkcja `sort()`
- ▶ Łączenie tablic
- ▶ Funkcja `slice()`
- ▶ Funkcja `splice()`
- ▶ Funkcja `fill()`



Przykładowe operacje na tablicach

► Wypełnianie tablicy

```
const tab = new Array(20);  
console.log(tab); //[empty x 20]  
tab.fill(1);  
console.log(tab); //[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```



Przykładowe operacje na tablicach

► Sortowanie prostych tablic

```
46     const liczby = [1, 2, 21, 2.1, 32, 3.1];
47     liczby.sort();
48
49     console.log(liczby); //[ 1, 2, 2.1, 21, 3.1, 32 ]
50
51     const imiona = ["Basia", "ania", "Celina", "agnieszka"];
52     imiona.sort();
53
54     console.log(imiona); //["Basia", "Celina", "agnieszka", "ania"]
```



Przykładowe operacje na tablicach

► Łączenie tablic

```
57  const zw1 = ["Pies", "Kot"];
58  const zw2 = ["Słoń", "Wieloryb"];
59  const zw3 = ["Chomik ninja", "Świnka morderca"];
60
61  console.log(zw1+zw2);    //wypisze Pies,KotSłoń,Wieloryb
62
63  const table = zw1.concat(zw2);
64  console.log(table);    //wypisze ["Pies", "Kot", "Słoń", "Wieloryb"]
65
66  const tableBig = zw1.concat(zw2, zw3);
67  console.log(tableBig); //wypisze ["Pies", "Kot", "Słoń", "Wieloryb", "Chomik ninja", "Świnka morderca"];
```



Tablice wielowymiarowe

- ▶ Elementami tablicy dwuwymiarowej są tablice.

```
var tablica = [];  
//deklaracja drugiego wymiaru tablicy  
for (var i = 0; i < iloscWierszy; i++) {  
    tablica[i] = [];  
}  
//wypełnianie tablicy wartościami  
for (var i = 0; i < iloscWierszy; i++) {  
    for (var j = 0; j < iloscKolumn; j++) {  
        tablica[i][j] = i + ', ' + j;  
    }  
}
```



Zadania

- ▶ Utwórz tablice zawierająca 10 liczb parzystych począwszy od 12.
- ▶ Znajdź ile wśród zapisanych liczb jest liczb podzielnych przez 3.
- ▶ Odwróć elementy tablicy nie używając funkcji wbudowanych.





FUNKCJE



FUNKCJE

► Deklaracja

```
function sum(a, b) {  
    var c = a + b;  
    return c;  
}
```

► Wywołanie

```
sum(5, 7);
```

■ Deklaracja

```
function getArea(width, height) {  
    return width*height;  
}
```

■ Wywołanie

```
getArea(3, 5);
```



FUNKCJE WYMAGAJĄCE INFORMACJI

▶ Deklaracja z parametrami

```
function calcArea(width, height) {  
    var area = width * height;  
    return area;}  

```

▶ Wywołanie

```
var areaOne = calcArea(3, 5);  
var areaTwo = calcArea(5, 5);  

```

■ Zwrot wielu wartości z funkcji

```
function getSize(width, height, depth) {  
    var area = width * height; //oblicza pole  
    var volume = width * height * depth; //oblicza pojemność  
    var sizes = [area, volume]; //wstawia wartości do tablicy  
    return sizes; }  

```

■ Wywołanie

```
var areaOne = getSize(3, 5, 4)[0]; //przechowuje pierwszą  
wartość w tablicy  
var volumeTwo = getSize(5, 5, 6)[1];  

```

Parametry a argumenty funkcji

- ▶ Technicznie rzecz biorąc, istnieje różnica między parametrami i argumentami, chociaż te dwa terminy są często używane zamiennie. Parametry są definiowane razem z funkcją, podczas gdy argumenty są przekazywane do funkcji w trakcie jej wywoływania.
- ▶ Użycie **arguments** pozwoli nam poprawić funkcję tak, by przyjmowała ona dowolną liczbę parametrów i dodawała je wszystkie.

```
function args() {  
    return arguments;  
}  
console.log(args());  
console.log(args( 1, 2, 3, 4, true, 'ninja'));
```



FUNKCJE

- ▶ JavaScript nie wymaga od nas, abyśmy przekazywali do funkcji wymaganą przez daną funkcję ilość wartości.
- ▶ Jeżeli nie zakładamy konkretnej liczby parametrów dla funkcji, możemy skorzystać z właściwości **arguments**, która zawiera w sobie wszystkie przekazane wartości:

```
1  function sum() {  
2      console.log(arguments);  
3  }  
4  
5  sum(); //[ ] Arguments  
6  sum(1, 2, 3, 4); //[1, 2, 3, 4] Arguments  
7  sum("ala", "ma", "kota"); //["ala", "ma", "kota"] Arguments
```

Obiekt **arguments** jest tablicą podobny, ale tak naprawdę nie jest tablicą.



FUNKCJE

► Parametry domyślne

```
function print(name = "Michał", status = "najlepszy") {  
    console.log(name + " jest " + status);  
}  
  
print(); // "Michał jest najlepszy"  
print("Karol"); // "Karol jest najlepszy"  
print("Paweł", "wysoki"); // "Paweł jest wysoki"  
print(undefined, "wysoki"); // "Michał jest wysoki"  
                                //- undefined jest traktowane jak niepodanie wartości
```

► Podczas określania domyślnych wartości parametrów można również odwoływać się do innych parametrów:

```
function t(fog_level=1, spark_level=fog_level){  
    console.log(`Poziom zamglenia: ${fog_level}, a natężenie iskier:  
    ${spark_level}`)  
    // Poziom zamglenia: 10, a natężenie iskier: 10  
}  
  
function s(fog_level=10, spark_level = fog_level*10){  
    console.log(`Poziom zamglenia: ${fog_level}, a natężenie iskier:  
    ${spark_level}`)  
    // Poziom zamglenia: 10, a natężenie iskier: 100  
}  
  
t(10);  
s(10);
```

Parametry reszty

- ▶ ES6 wprowadza parametry reszty (ang. *rest parameters*, *rest operator*). Parametry reszty pozwalają wysyłać do funkcji dowolną liczbę parametrów w postaci tablicy. Parametrem reszty może być tylko ostatni na liście parametrów i może być tylko jeden parametr reszty. Umieszczenie operatora reszty (...) przed ostatnim formalnym parametrem wskazuje, że ten parametr jest parametrem reszty.
- ▶ Zaleca się, aby zamiast operować na obiekcie **arguments**, używać **rest operator**, który zbiera przekazane argumenty w postaci klasycznej tablicy.



Parametry reszty

► Przykład 1

```
1  function superSum(...r) {  
2      console.log(r); //[1, 2, 3, 4]  
3  }  
4  
5  superSum(1, 2, 3, 4);
```

► Przykład 2

```
function sayThings(tone, ...quotes){  
    console.log(Array.isArray(quotes)); // true  
    console.log(`Głosem ${tone} mówię: ${quotes}`)  
}  
sayThings("Morgana Freemana", "Coś poważnego",  
    " Implodujący Wszechświat", " Amen");  
// Głosem Morgana Freemana mówię: Coś poważnego,  
//Implodujący Wszechświat, Amen
```



Zakres zmiennych

- ▶ Jeśli zmienna została zdefiniowana wewnątrz funkcji, nie jest widoczna poza nią. Natomiast zmienna zdefiniowana wewnątrz bloku if lub for jest widoczna poza blokiem.
- ▶ **Zmienne globalne** to zmienne definiowane poza funkcjami (w globalnym kodzie programu), natomiast **zmienne lokalne** to zmienne definiowane wewnątrz funkcji. Kod wewnątrz funkcji ma dostęp zarówno do zmiennych globalnych, jak i do swoich zmiennych lokalnych.

```
var global = 1;  
function f() {  
    var local = 2;  
    global++;  
    return global;  
}
```

Wynoszenie zmiennych (hoisting)

- ▶ Kiedy wykonywanie programu JavaScript wchodzi w nową funkcję, wszystkie zmienne zadeklarowane w różnych miejscach funkcji są przesuwane, czyli wynoszone w górę funkcji. Jest to ważna koncepcja, o której należy pamiętać. Co więcej, wynoszona jest tylko deklaracja, co oznacza, że tylko obecność zmiennej jest przenoszona na samą górę.

```
var a = 123;  
function f() {  
    alert(a);  
    var a = 1;  
    alert(a);  
}  
f();
```

Zakres bloku

- ▶ Poniższy przykład ilustruje zakres bloku:

```
var a = 1;  
{  
  let a = 2;  
  console.log( a ); // 2  
}  
console.log( a ); // 1
```



Funkcje anonimowe

- ▶ To funkcje nie posiadające nazwy

```
var f = function (a) {  
    return a;  
};
```

- ▶ Fragment `function (a) { return a; }` jest **wyrażeniem funkcyjnym (ang. *function expression*)**.

Zastosowanie:

- ▶ Funkcję anonimową można przekazać jako parametr do innej funkcji. Funkcja odbierająca może z przekazaną funkcją zrobić coś pożytecznego.
- ▶ Funkcje anonimowe można definiować i od razu wykonywać.



Funkcje strzałkowe

- ▶ Funkcjami strzałkowymi, to skrócony zapis funkcji, które pozwalają na poprawienie czytelności kodu
- ▶ Zamiast kodu:

```
1. var result = function(a, b) {  
2.     return a + b;  
3. }  
4. console.log(result(5, 5)) // rezultat w konsoli -> 10
```

- ▶ możemy zdefiniować funkcje strzałkową:

```
1. var result = (a, b) => {  
2.     return a + b;  
3. }  
4. console.log(result(5, 5)) // rezultat w konsoli -> 10
```

- ▶ Lub krótko:

```
1. var result = (a, b) => a + b;  
2. console.log(result(5, 5)) // wynik wyświetlony w konsoli -> 10
```



Wywołania zwrotne

- ▶ Skoro funkcje są jak wszystkie innego rodzaju dane przypisywane do zmiennych, można je definiować, kopiować i przekazywać jako argumenty do innych funkcji.
- ▶ Oto przykład funkcji, która przyjmuje dwie funkcje jako parametry, wykonuje je, po czym zwraca wynik będący sumą zwróconych przez te funkcje wartości:

```
function invokeAdd(a, b) {  
  return a() + b();  
}  
function one() {  
  return 1;  
}  
function two() {  
  return 2;  
}  
console.log(invokeAdd(one, two));
```

Można również zrobić to bardziej czytelnie w ten sposób:

```
console.log(invokeAdd(  
  function () { return 1; },  
  function () { return 2; }  
));
```

Wywołania zwrotne

Jakie zastosowania mają funkcje wywołania zwrotnego:

- ▶ wołania zwrotne umożliwiają przekazywanie funkcji bez konieczności ich nazywania, co oznacza, że potrzebnych jest mniej zmiennych.
- ▶ możemy przenieść obowiązek wywołania funkcji na inną funkcję, co oznacza, że musimy napisać krótszy kod.
- ▶ Wywołania zwrotne mogą korzystnie wpłynąć na wydajność aplikacji poprzez opóźnianie wykonywania lub odblokowywanie wywołań.



Funkcje wewnętrzne (prywatne)

```
function outer(param) {  
  function inner(theinput) {  
    return theinput * 2;  
  }  
  return 'Wynik wynosi ' + inner(param);  
}
```

Ze stosowania funkcji prywatnych płyną następujące korzyści:

- ▶ Nie dochodzi do zaśmiecenia globalnej przestrzeni nazw, co zmniejsza ryzyko powstawania kolizji nazw.
- ▶ Prywatność — na zewnątrz widoczne są tylko te funkcje, które programista chce udostępnić. Funkcjonalności nieprzeznaczone dla reszty aplikacji są ukryte.



Literatura

- ▶ Programowanie zorientowane obiektowo w języku JavaScript. Wydanie III.
- ▶ <https://kursjs.pl/kurs/>

