Cheatography

JavaScript basics Cheat Sheet

by sigeud via cheatography.com/188214/cs/39251/

Zmienne

Deklaracja zmiennej:

- **var** `nazwa = wartość` – deklaruje zmienną o zakresie funkcyjnym
- **let** `nazwa = wartość` – deklaruje zmienną o zakresie blokowym
- **const** `nazwa = wartość` – deklaruje stałą o zakresie blokowym, nie może ulec zmianie

Deklaracja funkcji

```
function addition(a,b) {
let result = a + b
consol e.l og( " Wynik dodawania: " + result)}
additi on(1, 2)
additi on(5, 2)
function multip ly(a,b) {
let result = a * b
return result}
consol e.l og( mul tip ly(5, 10))
```

Typy zmiennych - prymitywy

`let num = 7`	number
`let str = " tekst`	string
`let flag = true`	boolean
`let info`	null - specjalny prymityw, intencjonalnie nie zmienia wartości
`let mySymbol = Symbol ("Sy m ")`	symbol
`let bigInt = 1n`	bigint do przechowywania liczb większych niż `Number.MA X_ S AFE _IN TEGER`

Typy zmiennych - złożone

`let obj = new Object()`	Object
`let arr = new Array()`	obiekt Array
`let date = new Date()`	obiekt Date
`let err = new Error()`	obiekt Error

Sprawdzanie typów danych - typeof

```
typeof 42; // "number"
typeof " Hel lo"; // " str ing "
typeof true; // " boo lea n"
typeof []; // " obj ect "
typeof {}; // " obj ect "
typeof null; // " obj ect "
typeof undefined; // " und efi ned "
typeof function() {}; // " fun cti on"
```

Sprawdzanie typów danych - instanceof

```
const arr = [];
arr instanceof Array; // true
const obj = {};
obj instanceof Object; // true
const today = new Date();
today instanceof Date; // true
```

Sprawdzanie typów danych: porównywanie

```
typeof 42 === "number"; // true
typeof " Hel lo" === " str ing "; // true
typeof true === " boo lea n"; // true
```

Sprawdzanie typów danych - Array.isArray()

```
Array.isArray([]); // true
Array.i sA rra y({}); // false
Array.i sA rra y("h ell o"); // false
```

NaN - Not a Number

parseI nt(" hel lo")	NaN
Math.s qrt(-1)	NaN
Math.s qrt(-1)	NaN
isNaN(Num ber.NaN)	true
isNaN(NaN)	true
isNaN(78)	false

formatowanie z backtickiami

```
Wstawianie wartości zmiennych:
let name = " Joh n";
let age = 30;
let message = My name is ${name} and I am ${age} y
ears old.;
consol e.l og( mes sage); // "My name is John and I
am 30 years old."
Wyrażenia matema tyczne:
let a = 5;
let b = 3;
let sum = ${a} + ${b} = ${a + b};
consol e.l og( sum); // "5 + 3 = 8"
Wywoły wanie funkcji:
function greet( name) {
    returnHello, ${name}!;
}
let person = " Ali ce";
let greeting = greet( per son);
```

formatowanie z backtickiami (cont)

```
> console.log(greeting); // "Hello, Alice!"
Warunkowe wyrażenia:
let num = 7;
let isEven = ${num} is ${num % 2 === 0 ? " eve n" : " -
odd "};
console.log(isEven); // "7 is odd"
```

String

```
const s = 'string w apostrofach'
const str = " string w cudzys łow ie"
const txt = `string
w wielu
liniach`
```

Operacje na stringach

```
Właściwość length: Zwraca liczbę znaków w ciągu
znaków.
let str = " Hello, World! ";
consol e.l og( str.le ngth); // 13
Indeks owanie za pomocą nawiasów kwadra towych:
Można uzyskać dostęp do poszcz egó lnych
znaków ciągu znaków, odwołując się do ich indeksu
za pomocą nawiasów kwadra towych
i liczby
całkowitej reprez ent ującej pozycję znaku w ciągu
(indek sowanie zaczyna się od zera).
let str = " Hel lo";
consol e.l og( str [0]); // " H"
consol e.l og( str [1]); // " e"
consol e.l og( str [4]); // " o"
Metoda charAt (in dex): Zwraca znak znajdujący się
na określonym indeksie w ciągu znaków.
let str = " Hello, World! ";
consol e.l og( str.ch arA t(7)); // " W"
Metoda charCo deAt(): Metoda charCo deAt() zwraca
wartość kodu Unicode znaku na
```

Operacje na stringach (cont)

> określonym indeksie w ciągu znaków.
let str = "Hello";
console.log(str.charCodeAt(0)); // 72
console.log(str.charCodeAt(1)); // 101
Metoda indexOf(searchValue, fromIndex): Znajduje indeks pierwszego wystąpienia
określonej wartości w ciągu znaków.
let str = "Hello World";
console.log(str.indexOf("o")); // 4
console.log(str.indexOf("o", 5)); // 7 (rozpoczyna wyszukiwanie od indeksu 5)
console.log(str.indexOf("o", 10)); // -1 (nie znaleziono, rozpoczyna wyszukiwanie
//od indeksu 10)
console.log(str.indexOf("World")); // 6
console.log(str.indexOf("JavaScript")); // -1 (nie znaleziono)
Metoda lastIndexOf(): Metoda lastIndexOf() znajduje ostatnie wystąpienie podciągu
w ciągu znaków i zwraca indeks pierwszego znaku tego podciągu. Jeśli podciąg nie
zostanie znaleziony, metoda zwraca -1.
let str = "Hello World";
console.log(str.lastIndexOf("o")); // 7
console.log(str.lastIndexOf("o", 7)); // 7 (rozpoczyna wyszukiwanie od indeksu 7 wstecz)
console.log(str.lastIndexOf("o", 3)); // 2 (rozpoczyna wyszukiwanie od indeksu 3 wstecz)
console.log(str.lastIndexOf("World")); // 6
console.log(str.lastIndexOf("JavaScript")); // -1 (nie znaleziono)
Metoda slice(startIndex, endIndex) zwraca fragment tekstu między dwoma indeksami
(startIndex i endIndex). slice() działa włączając startIndex, ale wyłączając endIndex.
Jeśli startIndex jest większy niż endIndex, metoda zamienia te wartości. Można również
podać tylko startIndex, wtedy slice() zwróci fragment od startIndex do końca tekstu.
Metoda slice() jest jedyną metodą, która obsługuje ujemne
indeksy w celu wycinania fragmentów tekstu od końca. substring() i substr() nie

Operacje na stringach (cont)

> obsługują indeksów ujemnych i zamieniają je na 0.
let str = 'Hello, world!';
let sliced = str.slice(7, 12);
console.log(sliced); // "world"
Metoda substring(startIndex, endIndex): Zwraca fragment ciągu znaków od indeksu
startIndex do indeksu endIndex (bez końcowego indeksu).
Metoda substring(startIndex, endIndex) działa podobnie jak slice(), z tą różnicą, że
automatycznie zamienia wartości, jeśli startIndex jest większy niż endIndex. Tak samo
jak slice(), substring() zwraca fragment tekstu między indeksami, włączając startIndex,
ale wyłączając endIndex. Można również podać tylko startIndex, wtedy substring() zwróci
fragment od startIndex do końca tekstu.
let str = "Hello, World!";
console.log(str.substring(7, 12)); // "World"
Metoda substr(startIndex, length) zwraca fragment tekstu o określonej długości,
zaczynając od startIndex. Drugim argumentem jest length, który określa, ile znaków
ma zostać zwróconych. Jeśli length nie jest podane, substr() zwraca fragment tekstu
od startIndex do końca tekstu.
let str = 'Hello, world!';
let substr = str.substr(7, 5);
console.log(substr); // "world"
Metoda search() szuka dopasowania wzorca w ciągu znaków i zwraca indeks pierwszego
dopasowania.
let str = "Hello World";
console.log(str.search(/lo/)); // 3 (znaleziono dopasowanie na indeksie 3)
console.log(str.search("World")); // 6 (znaleziono dopasowanie na indeksie 6)
console.log(str.search(/JavaScript/)); // -1 (nie znaleziono dopasowania)
Metoda match() znajduje wszystkie dopasowania wzorca w ciągu znaków i zwraca
tablicę zawierającą dopasowania.

Operacje na stringach (cont)

> let str = "Hello World";
console.log(str.match(/lo/)); // ["lo"] (znaleziono dopasowanie "lo")
console.log(str.match(/o/g)); // ["o", "o"] (znaleziono wszystkie dopasowania "o")
console.log(str.match(/JavaScript/)); // null (nie znaleziono dopasowania)
Metoda trim() jest używana do usuwania białych znaków (spacje, tabulatory, znaki
nowej linii) z początku i końca ciągu znaków. Oto przykład użycia metody trim():
let str = " Hello World ";
console.log(str.trim()); // "Hello World" (usunięcie białych znaków z początku i końca)
let str2 = " Hello \n World ";
console.log(str2.trim()); // "Hello \n World" (białe znaki w środku nie są usuwane)
Metoda toLowerCase(): Zmienia wszystkie znaki w ciągu znaków na małe litery.
let str = "Hello, World!";
console.log(str.toLowerCase()); // "hello, world!"
Metoda toUpperCase(): Zmienia wszystkie znaki w ciągu znaków na wielkie litery.
let str = "Hello, World!";
console.log(str.toUpperCase()); // "HELLO, WORLD!"
Metoda split(separator):
let str = 'Hello, world!';
let splitted = str.split(', ');
console.log(splitted); // ["Hello", "world!"]
Metoda replace(searchValue, replaceValue):
let str = 'Hello, world!';
let replaced = str.replace('world', 'universe');
console.log(replaced); // "Hello, universe!"

Łączenie stringów

```
Operator +: Można użyć operatora + do konkatenacji dwóch ciągów znaków.
let str1 = " Hel lo";
let str2 = " Wor ld";
let result = str1 + " " + str2;
consol e.l og( res ult); // " Hello World"
Metoda concat(): Metoda concat() pozwala na łączenie wielu ciągów znaków.
let str1 = " Hel lo";
let str2 = " Wor ld";
let result = str1.c onc at( " ", str2);
consol e.l og( res ult); // " Hello World"
Metoda join(): Metoda join() pozwala na łączenie elementów tablicy w jeden ciąg znaków,
używając określ onego separa tora.
let arr = ["He llo ", " Wor ld"];
let result = arr.jo in( " ");
consol e.l og( res ult); // " Hello World"
```

Wyrażenia regularne

```
1. Tworzenie wyrażeń regularnych:
Wyrażenia regularne można utworzyć za pomocą dwóch sposobów: za pomocą literałów
/pattern/ lub przy użyciu konstr uktora RegExp - ('p att ern').
Na przykład: /hello/ lub new RegExp ('h ello').
let regex1 = /hello/; // za pomocą literału
let regex2 = new RegExp ('h ello'); // za pomocą konstr uktora RegExp
2.Metody do pracy z wyraże niami regula rnymi:
Metody takie jak test(), exec(), match(), search(), replace(), split() itp. mogą być
wykorz yst ywane do interakcji z wyraże niami regula rnymi i manipu lacji tekstami.
test():
let regex = /hello/;
let text = 'Hello, world!';
```

Wyrażenia regularne (cont)

> let result = regex.test(text);
console.log(result); // false
exec():
let regex = /lorem/;
let text = 'Lorem ipsum dolor sit amet.';
let result = regex.exec(text);
console.log(result); // ['lorem', index: 0, input: 'Lorem ipsum dolor sit amet.',
// groups: undefined]
split():
let regex = /\s+/;
let text = 'Hello world!';
let result = text.split(regex);
console.log(result); // ['Hello', 'world!']
3.Składnia wzorców:
Wzorce wyrażeń regularnych składają się z literałów, znaków specjalnych i zestawów
znaków. Przykłady: /hello/ - dopasowuje bezpośrednio słowo "hello", /[aeiou]/ -
dopasowuje dowolny samogłoskę.
let regex = /hello/;
let text = 'Hello, world!';
let result = regex.test(text);
console.log(result); // false
let regex = /[aeiou]/;
let text = 'Hello, world!';
let result = text.match(regex);
console.log(result); // ['e', 'o', 'o']
4.Znaki specjalne:
Wyrażenia regularne wykorzystują znaki specjalne, takie jak
^, $, ., *, +, ?, |, \, (), [], {}, itd., które mają specjalne znaczenie i służą

Wyrażenia regularne (cont)

> do definiowania różnych wzorców dopasowania.
Znaki specjalne:
Kropka .: Dopasowuje dowolny znak, z wyjątkiem nowej linii.
Znak zapytania ?: Oznacza, że poprzedni znak lub grupa może wystąpić zero
lub jeden raz.
Gwiazdka *: Oznacza, że poprzedni znak lub grupa może wystąpić zero lub więcej razy.
Plus +: Oznacza, że poprzedni znak lub grupa musi wystąpić co najmniej raz.
Nawiasy (): Tworzą grupę dopasowania.
5. Modifikatory:
Modifikatory są dodawane do wzorca wyrażenia regularnego w celu wpływania na sposób
dopasowywania.
Przykłady: g - dopasowanie globalne (wszystkie wystąpienia), i - dopasowanie bez
względu na wielkość liter, m - wielolinijkowe dopasowanie.

falsy values

" " // pusty string
0, -0
0n // 0 jako BigInt
NaN
null
undefined
false

To wartości, które poddane konwersji do boolean będą miały wartość false

Tablica

kolekcja elementów – obiekt, zmienna, w której są inne zmienne

```
let a = [3, „tekst”, 99, 12]
let b = new Array(10, 4, " txt ")
let c = Array(10, 4, " txt ")
```

`**let d** = Array.of(5) // pojedyncza wartość traktowana jest jako jeden element,
a nie ustawienie wartości length`

```
tab[1]  // element z tablicy, indeksy liczone są od 0
```

Tablica (Array) jest jednym z podstawowych typów danych w języku JavaScript,
który służy do przechowywania kolekcji elementów w uporządkowany sposób. Oto kilka dodatkowych informacji o tablicach:
- Indeksowanie: Elementy w tablicy są indeksowane numerycznie, zaczynając od zera.
Możesz uzyskać dostęp do elementów za pomocą ich indeksów.
Przykład:
const fruits = ['apple', 'banana', 'orange'];
console.log(fruits[0]); // "apple"
console.log(fruits[2]); // "orange"
- Długość tablicy: Właściwość length pozwala określić liczbę elementów w tablicy.
Przykład:
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.length); // 5
- Dodawanie i usuwanie elementów: Tablice mają wiele metod, które umożliwiają dodawanie i usuwanie elementów. Niektóre z tych metod to: push(), pop(), shift(), unshift(), splice() itp.
Przykład (dodawanie elementu na koniec tablicy):
const fruits = ['apple', 'banana'];
fruits.push('orange');
console.log(fruits); // ["apple", "banana", "orange"]
Przykład (usuwanie elementu z początku tablicy):
const numbers = [1, 2, 3, 4, 5];

Tablica (cont)

numbers.shift();
console.log(numbers); // [2, 3, 4, 5]
- Iteracja po tablicy: Możesz iterować po elementach tablicy za pomocą pętli, takich jak pętla for lub metody forEach(), map(), filter(), reduce() itp.
Przykład (użycie metody forEach()):
const fruits = ['apple', 'banana', 'orange'];
fruits.forEach(fruit => {
console.log(fruit);
});
- Metoda fill(value, start, end): Metoda fill() wypełnia wszystkie elementy tablicy określoną wartością. Można również określić opcjonalne parametry start i end, które wskazują zakres indeksów tablicy, które mają być wypełnione.
Przykład:
const numbers = [1, 2, 3, 4, 5];
numbers.fill(0);
console.log(numbers); // [0, 0, 0, 0, 0]
const fruits = ['apple', 'banana', 'orange'];
fruits.fill('pear', 1, 2);
console.log(fruits); // ['apple', 'pear', 'orange']
- Metoda find(callback): Metoda find() zwraca pierwszy element tablicy, który spełnia warunek zdefiniowany przez funkcję zwrotną (callback).
Przykład:
const numbers = [1, 2, 3, 4, 5];
const found = numbers.find(num => num > 3);
console.log(found); // 4
- Metoda findIndex(callback): Metoda findIndex() zwraca indeks pierwszego elementu
tablicy, który spełnia warunek zdefiniowany przez funkcję zwrotną (callback).
Przykład:

Tablica (cont)

```
const numbers = [1, 2, 3, 4, 5];
const index = numbers.findIndex(num => num > 3);
console.log(index); // 3
```

- Metoda copyWithin(target, start, end): Metoda copyWithin() kopiuje elementy z
określonego zakresu tablicy i wkleja je na pozycję początkową wewnątrz tej samej tablicy.
Parametr target wskazuje pozycję docelową, a parametry start i end określają zakres
elementów do skopiowania.
Przykład:

```
const numbers = [1, 2, 3, 4, 5];
numbers.copyWithin(0, 3, 5);
console.log(numbers); // [4, 5, 3, 4, 5]
```

- Metoda filter(callback): Metoda filter() tworzy nową tablicę zawierającą elementy,
dla których funkcja zwrotna (callback) zwraca wartość true.
Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

- Metoda reduce(callback, initialValue): Metoda reduce() wykonuje funkcję
zwrotną (callback) dla każdego elementu tablicy, redukując ją do pojedynczej
wartości. Parametr initialValue jest opcjonalny i określa początkową wartość akumulatora.
Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) =>
accumulator + currentValue, 0);
console.log(sum); // 15
```

- Metoda map(callback): Metoda map() tworzy nową tablicę, zawierającą wyniki
działania funkcji zwrotnej (callback) na każdym elemencie tablicy.
Przykład:

```
const numbers = [1, 2, 3, 4, 5];
```

Tablica (cont)

```
const squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers); // [1, 4, 9, 16, 25]
```

- Metoda every(callback): Metoda every() sprawdza, czy wszystkie elementy tablicy spełniają warunek zdefiniowany przez funkcję zwrotną (callback). Zwraca wartość true, jeśli wszystkie elementy spełniają warunek, w przeciwnym razie zwraca false.
Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // false
```

- Metoda some(callback): Metoda some() sprawdza, czy przynajmniej jeden element
tablicy spełnia warunek zdefiniowany przez funkcję zwrotną (callback). Zwraca wartość
true, jeśli przynajmniej jeden element spełnia warunek, w przeciwnym razie zwraca false.
Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // true
```

- Inne metody tablicowe: Tablice mają wiele innych przydatnych metod, które ułatwiają
manipulację i przetwarzanie danych. Niektóre z tych metod to: join(), slice(), concat(), sort(), reverse() itp.
Przykład (użycie metody join()):

```
const fruits = ['apple', 'banana', 'orange'];
const joinedFruits = fruits.join(', ');
console.log(joinedFruits); // "apple, banana, orange"
```

Operacje na tablicach

```
length: Zwraca liczbę elementów w tablicy. - można
też w tem sposób zmniejszyć
wielkość tablicy
let fruits = ["ap ple ", " ban ana ", " ora -
nge "];
consol e.l og( fru its.le ngth); // Wyświetli: 3
let a = [1,2,3 ,4, 5,6 ,7,8,9]
a.length = 3
consol e.l og(a) // [1, 2, 3]
push(): Dodaje jeden lub więcej elementów na koniec
tablicy i zwraca jej nową
długość.
let fruits = ["ap ple ", " ban ana "];
fruits.pu sh( " ora nge ");
consol e.l og( fru its); // Wyświetli: ["ap -
ple ", " ban ana ", " ora nge "]
pop(): Usuwa ostatni element z tablicy i zwraca
ten element.
let fruits = ["ap ple ", " ban ana ", " ora -
nge "];
let remove dFruit = fruits.pop();
consol e.l og( rem ove dFr uit); // Wyświetli: " -
ora nge "
consol e.l og( fru its); // Wyświetli: ["ap -
ple ", " ban ana "]
delete: Usuwa określony element z tablicy,
pozost awiając na jego miejscu pustą
wartość (undef ined).
let fruits = ["ap ple ", " ban ana ", " ora -
nge "];
delete fruits[1];
consol e.l og( fru its); // Wyświetli: ["ap -
ple ", undefined, " ora nge "]
concat(): Łączy dwie tablice lub więcej tablic i
zwraca nową tablicę.
let fruits1 = ["ap ple ", " ban ana "];
let fruits2 = ["or ang e", " gra pe"];
let combin edF ruits = fruits 1.c onc at( fru -
its2);
consol e.l og( com bin edF ruits); // Wyświetli:
["ap ple ", " ban ana ", " ora nge ", " gra pe"]
join(): Łączy wszystkie elementy tablicy w jeden
ciąg znaków, używając
```

Operacje na tablicach (cont)

```
> podanego separatora.
let fruits = ["apple", "banana", "orange"];
let joinedString = fruits.join(", ");
console.log(joinedString); // Wyświetli: "apple, banana, orange"
indexOf(): Znajduje indeks pierwszego wystąpienia podanego
elementu w
tablicy. Zwraca -1, jeśli element nie jest obecny.
let fruits = ["apple", "banana", "orange"];
let index = fruits.indexOf("banana");
console.log(index); // Wyświetli: 1
fill: Wypełnia wszystkie elementy tablicy określoną wartością.
let numbers = [1, 2, 3, 4, 5];
numbers.fill(0);
console.log(numbers); // Wyświetli: [0, 0, 0, 0, 0]
forEach: Wykonuje podaną funkcję dla każdego elementu tablicy.
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(number) {
  console.log(number);
});
// Wyświetli:
// 1
// 2
// 3
// 4
// 5
reverse: Odwraca kolejność elementów w tablicy.
let fruits = ["apple", "banana", "orange"];
fruits.reverse();
console.log(fruits); // Wyświetli: ["orange", "banana", "apple"]
```

Operacje na tablicach (cont)

> shift: Usuwa pierwszy element z tablicy i zwraca ten element.
let fruits = ["apple", "banana", "orange"];
let shiftedFruit = fruits.shift();
console.log(shiftedFruit); // Wyświetli: "apple"
console.log(fruits); // Wyświetli: ["banana", "orange"]
unshift: Dodaje jeden lub więcej elementów na początek tablicy i zwraca jej nową
długość.
let fruits = ["banana", "orange"];
fruits.unshift("apple");
console.log(fruits); // Wyświetli: ["apple", "banana", "orange"]
splice(start, deleteCount, item1, item2...): Modyfikuje zawartość tablicy, usuwając,
zamieniając lub dodając elementy.
let fruits = ["apple", "banana", "orange"];
fruits.splice(1, 1, "kiwi");
console.log(fruits); // Wyświetli: ["apple", "kiwi", "orange"]

Typy zdarzeń DOM

Przeglądarka wywołuje wiele zdarzeń. Pełna lista jest dostępna na MDN, ale poniżej możesz znaleźć te najpopularniejsze:
zdarzenia myszki (MouseEvent): mousedown, mouseup, click, dblclick, mousemove, mouseover, mousewheel, mouseout, contex-tmenu
zdarzenia dotykowe (TouchEvent): touchstart, touchmove, touchend, touchcancel
zdarzenia klawiatury (KeyboardEvent): keydown, keypress, keyup
zdarzenia formularzy: focus, blur, change, submit
zdarzenia okna przeglądarki: scroll, resize, hashchange, load, unload
Khan Academy

Fetch - pobieranie danych z serwera

```
fetch("https://swapi.dev/api/people/3/")
       .th en( res ponse => respon se.j son())
       .th en( pro ces sData)
function proces sDa ta( data) {
                        con sol e.l og( data);
                        con sol e.l og( " Imi -
ę:", dat a.n ame);
                        con sol e.l og( " Wys oko -
ść: ", data.h eight);
                        con sol e.l og( " Mas a:",
data.m ass);
                        con sol e.l og( " Kolor
oczu:", data.e ye_ color);
               }
```

Obiekt window i event handler

```
onload:
window.onload = function() {
   // Kod wykonywany po załado waniu strony
};
Ten event handler jest wywoływany po załado waniu
całej strony.
onresize:
window.on resize = function() {
   // Kod wykonywany po zmianie rozmiaru okna
przegl ądarki
};
Ten event handler jest wywoływany po zmianie
rozmiaru okna przegl ądarki.
onscroll:
window.on scroll = function() {
   // Kod wykonywany podczas przewi jania strony
};
Ten event handler jest wywoływany podczas przewi -
jania strony.
onkeydown:
window.on keydown = functi on( event) {
```

Obiekt window i event handler (cont)

```
>   // Kod wykonywany po wciśnięciu klawisza
  if (event.key === 'Enter') {
    // Wykonaj pewną akcję po wciśnięciu klawisza Enter
  }
};
```

Ten event handler jest wywoływany po wciśnięciu klawisza na klawiaturze.

onbeforeunload:

```
window.onbeforeunload = function(event) {
  // Kod wykonywany przed zamknięciem strony
  return 'Czy na pewno chcesz opuścić tę stronę?';
};
```

Ten event handler jest wywoływany przed zamknięciem strony.

Obiekt window i zdarzenia

```
Zdarzenie load:
window.ad dEv ent Lis ten er( 'load', function() {
    // Kod wykonywany po załado waniu strony
});
```

Ten event listener jest wywoływany po załado waniu całej strony.

```
Zdarzenie resize:
window.ad dEv ent Lis ten er( 're size', function()
{
    // Kod wykonywany po zmianie rozmiaru okna
przegl ądarki
});
```

Ten event listener jest wywoływany po zmianie rozmiaru okna przegl ądarki.

```
Zdarzenie scroll:
window.ad dEv ent Lis ten er( 'sc roll', function()
{
    // Kod wykonywany podczas przewi jania strony
});
```

Ten event listener jest wywoływany podczas przewi jania strony.

Obiekt window i zdarzenia (cont)

> Zdarzenie keydown:

```
window.addEventListener('keydown', function(event) {
  // Kod wykonywany po wciśnięciu klawisza
  if (event.key === 'Enter') {
    // Wykonaj pewną akcję po wciśnięciu klawisza Enter
  }
});
```

Ten event listener jest wywoływany po wciśnięciu klawisza na klawiaturze.

Zdarzenie beforeunload:

```
window.addEventListener('beforeunload', function(event) {
  // Kod wykonywany przed zamknięciem strony
  event.returnValue = 'Czy na pewno chcesz opuścić tę stronę?';
});
```

Ten event listener jest wywoływany przed zamknięciem strony.

Zdarzenia myszy

```
<!DOCTYPE html>
<ht ml>
<he ad>
    <ti tle >Zd arzenia myszy< /ti tle>
</h ead>
<bo dy>
    <p id=" my- par agr aph " >Najedź na ten
paragraf i kliknij go!</p>
    <sc rip t>
        const paragraph = docume nt.g et Ele men -
tBy Id( 'my -pa rag raph');
       // Obsługa zdarzenia najechania myszą
        par agr aph.ad dEv ent Lis ten er( 'mo -
use enter', functi on( event) {
            par agr aph.st yle.ba ckg rou ndColor =
'yellow';
       });
```

Zdarzenia myszy (cont)

```
>    // Obsługa zdarzenia opuszczenia myszy
   paragraph.addEventListener('mouseleave', function(event) {
     paragraph.style.backgroundColor = 'white';
   });
   // Obsługa zdarzenia kliknięcia myszą
   paragraph.addEventListener('click', function(event) {
     paragraph.textContent = 'Kliknięto!';
   });
   // Obsługa zdarzenia poruszania myszą
   paragraph.addEventListener('mousemove', function(event) {
     const x = event.clientX;
     const y = event.clientY;
     paragraph.textContent = X: ${x}, Y: ${y};
   });
  </script>
</body>
</html>
```

Obiekt window

```
alert(message): Wyświetla okno dialogowe z podaną
wiadomością.
window.al ert ("He llo !");
confir m(m ess age): Wyświetla okno dialogowe z
pytaniem i zwraca wartość logiczną
w zależności od wyboru użytko wnika.
const result = window.co nfi rm( "Czy jesteś
pewien ?");
consol e.l og( res ult); // true jeśli kliknięto
OK, false jeśli kliknięto Anuluj
prompt (me ssage, default): Wyświetla okno
dialogowe z pytaniem o wprowa dzenie
danych i zwraca wprowa dzoną wartość jako ciąg
znaków.
const name = window.pr omp t("Jak masz na imię?");
consol e.l og( name); // Wprowa dzona wartość
zostanie wyświe tlona w konsoli
```

Obiekt window (cont)

> setTimeout(callback, delay): Wywołuje funkcję (callback) po określonym opóźnieniu
(w milisekundach).

```
function greet() {
  console.log("Hello!");
}
window.setTimeout(greet, 2000); // Wywołuje funkcję greet po 2 sekundach
```

setInterval(callback, interval): Wywołuje funkcję (callback) co określony interwał
(w milisekundach), powtarzając ją w nieskończoność.

```
function printTime() {
  console.log(new Date());
}
window.setInterval(printTime, 1000); // Wywołuje funkcję printTime co sekundę
```

window.location: Obiekt dostarcza informacje o aktualnym adresie URL strony.

```
console.log(window.location.href); // Zwraca pełny adres URL strony
console.log(window.location.pathname); // Zwraca ścieżkę URL
console.log(window.location.host); // Zwraca nazwę hosta (np. www.example.com)
```

innerWidth i innerHeight: Właściwości innerWidth i innerHeight zwracają
szerokość i wysokość obszaru widocznego w oknie przeglądarki, bez uwzględniania paska przewijania i innych elementów interfejsu przeglądarki.

```
console.log(window.innerWidth); // Zwraca szerokość obszaru widocznego w pikselach
console.log(window.innerHeight); // Zwraca wysokość obszaru widocznego w pikselach
```

pageXOffset i pageYOffset: Właściwości pageXOffset i pageYOffset zwracają
przesunięcie strony w poziomie i pionie względem początku dokumentu. Są one
przydatne do określania aktualnego położenia przewijania strony.

```
console.log(window.pageXOffset); // Zwraca przesunięcie strony w poziomie
console.log(window.pageYOffset); // Zwraca przesunięcie strony w pionie
```

prompt(message, default): Metoda prompt wyświetla okno dialogowe z pytaniem
o wprowadzenie danych. Przyjmuje dwa argumenty: wiadomość wyświetlaną w oknie

Obiekt window (cont)

> dialogowym i wartość domyślną. Zwraca wprowadzoną wartość jako ciąg znaków lub
null, jeśli użytkownik kliknie Anuluj.
const result = window.prompt("Podaj swoje imię:", "Anonim");
console.log(result); // Zawiera wprowadzoną wartość lub null

Przedrostek new

Przedrostek new jest używany w JavaScript do tworzenia nowych instancji obiektów
na podstawie funkcji konstr uktora. Gdy używasz słowa kluczowego new przed
wywołaniem funkcji konstr uktora, oznacza to, że chcesz utworzyć nowy obiekt
na podstawie tego konstr uktora.
Kiedy używamy new przed funkcją konstr uktora, następuje kilka ważnych rzeczy:
1. Tworzony jest nowy pusty obiekt.
2. this wewnątrz funkcji konstr uktora odnosi się do tego nowego obiektu.
3. Właści wości i metody są dodawane do tego obiektu przy użyciu składni
this.p rop ert yName lub this.m eth odName.
4. Jeśli funkcja konstr uktora nie zwraca żadnej wartości, to automa tycznie zwracany
jest ten nowy obiekt.
5. Jeśli funkcja konstr uktora zwraca inny obiekt (np. poprzez użycie instrukcji
return), to zamiast nowego obiektu zwracany jest ten inny obiekt.
Bez przedr ostka new funkcja konstr uktora będzie po prostu zwykłą funkcją,
a nie zostanie utworzony nowy obiekt. W takim przypadku, this wewnątrz funkcji
konstr uktora odnosić się będzie do globalnego obiektu (np. window w przypadku
przegl ądarki) lub do kontekstu wywołania, w zależności od tego, w jakim kontekście
zostanie wywołana.

Obiekty

1. Object: Reprezentuje ogólny obiekt i jest podstawowym typem dla wszystkich innych obiektów JavaSc ript.
const obj = { name: 'John', age: 30 };
2. Array: Przech owuje dane w uporzą dko wanej sekwencji i zapewnia wiele funkcji do manipu lacji tablicami.
const arr = [1, 2, 3, 4, 5];
3. String: Reprez entuje sekwencję znaków tekstowych i zapewnia wiele funkcji do manipu lacji tekstami.
const str = 'Hello, World!';
4. Number: Reprez entuje wartość liczbową i zapewnia wiele funkcji matema tyc znych.
const num = 42;
5. Boolean: Reprez entuje wartość logiczną true lub false.
const bool = true;
6. Date: Reprez entuje datę i czas.
const curren tDate = new Date();
consol e.l og(cur ren tDate); // Wyświetli aktualną datę i czas
7. Math: Dostarcza funkcje matema tyczne i stałe.
const result = Math.s qrt (25);
8. RegExp: Reprez entuje wyrażenie regularne, używane do dopaso wywania i manipu lowania tekstami.
const regex = /hello/gi;
9. Function: Reprez entuje funkcję w JavaSc ript.
function greet(name) {
    con sol e.log(`Hello, ${name}!);
}
10. Error: Reprez entuje błąd wykonania programu.
const error = new Error('So mething went wrong.');
11. JSON: Zapewnia funkcje do przeks zta łcania danych JavaScript na format

Cheatography

JavaScript basics Cheat Sheet

by sigeud via cheatography.com/188214/cs/39251/

Obiekty (cont)

> JSON (JavaScript Object Notation) i vice versa.
const data = JSON.parse('{"name":"John","age":30}');
12. Map: Przechowuje pary klucz-wartość i zapewnia różne metody do
manipulacji mapą.
const map = new Map();
map.set('key', 'value');
13. Set: Przechowuje unikalne wartości i zapewnia różne metody do manipulacji zbiorami.
const set = new Set();
set.add(1);
14. Promise: Reprezentuje asynchroniczne wykonanie operacji i obsługę
wyników lub błędów.
```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Done');
  }, 1000);
});
```
15. ArrayBuffer: Przechowuje sekwencję bajtów i zapewnia interfejs do manipulacji
binarnymi danymi.
const buffer = new ArrayBuffer(16);
16. DataView: Pozwala na odczyt i zapis danych binarnych w ArrayBuffer.
const buffer = new ArrayBuffer(16);
const view = new DataView(buffer);
17. XMLHttpRequest: służy do wykonywania asynchronicznych żądań HTTP
w języku JavaScript. Umożliwia komunikację z serwerem i pobieranie danych
w tle bez konieczności przeładowywania strony. Pozwala na wysyłanie żądań
HTTP, odbieranie odpowiedzi, zarządzanie nagłówkami, przesyłanie danych i
obsługę różnych typów odpowiedzi (np. tekst, JSON, XML).

Obiekty (cont)

> const xhr = new XMLHttpRequest();
18. Window: reprezentuje globalne okno przeglądarki, w którym wyświetlana
jest strona internetowa. Dostarcza wiele funkcji i właściwości związanych z
interakcją użytkownika, manipulacją strukturą dokumentu, zarząd-zaniem
historią przeglądarki itp. Udostępnia interfejs do operacji na elementach DOM,
obsługę zdarzeń, manipulację historią przeglądarki, tworzenie nowych okien i
ram, zarządzanie czasem (np. setTimeout, setInterval) oraz wiele innych funkcjonalności.
window.alert('Hello, World!');
19.Document: reprezentuje aktualnie załadowany dokument HTML w przeglądarce.
Zapewnia dostęp do elementów DOM na stronie, takich jak elementy HTML,
style, zdarzenia itp. Udostępnia metody do manipulacji zawartością dokumentu,
takie jak tworzenie, usuwanie i modyfikowanie elementów HTML, manipulowanie
styli, dodawanie i usuwanie zdarzeń. Pozwala na dostęp do różnych kolekcji
elementów, takich jak elementy po tagu, po klasie, po identyfikatorze itp.
const element = document.createElement('div');

Serializacja danych

```
Serializacja danych to proces konwersji struktur
danych, takich jak obiekty,
tablice, liczby, ciągi znaków itp., na format,
który można przesłać lub przech ować,
na przykład w formie ciągu znaków. Najczę ściej
używane formaty serial izacji
danych to JSON (JavaS cript Object Notation) i XML
(eXten sible Markup Language).
Serial izacja danych jest użyteczna w wielu
przypa dkach, takich jak:
Przesy łanie danych przez sieć: Serial izacja
danych umożliwia przesłanie struktur
danych między aplika cjami lub serwerami za pomocą
protokołów komuni kac yjnych,
takich jak HTTP. Dane mogą być zamieniane na
tekstowe reprez ent acje, które są
łatwe do przesłania i odczyt ania.
Przech owy wanie danych: Serial izacja danych
umożliwia zapisanie struktur danych
na dysku lub w bazie danych w celu późnie jszego
odczyt ania. Przech owy wanie
danych w formie serial izo wanej jest przydatne do
utrwalania stanu aplikacji lub
```

Serializacja danych (cont)

> przechowywania danych persystentnych.
Współpraca z innymi językami programowania: Serializacja danych pozwala na
wymianę danych między aplikacjami napisanymi w różnych językach programowania.
Dzięki temu można zintegrować aplikacje działające na różnych platformach.
W JavaScript jednym z najczęściej używanych formatów serializacji danych jest
JSON. JavaScript dostarcza funkcje JSON.stringify() do konwersji obiektów na
JSON i JSON.parse() do konwersji JSON z powrotem na obiekty JavaScript. Przykład użycia:

```
const obj = { name: 'John', age: 30 };
// Serializacja obiektu do formatu JSON
const json = JSON.stringify(obj);
console.log(json); // {"name":"John","age":30}
// Deserializacja JSON do obiektu JavaScript
const parsedObj = JSON.parse(json);
console.log(parsedObj); // { name: 'John', age: 30 }
```

Serializacja danych jest powszechnie stosowaną techniką w programowaniu,
która umożliwia efektywną wymianę i przechowywanie danych w różnych formatach.

Error

W JavaScript obiekty typu Error są wykorzystywane do reprezentowania błędów i
wyjątków. Obiekt Error zawiera informacje o błędzie, takie jak wiadomość opisująca
rodzaj błędu oraz stos wywołań.
Przykład użycia obiektu Error:

```
try {
   // Kod, który może generować błąd
    throw new Error('To jest jakiś błąd');
} catch (error) {
   // Obsługa błędu
    con sol e.l og( err or.m es sage); // 'To jest jakiś błąd'
}
```

Error (cont)

> W powyższym przykładzie używamy słowa kluczowego throw, aby wygenerować nowy
obiekt Error z określoną wiadomością. Następnie używamy bloku catch do
przechwycenia błędu i obsługi go. W tym przypadku po prostu wypisujemy
wiadomość błędu na konsoli.
Obiekty Error mogą być dziedziczone przez konkretne typy błędów, takie jak
TypeError, ReferenceError, SyntaxError itp. Na przykład:

```
try {
  // Kod, który może generować błąd typu TypeError
  throw new TypeError('To jest błąd typu TypeError');
} catch (error) {
  // Obsługa błędu
  console.log(error instanceof TypeError); // true
  console.log(error.message); // 'To jest błąd typu TypeError'
}
```

W powyższym przykładzie używamy obiektu TypeError do reprezentowania błędu
typu TypeError. Możemy sprawdzić typ błędu za pomocą operatora instanceof oraz
uzyskać jego wiadomość za pomocą właściwości message.
Obiekty Error posiadają również inne właściwości, takie jak name (nazwa błędu) i
stack (stos wywołań), które zawierają dodatkowe informacje diagnostyczne dotyczące
błędu.
Błędy i wyjątki są ważnym elementem w zarządzaniu błędami w JavaScript,
pozwalając na łatwiejszą identyfikację i obsługę różnych sytuacji błędnych w kodzie.
W języku JavaScript istnieje kilka wbudowanych typów błędów (errors), które dziedziczą
o klasie Error. Oto niektóre z najczęściej spotykanych typów błędów:
- Error: Podstawowy typ błędu. Może być używany jako ogólny typ błędu.
- SyntaxError: Występuje, gdy w kodzie występuje błąd składniowy. Na przykład,
nieprawidłowe użycie składni języka JavaScript.
- ReferenceError: Występuje, gdy odwołujemy się do nieistniejącej zmiennej lub obiektu.

Error (cont)

> Na przykład, próba odwołania się do niezadeklarowanej zmiennej.
- TypeError: Występuje, gdy wykonywane jest nieprawidłowe działanie na typie danych.
Na przykład, próba wykonania działania matematycznego na wartości niebędącej liczbą.
- RangeError: Występuje, gdy wartość znajduje się poza zakresem, który jest dozwolony.
Na przykład, próba utworzenia tablicy o zbyt dużym rozmiarze.
- EvalError: Występuje, gdy wystąpi błąd w funkcji eval().
- URIError: Występuje, gdy URI (Uniform Resource Identifier) nie jest poprawnie
zakodowany.
- AggregateError: Występuje w przypadku wielu błędów zgromadzonych w jednym
obiekcie. Jest używany w przypadku obsługi wielu promisów.
Warto zauważyć, że możemy również tworzyć własne typy błędów, dziedzicząc po
klasie Error. Tworzenie niestandardowych typów błędów może być przydatne w celu
lepszego zorganizowania obsługi błędów w naszym kodzie.
Przykład użycia niestandardowego typu błędu:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = 'CustomError';
  }
}
try {
  throw new CustomError('To jest niestandardowy błąd');
} catch (error) {
  console.log(error instanceof CustomError); // true
  console.log(error.message); // 'To jest niestandardowy błąd'
}
```

W powyższym przykładzie tworzymy niestandardowy typ błędu o nazwie
CustomError, który dziedziczy po klasie Error. Możemy go obsługiwać podobnie

Error (cont)

> jak inne wbudowane typy błędów.

XMLHttpRequest

```
XMLHttpRequest jest obiektem wbudowanym w
przeglądarki internetowe,
które umożliwia komuni kację z serwerem za pomocą
protokołu HTTP lub HTTPS.
Służy do wysyłania asynch ron icznych żądań HTTP i
odbierania odpowiedzi w
formacie tekstowym, XML lub JSON.
Główne kroki w użyciu XMLHtt pRe quest obejmują:
Tworzenie obiektu XMLHtt pRe quest:
const xhr = new XMLHtt pRe que st();
Konfig uracja żądania HTTP:
xhr.op en( method, url, async);
method: Metoda HTTP, np. " GET ", " POS T", " -
PUT ", " DEL ETE ".
url: Adres URL, do którego wysyłane jest żądanie.
async (opcjo nalne): Określa, czy żądanie ma być
asynch ron iczne (domyślnie true) lub
synchr oniczne (false).
Ustalenie funkcji obsług ującej zdarzenie readyS -
tat eCh ange:
xhr.on rea dys tat echange = function () {
   if (xhr.r ead yState === 4 && xhr.status ===
200) {
       // Obsługa zakońc zonego żądania
   }
};
readyState reprez entuje stan obiektu XMLHtt pRe -
quest i może przyjmować różne
wartości od 0 do 4. Wartość 4 oznacza, że żądanie
zostało zakończone i odpowiedź
jest gotowa do odczytu.
Wysłanie żądania:
xhr.se nd( data);
data (opcjo nalne) reprez entuje dane, które mogą
być przesłane wraz z żądaniem,
np. w przypadku metod POST lub PUT.
```

XMLHttpRequest (cont)

> Obsługa odpowiedzi:
xhr.responseText; // Odpowiedź jako tekst
xhr.responseXML; // Odpowiedź jako dokument XML
xhr.status; // Kod stanu odpowiedzi HTTP
Odpowiedź z serwera jest dostępna za pomocą właściwości responseText lub responseXML. Właściwość status zawiera kod stanu odpowiedzi HTTP, np. 200 oznacza sukces.
XMLHttpRequest dostarcza również inne funkcje i metody, takie jak setRequestHeader(), która umożliwia ustawianie nagłówków żądania HTTP, oraz
abort(), która służy do anulowania żądania w trakcie trwania.
Warto również zaznaczyć, że wraz z pojawieniem się nowszych standardów w
JavaScript, takich jak Fetch API i XMLHttpRequest zostało zastąpione przez te
bardziej nowoczesne metody obsługi żądań HTTP.

Iteracja po elementach kolekcji

W języku JavaScript istnieje kilka sposobów iteracji po elementach kolekcji, takich jak tablice lub obiekty. Oto kilka popula rnych sposobów iteracji:
- Pętla for: Tradycyjna pętla for pozwala na iterację po indeksach kolekcji. Jest szczeg ólnie przydatna do iteracji po tablicach.
Przykład:

```
const tablica = [1, 2, 3, 4];
for (let i = 0; i < tablic a.l engths; i++) {
    con sol e.l og( tab lic a[i]);
}
```

- Pętla for...of: Pętla for...of umożliwia iterację po elementach kolekcji, takich jak tablice
lub obiekty iterow alne, bez koniec zności odwoły wania się do indeksów.
Przykład:

```
const tablica = [1, 2, 3, 4];
for (let element of tablica) {
    con sol e.l og( ele ment);
```

Iteracja po elementach kolekcji (cont)

> }
- Metoda forEach: Metoda forEach jest dostępna dla tablic i pozwala na wykonanie
określonej funkcji dla każdego elementu tablicy.
Przykład:

```
const tablica = [1, 2, 3, 4];
tablica.forEach((element) => {
  console.log(element);
});
```

- Metoda map: Metoda map również jest dostępna dla tablic i pozwala na wykonanie
określonej funkcji dla każdego elementu tablicy, a następnie zwrócenie nowej tablicy
zawierającej wyniki.
Przykład:

```
const tablica = [1, 2, 3, 4];
const nowaTablica = tablica.map((element) => {
  return element * 2;
});
console.log(nowaTablica);
```

- Pętla for...in: Pętla for...in umożliwia iterację po nazwach właściwości obiektu.
Jest szczególnie przydatna do iteracji po obiektach.
Przykład:

```
const obiekt = { a: 1, b: 2, c: 3 };
for (let key in obiekt) {
  console.log(key, obiekt[key]);
}
```

Klasa

Klasy są wprowadzone w języku JavaScript w standardzie ECMAScript 2015 (ES6)
jako syntak tyczny cukier nad protot ypami, umożli wiający defini owanie obiektów i ich właści wości oraz metody w bardziej deklar atywny sposób. Klasy pozwalają na tworzenie obiektów o podobnej strukturze i zachow aniu.
Oto kilka kluczowych cech klas w JavaSc ript:
Deklaracja klasy: Klasa może być zadekl arowana za pomocą słowa kluczowego class, po którym podajemy nazwę klasy.
Przykład:

```
class Klasa {
   // Ciało klasy
}
```

Konstr uktor: Konstr uktor jest specjalną metodą w klasie, która jest wywoływana podczas tworzenia nowego obiektu na podstawie klasy. Może być używany do inicja lizacji właści wości obiektu.
Przykład:

```
class Klasa {
    con str uct or( par ametr) {
        thi s.w las cowosc = parametr;
   }
}
const obiekt = new Klasa( 'wa rto ść');
```

Metody: Metody to funkcje zdefin iowane wewnątrz klasy, które wykonują określone operacje na obiekcie. Mogą być używane do manipu - lacji właści woś ciami obiektu lub wykony wania innych operacji.
Przykład:

```
class Klasa {
    met oda() {
        con sol e.l og( 'Wy wołanie metody');
> }
}
const obiekt = new Klasa();
obiekt.metoda();
```

Klasa (cont)

Dziedziczenie: Klasa może dziedziczyć właściwości i metody z innej klasy za pomocą słowa kluczowego extends. Dziedziczenie pozwala na tworzenie hierarchii klas i ponowne wykorzystanie kodu.
Przykład:

```
class KlasaPodstawowa {
  metoda() {
    console.log('Metoda z klasy podstawowej');
  }
}
class KlasaPochodna extends KlasaPodstawowa {
  // Klasa pochodna dziedziczy metody z klasy podstawowej
}
const obiekt = new KlasaPochodna();
obiekt.metoda(); // Wywołanie metody z klasy podstawowej
```

Słowo kluczowe super w języku JavaScript jest używane w kontekście dziedziczenia klas, aby odwoływać się do właściwości i metod z klasy nadrzędnej (klasy bazowej). Pozwala to na rozszerzanie funkcjonalności klasy nadrzędnej w klasie pochodnej oraz wywoływanie konstruktora klasy nadrzędnej.
Oto kilka zastosowań słowa kluczowego super:
Wywołanie konstruktora klasy nadrzędnej: Przy dziedziczeniu klas, konstruktor klasy pochodnej może wywołać konstruktor klasy nadrzędnej za pomocą super(). Pozwala to na inicjalizację właściwości klasy nadrzędnej przed zainicjalizo- waniem właściwości klasy pochodnej.
Przykład:

Klasa (cont)

```
> class KlasaNadrzedna {
  constructor(parametr) {
    this.wlascowoscNadrzedna = parametr;
  }
}
class KlasaPochodna extends KlasaNadrzedna {
  constructor(parametr1, parametr2) {
    super(parametr1); // Wywołanie konstruktora klasy nadrzędnej
    this.wlascowoscPochodna = parametr2;
  }
}
const obiekt = new KlasaPochodna('wartość1', 'wartość2');
console.log(obiekt.wlascowoscNadrzedna); // wartość1
console.log(obiekt.wlascowoscPochodna); // wartość2
```

Odwoływanie się do metody klasy nadrzędnej: Słowo kluczowe super może być
używane do odwoływania się do metody z klasy nadrzędnej w klasie pochodnej.
Umożliwia to rozszerzanie funkcjonalności metody z klasy nadrzędnej lub nadpisywanie
jej zachowania.
Przykład:

```
class KlasaNadrzedna {
  metoda() {
    console.log('Metoda z klasy nadrzędnej');
  }
}
class KlasaPochodna extends KlasaNadrzedna {
  metoda() {
    super.metoda(); // Wywołanie metody klasy nadrzędnej
    console.log('Metoda z klasy pochodnej');
```

Klasa (cont)

```
>   }
}
const obiekt = new KlasaPochodna();
obiekt.metoda();
// Wyjście:
// Metoda z klasy nadrzędnej
// Metoda z klasy pochodnej
```

Słowo kluczowe super jest używane do odwoływania się do elementów klasy
nadrzędnej w kontekście dziedziczenia. Dzięki temu możliwe jest rozszerzanie i dostęp
do funkcjonalności klasy nadrzędnej w klasie pochodnej.
Klasy w JavaScript zapewniają bardziej zwięzłą i deklaratywną składnię do tworzenia
obiektów i zarządzania ich zachowaniem. Mogą być używane do implementacji
obiektowo zorientowanego programowania w JavaScript.

```
    let Animal = class BasicAnimal {
      constructor(name) {
        this.name = name
        this._age = 1;

        if (Animal.count === undefined) Animal.count = 0;
        Animal.count++; // emulujemy statyczne property w ES6
      }
      getName = () => {
        return this.name;
      }
      set age(value) {
        if (value > 0) this._age = value;
      }
      get age() {
        return this._age;
```

Klasa (cont)

```
>         }
        static getNewAnimal() {
            return new Animal("Default animal");
        }
        static getAnimalCount() {
            return Animal.count;
        }
    }
    const animal1 = new Animal("Tiger");
    console.log( animal1.getName() ); // "Tiger"
    console.log( Animal.name ); // BasicAnimal
    animal1.age = 10;
    // animal1._age = 22; // można zmodyfikować, nie ma private w
ES6
    console.log( animal1.age ); // 10
    const animal2 = Animal.getNewAnimal();
    console.log(animal2.getName()); // Default animal
    console.dir(Animal);
    console.log( Animal.getAnimalCount() ); // 2
```

Closures

Closures (domknięcia) są ważnym konceptem w języku JavaScript. Oznaczają one zdolność funkcji do zapami ętania i uzyski wania dostępu do zmiennych ze swojego otoczenia, nawet po opuszc zeniu tego otoczenia. Oto prosty przykład, który demons truje closures:

```
function outer() {
   var outerV ariable = 'Hello';
    fun ction inner() {
       var innerV ariable = 'World';
        con sol e.l og( out erV ariable + ' ' +
innerV ari able);
```

Closures (cont)

```
> }
  return inner;
}
var closureFn = outer();
closureFn(); // Wynik: "Hello World"
```

W tym przykładzie funkcja inner jest zagnieżdżona w funkcji outer. Funkcja inner ma dostęp do zmiennej outerVariable z funkcji outer, nawet po tym jak funkcja outer zakończyła swoje działanie i została wywołana. Closures są przydatne w wielu sytuacjach, takich jak tworzenie prywatnych zmiennych, implementacja funkcji zwrotnych (callback) i zarządzanie stanem w programowaniu asynchronicznym. Pozwalają one na utrzymanie stanu i zapamiętywanie danych między różnymi wywołaniami funkcji. Warto zauważyć, że closures w języku JavaScript mogą mieć znaczący wpływ na zarządzanie pamięcią. Jeśli closure nadal utrzymuje referencję do obiektów lub zmiennych, które nie są już potrzebne, może to prowadzić do wycieków pamięci.

Currying

Currying to technika programistyczna, która polega na przekształceniu funkcji o wielu argume ntach w sekwencję funkcji o pojedy nczych argume ntach. Pozwala to na tworzenie bardziej elasty cznych i modułowych funkcji. W curryingu funkcja przyjmuje jeden argument i zwraca nową funkcję, która również przyjmuje jeden argument, a tak dalej, aż wszystkie argumenty zostaną dostar czone i funkcja zostanie ostate cznie wywołana. Oto prosty przykład:

```
function add(a) {
    return functi on(b) {
        return a + b;
    }
}
```

Currying (cont)

> var add5 = add(5); // Zwraca funkcję, która dodaje 5 do przekazanego argumentu
console.log(add5(2)); // Wynik: 7
console.log(add5(10)); // Wynik: 15
W powyższym przykładzie funkcja add jest funkcją currying. Przyjmuje ona argument
a i zwraca funkcję, która dodaje przekazany argument b do a. Wywołanie add(5) zwraca
funkcję, która dodaje 5 do przekazanego argumentu. Następnie możemy wielokrotnie
wywoływać zwróconą funkcję, przekazując różne argumenty. Currying jest przydatne w wielu sytuacjach, szczególnie przy tworzeniu funkcji
cząstkowych (partial functions) i kompozycji funkcji. Umożliwia tworzenie bardziej
elastycznych funkcji, które można łatwo dostosować do różnych przypadków użycia.
Warto również wspomnieć, że w języku JavaScript można używać metod takich jak
bind() do osiągnięcia efektu podobnego do currying. Metoda bind() pozwala na utworzenie
nowej funkcji, która jest związana z określonym kontekstem i zestawem argumentów.

ECMAScript 2021 (ES12)

```
- Private Fields and Methods (Pola i metody prywatne):
ES12 wprowadza możliwość defini owania pól i metod prywatnych w klasach
za pomocą prefixu #. Przykład:
class MyClass {
    #pr iva teField = 10;
    #pr iva teM ethod() {
        return this.# pri vat eField;
   }
    get Pri vat eFi eld Value() {
        return this.# pri vat eField;
   }
}
const myObject = new MyClass();
consol e.l og( myO bje ct.g et Pri vat eFi eld - Val ue()); // 10
```

ECMAScript 2021 (ES12) (cont)

> console.log(myObject.privateField); // undefined
console.log(myObject.privateMethod()); // Error: privateMethod is not a function
- Logical Assignment Operators (Operatorzy logiczne przypisania):
ES12 wprowadza skrócone zapisy operacji logicznych z przypisaniem, takie jak
||=, &&=, ??=. Pozwalają one na wykonanie operacji logicznej i przypisania wartości
w jednym kroku. Przykład:
let x = 5;
x ||= 10; // Jeśli x jest fałszywe (falsy), przypisz mu wartość 10
console.log(x); // 5
let y = null;
y ??= 'Hello'; // Jeśli y jest null lub undefined, przypisz mu wartość 'Hello'
console.log(y); // 'Hello'
- Numeric Separator (Separator numeryczny):
ES12 wprowadza możliwość stosowania podkreślenia _ jako separatora w liczbach,
aby ułatwić czytanie i zrozumienie dużych liczb. Przykład:
const bigNumber = 1_000_000;
console.log(bigNumber); // 1000000
- Promise.any():
ES12 wprowadza funkcję Promise.any(), która przyjmuje tablicę obiektów Promise i
zwraca pierwszy rozwiązany Promise. Przykład:

```
const promises = [
  fetch('https://api.example.com/data1'),
  fetch('https://api.example.com/data2'),
  fetch('https://api.example.com/data3')
];
Promise.any(promises)
  .then(response => console.log(response))
  .catch(error => console.error(error));
```

ECMAScript 2021 (ES12) (cont)

> - WeakRef to nowa funkcjonalność wprowadzona w ECMAScript 2021 (ES12).
Służy do tworzenia słabych referencji do obiektów, co umożliwia bardziej elastyczne
zarządzanie pamięcią w JavaScript.
Słabe referencje są szczególnie przydatne w sytuacjach, gdy chcemy monitorować
obiekty, ale jednocześnie pozwolić na ich automatyczne usuwanie przez mechanizm
zarządzania pamięcią, jeśli nie są już używane. Słabe referencje nie zatrzymują
automatycznie obiektów przed usunięciem przez garbage collector.
Oto prosty przykład użycia WeakRef:

```
let obj = { data: "Hello" };
const ref = new WeakRef(obj);
// Sprawdzanie, czy obiekt wciąż istnieje
console.log(ref.deref()); // { data: "Hello" }
obj = null; // Usunięcie referencji do obiektu
// Sprawdzanie, czy obiekt został usunięty
console.log(ref.deref()); // null
```

W powyższym przykładzie tworzymy obiekt obj, a następnie tworzymy słabą
referencję ref do tego obiektu za pomocą konstruktora WeakRef. Możemy użyć
metody deref() na obiekcie ref, aby sprawdzić, czy obiekt wciąż istnieje.
Po usunięciu referencji do obiektu poprzez przypisanie null do zmiennej obj,
ponowne wywołanie deref() zwraca null, co oznacza, że obiekt został usunięty.
Słabe referencje są szczególnie przydatne w przypadku tworzenia pamięciochłonnych
obiektów, które mogą być dynamicznie usuwane, gdy nie są już potrzebne.

ECMAScript 2019 (ES10)

Oto lista niektórych nowych funkcji i zasad wprowadzonych w ECMAScript 2019 (ES10):
- Array.f lat() i Array.f la tMap(): Metoda flat() służy do spłasz czania tablicy, usuwając
zagnie żdżone tablice i łącząc je w jedną płaską tablicę. Metoda flatMap() pozwala na
jednoc zesne spłasz czenie tablicy i zastos owanie funkcji do każdego elementu.
Przykład:

ECMAScript 2019 (ES10) (cont)

```
> const nestedArray = [1, [2, 3], [4, [5, 6]]];
const flattenedArray = nestedArray.flat();
console.log(flattenedArray); // [1, 2, 3, 4, [5, 6]]
const numbers = [1, 2, 3];
const doubledArray = numbers.flatMap(num => [num, num * 2]);
console.log(doubledArray); // [1, 2, 2, 4, 3, 6]
```

- String.trimStart() i String.trimEnd(): Metody trimStart() i trimEnd() służą do usuwania
spacji na początku i końcu łańcucha znaków. Zastępują one metody trimLeft() i trimRight()
używane wcześniej. Przykład:

```
const text = ' Hello World ';
console.log(text.trimStart()); // 'Hello World '
console.log(text.trimEnd()); // ' Hello World'
```

- Optional Catch Binding: Pozwala na użycie pustego identyfikatora w bloku catch,
jeśli nie jest potrzebny dostęp do błędu. Przykład:

```
try {
  // kod generujący błąd
} catch {
  console.log('Wystąpił błąd');
}
```

- Object.fromEntries(): Metoda fromEntries() przekształca listę par klucz-wartość
na obiekt. Przykład:

```
const entries = [['a', 1], ['b', 2], ['c', 3]];
const obj = Object.fromEntries(entries);
console.log(obj); // { a: 1, b: 2, c: 3 }
```

- Symbol Description Accessor: Umożliwia uzyskanie dostępu do opisu symbolu za
pomocą metody description. Przykład:

```
const symbol = Symbol('My Symbol');
console.log(symbol.description); // 'My Symbol'
```

ECMAScript 2017 (ES8)

Oto kilka nowości wprowadzonych w ECMAScript 2017 (ES8) wraz z przykładami:

- Async Functions: Wprowa dzono asynch ron iczne funkcje, które ułatwiają pracę z asynch ron icznym kodem. Wykorz ystuje się do tego słowo kluczowe async oraz await.

Przykład:

```
async function fetchD ata() {
   try {
        const response = await fetch( 'ht -
tps :// api.ex amp le.c om /da ta');
        const data = await respon se.j son();
        return data;
   } catch (error) {
        con sol e.e rro r(' Err or:', error);
   }
}
```

- Object.va lues(): Metoda Object.va lues() zwraca tablicę zawier ającą wartości wszystkich właści - wości obiektu. Przykład:

```
const obj = { a: 1, b: 2, c: 3 };
const values = Object.va lue s(obj);
consol e.l og( val ues); // [1, 2, 3]
```

- Object.en tri es(): Metoda Object.en tries() zwraca tablicę zawier ającą tablice składające się z klucza i wartości każdej właści wości obiektu. Przykład:

```
const obj = { a: 1, b: 2, c: 3 };
const entries = Object.en tri es( obj);
consol e.l og( ent ries); // [["a ", 1], ["b", 2], ["c", 3]]
```

- String padding: Dodano możliwość wyrówn ywania łańcucha znaków poprzez uzupeł nianie go odpowi ednią ilością znaków. Wykorz ystuje się do tego metody padStart() i padEnd(). Przykład:

```
const str = 'Hello';
consol e.l og( str.pa dSt art(10,' ');' // "Hello"
```

ECMAScript 2017 (ES8) (cont)

```
> console.log(str.padEnd(10, '-')); // "Hello-----"
```

- Object.getOwnPropertyDescriptors(): Metoda Object.getOwnPr- opertyDescriptors() zwraca wszystkie deskryptory właściwości obiektu. Przykład:

```
const obj = {
  name: 'John',
  age: 30
};
const descriptors = Object.getOwnPropertyDescriptors(obj);
console.log(descriptors);
/*
{
  name: { value: 'John', writable: true, enumerable: true, configurable: true },
  age: { value: 30, writable: true, enumerable: true, configurable: true
}
}
*/
```

- SharedArrayBuffer i Atomics: Wprowadzono nowe obiekty Shared- ArrayBuffer i Atomics do obsługi współdzielonych buforów pomiędzy wątkami w środowisku wielowątkowym.

-Async Iteration: Wprowadzono możliwość iteracji asynchronicznej przy użyciu pętli for-await-of, co ułatwia pracę z asynchronicznymi operacjami. Przykład:

```
async function* asyncGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
(async () => {
  for await (const num of asyncGenerator()) {
    console.log(num);
  }
```

ECMAScript 2017 (ES8) (cont)

```
> })();
```

ECMAScript 2016 (ES7)

ES7, również znany jako ECMAScript 2016, wprowadził niewiele nowych funkcji w porównaniu do poprze dnich wersji języka JavaSc - ript. Poniżej znajduje się lista kilku zmian wprowa dzonych w ES7 wraz z przykł adami:

- Includes:

Metoda includes została dodana do prototypu tablic w celu sprawd zania, czy dany element istnieje w tablicy. Zwraca wartość logiczną true, jeśli element istnieje, lub false, jeśli nie istnieje. Przykład:

```
const numbers = [1, 2, 3, 4, 5];
consol e.l og( num ber s.i ncl ude s(3)); // true
consol e.l og( num ber s.i ncl ude s(6)); // false
```

- Potęgo wanie:

Operator potęgo wania ** został wprowa dzony w ES7 do obliczania potęgi liczby. Przykład:

```
consol e.log(2 ** 3); // 8
consol e.log(5 ** 2); // 25
```

- Rest Proper ties:

W ES7 dodano obsługę reszty właści wości w destru ktu ryzacji obiektów. Umożliwia to zbieranie pozost ałych właści wości obiektu do jednej zmiennej. Przykład:

```
const { name, age, ...rest } = { name: 'John', age: 30, gender: 'male', city: 'New York' };
consol e.l og( name); // 'John'
consol e.l og( age); // 30
consol e.l og( rest); // { gender: 'male', city: 'New York' }
```

- Async Functions:

W ES7 wprowa dzono słowo kluczowe async, które umożliwia tworzenie funkcji asynch ron icz nych. Funkcje asynch ron iczne obsługują Promise i umożli wiają korzys tanie z await wewnątrz funkcji, co ułatwia progra - mowanie asynch ron iczne. Przykład:

ECMAScript 2016 (ES7) (cont)

```
> async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
}
```

- Shared Memory i Atomics (Atomics API):

ES7 wprowadza obsługę pamięci współdzielonej i wprowadza nowe API Atomics do wykonywania operacji atomowych na pamięci współdzielonej. Ta funkcjonalność jest szczególnie przydatna w programowaniu równoległym i wielowątk- owym, ale wymaga bardziej zaawansowanych technik programistycznych.

Warto zauważyć, że ES7 nie wprowadził dużej liczby nowych funkcji, ale skupił się głównie na poprawie i rozszerzeniu istniejących funkcjonalności języka JavaScript.

Instrukcje warunkowe: if else if

```
if (tempe rature < 0) consol e.l og( " Bie rzemy kurt
else if (tempe rature == 0) consol e.l og( " Bie rzem
rtkę ")
else if (tempe rature >= 0) consol e.l og( " Bie rzem
ękaw")
else consol e.l og( " Zos tajemy w domu")
```

switch case

```
switch (data3) {
                        case 0:
                                con sol e.l -
og( "data równe 0")
                                break
                        case 1:
                                con sol e.l -
og( "data równe 1")
                                break
                        case 3:
                                con sol e.l -
og( "data równe 3")
```

switch case (cont)

```
>            break
        case 5:
           console.log("data równe 5")
           break
        default:
           console.log("data równe jest dokładnie:", data3)
     }
```

while

```
while ( i < 3) {console.log("while oraz wartość i:
" + i)
                        i = i + 1 }
```

do while

```
do {console.log("do while oraz wartość b:", b)
             b++ // to samo co b = b + 1
               } while (b < 4)
```

for

```
for (let c = 0; c < 5; c++) {
consol e.l og( " pętla c, wartość c:", c)}
```

Number - bezpieczne zakresy

Number.MA X_S AFE _IN TEGER	900719 925 474099 1
Number.MI N_S AFE _IN TEGER	-90071 992 547 40 991
Number.is Saf eIn teger(Math.p ow(2,53) - 1)	true
Number.is Saf eIn teger(Math.p ow(2,53))	false

Infinity

consol e.log(1 / 0)	Infinity
consol e.l og(-1 / 0)	-Infinity
Number.is Fin ite(35)	true
Number.is Fin ite (1/0)	false

Numbers - właściwości i funkcje

```
console.log(Number.MAX_VALUE); //
1.7976931348623157e+308
consol e.l og( Num ber.MI N_V ALUE); // 5e-324
Funkcja Number.pa rse Int(): Konwertuje ciąg
znaków na liczbę całkowitą.
consol e.l og( Num ber.pa rse Int ("42 ")); // 42
consol e.l og( Num ber.pa rse Int ("3.1 4")); // 3
consol e.l og( Num ber.pa rse Int ("ab c")); // NaN
Funkcja Number.pa rse Flo at(): Konwertuje ciąg
znaków na liczbę zmienn opr zec inkową.
consol e.l og( Num ber.pa rse Flo at( " 3.1 4"));
// 3.14
consol e.l og( Num ber.pa rse Flo at( " 42.0 "));
// 42
consol e.l og( Num ber.pa rse Flo at( " abc ")); //
NaN
Funkcja Number.is Int eger(): Sprawdza, czy
wartość jest liczbą całkowitą.
consol e.l og( Num ber.is Int ege r(42)); // true
consol e.l og( Num ber.is Int ege r(3.14)); //
false
consol e.l og( Num ber.is Int ege r("4 2")); //
false
consol e.l og( Num ber.is Int ege r(N aN)); //
false
Metoda Number.to Fix ed(): Określa liczbę miejsc
dziesi ętnych po przecinku dla liczby
zmienn opr zec inkowej i zwraca wynik jako ciąg
znaków.
let num = 3.14159;
consol e.l og( num.to Fix ed(2)); // " 3.1 4"
consol e.l og( num.to Fix ed(0)); // " 3"
consol e.l og( num.to Fix ed(5)); // " 3.1 415 9"
Uwaga: Number.to Fixed() zwraca wynik jako string,
a nie jako liczbę.
Jeśli chcesz uzyskać liczbę z ustaloną ilością
miejsc dziesi ętnych, można
```

Numbers - właściwości i funkcje (cont)

> użyć parseFloat() lub Number().

```
let num = 3.14159;
console.log(parseFloat(num.toFixed(2))); // 3.14
console.log(Number(num.toFixed(2))); // 3.14
```

Zaokrąglanie liczb

```
Metoda Math.round(): Zaokrągla liczbę do
najbliższej wartości całkowitej.
consol e.l og( Mat h.r oun d(3.7)); // 4
consol e.l og( Mat h.r oun d(3.2)); // 3
Metoda Math.f loor(): Zaokrągla liczbę w dół do
najbli ższej wartości całkowitej
mniejszej lub równej danej liczbie.
consol e.l og( Mat h.f loo r(3.7)); // 3
consol e.l og( Mat h.f loo r(3.2)); // 3
Metoda Math.c eil(): Zaokrągla liczbę w górę do
najbli ższej wartości całkowitej
większej lub równej danej liczbie.
consol e.l og( Mat h.c eil (3.7)); // 4
consol e.l og( Mat h.c eil (3.2)); // 4
Metoda Math.t runc(): Zwraca tylko całkowitą część
liczby, odrzucając jej
część dziesi ętną.
consol e.l og( Mat h.t run c(3.7)); // 3
consol e.l og( Mat h.t run c(3.2)); // 3
Metoda Number.to Fix ed(): Zaokrągla liczbę do
określonej liczby miejsc
dziesi ętnych i zwraca wynik jako ciąg znaków.
consol e.l og( (3.1 41 59).to Fix ed(2)); // " -
3.1 4"
consol e.l og( (3.1 41 59).to Fix ed(0)); // " 3"
Metody Math.f loor(), Math.c eil(), Math.t runc()
można także wykorz ystać w połączeniu
z mnożeniem i dziele niem, aby zaokrąglić liczbę
do określonej dokład ności miejsc
dziesi ętnych.
let num = 3.14159;
```

Zaokrąglanie liczb (cont)

> let roundedNum = Math.floor(num * 100) / 100;

```
console.log(roundedNum); // 3.14
```

Operatory przypisania

```
+= - Dodawanie i przypisanie
Przykład: x += 2; // równoważne zapisowi: x = x +
2;
-= - Odejmo wanie i przypi sanie
Przykład: x -= 3; // równoważne zapisowi: x = x -
3;
*= - Mnożenie i przypi sanie
Przykład: x = 4; // równoważne zapisowi: x = x  4;
/= - Dzielenie i przypi sanie
Przykład: x /= 2; // równoważne zapisowi: x = x /
2;
%= - Reszta z dzielenia i przypi sanie
Przykład: x %= 3; // równoważne zapisowi: x = x %
3;
**= - Potęgo wanie i przypi sanie
Przykład: x = 2; // równoważne zapisowi: x = x  2;
<<= - Przesu nięcie bitowe w lewo i przypi sanie
Przykład: x <<= 3; // równoważne zapisowi: x = x
<< 3;
>>= - Przesu nięcie bitowe w prawo (ze znakiem) i
przypi sanie
Przykład: x >>= 2; // równoważne zapisowi: x = x
>> 2;
>>>= - Przesu nięcie bitowe w prawo (bez znaku) i
przypi sanie
Przykład: x >>>= 4; // równoważne zapisowi: x = x
>>> 4;
&= - Bitowe AND i przypi sanie
Przykład: x &= 5; // równoważne zapisowi: x = x &
5;
|= - Bitowe OR i przypi sanie
Przykład: x |= 8; // równoważne zapisowi: x = x |
8;
^= - Bitowe XOR i przypi sanie
Przykład: x = 3; // równoważne zapisowi: x = x 3;
```

Operatory arytmetyczne

```
+ - Dodawanie
Przykład: 2 + 3 // wynik: 5
- - Odejmo wanie
Przykład: 5 - 2 // wynik: 3
* - Mnożenie
Przykład: 2 * 3 // wynik: 6
/ - Dzielenie
Przykład: 6 / 2 // wynik: 3
% - Reszta z dzielenia (modulo)
Przykład: 7 % 3 // wynik: 1
** - Potęgo wanie
Przykład: 2 ** 3 // wynik: 8
++ - Inkrem entacja (zwięk szenie o 1)
Przykład: let x = 5; x++; // x będzie miało wartość
6
-- - Dekrem entacja (zmnie jszenie o 1)
Przykład: let x = 5; x--; // x będzie miało
wartość 4
```

Operatory porównania (relacyjne)

```
== - Równy wartości (konwertuje typy)
Przykład: 2 == '2' // wynik: true
=== - Równy wartości i typu (nie konwertuje typów)
Przykład: 2 === '2' // wynik: false
!= - Nie równy wartości (konwe rtuje typy)
Przykład: 2 != '3' // wynik: true
!== - Nie równy wartości i typu (nie konwertuje
typów)
Przykład: 2 !== '3' // wynik: true
> - Większy niż
Przykład: 5 > 3 // wynik: true
< - Mniejszy niż
```

Operatory porównania (relacyjne) (cont)

> Przykład: 3 < 5 // wynik: true
>= - Większy lub równy
Przykład: 5 >= 5 // wynik: true
<= - Mniejszy lub równy
Przykład: 3 <= 5 // wynik: true
Operatory porównania z obiektami (== i ===) porównują referencje obiektów,
a nie ich wartości. Dlatego dla dwóch różnych obiektów o identycznym zewnętrznym
wyglądzie, operator porównania zwróci false, chyba że obiekt jest bezpośrednio
porównywany z samym sobą.

```
let obj1 = { value: 5 };
let obj2 = { value: 5 };
let obj3 = obj1;
console.log(obj1 == obj2); // false
console.log(obj1 === obj2); // false
console.log(obj1 === obj3); // true
```

Operatory logiczne

```
&& - Operator logiczny "i" (AND)
Zwraca true, jeśli oba wyrażenia są prawdziwe, w
przeciwnym razie zwraca false.
Przykład: true && false // wynik: false
|| - Operator logiczny " lub " (OR)
Zwraca true, jeśli przyna jmniej jedno z wyrażeń
jest prawdziwe, w przeciwnym razie
zwraca false.
Przykład: true || false // wynik: true
! - Operator logiczny " nie " (NOT)
Zwraca wartość przeciwną do wartości wyrażenia,
czyli true zamienia na false
i vice versa.
Przykład: !true // wynik: false
Operatory logiczne && i || wykorz ystują " sho -
rt- circuit evalua tio n" (ocenianie
```

Operatory logiczne (cont)

> krótkiego obwodu). Oznacza to, że jeśli wynik można określić na podstawie
pierwszego wyrażenia, drugie wyrażenie nie jest oceniane.

```
let a = 5;
let b = 10;
if (a > 0 && b > 0) {
  console.log("Oba warunki są spełnione");
} else {
  console.log("Co najmniej jeden warunek nie jest spełniony");
}
```

W tym przypadku, jeśli a i b są większe od zera, to oba warunki są spełnione i
zostanie wyświetlony odpowiedni komunikat. Jeśli przynajmniej jeden z warunków
nie jest spełniony, zostanie wyświetlony inny komunikat.

Operatory bitowe

& - Operator bitowy AND
Wykonuje operację logicznego AND bit po bicie na dwóch operan dach.
Przykład: 5 & 3 // wynik: 1 (binarnie: 0101 & 0011 = 0001)
| - Operator bitowy OR
Wykonuje operację logicznego OR bit po bicie na dwóch operan dach.
Przykład: 5 | 3 // wynik: 7 (binarnie: 0101 | 0011 = 0111)
^ - Operator bitowy XOR (exclusive OR)
Wykonuje operację logicznego XOR bit po bicie na dwóch operan dach.
Przykład: 5 ^ 3 // wynik: 6 (binarnie: 0101 0011 = 0110)
~ - Operator bitowy NOT (negacja)
Wykonuje negację bitową na jednym operancie. Odwraca bity.
Przykład: ~5 // wynik: -6 (binarnie: ~0101 = 1010)
<< - Operator bitowy przesu nięcia w lewo
Przesuwa bity w lewo o określoną ilość pozycji.
Przykład: 5 << 2 // wynik: 20 (binarnie: 0101 << 2 = 10100)

Operatory bitowe (cont)

> >> - Operator bitowy przesunięcia w prawo ze znakiem
Przesuwa bity w prawo o określoną ilość pozycji, zachowując znak liczby.
Przykład: 5 >> 1 // wynik: 2 (binarnie: 0101 >> 1 = 0010)
>>> - Operator bitowy przesunięcia w prawo bez znaku
Przesuwa bity w prawo o określoną ilość pozycji, wypełniając zerami z lewej strony.
Przykład: 5 >>> 1 // wynik: 2 (binarnie: 0101 >>> 1 = 0010)

Inne operatory

Operator warunkowy (ternarny, trójargumentowy, trójelementowy):
Przykład: let result = condition ? value1 : value2;
Operator konkat enacji (dla łączenia stringów):
Przykład: let fullName = " Joh n" + " " + " Doe ";
Operator dostępu do właści wości:
Przykład: let length = array.l ength;
Operator przecinka ,:
Operator przecinka w JavaScript służy do oddzie - lenia wielu wyrażeń i wykony wania
ich sekwen cyjnie. Wykonuje się wyrażenie po lewej stronie przecinka, a następnie
wyrażenie po prawej stronie przecinka. Operator przecinka zwraca wartość
ostatniego wyrażenia.

```
let a = 1, b = 2, c = 3;
consol e.l og(a, b, c); // wynik: 1 2 3
```

Operator jednoa rgu mentowy (unary):
Operator jednoa rgu mentowy działa na jednym argume ncie. Przykł adami operatorów
jednoa rgu men towych w JavaScript są:
+ - Konwersja na liczbę dodatnią
- - Negacja liczby
++ - Inkrem entacja
-- - Dekrem entacja
! - Negacja logiczna (NOT)

Inne operatory (cont)

```
> let a = 5;
console.log(-a); // wynik: -5
console.log(++a); // wynik: 6
console.log(!true); // wynik: false
```

Pierwszeństwo operatorów

W języku JavaScript istnieje hierarchia lub priorytet operatorów, która określa kolejność wykony wania operacji. Poniżej przeds - tawiam przykł adową listę operatorów według priory tetu, od najwyż szego do najniż - szego:

Nawiasy: ()
Operator indeks owania: []
Operator wywołania funkcji: ()
Postin kre men tacja (++) i postde kre men tacja (--)
Preink rem entacja (++) i predek rem entacja (--)
Unarne operatory arytme tyczne: +, -
Unarne operatory logiczne: !, typeof, void, delete
Potęgo wanie: **
Mnożenie: *, dzielenie: /, reszta z dzielenia: %
Dodawanie: +, odejmo wanie: -
Operatory porówn ania: >, <, >=, <=, instan ceof, in
Operatory równości: ==, !=, ===, !==
Operatory logiczne: &&, ||
Operator warunkowy (terna rny): ? :
Operator przypi sania: =, +=, -=, =, /=, %=, *=, <<=, >>=, >>>=, &=, |=, ^=

W przypadku wyrażeń z operatorem o tym samym poziomie priory tetu, wykonywane są od lewej do prawej.

Konwersja typów

W języku JavaScript istnieją dwa główne rodzaje konwersji typów: niejawna (autom aty czna) konwersja typów i jawna (ręczna) konwersja typów.

Niejawna (autom aty czna) konwersja typów:

Niejawna konwersja typów w JavaScript zachodzi automa tycznie w pewnych sytuac jach, gdy wartości są używane w kontek ście, który oczekuje innych typów danych. Np. przy dodawaniu liczby do stringa, JavaScript automa - tycznie zamienia liczbę na string i wykonuje konkat enację.

Przykład niejawnej konwersji:

```
let a = 5;
let b = " 10";
let result = a + b; // Wartość result będzie " - 510 ", ponieważ liczba 5 zostanie skonwe rtowana na string i nastąpi konkat enacja.
```

Jawna (ręczna) konwersja typów:

Jawna konwersja typów w JavaScript polega na ręcznym zmienianiu typu wartości za pomocą odpowi ednich funkcji i operacji.

Przykłady jawnej konwersji:

Konwersja na liczbę:

```
let a = " 5";
let b = Number(a); // jawna konwersja na liczbę
consol e.l og( typeof b); // " num ber "
```

Funkcje

Deklaracja funkcji:

Funkcje w JavaScript mogą być deklar owane za pomocą słowa kluczowego function.

Przykład deklaracji funkcji:

```
function greet( name) {
        con sol e.l og( " Hello, " + name + " !");
}
```

Wyrażenie funkcyjne:

Funkcje (cont)

> Funkcje w JavaScript mogą być przypisywane do zmiennych lub wartości.
Przykład wyrażenia funkcyjnego:

```
const greet = function(name) {
    console.log("Hello, " + name + "!");
};
```

Wywołanie funkcji:
Funkcje są wywoływane poprzez podanie nazwy funkcji i nawiasów () z odpowiednimi
argumentami.
Przykład wywołania funkcji:

```
greet("John"); // Wywołanie funkcji greet z argumentem "John"
```

Funkcje strzałkowe:
Funkcje strzałkowe to skrócona składnia do tworzenia funkcji anonimowych.
Przykład funkcji strzałkowej:

```
const greet = (name) => {
    console.log("Hello, " + name + "!");
};
```

Zasięg (scope) zmiennych:
Funkcje w JavaScript mają własne zasięgi zmiennych. Zmienne zdefiniowane wewnątrz
funkcji są lokalne dla funkcji i nie są dostępne poza nią. Zmienne zdefiniowane poza
funkcją są globalne lub mają zasięg zdefiniowany przez blok kodu (w przypadku bloków
takich jak if czy for).
Wartość zwracana:
Funkcje w JavaScript mogą zwracać wartość za pomocą słowa kluczowego return.
Zwrócona wartość może być przypisana do zmiennej lub używana bezpośrednio w
innym kontekście.
Przykład zwracania wartości:

```
function add(a, b) {
    return a + b;
```

Funkcje (cont)

> }

```
const sum = add(2, 3); // Przypisanie zwracanej wartości do zmiennej
console.log(sum); // Wyświetlenie wartości 5
```

Metoda call():
Metoda call() pozwala na wywołanie funkcji i przekazanie innego obiektu jako jej
kontekstu (this).
Przykład użycia metody call():

```
const person = {
    name: "John",
    greet: function() {
        console.log("Hello, " + this.name + "!");
    }
};
const anotherPerson = {
    name: "Jane"
};
person.greet.call(anotherPerson); // Wywołanie metody greet z kontekstem anotherPerson
```

Metoda apply():
Metoda apply() działa podobnie do metody call(), ale argumenty są przekazywane
jako tablica.
Przykład użycia metody apply():

```
const person = {
    name: "John",
    greet: function(greeting) {
        console.log(greeting + ", " + this.name + "!");
    }
};
const greetings = ["Hello", "Bonjour", "Hola"];
```

Funkcje (cont)

> person.greet.apply(person, greetings); // Wywołanie metody greet z kontekstem person
i przekazanie tablicy greetings jako argumentów
Rekurencja:
Rekurencja to technika polegająca na wywoływaniu funkcji samej siebie. Jest często
używana do rozwiązywania problemów, które można zdefiniować w sposób rekurencyjny.
Przykład rekurencji:

```
function countdown(n) {
    if (n === 0) {
        console.log("Lift off!");
    } else {
        console.log(n);
        countdown(n - 1);
    }
}
countdown(5); // Wywołanie funkcji countdown z argumentem 5
```

Funkcje przekazywane jako argument do funkcji (funkcje zwrotne lub funkcje
wyższego rzędu) pozwalają na wykonanie pewnych działań wewnątrz funkcji
na podstawie logiki zawartej w przekazanej funkcji.
Oto przykład, który ilustruje przekazywanie funkcji jako argumentów:

```
function sayHello() {
    console.log("Hello!");
}
function runCallback(callback) {
    console.log("Running callback...");
    callback(); // Wywołanie przekazanej funkcji
}
runCallback(sayHello); // Przekazanie funkcji sayHello jako argument do funkcji
runCallback
```

Funkcje (cont)

> W powyższym przykładzie funkcja sayHello jest przekazywana jako argument do funkcji
runCallback. Gdy funkcja runCallback jest wywoływana, przekazana funkcja sayHello jest
wywoływana wewnątrz funkcji runCallback. To umożliwia wykonanie dodatkowych działań
wewnątrz runCallback, takich jak logowanie, przed lub po wywołaniu przekazanej funkcji.
Funkcje anonimowe w języku JavaScript to funkcje, które nie posiadają nazwy.
Są one definiowane bez podawania nazwy funkcji i mogą być przypisywane do
zmiennych lub przekazywane jako argumenty do innych funkcji. Funkcje anonimowe
są często wykorzystywane w przypadkach, gdy potrzebujemy jednorazowej funkcji lub
gdy chcemy przekazać logikę funkcji bez konieczności nadawania jej nazwy.
Przypisanie funkcji anonimowej do zmiennej:

```
const sayHello = function() {
  console.log("Hello!");
};
sayHello(); // Wywołanie funkcji anonimowej
```

Przekazanie funkcji anonimowej jako argumentu:

```
function runCallback(callback) {
  console.log("Running callback...");
  callback(); // Wywołanie przekazanej funkcji anonimowej
}
runCallback(function() {
  console.log("Hello from anonymous function!");
});
```

Wykorzystanie funkcji anonimowej bezpośrednio w wyrażeniu:

```
setTimeout(function() {
  console.log("Delayed message");
}, 2000);
```

Obiekt document

getElementById():
const element = docume nt.g et Ele men tBy - Id('my Ele ment');
Ta funkcja zwraca element DOM o określonym identy fik atorze. Przykład pokazuje pobranie elementu o identy fik atorze " myE lem - ent ".
getEle men tsB yCl ass Name():
const elements = docume nt.g et Ele men tsB yCl - ass Nam e(' myC lass');
Ta funkcja zwraca kolekcję elementów DOM, które mają określoną klasę. Przykład pokazuje pobranie elementów o klasie " myC las s".
getEle men tsB yTa gNa me():
const elements = docume nt.g et Ele men tsB yTa - gNa me('p');
Ta funkcja zwraca kolekcję elementów DOM, które mają określony tag HTML. Przykład pokazuje pobranie wszystkich elementów <p>.
queryS ele ctor():
const element = docume nt.q ue ryS ele cto r(' - #my Element .myCla ss');
Ta funkcja zwraca pierwszy element DOM, który pasuje do podanego selektora CSS.
Przykład pokazuje pobranie pierwszego elementu, który ma identy fikator " myE lem ent " i klasę " myC las s".
queryS ele cto rAll():
const elements = docume nt.q ue ryS ele cto rAl - l('.my Cla ss');
Ta funkcja zwraca wszystkie elementy DOM, które pasują do podanego selektora CSS.
Przykład pokazuje pobranie wszystkich elementów o klasie " myC las s".
create Ele ment():
const element = docume nt.c re ate Ele men t(' - div');
Ta funkcja tworzy nowy element DOM o podanym tagu HTML. Przykład pokazuje utworzenie nowego elementu <di v>.
append Chi ld():
parent Ele men t.a ppe ndC hil d(n ewE lem ent);
Ta funkcja dodaje nowy element jako dziecko do określ onego elementu DOM.

Obiekt document (cont)

> Przykład pokazuje dodanie nowego elementu jako dziecka do elementu nadrzędnego.
removeChild():
parentElement.removeChild(childElement);
Ta funkcja usuwa określony element potomny z danego elementu DOM.
Przykład pokazuje usunięcie określonego elementu potomnego z elementu nadrzędnego.
prepend():
parentElement.prepend(newElement);
Ta funkcja dodaje nowy element jako pierwsze dziecko do określonego elementu DOM.
Przykład pokazuje dodanie nowego elementu jako pierwszego dziecka do elementu
nadrzędnego.
const parentElement = document.getElementById('myElement');
const newElement = document.createElement('div');
newElement.textContent = 'Hello, World!';
parentElement.prepend(newElement);
createElement():
const element = document.createElement('div');
Ta funkcja tworzy nowy element DOM o podanym tagu HTML. Można również użyć tej
funkcji do tworzenia innych rodzajów elementów, takich jak <p>, , <img>, itp.
Przykład pokazuje tworzenie nowego elementu .
const element = document.createElement('p');
element.textContent = 'Hello, World!';
Funkcja createElement() jest używana w połączeniu z innymi funkcjami, takimi jak
appendChild() lub prepend(), aby dodać nowo utworzony element do drzewa DOM.

Manipulowanie elementami html i css

setAttribute():
elemen t.s etA ttr ibu te('at tri but eName', 'attri but eVa lue');
Ta funkcja ustawia wartość określ onego atrybutu na elemencie. Może być używana do ustawienia dowolnego atrybutu, na przykład class, id, src, href itp.
style:
elemen t.s tyl e.p roperty = 'value';
Można użyć właści wości style na elemencie, aby zmieniać jego style CSS. Właści wości style odzwie rci edlają style inline dla danego elementu.
classList:
elemen t.c las sLi st.a dd ('c las sNa me');
elemen t.c las sLi st.r em ove ('c las sNa me');
elemen t.c las sLi st.t og gle ('c las sNa me');
Właściwość classList umożliwia manipu lowanie klasami elementu. Można użyć metod add(), remove() i toggle() do dodawania, usuwania i przełą czania klas na elemencie.
innerHTML obiektu document służy do odczyt ywania i modyfi kowania zawartości HTML danego elementu. Przech owuje ona kod HTML znajdujący się wewnątrz danego elementu.
Przykład użycia innerHTML do odczytania zawartości HTML elementu:

```
const element = docume nt.g et Ele men tBy -
Id( 'my Ele ment');
const htmlCo ntent = elemen t.i nne rHTML;
consol e.l og( htm lCo ntent);
```

Przykład użycia innerHTML do modyfi kowania zawartości HTML elementu:

```
const element = docume nt.g et Ele men tBy -
Id( 'my Ele ment');
elemen t.i nne rHTML = '<p >Nowa zawart ość </p >';
```

Własne funkcje konstruujące

W JavaScript możemy tworzyć własne funkcje konstrukt do tworzenia nowych obiektów na podstawie określ onego wzorca. Fu uktory są używane do defini owania " kla s" obiektów i tworzenia inst obiektów.
Aby zdefin iować własną funkcję konstr uktora, stosu CamelCase dla nazwy funkcji (zaczynamy wielką literą) i defini ujemy jej konstr uktora może zawierać parametry, które są przeka zywane przy twor instancji obiektu.
Wewnątrz funkcji konstr uktora możemy definiować wła obiektu, które będą różne dla każdej instancji, oraz metody, które będą współd wszystkie instancje.

```
function Person (name, age) {
    thi s.name = name;
    thi s.age = age;
    thi s.greet = function() {
        con sol e.log( Hello, my name is ${this.name} and I
ge} years old.);
   };
}
// Tworzenie nowej instancji obiektu za pomocą funkc
const person1 = new Person ('J ohn', 25);
const person2 = new Person ('J ane', 30);
// Wywoły wanie metody na instan cjach obiektu
person 1.g reet(); // Hello, my name is John and I'm
person 2.g reet(); // Hello, my name is Jane and I'm
```

call apply bind

call, apply i bind to metody, które pozwalają na manipulację kontekstem (this) funkcji w JavaSc ript. Oto ich krótkie wyjaśn - ienie:
- call: Metoda call pozwala na wywołanie funkcji z określonym kontekstem (this) oraz przeka zanie argumentów osobno.

```
function sayHello() {
    con sol e.log( Hello, ${this.name}!);
```

call apply bind (cont)

> }
const person = { name: 'John' };
sayHello.call(person); // Wywołanie funkcji sayHello z kontekstem person
- apply: Metoda apply działa podobnie do call, ale przekazuje argumenty jako tablicę.
function sayHello(greeting) {
 console.log(${gree ting}, ${this.name}!);
}
const person = { name: 'John' };
sayHello.apply(person, ['Hello']); // Wywołanie funkcji sayHello z kontekstem person
i argumentem 'Hello'
- bind: Metoda bind tworzy nową funkcję, która ma przypisany określony kontekst (this),
ale nie jest natychmiast wywoływana. Zamiast tego zwraca funkcję, która może być
wywołana później.
function sayHello() {
 console.log(Hello, ${this.name}!);
}
const person = { name: 'John' };
const greetPerson = sayHello.bind(person); // Utworzenie nowej funkcji greetPerson
z przypisanym kontekstem person
greetPerson(); // Wywołanie funkcji greetPerson
Metody call i apply są używane, gdy chcemy natychmiastowo wywołać funkcję z
określonym kontekstem i argumentami. Metoda bind jest przydatna, gdy chcemy
utworzyć nową funkcję, która będzie miała przypisany kontekst i będzie mogła być
wywołana w przyszłości.
Te trzy metody są często używane do manipulacji kontekstem (this) w JavaScript i
pozwalają na większą elastyczność w programowaniu.

Object

Właściwości obiektu:
- Object.pr oto typ e.c ons tru ctor: Zwraca referencję do funkcji konstr uktora,
która została użyta do utworzenia obiektu.
- Object.pr oto typ e.h asO wnP rop erty(): Sprawdza, czy dany obiekt posiada
własność o podanej nazwie.
- Object.pr oto typ e.p rop ert yIs Enu mer able(): Sprawdza, czy dana własność
obiektu jest wylicz alna.
Funkcje obiektu:
- Object.as sign(): Kopiuje wartości wszystkich wylicz alnych własności z jednego
lub więcej obiektów źródłowych do obiektu docelo - wego.
- Object.ke ys(): Zwraca tablicę zawier ającą nazwy wszystkich wylicz alnych
własności obiektu.
- Object.va lues(): Zwraca tablicę zawier ającą wartości wszystkich wylicz alnych
własności obiektu.
- Object.en tri es(): Zwraca tablicę zawier ającą tablice [klucz, wartość] dla
wszystkich wylicz alnych własności obiektu.
- Object.fr eeze(): Zamraża obiekt, uniemo żli - wiając zmianę jego własności.
- Object.se al(): Zapiec zęt owuje obiekt, uniemo żli wiając dodawanie i usuwanie
własności, ale umożli wiając modyfi kację istnie - jących własności.
- Object.cr eate(): Tworzy nowy obiekt z określonym prototypem i opcjon alnymi
własno ściami.
Metody prototypu obiektu:
- toStri ng(): Zwraca ciąg znaków reprez ent ujący obiekt.
- hasOwn Pro per ty(): Sprawdza, czy obiekt ma daną własność jako własną,
a nie odzied zic zoną.
- valueOf(): Zwraca wartość pierwotną obiektu.
Poniżej znajduje się przykład użycia niektórych z tych właści wości i funkcji:
const obj = { name: 'John', age: 30 };

Object (cont)

```
> console.log(Object.keys(obj)); // Output: ['name', 'age']
console.log(Object.values(obj)); // Output: ['John', 30]
console.log(Object.entries(obj)); // Output: [['name', 'John'], ['age',
30]]
const newObj = Object.assign({}, obj);
console.log(newObj); // Output: { name: 'John', age: 30 }
const sealedObj = Object.seal(obj);
sealedObj.name = 'Jane';
delete sealedObj.age;
console.log(sealedObj); // Output: { name: 'Jane', age: 30 }
const frozenObj = Object.freeze(obj);
frozenObj.name = 'Jane';
console.log(frozenObj); // Output: { name: 'John', age: 30 }
console.log(obj.hasOwnProperty('name')); // Output: true
console.log(obj.hasOwnProperty('toString')); // Output: false
```

Gettery i settery to specjalne metody używane w języku JavaScript do kontroli
dostępu do właściwości obiektów. Pozwalają na bardziej elastyczne zarządzanie
właściwościami obiektu i umożliwiają wykonanie dodatkowej logiki podczas
odczytu i zapisu wartości.
Getter jest metodą, która zwraca wartość właściwości obiektu. Jest definiowany
za pomocą słowa kluczowego get i nazwy właściwości. Kiedy odwołujemy się do
tej właściwości, getter jest automatycznie wywoływany i zwraca oczekiwany wynik.
Setter jest metodą, która ustawia wartość właściwości obiektu. Jest definiowany
za pomocą słowa kluczowego set i nazwy właściwości. Kiedy przypisujemy wartość
do tej właściwości, setter jest automatycznie wywoływany, co pozwala na wykonanie
określonych operacji lub walidację przed zapisaniem wartości.
Oto przykład użycia gettera i settera:

```
const person = {
  firstName: '',
```

Object (cont)

```
>   lastName: '',

  get fullName() {
    return this.firstName + ' ' + this.lastName;
  },

  set fullName(name) {
    const [firstName, lastName] = name.split(' ');
    this.firstName = firstName;
    this.lastName = lastName;
  }
};
person.fullName = 'John Doe'; // Wywołanie settera
console.log(person.firstName); // Wyświetli "John"
console.log(person.lastName); // Wyświetli "Doe"
console.log(person.fullName); // Wyświetli "John Doe", getter
```

automatycznie wywołany
Object.defineProperty() jest metodą wbudowaną w JavaScript, która służy do
definiowania nowej właściwości na obiekcie lub modyfikowania istniejącej właściwości
na obiekcie. Metoda ta umożliwia dokładne kontrolowanie konfiguracji właściwości,
takich jak możliwość zapisu (writable), możliwość konfiguracji (configurable), możliwość
wyliczania (enumerable) itd.
Przykład użycia Object.defineProperty():

```
const obj = {};
// Definiowanie nowej właściwości "name" na obiekcie
Object.defineProperty(obj, 'name', {
  value: 'John',
  writable: false, // Wartość nie może być zmieniana
  enumerable: true, // Właściwość jest wyliczalna
  configurable: true // Właściwość może być modyfikowana lub
usunięta
});
```

Object (cont)

```
> console.log(obj.name); // John
obj.name = 'Jane'; // Nie spowoduje zmiany, ponieważ writable jest ustawione na false
console.log(obj.name); // John
// Modyfikowanie istniejącej właściwości
Object.defineProperty(obj, 'name', {
  value: 'Jane' // Zmiana wartości
});
console.log(obj.name); // Jane
```

Enumeracja

Enumeracja obiektu w języku JavaScript polega na przejściu przez wszystkie jego właści wości. Istnieją różne metody umożli - wiające iterację po właści woś ciach obiektu i wykonanie określ onych operacji dla każdej z nich.

Najpop ula rni ejszymi metodami iteracji po właści woś ciach obiektu są:

Pętla for...in: Pętla for...in iteruje po wszystkich wylicz alnych właści woś ciach obiektu (w tym również w łaściw ościach dziedz icz onych). Dla każdej właści wości można wykonać określone operacje.

```
const obj = { a: 1, b: 2, c: 3 };
for (const prop in obj) {
    con sol e.l og(prop + ': ' + obj[pr op]);
}
// Wynik:
// a: 1
// b: 2
// c: 3
```

Metoda Object.ke ys(): Metoda Object.keys() zwraca tablicę zawier ającą wszystkie własne właści wości wyliczalne obiektu. Można na niej iterować za pomocą pętli for...of.

```
const obj = { a: 1, b: 2, c: 3 };
for (const prop of Object.ke ys( obj)) {
>   console.log(prop + ': ' + obj[prop]);
}
// Wynik:
// a: 1
// b: 2
// c: 3
```

Metoda Object.entries(): Metoda Object.entries() zwraca tablicę zawierającą tablice [klucz, wartość] dla każdej wyliczalnej własnej właściwości obiektu. Można na niej iterować za pomocą pętli for...of.

```
const obj = { a: 1, b: 2, c: 3 };
for (const [key, value] of Object.entries(obj)) {
  console.log(key + ': ' + value);
}
// Wynik:
// a: 1
// b: 2
// c: 3
```

Te metody iteracji pozwalają na dostęp do właściwości obiektu i wykonanie operacji na nich. Warto zauważyć, że kolejność iteracji nie jest gwarantowana, więc nie należy polegać na porządku, w jakim właściwości są zwracane.

Enumeracja (cont)

Proxy API

Proxy API to funkcjonalność dostępna w języku JavaScript, która umożliwia przech wyt ywanie i dostos owy wanie zachowań podsta wowych operacji na obiektach.

API to dostarcza możliwość tworzenia proxy (osłonki) wokół istnie jących obiektów i defini owania niesta nda rdowych operacji, które mają być wykonywane przy różnych intera kcjach z tymi obiektami.

Oto kilka cech i przykładów użycia Proxy API:

- Przech wyt ywanie dostępu do właści wości:

Proxy API (cont)

> Proxy pozwala przechwycić próby odczytu (get) i zapisu (set) właściwości obiektu oraz reagować na nie. Przykład:

```
const obj = {
  name: "John",
  age: 30
};
const proxy = new Proxy(obj, {
  get(target, property) {
    console.log(`Odczytano wartość właści wości " ${p rop ert y}"`);
    return target[property];
  },
  set(target, property, value) {
    console.log(`Ustawiono wartość właści wości " ${p rop ert y}" na "${v alu e}"`);
    target[property] = value;
  }
});
console.log(proxy.name); // Odczytano wartość właściwości "name", wynik: "John"
proxy.age = 40; // Ustawiono wartość właściwości "age" na "40"
```

- Przechwytywanie wywołań metod:

Proxy pozwala przechwycić wywołania metod na obiekcie i dostosować ich zachowanie.

Przykład:

```
const obj = {
  greet(name) {
    console.log(`Witaj, ${name}!`);
  }
};
const proxy = new Proxy(obj, {
  apply(target, thisArg, argumentsList) {
    console.log(`Wywołano metodę " ${a rgu men tsL ist -[0] }"`);
    return target.apply(thisArg, argumentsList);
  }
});
proxy.greet("John"); // Wywołano metodę "greet", wynik: "Witaj, John!"
```

- Inne operacje i przechwytywanie zdarzeń:

Proxy umożliwia przechwytywanie innych operacji, takich jak sprawdzanie istnienia właściwości (has), iteracja (enumerate), usuwanie właściwości (deleteProperty) itp.

Można również przechwycić zdarzenia dotyczące proxy za pomocą specjalnych metod, takich jak getPrototypeOf, setPrototypeOf, isExtensible, preventExtensions, getOwnPropertyDescriptor, defineProperty, ownKeys itp.

Proxy API dostarcza potężny mechanizm do tworzenia niestandardowych zachowań dla obiektów w języku JavaScript. Można go wykorzystać do wprowadzania walidacji, filtrowania, zabezpieczeń, śledzenia zmian i wielu innych operacji na obiektach.

Reflect API

Reflect API to zestaw metod i funkcji, które dostarczają interfejsu programistycznego do operacji na obiektach w języku JavaScript. API to umożliwia wykonywanie różnych operacji na obiektach, takich jak tworzenie, modyfikacja, usuwanie właściwości, wywoływanie funkcji itp.

Oto kilka cech i przykładów użycia Reflect API:

1. Metody do tworzenia i manipulowania obiektami:

- `Reflec t.c ons tru ct( target, argume nts List)`: Tworzy nowy obiekt na podstawie danego konstruktora i argumentów.
- `Reflec t.g etP rot oty peO f(o bject)`: Zwraca prototyp danego obiektu.
- `Reflec t.s etP rot oty peO f(o bject, prototype)`: Ustawia nowy prototyp dla danego obiektu.

2. Metody do manipulacji właściwościami obiektów:

- `Reflec t.g et( object, proper tyKey)`: Zwraca wartość właściwości danego obiektu.
- `Reflec t.s et( object, proper tyKey, value)`: Ustawia wartość właściwości danego obiektu.

Reflect API (cont)

- `Reflec t.h as( object, proper tyKey)`: Sprawdza, czy dany obiekt zawiera właściwość o podanym kluczu.
- `Reflec t.d ele teP rop ert y(o bject, proper tyKey )`: Usuwa właściwość z danego obiektu.

3. Metody do wywoływania funkcji:

- `Reflec t.a ppl y(t arget, thisAr gument, argume nt s List)`: Wywołuje daną funkcję z podanym kontekstem (`this`) i listą argumentów.
- `Reflec t.c ons tru ct( target, argume nts List, ne wTarget)`: Wywołuje konstruktor z podanymi argumentami i opcjonalnie nowym obiektem docelowym.

4. Obsługa zdarzeń:

- `Reflec t.g et( target, proper tyKey, receiver)`: Obsługuje pobieranie wartości właściwości, umożliwiając działanie na pośredniku (`receiver`).
- `Reflec t.s et( target, proper tyKey, value, receiv er)`: Obsługuje ustawianie wartości właściwości, umożliwiając działanie na pośredniku (`receiver`).

Reflect API dostarcza bardziej jednolitej i przejrzystej składni do operacji na obiektach w porównaniu do tradycyjnych operacji, takich jak `obj.pr operty` lub `obj["pr ope rty "]`. Zapewnia również większą elastyczność i kontrolę nad operacjami na obiektach, umożliwiając manipulację właściwościami, wywoływanie funkcji i zarządzanie prototypami.

Generatory

```
Generatory w języku JavaScript to specjalny rodzaj
funkcji, które pozwalają na
wznawialne wykony wanie się kodu. Generator jest
tworzony przy użyciu słowa kluczowego
function, ale zamiast zwykłego return, używa się
instrukcji yield, która zwraca wartość,
ale nie kończy działania funkcji.
Oto kilka cech genera torów:
Wznawialne wykonanie: Główną cechą genera torów
jest możliwość wznawi alnego
wykonania kodu. Po wywołaniu genera tora, nie jest
on urucha miany od początku do końca,
ale zatrzymuje się na każdej instrukcji yield,
zwracając wartość i zapami ętując swój stan.
Kolejne wywołanie generatora wznawia jego
wykonanie od momentu, w którym został
zatrzy many.
```

Generatory (cont)

> Wydajne zarządzanie pamięcią: Generatory pozwalają na leniwe generowanie wartości,
co oznacza, że nie muszą generować wszystkich wartości od razu. Mogą generować
wartości na żądanie, co jest szczególnie przydatne w przypadku generowania dużych kolekcji danych.
Iterator: Generatory są jednocześnie iteratorami. Implementują protokół iteratora, co
oznacza, że można ich używać w pętlach for...of lub wykorzystywać do iteracji po danych.
Potokowe przetwarzanie danych: Generatory mogą być łączone w tzw. potoki, co
umożliwia przetwarzanie danych w sposób sekwencyjny. Wynik jednego generatora
może być przekazywany do innego generatora, co pozwala na składanie różnych
operacji na danych.
Przykład użycia generatora:

```
function* generateNumbers() {
  let number = 1;
  while (true) {
    yield number;
    number++;
  }
}
const numberGenerator = generateNumbers();
console.log(numberGenerator.next().value); // 1
console.log(numberGenerator.next().value); // 2
console.log(numberGenerator.next().value); // 3
```

W powyższym przykładzie mamy generator generateNumbers, który generuje kolejne
liczby naturalne. Po każdym wywołaniu numberGenerator.next(), otrzymujemy obiekt z
wartością generowaną przez generator. Dzięki temu możemy generować kolejne liczby w
nieskończoność, zatrzymując się i wznawiając wykonanie generatora w dowolnym
momencie.
Generatory są użytecznym narzędziem do manipulacji danymi i zarządzania ich

Generatory (cont)

> przepływem. Pozwalają na bardziej elastyczną kontrolę nad generowaniem i
przetwarzaniem danych w języku JavaScript.

Moduły

Moduły w języku JavaScript są mechanizmem, który umożliwia podział kodu na
mniejsze, samodz ielne jednostki. Pozwala to na lepszą organi zację i zarząd zanie
kodem, a także ułatwia jego ponowne wykorz yst - anie.
W JavaScript istnieją różne metody implem entacji modułów, takie jak moduły CommonJS,
moduły AMD (Async hronous Module Defini tion) i moduły ES6 (ECMAS cript 6).
Moduły ES6 są wbudowanym mechan izmem języka od wersji ES6, który zapewnia
prostą i wydajną składnię do defini owania i ekspor towania modułów.
Główne cechy modułów ES6 to:
Deklaracja modułu: Moduł jest tworzony poprzez utworzenie pliku z rozsze rzeniem
".js ", który zawiera kod JavaSc ript. Moduł jest oddzielną jednostką, która posiada swoje
własne zakresy i niezal eżność od innych modułów.
Ekspor towanie danych: Ekspor towanie danych z modułu odbywa się za pomocą
słowa kluczowego export. Możemy ekspor tować funkcje, zmienne, stałe, klasy i wiele innych.
Przykład eksportu funkcji:

```
export function sayHello() {
    con sol e.l og( 'He llo!');
}
```

Import owanie danych: Import owanie danych z innego modułu odbywa się za pomocą
słowa kluczowego import. Import ujemy określone elementy z modułu i przypi sujemy je do zmiennych.
Przykład importu funkcji:

```
import { sayHello } from './mod ule.js';
sayHel lo(); // Wywołanie funkcji z innego modułu
```

Domyślne ekspor tow anie: Możemy także domyślnie ekspor tować jedno element z
modułu. Może to być funkcja, klasa lub inny obiekt.

Moduły (cont)

> Przykład domyślnego eksportu klasy:

```
export default class MyClass {
  constructor() {
    // Konstruktor klasy
  }
}
```

Moduły w JavaScript umożliwiają tworzenie modularnej struktury kodu, gdzie poszczególne
części aplikacji są łatwo odizolowane i mogą być używane i modyfikowane niezależnie od siebie.
Jest to przydatne w większych projektach, które wymagają modularności i skalowalności.

isin

Jeśli chodzi o sprawdzenie, czy określona wartość jest obecna w danym tekście,
obiekcie lub innym zbiorze danych, istnieje kilka sposobów:
- Metoda includ es(): Metoda includes() jest dostępna dla typów danych takich jak ciągi
znaków (stringi) i tablice. Służy do sprawd zenia, czy dana wartość jest zawarta w danym
ciągu znaków lub tablicy. Zwraca wartość logiczną true lub false.
Przykład (użycie na stringu):

```
const text = " Lorem ipsum dolor sit amet";
consol e.l og( tex t.i ncl ude s("i psu m")); // true
consol e.l og( tex t.i ncl ude s("f oo")); // false
```

- Operator in: Operator in jest używany do sprawd zenia, czy dany klucz istnieje w
obiekcie. Może być stosowany na obiektach i tablicach. Zwraca wartość logiczną true
lub false.
Przykład (użycie na obiekcie):

```
const person = { name: " Joh n", age: 30 };
consol e.l og( " nam e" in person); // true
consol e.l og( " add res s" in person); // false
```

- Metoda indexOf(): Metoda indexOf() jest dostępna dla ciągów znaków (stringów) i

isin (cont)

> tablic. Służy do znalezienia indeksu pierwszego wystąpienia danej wartości w ciągu
znaków lub tablicy. Jeśli wartość nie jest znaleziona, zwraca -1.
Przykład (użycie na stringu):
const text = "Lorem ipsum dolor sit amet";
console.log(text.indexOf("dolor")); // 12
console.log(text.indexOf("foo")); // -1
- Metoda hasOwnProperty(): Metoda hasOwnProperty() jest dostępna dla obiektów.
Służy do sprawdzenia, czy obiekt posiada określony klucz jako własność. Zwraca wartość
logiczną true lub false.
Przykład (użycie na obiekcie):
const person = { name: "John", age: 30 };
console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("address")); // false
- Metoda match(): Metoda match() jest dostępna dla ciągów znaków (stringów) i służy do znalezienia dopasowań do określonego wzorca za pomocą wyrażenia regularnego.
Jeśli dopasowanie jest znalezione, zwraca tablicę dopasowanych wartości; w przeciwnym
razie zwraca null.
Przykład (użycie na stringu):
const text = "Lorem ipsum dolor sit amet";
console.log(text.match(/ipsum/)); // ["ipsum"]
console.log(text.match(/foo/)); // null
- Metoda find(): Metoda find() jest dostępna dla tablic i służy do znalezienia pierwszego
elementu, który spełnia określony warunek w funkcji zwrotnej.
Zwraca wartość znalezionego elementu lub undefined, jeśli nie znaleziono pasującego elementu.
Przykład (użycie na tablicy):
const numbers = [1, 2, 3, 4, 5];
const found = numbers.find(num => num > 3);
console.log(found); // 4

isin (cont)

> - Operator some() (dla tablic): Operator some() działa na tablicach i sprawdza, czy
przynajmniej jeden element w tablicy spełnia określony warunek w funkcji zwrotnej.
Zwraca wartość logiczną true lub false.
Przykład (użycie na tablicy):
const numbers = [1, 2, 3, 4, 5];
const hasEvenNumber = numbers.some(num => num % 2 === 0);
console.log(hasEvenNumber); // true

Symbol

Symbol jest jednym z typów danych w języku JavaScript, wprowadzonym w
ECMAScript 2015 (ES6). Jest to wartość unikalna i niezmi enna, która może być używana
jako klucz identy fik ujący właści wości obiektów. Symbole są przydatne w kontekście
tworzenia i zarząd zania własnymi właści woś ciami obiektów, które są ukryte i niekol idują
z innymi właści woś ciami.
Oto kilka cech i zastosowań symboli:
Unikal ność: Każdy symbol jest unikalny i różni się od innych symboli. Nawet jeśli dwa
symbole mają taką samą nazwę, są one od siebie różne.
Prywat ność: Symbole są przydatne do tworzenia ukrytych lub prywatnych właści wości
obiektów. Można je wykorz ystać do dodania własności do obiektu, które nie będą
konfli ktować z innymi właści woś ciami i nie będą przypa dkowo nadpis ywane.
Użycie jako klucze: Symbole mogą być używane jako klucze w obiektach.
Dzięki temu można tworzyć niesta nda rdowe właści wości, które nie zostaną
przypa dkowo nadpisane lub skonfl ikt owane z innymi właści woś ciami.
Oto przykł adowe użycie symboli:

```
// Tworzenie symbolu
const symbol = Symbol();
// Używanie symbolu jako klucza
const obj = {
    [sy mbol]: 'Wartość symbolu'
```

Symbol (cont)

> };
// Dostęp do wartości za pomocą symbolu
console.log(obj[symbol]); // Wyświetli: 'Wartość symbolu'
// Porównywanie symboli
const symbol1 = Symbol('Symbol');
const symbol2 = Symbol('Symbol');
console.log(symbol1 === symbol2); // false
console.log(symbol1 == symbol2); // false
Symbole są przydatne w przypadkach, gdy potrzebujemy dodatkowej warstwy
prywatności, unikalnych identyfikatorów lub niestandardowych właściwości obiektów.
let obj = {
  a: 10
};
obj[symbol1] = 1000;
console.log(obj[symbol1]); // 1000
for (const v in obj) console.log(v); // a
console.log(Object.getOwnPropertySymbols(obj));

Promise

Obiekt Promise w języku JavaScript jest używany do obsługi asynchronicznych
operacji i zarząd zania ich wynikami. Promise reprez entuje wartość, która może być
dostępna teraz, w przysz łości lub nigdy. Jest to sposób elegan ckiego rozwią zania
problemów związanych z asynch ron icz nością, takich jak pobieranie danych z serwera, wykony - wanie żądań sieciowych czy operacje na plikach.
Główne cechy Promise:
Stan: Promise może znajdować się w jednym z trzech stanów:
- Pending (oczek ujący): Początkowy stan, w którym Promise oczekuje na wykonanie.
- Fulfilled (zreal izo wany): Stan, w którym Promise został z powodz eniem zakończony i zwraca wartość.

Promise (cont)

> - Rejected (odrzucony): Stan, w którym Promise zakończył się niepowodzeniem i
zwraca błąd.
Metody: Promise udostępnia kilka przydatnych metod do obsługi asynchroniczności:
- then(): Wykonuje się, gdy Promise zostanie zrealizowany, i obsługuje wynik sukcesu.
- catch(): Wykonuje się, gdy Promise zostanie odrzucony, i obsługuje błąd.
- finally(): Wykonuje się bez względu na wynik Promise, zarówno po zrealizowaniu,
jak i odrzuceniu.
- Promise.all(): Pozwala na równoczesne wykonywanie wielu Promise i czeka na
ich zakończenie.
- Promise.race(): Czeka na zakończenie dowolnego z podanych Promise.
Przykład użycia Promise:
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Przykładowe dane';
      // Symulacja sukcesu
      resolve(data);
      // Symulacja błędu
      // reject(new Error('Wystąpił błąd'));
    }, 2000);
  });
};
fetchData()
  .then((data) => {
    console.log('Pomyślnie pobrane dane:', data);
  })
  .catch((error) => {
    console.error('Wystąpił błąd:', error);

Promise (cont)

```
>  })
  .finally(() => {
    console.log('Operacja zakończona');
  });
```

W tym przykładzie tworzymy funkcję fetchData, która zwraca Promise. Wewnątrz
Promise symulujemy opóźnienie za pomocą funkcji setTimeout. W zależności od
rezultatu symulujemy sukces lub błąd, używając funkcji resolve i reject.
Następnie używamy metody then() do obsługi wyniku sukcesu i metody catch() do
obsługi błędu. Metoda finally() jest wywoływana bez względu na wynik Promise.
Dzięki obiektowi Promise możemy w łatwy sposób zarządzać asynchronicznymi
operacjami, a także w ładny sposób obsługi.

Destrukturyzacja

Destrukturyzacja (destrukcyjne przypisanie) to składnia w JavaScript, która umożliwia wygodne i eleganckie wyciąganie wartości z obiektów i tablic oraz przypi sywanie ich do zmiennych. Dzięki destru ktu ryzacji można łatwo dostępować do właści wości obiektów lub elementów tablicy bez koniec zności odwoły wania się do nich za pomocą tradyc yjnych notacji.

```
Destru ktu ryzacja obiektów:
const person = { name: 'John', age: 30 };
// Wyciąganie wartości z obiektu do zmiennych
const { name, age } = person;
consol e.l og( name); // 'John'
consol e.l og( age); // 30
Destru ktu ryzacja tablic:
const numbers = [1, 2, 3, 4, 5];
// Wyciąganie wartości z tablicy do zmiennych
const [first, second, ...rest] = numbers;
consol e.l og( first); // 1
consol e.l og( sec ond); // 2
```

Destrukturyzacja (cont)

```
> console.log(rest); // [3, 4, 5]
```

Destrukturyzacja może być również stosowana w bardziej złożonych przypadkach,
takich jak destrukturyzacja zagnieżdżonych obiektów lub tablic, przypisywanie
domyślnych wartości, ignorowanie niektórych elementów itp.
Przykład destrukturyzacji zagnieżdżonych obiektów:

```
const person = { name: 'John', age: 30, address: { city: 'New York', country: 'USA' } };
// Wyciąganie zagnieżdżonych wartości
const { name, address: { city } } = person;
console.log(name); // 'John'
console.log(city); // 'New York'
```

Przykład destrukturyzacji z domyślnymi wartościami:

```
const person = { name: 'John' };
// Przypisanie domyślnych wartości, jeśli nie ma odpowiedniego klucza w obiekcie
const { name, age = 30 } = person;
console.log(name); // 'John'
console.log(age); // 30
```

Destrukturyzacja jest bardzo przydatnym narzędziem w JavaScript, które pozwala
na bardziej ekspresywny i zwięzły kod, szczególnie przy manipulacji obiektami i tablicami.

ECMAScript 6 (ES6)

```
- Zmienne let i const:
let x = 5;
const PI = 3.14;
- Bloki z zakresem (block scope):
if (true) {
   let x = 10;
    con sol e.l og(x); // 10
}
consol e.l og(x); // Error: x is not defined
```

ECMAScript 6 (ES6) (cont)

> - Szablony łańcuchowe (Template literals):

```
const name = 'John';
const message = Hello, ${name}!;
console.log(message); // Hello, John!
```

- Funkcje strzałkowe (Arrow functions):

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5
```

- Parametry domyślne:

```
function greet(name = 'Guest') {
  console.log(Hello, ${name}!);
}
greet(); // Hello, Guest!
greet('John'); // Hello, John!
```

- Domyślne wartości obiektów:

```
function getUserInfo({ name = 'Anonymous', age = 0 } = {}) {
  console.log(Name: ${name}, Age: ${age});
}
getUserInfo(); // Name: Anonymous, Age: 0
getUserInfo({ name: 'John', age: 25 }); // Name: John, Age: 25
```

- Destrukturyzacja (Destructuring):

```
const person = {
  name: 'John',
  age: 30,
  country: 'USA'
};
const { name, age } = person;
console.log(name); // John
console.log(age); // 30
```

ECMAScript 6 (ES6) (cont)

> - Moduły (Modules):

```
// file1.js
export const PI = 3.14;
export function multiply(a, b) {
  return a * b;
}
// file2.js
import { PI, multiply } from './file1.js';
console.log(PI); // 3.14
console.log(multiply(2, 3)); // 6
```

- Klasy (Classes):

```
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(Hello, ${this.name}!);
  }
}
const person = new Person('John');
person.sayHello(); // Hello, John!
```

- Rest parameters: umożliwia przekazywanie dowolnej liczby argumentów do funkcji w postaci tablicy. Pozwala to na bardziej elastyczne i dynamiczne definiowanie funkcji.

```
function sum(...numbers) {
  return numbers.reduce((acc, num) => acc + num, 0);
}
console.log(sum(1, 2, 3)); // 6
```

Kiedy funkcja zostanie wywołana, wszystkie przekazane argumenty zostaną zebrane

ECMAScript 6 (ES6) (cont)

> w tablicę i przypisane do parametru parametry. Ta tablica może mieć dowolną liczbę
elementów, w zależności od liczby przekazanych argumentów.
Przykład użycia rest parameters:

```
function sumuj(...liczby) {
  let suma = 0;
  for (let liczba of liczby) {
    suma += liczba;
  }
  return suma;
}
console.log(sumuj(1, 2, 3, 4)); // Output: 10
console.log(sumuj(5, 10, 15)); // Output: 30
console.log(sumuj(2)); // Output: 2
console.log(sumuj()); // Output: 0
```

Rest parameters można również wykorzystać w połączeniu z innymi parametrami
funkcji. Jednak rest parameters musi być ostatnim parametrem w definicji funkcji,
ponieważ zbiera wszystkie pozostałe argumenty.

```
function foo(a, b, ...rest) {
  console.log('a:', a);
  console.log('b:', b);
  console.log('reszta:', rest);
}
foo(1, 2, 3, 4, 5);
// Output:
// a: 1
// b: 2
// reszta: [3, 4, 5]
```

- Spread operator: Operator rozproszenia (spread operator) w JavaScript jest oznaczony

ECMAScript 6 (ES6) (cont)

> trzema kropkami (...) i służy do rozwinięcia (rozproszenia) elementów z kolekcji (np. tablicy)
lub obiektu w miejscach, gdzie oczekiwane są oddzielne elementy. Operator rozproszenia umożliwia wygodne kopiowanie, łączenie i rozszerzanie danych.
Oto kilka przykładów zastosowania operatora rozproszenia:
Rozproszenie tablicy:

```
const numbers = [1, 2, 3];
const copiedNumbers = [...numbers];
console.log(copiedNumbers); // [1, 2, 3]
```

Łączenie tablic:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArray = [...arr1, ...arr2];
console.log(combinedArray); // [1, 2, 3, 4, 5, 6]
```

Rozproszenie obiektu:

```
const person = { name: 'John', age: 30 };
const copiedPerson = { ...person };
console.log(copiedPerson); // { name: 'John', age: 30 }
```

Przekazanie argumentów do funkcji:

```
const numbers = [1, 2, 3, 4, 5];
const maxNumber = Math.max(...numbers);
console.log(maxNumber); // 5
```

Tworzenie kopii obiektu z modyfikacją:

```
const person = { name: 'John', age: 30 };
const modifiedPerson = { ...person, age: 40 };
console.log(modifiedPerson); // { name: 'John', age: 40 }
```

Operator rozproszenia jest potężnym narzędziem, które umożliwia łatwe manipulowanie
danymi w JavaScript. Może być stosowany na różnych typach danych, takich jak tablice,
obiekty, stringi itp.

ECMAScript 6 (ES6) (cont)

> - Rest parameters i spread operator to dwie różne funkcjonalności w języku JavaScript,
ale mają podobną składnię i korzystają z tych samych symboli: trzech kropek (...).
Rest parameters (...) używane jest w definicji funkcji do zbierania dowolnej liczby
argumentów i pakowania ich w tablicę.
Spread operator (...) używany jest w miejscu wywołania funkcji, tablicy lub obiektu
do rozwinięcia jego elementów i przekazania ich jako pojedyncze argumenty lub
elementy do innej funkcji, tablicy lub obiektu.

ECMAScript 2018 (ES9)

```
- Rest/Spread Properties: Rest/Spread Properties
umożliwiają rozwinięcie i zbieranie
właści wości obiektów. Przykład:
const person = { name: 'John', age: 30, city: 'New
York' };
const { name, ...rest } = person;
consol e.l og( name); // " Joh n"
consol e.l og( rest); // { age: 30, city: 'New
York' }
- Promis e.f ina lly(): Dodano metodę finally() do
obiektu Promise, która pozwala zdefin iować
blok kodu, który zostanie wykonany niezal eżnie od
tego, czy Promise zakończy się sukcesem,
czy odrzuc eniem. Przykład:
fetch( 'ht tps :// api.ex amp le.c om /data')
   .th en( res ponse => respon se.j son())
   .th en(data => consol e.l og( data))
   .ca tch (error => consol e.e rro r(e rror))
   .fi nal ly(() => consol e.l og( 'Za koń czono
żądani e'));
- Asynch ron iczne iterowanie po obiektach:
Wprowa dzono możliwość asynch ron icznego
iterowania po obiektach przy użyciu pętli for-aw -
ait-of. Przykład:
const obj = { a: 1, b: 2, c: 3 };
(async () => {
   for await (const value of Object.va lue s(obj))
{
        con sol e.l og( value);
```

ECMAScript 2018 (ES9) (cont)

```
>  }
})();
- Obiekty RegExp z grupami named capture: Dodał możliwość
nazwanych grup
przechwytywania w wyrażeniach regularnych. Przykład:
const regex = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;
const match = regex.exec('2021-09-30');
console.log(match.groups.year); // "2021"
console.log(match.groups.month); // "09"
console.log(match.groups.day); // "30"
- Async Iteration: Umożliwia iterację po obiektach asynchronicznych,
takich jak obiekty
generujące asynchroniczne wartości. Wykorzystuje nowy interfejs
Symbol.asyncIterator i
for-await-of do obsługi iteracji. Przykład:
async function* getData() {
  yield await fetchData1();
  yield await fetchData2();
  yield await fetchData3();
}
(async () => {
  for await (const data of getData()) {
    console.log(data);
  }
})();
```

ECMAScript 2020 (ES11)

```
Oto lista niektórych nowych funkcji i zasad
wprowadzonych w ECMAScript 2020 (ES11):
- Optional Chaining (Optional Property Access):
Operator ?. pozwala na bezpieczne
odwoły wanie się do właści wości obiektu, nawet
jeśli wcześn iejsze właści wości są
undefined lub null. Przykład:
const user = {
```

ECMAScript 2020 (ES11) (cont)

```
>   name: 'John',
  address: {
    city: 'New York'
  }
};
console.log(user.address?.city); // 'New York'
console.log(user.address?.street); // undefined
```

- Nullish Coalescing Operator: Operator ?? pozwala na wybór wartości domyślnej tylko
wtedy, gdy wartość jest null lub undefined, a nie w przypadku innych "fałszywych"
wartości, takich jak false, 0 czy pusty łańcuch. Przykład:

```
const name = null;
const defaultName = 'John';
console.log(name ?? defaultName); // 'John'
const count = 0;
const defaultCount = 10;
console.log(count ?? defaultCount); // 0
```

- BigInt: Typ danych BigInt został dodany do języka JavaScript, umożliwiając obsługę
liczb całkowitych o dowolnej precyzji. Przykład:

```
const bigNumber = BigInt(12345678901234567890123456789012-
34567890);
console.log(bigNumber); // 12345678901234567890123456789012-
4567890n
```

- Promise.allSettled(): Metoda Promise.allSettled() pozwala na oczekiwanie na
zakończenie wykonania wszystkich obietnic bez względu na ich rezultat (sukces,
odrzucenie lub wyjątek). Przykład:

```
const promises = [
  Promise.resolve('Success'),
  Promise.reject('Error'),
  Promise.resolve('Another success')
];
```

ECMAScript 2020 (ES11) (cont)

```
> Promise.allSettled(promises)
  .then(results => {
    results.forEach(result => {
      console.log(result.status); // 'fulfilled' lub 'rejected'
      console.log(result.value); // wartość lub powód odrzucenia
    });
  });
```

- String.matchAll(): Metoda matchAll() służy do iteracji po wszystkich dopasowaniach
wzorca w łańcuchu znaków z wykorzystaniem wyrażeń regularnych. Przykład:

javascript
Copy code

```
const text = 'Hello World!';
const regex = /[a-z]/g;
const matches = text.matchAll(regex);
for (const match of matches) {
  console.log(match); // kolejne dopasowania
}
```

- globalThis: Obiekt globalThis dostarcza odniesienie do globalnego obiektu niezależnie
od środowiska (przeglądarka, Node.js itp.). Dzięki temu można jednolicie odwoływać się
do globalnego obiektu bez względu na kontekst. Na przykład:

```
console.log(globalThis.setTimeout === setTimeout); // true
```

import() dynamiczne importowanie: import() to asynchroniczna metoda, która umożliwia dynamiczne importowanie modułów w czasie wykonywania. Umożliwia to dynamiczną
załadowanie kodu modułu na żądanie. Przykład:

```
const module = await import('./module.js');
```

- String.prototype.replaceAll(): Metoda replaceAll() zastępuje wszystkie wystąpienia podanej
frazy w łańcuchu inną wartością. Przykład:

```
const text = 'Hello World!';
```

ECMAScript 2020 (ES11) (cont)

> const newText = text.replaceAll('l', 'X');
console.log(newText); // 'HeXXo WorXd!'

ECMAScript 2022 (ES13)

- Deklaracja pól klas
Do tej pory jeśli chcieliśmy zdefin iować pole klasy, mogliśmy to zrobić tylko poprzez konstr - uktor. Czyli …

```
class Cat {
        con str uctor() {
                thi s.name = 'Szarek';
                thi s.age = 1;
       }
}
```

w ES13 możemy to zrobić w bardziej intuicyjny sposób. Możemy więc napisać:

```
class Cat {
       name = 'Szarek';
       age = 1;
}
```

- Prywatne metody i pola klasy
W JavaScript przy posług iwaniu się klasami nie mieliśmy możliwości tworzenia prywatnych pól klas ani prywatnych metod. Można było stosować konwencję, która mówiła, że prywatne "rzeczy" wyróżniamy przedr ostkiem _. Wyglądało to w ten sposób:

```
class Cat {
        _name = 'Szarek';
       _age = 1;

        _ge tName() {
                return this._ name;
       }
}
```

ECMAScript 2022 (ES13) (cont)

> Jednak w zachowaniu kodu nie miało to żadnego odzwierciedlenia. Więc jeśli spróbowaliśmy
pobrać wartość bezpośrednio z pola klasy:
const cat = new Cat();
console.log(cat._name); // -> 'Szarek'
… dostaliśmy bez problemu wartość pola. Z pomocą śpieszy nam oczywiście
ECMAScript 2022. Wprowadza ona prywatne pola klas oraz prywatne metody!
Prywatne metody oraz prywatne pola klas definiujemy od tej wersji za pomocą przedrostka
Czyli nasz kod wyglądałby w ten sposób:

```
class Cat {
   #name = 'Szarek';
   #age = 1;

   #getName() {
      return this.#name;
   }

   getAge() {
      return this.#age;
   }
}
```

I teraz jeśli próbujemy się dostać do prywatnego pola klasy:
console.log(cat.#name); // -> SyntaxError: Private name #name is not defined.
do metody prywatnej:
console.log(cat.getName()); // -> TypeError: cat.getName is not a function
no i do metody publicznej:
console.log(cat.getAge()); // -> 1
… tak jak byśmy tego oczekiwali, dostajemy wartość.
- Operator await w najwyższym scope'ie
Temat dotyczy kombinacji operatorów async - await. W skrócie - operatora await nie
mogliśmy używać poza funkcją oznaczoną operatorem async. Więc taki zapis generował

ECMAScript 2022 (ES13) (cont)

> błąd:

```
const waitPrmise = () => {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve('RESOLVE!');
        }, 5000);
    });
};
await waitPrmise(); // -> // SyntaxError: await is only valid in async functions
```

Nowa wersja, znosi takie ograniczenie i przykład powyżej będzie w niej działał poprawnie.

4. Statyczne pola klas oraz statyczne prywatne metody

Od wersji ES13 możemy definiować statyczne prywatne pola klas oraz statyczne
prywatne metody klas. Statyczne metody mają dostęp do innych statycznych
(prywatnych i publicznych) metod poprzez this.

```
class Pet {
    static #count = 0;
    static getCount() {
        return this.#count;
    }
    constructor() {
        this.constructor.#incrementCount();
    }
    static #incrementCount() {
        this.#count++;
    }
}
const pet1 = new Pet();
const ppet2 = new Pet();
```

ECMAScript 2022 (ES13) (cont)

```
> console.log(Pet.getCount()); // -> 2
```

- Statyczne bloki klas

ECMSScript 2022 pozwala nam definiować stateczne bloki kodu wewnątrz klasy.
Deklarujemy je za pomocą: static {}. Statyczne bloki kodu są wykonywane tylko raz
podczas inicjalizacji klasy.
Klasa może posiadać dowolną ilość statycznych bloków kodu. Bloki te będą wykonywane
zgodnie z kolejnością ich deklaracji.

```
class Pet {
    static count = 0;
    static {
        this.count++;
    }
    static {
        this.count++;
    }
}
console.log(Pet.count); // -> 2
```

- Metoda at() dla tablic, stringów oraz obiektów TypedArray:

Jeśli chcemy pobrać element z tablicy o konkretnym indeksie to z reguły używamy w tym celu nawiasów kwadratowych [], w ten sposób:

```
const letters = ['a', 'b', 'c', 'd', 'e'];
console.log(letters[2]); // -> c
```

Pamiętamy, że indeksy zaczynają się od 0. Jeżeli chcemy pobrać ostatni element tablicy,
to musimy zrobić takie fiku-miku:

```
console.log(letters[letters.length - 1]); // -> e
```

letters.lenght zwróci nam długość tablicy 5 i od tego odejmujemy 1, żeby zgodził się indeks.
Dzięki nowej metodzie at() możemy wyciągać wartości z tablic w bardziej naturalny sposób.

ECMAScript 2022 (ES13) (cont)

```
> console.log(letters.at(0)); // -> a
console.log(letters.at(1)); // -> b
console.log(letters.at(-1)); // -> e
console.log(letters.at(-2)); // -> d
```

Jak widzisz pobieranie ostatniego elementu tablicy zostało uproszczone. Wystarczy, że do metody przekażemy ujemną liczbę, a pozycje będą brane od końca tablicy.

Metoda at() oprócz tablic została dodana także do string'ów oraz tablic typu TypedArray.

7. Metoda Object.hasOwn()

Kto wie jak sprawdzić czy klasa posiada jakieś pole? Możemy to zrobić używając metody hasOwnProperty w ten sposób:

```
class Cat {
    name = 'Szarek';
}
const cat = new Cat();
cat.hasOwnProperty('name'); // -> true
cat.hasOwnProperty('age'); // -> false
```

Problem polega na tym, że tą metodę można w bardzo prosty sposób nadpisać i
kompletnie zmienić jej logikę, więc nie powinniśmy takiego rozwiązania stosować w
naszym kodzie.

Rozwiązaniem tego problemu jest skorzystanie z metody call na prototypie: Object.prototype.hasOwnProperty.call(obj, propertyKey) i tutaj jesteśmy już bardziej
bezpieczni.

Jednak w najnowszej wersji JavaScript pojawiło się lepsze i bardziej eleganckie
rozwiązanie. Dostaliśmy do dyspozycji metodę Object.hasOwn(), która wykonuje taką
samą pracę jak Object.prototype.hasOwnProperty.call(obj, propertyKey).

```
class Cat {
    name = 'Szarek';
}
```

ECMAScript 2022 (ES13) (cont)

```
> const cat = new Cat();
Object.hasOwn(cat, 'name'); // -> true
Object.hasOwn(cat, 'age'); // -> false
```

- Metody findLast oraz findLastIndex

Teraz dostaliśmy do kolekcji dwie nowe metody tego typu, czyli findLast i findLastIndex.

Działają one tak samo, ale zwracają co innego:

```
const letters = [
    { value: 'a' },
    { value: 'b' },
    { value: 'c' },
    { value: 'b' },
    { value: 'd' },
];
letters.findLast((element) => element.value === 'b'); // -> { value: 'b' }
letters.findLastIndex((element) => element.value === 'b'); // -> 3
```

W odróżnieniu od find i findIndex, nowe metody szukają od końca tablicy.

Więc znajdą ostatni pasujący element.

Metoda findLast zwróci znaleziony element, natomiast metoda findLastIndex zwróci indeks znalezionego elementu.