**Zadanie 1**

Wymień jakie mamy typy danych w Javascript wraz z przykładami ich deklaracji i inicjacji. Opisz również przkłady wykorzystyania operatora „typeof”.

1. Typ liczbowy (Number):

- Deklaracja i inicjacja:

let liczba = 42;

- Użycie operatora `typeof`:

     console.log(typeof liczba); // "number"

2. Typ łańcuchowy (String):

- Deklaracja i inicjacja:

let tekst = "To jest przykład tekstu.";

- Użycie operatora `typeof`:

console.log(typeof tekst); // "string"

3. Typ logiczny (Boolean):

- Deklaracja i inicjacja:

let prawda = true;

let falsz = false;

- Użycie operatora `typeof`:

console.log(typeof prawda); // "boolean"

console.log(typeof falsz);  // "boolean"

4. Typ obiektowy (Object):

- Deklaracja i inicjacja:

let osoba = {

     imie: "Jan",

     nazwisko: "Kowalski"

   };

- Użycie operatora `typeof`:

console.log(typeof osoba); // "object"

5. Typ funkcji (Function):

- Deklaracja i inicjacja:

function dodaj(a, b) {

     return a + b;

   }

- Użycie operatora `typeof`:

console.log(typeof dodaj); // "function"

6. Typ undefined:

- Deklaracja i inicjacja:

let zmiennaNiezdefiniowana;

- Użycie operatora `typeof`:

console.log(typeof zmiennaNiezdefiniowana); // "undefined"

7. Typ null:

- Deklaracja i inicjacja:

let nulowaZmienna = null;

- Użycie operatora `typeof`:

console.log(typeof nulowaZmienna); // "object"

8. Typ symbol (Symbol):

- Deklaracja i inicjacja:

const symbol = Symbol("opisSymbolu");

- Użycie operatora `typeof`:

console.log(typeof symbol); // "symbol"

9. Typ tablicowy (Array):

- Deklaracja i inicjacja:

let tablica = [1, 2, 3];

- Użycie operatora `typeof`:

console.log(typeof tablica); // "object"

**ZADANIE 2**

W JavaScript można zdefiniować wartość matematyczną PI na dwa różne sposoby:

1. Używając wbudowanej stałej `Math.PI`:

const pi = Math.PI;

console.log(pi); // 3.141592653589793

2. Ręcznie definiując wartość PI:

const pi = 3.141592653589793;

   console.log(pi); // 3.141592653589793

**Zadanie 3**

**Definicja identyfikatora w JavaScript:**

1. Identyfikator jest sekwencją liter (a-z, A-Z), cyfr (0-9), znaku podkreślenia (\_) oraz znaku dolara ($).
2. Identyfikatory muszą zaczynać się literą, znakiem podkreślenia (\_) lub znakiem dolara ($). Nie mogą zaczynać się od cyfry.
3. Identyfikatory mogą zawierać dowolną liczbę liter, cyfr, znaków podkreślenia i znaków dolara.
4. Identyfikatory są wrażliwe na wielkość liter (rozróżniają małe i duże litery).
5. Identyfikatory nie mogą być słowami kluczowymi ani zarezerwowanymi słowami w języku JavaScript.

**Zadanie 4:**

**Algorytmy iteracyjne** to algorytmy, które używają pętli do powtarzalnego wykonywania określonych kroków lub operacji. Poniżej przedstawiam definicję i kilka przykładów prostych algorytmów iteracyjnych w języku JavaScript:

**Definicja** Algorytm iteracyjny to sekwencja kroków lub operacji, która jest wykonywana powtórnie, aż spełniony zostanie pewien warunek stopu.

Przykład 1: Sumowanie liczb od 1 do n za pomocą pętli `for`:

function sumaLiczbOd1DoN(n) {

     let suma = 0;

     for (let i = 1; i <= n; i++) {

       suma += i;

     }

     return suma;

   }

   const wynik = sumaLiczbOd1DoN(5); // Wynik to 1 + 2 + 3 + 4 + 5 = 15

   console.log(wynik);

W tym przykładzie funkcja `sumaLiczbOd1DoN` używa pętli `for` do sumowania liczb od 1 do `n`.

Przykład 2: Obliczanie silni liczby za pomocą pętli `while`:

function obliczSilnie(n) {

     let wynik = 1;

     let i = 1;

     while (i <= n) {

       wynik \*= i;

       i++;

     }

     return wynik;

   }

   const silniaZ5 = obliczSilnie(5); // 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

   console.log(silniaZ5);

W tym przykładzie funkcja `obliczSilnie` używa pętli `while` do obliczenia silni liczby `n`.

Przykład 3: Wyszukiwanie elementu w tablicy za pomocą pętli `for`:

function znajdzElement(tablica, szukanyElement) {

     for (let i = 0; i < tablica.length; i++) {

       if (tablica[i] === szukanyElement) {

         return i; // Zwraca indeks znalezionego elementu

       }

     }

     return -1; // Zwraca -1, jeśli element nie został znaleziony

   }

   const liczby = [10, 20, 30, 40, 50];

   const indeks = znajdzElement(liczby, 30); // Indeks 2 (30 jest na indeksie 2)

   console.log(indeks);

W tym przykładzie funkcja `znajdzElement` używa pętli `for` do wyszukiwania elementu w tablicy i zwraca jego indeks.

Algorytmy iteracyjne są użyteczne w wielu różnych sytuacjach, gdzie konieczne jest powtarzalne wykonywanie operacji. Warto pamiętać o odpowiednich warunkach stopu, aby uniknąć nieskończonych pętli.

**ZADANIE 5**

**\*\*Definicja algorytmu rekurencyjnego:\*\***

Algorytm rekurencyjny to algorytm, który rozwiązuje problem poprzez podział go na mniejsze podproblemy tego samego typu, a następnie rozwiązuje te podproblemy za pomocą tych samych kroków. Algorytm wykorzystuje rekurencyjne wywołania funkcji, aby rozwiązać te podproblemy.

\*\*Przykłady algorytmów rekurencyjnych w JavaScript:\*\*

1. \*\*Obliczanie silni:\*\*

function silnia(n) {

     if (n === 0) {

         return 1;

     } else {

         return n \* silnia(n - 1);

     }

 }

 console.log(silnia(5)); // Wynik: 120

2. \*\*Suma elementów tablicy:\*\*

function sumaTablicy(tablica) {

     if (tablica.length === 0) {

         return 0;

     } else {

         return tablica[0] + sumaTablicy(tablica.slice(1));

     }

 }

 console.log(sumaTablicy([1, 2, 3, 4, 5])); // Wynik: 15

3. \*\*Fibonacci:\*\*

function fibonacci(n) {

     if (n <= 1) {

         return n;

     } else {

         return fibonacci(n - 1) + fibonacci(n - 2);

     }

 }

 console.log(fibonacci(6)); // Wynik: 8 (0, 1, 1, 2, 3, 5, 8)

4. \*\*Obliczanie potęgi:\*\*

function potega(x, n) {

     if (n === 0) {

         return 1;

     } else {

         return x \* potega(x, n - 1);

     }

 }

 console.log(potega(2, 3)); // Wynik: 8 (2^3)

5. \*\*Wyszukiwanie binarne w posortowanej tablicy:\*\*

function wyszukiwanieBinarne(tablica, element, lewy = 0, prawy = tablica.length - 1) {

     if (lewy > prawy) {

         return -1;

     }

     const srodek = Math.floor((lewy + prawy) / 2);

     if (tablica[srodek] === element) {

         return srodek;

     } else if (tablica[srodek] < element) {

         return wyszukiwanieBinarne(tablica, element, srodek + 1, prawy);

     } else {

         return wyszukiwanieBinarne(tablica, element, lewy, srodek - 1);

     }

 }

 const posortowanaTablica = [1, 2, 3, 4, 5, 6, 7, 8, 9];

 console.log(wyszukiwanieBinarne(posortowanaTablica, 5)); // Wynik: 4

Algorytmy rekurencyjne mogą być bardzo eleganckie i intuicyjne, ale należy pamiętać, że nieodpowiednie wykorzystanie rekursji może prowadzić do błędów przepełnienia stosu. Dlatego ważne jest ostrożne planowanie i testowanie algorytmów rekurencyjnych.

**ZADANIE 6**

Silnia jest matematyczną funkcją, która jest często używana w programowaniu do rozwiązywania różnych problemów. Dla programisty silnia jest definowana jako iloczyn wszystkich dodatnich liczb całkowitych od 1 do danej liczby całkowitej n, oznaczanej jako "n!" (czytaj: "n silnia"). Zapisuje się to jako:

n! = n \* (n-1) \* (n-2) \* ... \* 2 \* 1

Główne cechy silnii dla programisty to:

1. **Rekurencyjna definicja**: Silnia jest często definiowana rekurencyjnie, co oznacza, że silnia n jest obliczana na podstawie silni (n-1), a ta z kolei na podstawie (n-2), aż do osiągnięcia silni 1, która wynosi 1.

2. **Silnia zdefiniowana dla 0:** Dla programisty, wartość silni dla 0 jest zazwyczaj przyjmowana jako 1 (0! = 1), chociaż istnieją różne definicje w matematyce.

3. **Duże liczby**: Silnia rośnie bardzo szybko wraz ze wzrostem n. Dla większych wartości n, silnia może przekroczyć zakres liczby całkowitej, co prowadzi do błędów w programach. W takich przypadkach stosuje się techniki, takie jak obliczanie silni modulo, aby uniknąć przepełnienia.

Silnia jest szeroko wykorzystywana w programowaniu do rozwiązywania problemów z dziedzin takich jak kombinatoryka, prawdopodobieństwo, analiza algorytmów itp. Może być również używana do obliczania wartości funkcji rekurencyjnych i analizy złożoności obliczeniowej algorytmów. W językach programowania, takich jak JavaScript, można napisać funkcje do obliczania silni, co jest przydatne w wielu aplikacjach programistycznych.

**ZADANIE 7**

Ciąg Fibonacciego to ciąg liczb naturalnych, w którym każdy kolejny element jest sumą dwóch poprzednich elementów. Zaczyna się on od dwóch pierwszych liczb: 0 i 1, a następnie każda kolejna liczba w ciągu jest sumą dwóch poprzednich. Oto kilka początkowych liczb w ciągu Fibonacciego:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Dla programisty ciąg Fibonacciego jest często wykorzystywany jako przykład do nauki programowania i algorytmów. Może być używany do demonstracji różnych koncepcji i technik, takich jak rekurencja, iteracja, dynamiczne programowanie, i wiele innych.

**ZADANIE 8**

// Metoda prostokątów

function rectangleMethod(func, a, b, n) {

     const h = (b - a) / n;

     let sum = 0;

     for (let i = 0; i < n; i++) {

         const x = a + i \* h;

         sum += func(x);

     }

     return h \* sum;

 }

 // Metoda trapezów

 function trapezoidMethod(func, a, b, n) {

     const h = (b - a) / n;

     let sum = 0;

     for (let i = 0; i < n; i++) {

         const x0 = a + i \* h;

         const x1 = a + (i + 1) \* h;

         sum += (func(x0) + func(x1)) / 2;

     }

     return h \* sum;

 }

 // Przykład użycia

 const f = x => x \* x; // Przykładowa funkcja, którą chcemy całkować

 const a = 0; // Dolna granica całkowania

 const b = 2; // Górna granica całkowania

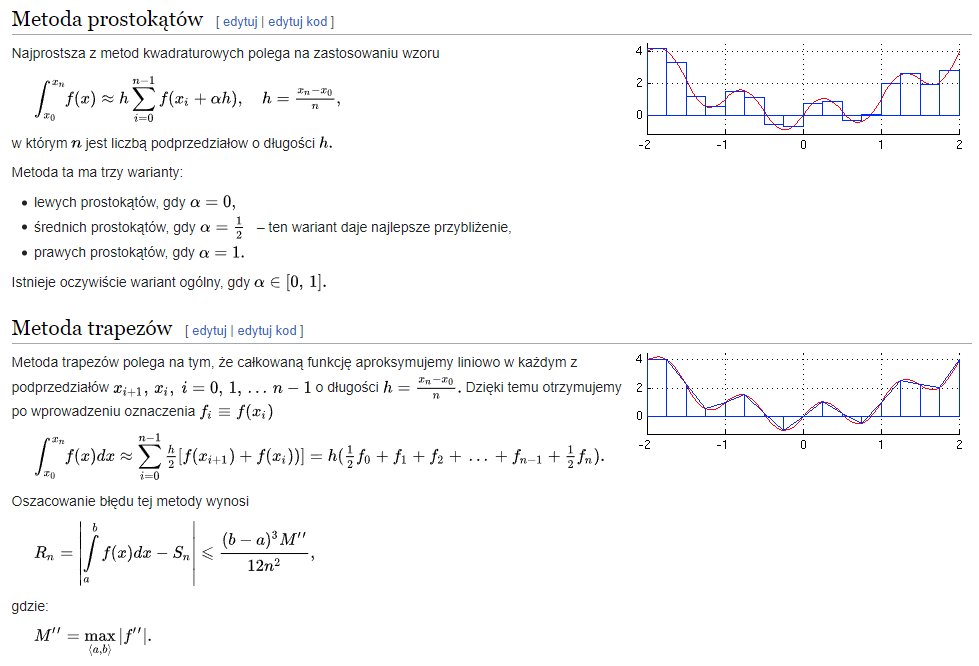
 const n = 1000; // Liczba podziałów przedziału

 const resultRectangle = rectangleMethod(f, a, b, n);

 const resultTrapezoid = trapezoidMethod(f, a, b, n);

 console.log(`Metoda prostokątów: ${resultRectangle}`);

 console.log(`Metoda trapezów: ${resultTrapezoid}`);



**ZADANIE 9**

// Deklaracja pustej tablicy

var mojaTablica = [];

// Dodawanie elementów do tablicy

for (var i = 0; i < 10000; i++) {

  mojaTablica.push(i);

}

// Teraz tablica "mojaTablica" zawiera 10 000 elementów o wartościach od 0 do 9999.

**ZADANIE 10**

Algorytm sortowania szybkiego (quicksort):

Algorytm sortowania szybkiego jest efektywnym algorytmem sortowania, który działa na zasadzie podziału tablicy na mniejsze podtablice, a następnie sortuje te podtablice rekurencyjnie. Algorytm ten działa na miejscu, co oznacza, że nie tworzy dodatkowych tablic.

1. Wybierz element pivot (punkt odniesienia). Możesz wybrać dowolny element z tablicy, na przykład pierwszy, ostatni lub środkowy.
2. Podziel tablicę na dwie części:
   * - Elementy mniejsze od pivota przenieś na lewą stronę.
   * - Elementy większe od pivota przenieś na prawą stronę.
3. Rekurencyjnie sortuj lewą i prawą część tablicy.
4. Po zakończeniu sortowania części, połącz je razem z pivotem, aby uzyskać posortowaną tablicę.

Poniżej znajduje się przykład sortowania szybkiego w JavaScript:

function quickSort(arr) {

    if (arr.length <= 1) {

        return arr;

    }

    const pivot = arr[0];

    const left = [];

    const right = [];

    for (let i = 1; i < arr.length; i++) {

        if (arr[i] < pivot) {

            left.push(arr[i]);

        } else {

            right.push(arr[i]);

        }

    }

    return [...quickSort(left), pivot, ...quickSort(right)];

}

const nieuporzadkowanaTablica = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];

const posortowanaTablica = quickSort(nieuporzadkowanaTablica);

console.log(posortowanaTablica); // Wyświetli posortowaną tablicę

W tym przykładzie `quickSort` jest funkcją rekurencyjną, która sortuje tablicę. Algorytm ten jest wydajny i działa dobrze dla większych zbiorów danych.