Programowanie Front-End

Wprowadzenie do JavaScript

mgr inż. Jakub Gogola

Czym jest JavaScript?

- wysokopoziomowy język programowania
- początkowo zaprojektowany do zastosowań webowych, obecnie - również do zastosowań backendowych, skryptowych
- multiplatformowy
- asynchroniczny
- multiparadygmatowy *(procedural, functional, OOP, event-driven)*

Czym jest JavaScript

Historia

- stworzony w **1995r.** przez **Brendana Eich'a**
- początkowo nazwany *LiveScript*, a następnie na obecne nazwę (na kanwie popularności języka Java)
- W **1996r.** Organizacja **ECMA** rozpoczęła pracę nad opracowaniem standardu Języka
  - **ES3** (2003) - stabilna języka *(RegExp, try-catch)*
  - **ES5** (2009) - *strict mode*, *JSON support*
  - **ES6/ES2015** (2015) - jeden z największych update'w (*let/const, arrow functions, classed, modules, Promises)*

JavaScript
Zmienne

- W języku JavaScript można wyróżnić dwa sposoby deklarowania zmiennych:
  - `let` - nie wymaga inicjalizacji, pozwala na zmianę wartości/przypisania, dostępna tylko w danym bloku (pętla, funkcja…)
  - `const` - wymaga inicjalizacji, wartość/przypisanie nie może być zmienione
    - przypisanie do zmiennej typu *const* tablicy lub obiektu **nie czyni ich niemutowalnymi**, a jedynie nie pozwala zmienić przypisania referencji do zmiennej

JavaScript

Zmienne

```
let counter;

console.log(counter);

counter = 10;

console.log(counter);

let name = 'John';

console.log(name);

name = 'Jane';

console.log(name);

const age = 30;

console.log(age);
```

```
const person = {
    name: 'John',
    age: 30,
};

console.log(person);

person.name = 'Jane';

console.log(person);

person = {
    name: 'Jane',
    age: 30,
}; // Error: Assignment to constant variable
```

JavaScript

Primitive types

- `String` - reprezentuje ciąg znaków
- `Number` - liczby całkowite i zmiennoprzecinkowe (64-bitowy zapis, podwójnej precyzji, zgodny ze standardem IEEE 754)
- `BigInt` - liczby całkowite wykraczające poza zakres typu `Number`
- `Boolean` - typ logiczny
- `Undefined` - wartość zmiennej, która została zadeklarowana, ale nie zainicjalizowana,
- `Null` - **zamierzony** brak wartości/referencji
- `Symbol` - unikatowy i niemutowany identyfikator

JavaScript

Primitive types

```
// String
const greeting = "Hello, world!";

// Number
const age = 30;
const price = 19.99;

// BigInt
const bigNum = 1234567890123456789012345678901234567890n;

// Boolean
const isStudent = true;

// Undefined
let value;
console.log(value); // undefined

// Null
const empty = null;

// Symbol
const id = Symbol('id');
```

JavaScript

Object types

- `Object` - złożony typ danych, który pozwala na przechowywanie zbiorów danych oraz bardziej złożonych struktur
  - `Object`
  - `Array`
  - `Function`
  - …

JavaScript

Object types

```
// Object
const person = {
    name: "John",
    age: 30
};

// Array
const numbers = [1, 2, 3, 4]; // Array, a special type of object

const sayHello = function() {
    console.log("Hello, world!");
}
```

Operatory
Arytmetyczne

- – - odejmowanie
- + - dodawanie
- * - mnożenie
- / - dzielenie
- % - modulo
- ** - potęgowanie
- ++ - inkrementacja (+1)
- — - dekrementacja (-1)

Operatory

Arytmetyczne

```
let x = 10;
let y = 3;

// Addition
console.log(x + y); // 13

// Subtraction
console.log(x - y); // 7

// Multiplication
console.log(x * y); // 30

// Division
console.log(x / y); // 3.3333...

// Modulus (remainder)
console.log(x % y); // 1
```

```
// Exponentiation
console.log(x ** y); // 1000

// Increment (postfix)
console.log(x++); // 10;
console.log(x); // 11

// Increment (prefix)
console.log(++x); // 12
console.log(x); // 12

// Decrement (postfix)
console.log(y--); // 3
console.log(y); // 2

// Decrement (prefix)
console.log(--y); // 1
console.log(y); // 1
```

Operatory

Przypisania

- = - przypisanie
- += - przypisanie z dodawaniem
- −= - przypisanie z odejmowaniem
- *= - przypisanie z odejmowaniem
- /= - przypisanie z dzieleniem
- %= - przypisanie z modulo
- **= - przypisanie z potęgowaniem

Operatory

Przypisania

```
let a = 5;

// Basic assignment
a = 10;
console.log(a); // 10

// Addition assignment
a += 5; // Equivalent to: a = a + 5
console.log(a); // 15

// Subtraction assignment
a -= 2; // Equivalent to: a = a - 2
console.log(a); // 13

// Multiplication assignment
a *= 3; // Equivalent to: a = a * 3
console.log(a); // 39

// Division assignment
a /= 3; // Equivalent to: a = a / 3
console.log(a); // 13

// Modulus assignment
a %= 4; // Equivalent to: a = a % 4
console.log(a); // 1

// Exponentiation assignment
a **= 3; // Equivalent to: a = a ** 3
console.log(a); // 1 (1^3 is still 1)
```

Operatory
Porównania

- == - równe *(sprawdzenie wartości)*
- === - ściśle równe *(sprawdzenie wartości oraz typu)*
- != - różne
- !== - ściśle różne
- > - większe
- < - mniejsze
- >= - większe równe
- <= - mniejsze równe

Operatory

Porównania

```
let b = 10;
let c = "10";

// Equal to (value only)
console.log(b == c); // true

// Strict equal to (value and type)
console.log(b === c); // false

// Not equal to (value only)
console.log(b != c); // false

// Strict not equal to (value and type)
console.log(b !== c); // true

// Greater than
console.log(b > 5); // true

// Less than
console.log(b < 15); // true

// Greater than or equal to
console.log(b >= 10); // true

// Less than or equal to
console.log(b <= 10); // true
```

Operatory

Logiczne

- && - (AND) koniunkcja
- || - (OR) alternatywa
- ! - (NOT) negacja

Operatory

Logiczne

```
let isLoggedIn = true;
let hasPermission = false;

// AND
console.log(isLoggedIn && hasPermission); // false

// OR
console.log(isLoggedIn || hasPermission); // true

// NOT
console.log(!isLoggedIn); // false
```

Operatory

Bitowe

- & - AND
- | - OR
- ^ - XOR
- ~ - NOT
- << - *left shift*
- >> - *right shift*
- >>> - *zero-fill right shift*

Operatory

Bitowe

```
let d = 5;   // 0101 in binary
let e = 1;   // 0001 in binary

// AND
console.log(d & e); // 1 (0001)

// OR
console.log(d | e); // 5 (0101)

// XOR
console.log(d ^ e); // 4 (0100)

// NOT
console.log(~d); // -6

// Left shift
console.log(d << 1); // 10 (1010)

// Right shift
console.log(d >> 1); // 2 (0010)

// Zero-fill right shift
console.log(d >>> 1); // 2 (0010)
```

Operatory

Warunkowe *(ternary)*

- `condition ? expressionIfTrue : expressionIfFalse` - skrócony zapis warunku *if-else*

Operatory

Warunkowe *(ternary)*

```
let age = 20;
let access = age >= 18 ? "Granted" : "Denied";
console.log(access); // "Granted"
```

Operatory
Typu

- `typeof` - zwraca typ zmiennej lub wyrażenia
- `instanceof` - sprawdza czy obiekt jest instancją danej klasy lub konstruktora

Operatory

Typu

```
let name = "Alice";
let num = 42;
let isTrue = true;
let person = { name: "Alice", age: 25 };

// typeof operator
console.log(typeof name); // "string"
console.log(typeof num); // "number"
console.log(typeof isTrue); // "boolean"
console.log(typeof person); // "object"

// instanceof operator
console.log(person instanceof Object); // true

function Person() {}
let john = new Person();
console.log(john instanceof Person); // true
```

Komentarze

- W języku JavaScript można wyróżnić następujące rodzaje komentarzy
  - jednoliniowe
  - wieloliniowe
  - dokumentujące

Komentarze

Przykłady

```
// This is a single-line comment
let name = "Alice"; // Comment after a statement

// Uncomment the next line to see it in action
// console.log(name);

/*
This is a multi-line comment.
It can span multiple lines.
*/
let age = 25;

/*
Commenting out a block of code:

console.log("This code will not run.");
console.log("This is also commented out.");
*/
```

```
/**
 * Calculates the sum of two numbers.
 *
 * @param {number} a - The first number.
 * @param {number} b - The second number.
 * @returns {number} The sum of the two numbers.
 */
function add(a, b) {
    return a + b;
}
```

Wyrażenia warunkowe

if-else

```
let age = 20;

if (age < 0) {
    console.log("Invalid value.");
}

if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

Wyrażenia warunkowe

switch-case

```
const breed = "Golden Retriever";

switch (breed) {
    case "Labrador Retriever":
        console.log("Labrador Retrievers are friendly and outgoing.");
        break;
    case "German Shepherd":
        console.log("German Shepherds are loyal and highly trainable.");
        break;
    case "Golden Retriever":
        console.log("Golden Retrievers are intelligent and friendly.");
        break;
    case "Bulldog":
        console.log("Bulldogs are calm and courageous.");
        break;
    case "Poodle":
        console.log("Poodles are highly intelligent and hypoallergenic.");
        break;
    default:
        console.log("Unknown breed. Each dog is unique and special!");
        break;
}
```

Pętle

for, for…of, for…in

```
for (let i = 0; i < 5; i++) {
    console.log("Iteration
number:", i);
}
// Output: Iteration number: 0,
Iteration number: 1, ..., Iteration
number: 4
```

```
const dogs = ["Labrador", "Beagle",
"Bulldog", "Poodle"];

for (const breed of dogs) {
    console.log(breed);
}
// Output: Labrador, Beagle,
Bulldog, Poodle
```

```
const dog = {
    name: "Buddy",
    breed: "Golden Retriever",
    age: 3
};

for (const property in dog) {
    console.log(`${property}: $
{dog[property]}`);
}
// Output: name: Buddy, breed:
Golden Retriever, age: 3
```

Pętle

while, do…while

```
let count = 0;

while (count < 5) {
    console.log("Count is:", count);
    count++;
}
// Output: Count is: 0, Count is: 1, ..., Count is: 4
```

```
let number = 0;

do {
    console.log("Number is:", number);
    number++;
} while (number < 3);
// Output: Number is: 0, Number is: 1, Number is: 2
```

Instrukcje sterowania pętlą

break, continue

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break; // Exit the loop when i is 5
    }
    console.log(i);
}
// Output: 0, 1, 2, 3, 4
```

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        continue; // Skip the iteration when i is 2
    }
    console.log(i);
}
// Output: 0, 1, 3, 4 (skips 2)
```

Funkcje

Standardowe

- Standardowe funkcje w języku JavaScript definiujemy za pomocą słowa kluczowego `function`.
- Funkcje standardowe **są *hoistowane***, to znaczy - można je wywoływać przed ich deklaracją w kodzie
- Posiadają własny kontakt *this*

```
function describeDog(breed) {
    console.log(`${breed}s are wonderful dogs!`);
}

describeDog("Labrador"); // Output: "Labradors are wonderful dogs!"
```

Funkcje

Strzałkowe *(arrow functions)*

- Funkcje strzałkowe **nie są *hoistowane***, to znaczy - nie można ich wywoływać przed ich deklaracją w kodzie
- Nie posiadają własnego kontekstu *this* (dziedziczą go od *rodzica*)

```
const describeDog = (breed) => {
    console.log(`${breed}s are wonderful dogs!`);
};

describeDog("Beagle"); // Output: "Beagles are wonderful dogs!"
```

Funkcje

ze zwracaną wartością

- Funkcje mogą również zwracać wartość o odpowiednim typie
- Złą praktyką jest zwracanie wielu różnych typów w jednej funkcji

```
function getDogDescription(breed) {
    return `${breed}s are very friendly and loyal.`;
}

let description = getDogDescription("Golden Retriever");
console.log(description); // Output: "Golden Retrievers are very friendly and loyal."

const getDogDescription = (breed) => {
    return `${breed}s are very friendly and loyal.`;
};

let description = getDogDescription("Bulldog");
console.log(description); // Output: "Bulldogs are very friendly and loyal."

const getDogName = (name) => `This dog's name is ${name}.`;

console.log(getDogName("Buddy")); // Output: "This dog's name is Buddy."
```

Funkcje

bez zwracanego typu

- Funkcje, które nie zwracają żadnego typu domyślnie zwracają wartość `undefined`

```
function bark() {
    console.log("Woof! Woof!");
}

bark(); // Output: "Woof! Woof!"

const bark = () => {
    console.log("Woof! Woof!");
};

bark(); // Output: "Woof! Woof!"
```

Obiekty

Definicja

- Obiekt to zbiór (kolekcja) powiązanych ze sobą danych i funkcji (metod)
- Obiekt składa się z nieuporządkowanego zbioru par klucz-wartość *(key-value)* nazywanymi właściwościami *(properties)*, gdzie:
  - *key* to identyfikator
  - *value* - wartość (referencja) przypisana do danego klucza

Obiekty

Object literals

- Najprostsza metoda utworzenia obiektu to użycie *object literal*

```
const dog = {
    name: "Buddy",
    breed: "Golden Retriever",
    age: 3,
    bark: function() {
        console.log("Woof! Woof!");
    }
};

console.log(dog.name); // Output: "Buddy"
dog.bark(); // Output: "Woof! Woof!"
```

Obiekt

new Object()

- Obiekty można również tworzyć używając konstruktora `Object`

Obiekty

Constructor functions

- *Constructor functions* to sposób na wielokrotne wykorzystanie struktury obiektu poprzez parametryzację.

```
function Dog(name, breed, age) {
    this.name = name;
    this.breed = breed;
    this.age = age;
    this.bark = function() {
        console.log("Woof! Woof!");
    };
}

const myDog = new Dog("Max", "Labrador", 5);
console.log(myDog.breed); // Output: "Labrador"
myDog.bark(); // Output: "Woof! Woof!"
```

Obiekty

Klasy (ES6+)

- Klasy zapewniają *najczystszy* i najbardziej złożony sposób tworzenia obiektu
- Definiuje się je podobnie jak w innych językach implementujących paradygmat programowania obiektowego
- W językach prototypowych takich jak JavaScript klasy najczęściej to *syntactic sugar* na system prototypów

Obiekty

Klasy (ES6+)

```
class Dog {
    constructor(name, breed, age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    bark() {
        console.log("Woof! Woof!");
    }
}

const myDog = new Dog("Bella", "Beagle", 2);
console.log(myDog.age); // Output: 2
myDog.bark(); // Output: "Woof! Woof!"
```

Obiekty

Operacje na obiekcie

- Na obiektach można wykonywać różne operacje

```
const dog = {
    name: "Buddy",
    breed: "Golden Retriever",
    age: 3,
    bark: function() {
        console.log("Woof! Woof!");
    }
};

console.log(dog.name); // Output: "Buddy"
dog.bark(); // Output: "Woof! Woof!"

const prop = "breed";
console.log(dog[prop]); // Output: "Golden Retriever"
```

```
dog.color = "Golden";
dog.age = 4; // Modifying an existing property
console.log(dog.color); // Output: "Golden"
console.log(dog.age); // Output: 4

delete dog.age;
console.log(dog.age); // Output: undefined
```

Obiekty
Zagnieżdżanie obiektów

- Obiekty można w sobie zagnieżdżać

```
const kennel = {
    location: "Main Street",
    dogs: [
        { name: "Rex", breed: "Bulldog" },
        { name: "Bella", breed: "Beagle" }
    ]
};

console.log(kennel.dogs[1].name); // Output: "Bella"
```

Obiekty

Metody wbudowane

- Dla wszystkich obiektów można stosować statyczne metody przynależące do prototypu `Object`

```
const dog = {
    name: "Buddy",
    breed: "Golden Retriever",
    age: 3,
    bark: function() {
        console.log("Woof! Woof!");
    }
};

const keys = Object.keys(dog);
console.log(keys); // Output: ["name", "breed", "bark"]

const values = Object.values(dog);
console.log(values); // Output: ["Buddy", "Golden Retriever", function]

const entries = Object.entries(dog);
console.log(entries);
// Output: [["name", "Buddy"], ["breed", "Golden Retriever"], ["bark", function]]
```

Prototypy

Mechanizm prototypów

- JavaScript jest językiem opartym na mechanizmie prototypów
- Prototypy pozwalają na dziedziczenie funkcjonalności pomiędzy obiektami
- Każdy obiekt może posiadać prototyp - działa on jak obiekt szablonowy, po którym można dziedziczyć właściwości i metody
- Prototypy pozwalają na linkowanie obiektów, które po sobie dziedziczą.

Prototypy

Definicja

- **Prototyp** to po prostu obiekt, po którym inne obiekty dziedziczą własności *(properties)*.
- Każdy obiekt ma własność `[[Prototype]]` (`__proto__`), która wskazuje na iny obiekt
- Każdy obiekt utworzony przez użycie konstruktora (słowo kluczowe new) ma własność `[[Prototype]]`, która wskazuje na konstruktor prototypu tego obiektu.
- W przypadku, gdy w danym obiekcie nie istnieje jakaś własność/metoda to używana jest ta zdefiniowana w prototypie.

Prototypy

Przykład działania prototypu

```
const dog = {
  bark() {
    console.log("Woof! Woof!");
  },
};

const myDog = Object.create(dog);
myDog.name = "Buddy";

console.log(myDog.name); // "Buddy" (own property)
myDog.bark(); // "Woof! Woof!" (inherited from dog prototype)
```

Prototypy

Łańcuch prototypów *(prototype chain)*

- **Łańcuch prototypów** *(prototype chain)* to ciąg powiązań pomiędzy obiektami - każdy obiekt wskazuje na swój prototyp

```
function Animal() {
    this.isLiving = true;
}

Animal.prototype.eat = function() {
    console.log("Eating...");
};

function Dog(name) {
    this.name = name;
}

Dog.prototype = Object.create(Animal.prototype); // Dog inherits from Animal
Dog.prototype.bark = function() {
    console.log("Woof! Woof!");
};

const myDog = new Dog("Buddy");
myDog.eat(); // Output: "Eating..." (inherited from Animal)
myDog.bark(); // Output: "Woof! Woof!" (own method)
```

Prototypy

Properties shadowing

- *Z properties shadowing* mamy do czynienia, gdy jakiś obiekt przysłania (nadpisuje) własność o tej samej nazwie, którą posiada jego prototyp/

```
function Animal() {
    this.isLiving = true;
}

function Dog(name) {
    this.name = name;
}

Dog.prototype = Object.create(Animal.prototype); // Dog inherits from Animal
Dog.prototype.bark = function() {
    console.log("Woof! Woof!");
};

const myDog = new Dog("Buddy");

myDog.isLiving = false; // Shadows the `isLiving` property on the prototype
console.log(myDog.isLiving); // Output: false
```

Prototypy

Own properties

- *Own properties* to te własności, które są zdefiniowane bezpośrednio w obiekcie i nie należą do jego prototypu

```
const dog = {
  breed: "Labrador",
};

const myDog = Object.create(dog);
myDog.breed = "Beagle"; // Shadows the prototype's breed property
myDog.age = 2;

console.log(myDog.breed); // "Beagle" (own property)
console.log(dog.breed); // "Labrador" (prototype property remains unchanged)

console.log(myDog.hasOwnProperty("breed")); // true, shadowed property
console.log(myDog.hasOwnProperty("age")); // true, defined on the object, doesn't belong to the prototype
console.log(myDog.hasOwnProperty("bark")); // false (inherited from the prototype)
```

Prototypy

Prototypy a dziedziczenie

- W języku JavaScript dziedziczenie jest oparte na prototypach a nie na klasach. Oznacza to, że obiekty dziedziczą bezpośrednio po innych obiektach.
- Kiedy prototyp danego obiektu wskazuje na inny obiekt to tworzony jest link pomiędzy obiektami pełniący funkcję dziedziczenia.
- Do prototypu każdego obiektu można dodawać nowe własności i każdy inny obiekt, który wskazuje na prototyp tego obiektu będzie miał do nich dostęp

Prototypy

Prototypy a dziedziczenie

```
function Animal() {
    this.isLiving = true;
}

Animal.prototype.eat = function() {
    console.log("Eating...");
};

function Dog(name) {
    this.name = name;
}

Dog.prototype = Object.create(Animal.prototype); // Dog
inherits from Animal
Dog.prototype.bark = function() {
    console.log("Woof! Woof!");
};

const myDog = new Dog("Buddy");
myDog.eat(); // Output: "Eating..." (inherited from
Animal)
myDog.bark(); // Output: "Woof! Woof!" (own method)
```

```
Dog.prototype = Object.create(Animal.prototype); // Dog
inherits from Animal

Animal.prototype.sleep = function() {
    console.log("Sleeping...");
};

myDog.sleep(); // Output: "Sleeping..." (inherited from
Animal prototype)
```

Klasy
OOP

- Język JavaScript osiąga dziedziczenie pomiędzy obiektami używając mechanizmu prototypów.

- Począwszy od standardu ES6 wprowadzono do języka mechanizm klas, który jest warstwą abstrakcji nad mechanizmem prototypów *(syntactic sugar)* i pozwala w wygodny sposób realizować paradygmat OOP.

Klasy

Tworzenie klasy

```
class Dog {
    constructor(name, breed, age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    bark() {
        console.log("Woof! Woof!");
    }
}

// Creating an instance of the Dog class
const myDog = new Dog("Buddy", "Golden Retriever", 3);
console.log(myDog.name); // Output: "Buddy"
myDog.bark(); // Output: "Woof! Woof!"
```

Klasy

Metody, metody statyczne

```
class Cat {
    constructor(name) {
        this.name = name;
    }

    meow() {
        console.log("Meow!");
    }
}

const myCat = new Cat("Whiskers");
myCat.meow(); // Output: "Meow!"

class MathUtil {
    static add(x, y) {
        return x + y;
    }
}

console.log(MathUtil.add(5, 3)); // Output: 8
```

Klasy

Dziedziczenie

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    eat() {
        console.log(`${this.name} is eating.`);
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Calls the parent class's constructor
        this.breed = breed;
    }

    bark() {
        console.log("Woof! Woof!");
    }
}

const myDog = new Dog("Max", "Labrador");
myDog.eat(); // Output: "Max is eating." (inherited from Animal)
myDog.bark(); // Output: "Woof! Woof!"
```

Tablice

Definicja

- **Tablice** w języku JavaScript to **specjalny rodzaj obiektu**, który pozwala na przechowywanie **uporządkowanego zbioru** (kolekcji) wartości.
- Tablice mogą przechowywać wartości wszystkich typów.
- Indeksowanie rozpoczyna się od **zera**.

Tablice

Podstawowe operacje

- Podstawowymi operacjami na tablicy są jej tworzenie, sprawdzanie jej zawartości, modyfikacja zawartości oraz sprawdzanie długości tablicy.

```
const dogBreeds = ["Labrador Retriever", "Beagle", "Bulldog", "Poodle", "Golden Retriever"];

// Accessing elements
console.log(dogBreeds[0]); // Output: "Labrador Retriever"
console.log(dogBreeds[2]); // Output: "Bulldog"

// Modifying elements
dogBreeds[1] = "German Shepherd";
console.log(dogBreeds);
// Output: ["Labrador Retriever", "German Shepherd", "Bulldog", "Poodle", "Golden Retriever"]

// Adding elements
dogBreeds[1] = "German Shepherd";
console.log(dogBreeds);
// Output: ["Labrador Retriever", "German Shepherd", "Bulldog", "Poodle", "Golden Retriever"]

// Checking the length
console.log(dogBreeds.length); // Output: 5
```

Tablice

Dodawanie i usuwanie elementów

- Elementy w tablicy można usuwać jak i również je dodać

```
const dogBreeds = ["Labrador Retriever", "Beagle", "Bulldog", "Poodle", "Golden Retriever"];

// Adding a breed to the end
dogBreeds.push("Chihuahua");
console.log(dogBreeds);
// Output: ["Labrador Retriever", "German Shepherd", "Bulldog", "Poodle", "Golden Retriever", "Chihuahua"]

// Removing the last breed
dogBreeds.pop();
console.log(dogBreeds);
// Output: ["Labrador Retriever", "German Shepherd", "Bulldog", "Poodle", "Golden Retriever"]

// Removing the first breed
dogBreeds.shift();
console.log(dogBreeds);
// Output: ["German Shepherd", "Bulldog", "Poodle", "Golden Retriever"]

// Adding a breed to the beginning
dogBreeds.unshift("Boxer");
console.log(dogBreeds);
// Output: ["Boxer", "German Shepherd", "Bulldog", "Poodle", "Golden Retriever"]
```

Tablice

Inne metody

```
const dogBreeds = ["Labrador Retriever", "Beagle",
"Bulldog", "Poodle", "Golden Retriever"];

const smallBreeds = ["Chihuahua", "Pug",
"Dachshund"];
const allBreeds = dogBreeds.concat(smallBreeds);

// Concats two arrays
console.log(allBreeds);
// Output: ["Boxer", "German Shepherd", "Bulldog",
"Poodle", "Golden Retriever", "Chihuahua", "Pug",
"Dachshund"]

// Allows to create sub-array without modifying the
original array
const popularBreeds = dogBreeds.slice(1, 4);
console.log(popularBreeds);
// Output: ["German Shepherd", "Bulldog", "Poodle"]
```

```
// Modifies the original array in place, allowing to
add or remove elements
// Adding "Cocker Spaniel" and "Shih Tzu" at index 2
dogBreeds.splice(2, 0, "Cocker Spaniel", "Shih Tzu");
console.log(dogBreeds);
// Output: ["Boxer", "German Shepherd", "Cocker
Spaniel", "Shih Tzu", "Bulldog", "Poodle", "Golden
Retriever"]

console.log(dogBreeds.indexOf("Bulldog")); // Output:
4
console.log(dogBreeds.includes("Poodle")); // Output:
true
console.log(dogBreeds.includes("Husky")); // Output:
false

// Sorts the array in place
dogBreeds.sort();
console.log(dogBreeds);
// Output: ["Boxer", "Bulldog", "Cocker Spaniel",
"German Shepherd", "Golden Retriever", "Poodle",
"Shih Tzu"]
```

Programowanie funkcyjne

Definicja

- **Programowanie funkcyjne** to paradygmat, który skupia się na wykorzystaniu *pure functions* i niezmienność/niemutowalność *(immutability)*.

- W tym paradygmacie funkcje traktowane są jako *first-class citizens* - mogą być przypisywane jako wartości zmiennych, przekazywane jako argumenty do funkcji/metod albo z nich zwracane.

Programowanie funkcyjne

Pure function

- *Pure function* to taka funkcja, która dla tych samych wartości podanych na wejściu (przez argumenty) zwróci zawsze taką samą wartość na wyjściu (wynik).
- Nie modyfikuje ona żadnych zmiennych, które nie należą do jej kontekstu (np. są globalne).

```
// Pure function
function add(x, y) {
    return x + y;
}
```

Programowanie funkcyjne

Immutability

- W paradygmacie programowania funkcyjnego ważna jest **niemutowalność** - tworzone dane (wartości zmiennych) nie są modyfikowane po inicjalizacji - zamiast tego tworzone są ich kopie, na których przeprowadza się operacje.

```
// Immutability
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4]; // Creates a new array
console.log(newNumbers); // Output: [1, 2, 3, 4]
```

Programowanie funkcyjne

Funkcje jako *first-class citizens*

- Przykładem wykorzystania funkcji jako *first-class citizens* są ***higher-order functions*** (funkcje wyższego rzędu), które jako przyjmują jako argumenty inne funkcje lub zwracają je jako wynik.
- *Z higher-order functions* mamy do czynienia w przypadku metod specyficznych dla tablic.

```
// First-class citizens - higher-order functions
const numbersArray = [1, 2, 3, 4, 5];

// Using `map` to create a new array where each element is doubled
const doubled = numbersArray.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

Tablice a programowanie funkcyjne

map()

- Metoda `map()` tworzy nową tablicę aplikując do każdego elementu oryginalnej tablicy funkcję, która modyfikuje jego wartość

```
const numbers = [1, 2, 3];
const squared = numbers.map(num => num * num);
console.log(squared); // Output: [1, 4, 9]
```

Tablice a programowanie funkcyjne

filter()

- Metoda `filter()` tworzy nową tablicę, która zawiera jedynie elementy spełniające warunek zawarty w funkcji podanej jako argument.

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

Tablice a programowanie funkcyjne

reduce()

- `reduce()` akumuluje wszystkie elementy tablicy w pojedynczej wartości (liczba, ciąg znaków, obiekt, …)

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(sum); // Output: 10
```

Tablice a programowanie funkcyjne

some(), every()

- `some()` sprawdza czy przynajmniej jeden element tablicy spełnia podany warunek
- `every()` sprawdza czy wszystkie elementy tablicy spełniają podany warunek

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

Programowanie funkcyjne

Zalety

- **przewidywalność** - *pure functions* i niemutowalność sprawiają, że kod jest bardziej przewidywalny
- **modularność** - paradygmat funkcyjny zmusza programistę do tworzenia małych, prostych i reużywalnych funkcji, które pomagają w porządkowaniu kodu i poprawiają jego przejrzystość
- **łatwe testowanie** - *pure functions* nie powodują efektów ubocznych *(side effects)* co ułatwia testowanie kodu

Asynchroniczność

Definicja

- JavaScript jest **jednowątkowym** i synchronicznym językiem programowania - kod jest wykonywany *linijka po linijce* i każda z operacji musi zakończyć się zanim zacznie się wykonywanie kolejnej

- Mechanizm **asynchroniczności** pozwala na rozpoczęcie danej operacji bez czekania na jej wynik i kontynuację wykonywania kolejnych poleceń.

Asynchroniczność

Zalety

- Programowanie asynchroniczne ma bardzo duże zadanie dla operacji, które zajmują sporo czasu - zapytania do API, operacje na plikach, komunikacja z bazą danych, …
- Użycie mechanizmu asynchroniczność w aplikacji pozwala usprawnić jej resaponsywność i wydajność

Asynchroniczność

Callbacks

- *Callback* to funkcja przekazana jako argument do innej funkcji i wykonana w momencie, gdy zakończy się wykonywanie asynchronicznego kodu np. timeout’u

```
// Callbacks
function fetchDogBreed(callback) {
    setTimeout(() => {
        callback("Labrador Retriever");
    }, 2000);
}

fetchDogBreed(breed => {
    console.log(`Fetched breed: ${breed}`);
});
// Output after 2 seconds: "Fetched breed: Labrador Retriever"
```

Asynchroniczność

Promises

- *Promise* to obiekt, który reprezentuje wynik asynchronicznej operacji (sukces lub niepowodzenie).
- Obiekty typu *Promise* mogą znajdować się w jednym z trzech stanów:
  - p*ending* - operacja jest w trakcie wykonywania
  - *fullfiled* - operacja zakończyła się sukcesem
  - r*ejected* - operacja zakończyła się niepowodzeniem

Asynchroniczność

Promises

```
function fetchDogBreed() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Golden Retriever");
        }, 2000);
    });
}

fetchDogBreed()
    .then(breed => {
        console.log(`Fetched breed: ${breed}`);
    })
    .catch(error => {
        console.error(`Error fetching breed: ${error}`);
    });
// Output after 2 seconds: "Fetched breed: Golden Retriever"
```

Asynchroniczność

async/await

- Składania *async/await* to *syntactic sugar* na mechanizm *Promise* - został wprowadzony w standardzie ES8 w 2017r.
- Jego celem aby zachowanie i składnię asynchronicznego kodu upodobnić do jego synchronicznej wersji oraz uprościć korzystanie z *Promise'ów*.

Asynchroniczność

async/await

```
async function fetchDogBreed() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Beagle");
        }, 2000);
    });
}

async function displayBreed() {
    try {
        const breed = await fetchDogBreed();
        console.log(`Fetched breed: ${breed}`);
    } catch (error) {
        console.error(`Error fetching breed: ${error}`);
    }
}

displayBreed();
// Output after 2 seconds: "Fetched breed: Beagle"
```

Asynchroniczność

Chaining promises

```
// Chaining promises
function getDogBreed() {
    return new Promise(resolve => {
        setTimeout(() => resolve("Poodle"), 1000);
    });
}

function getBreedSize(breed) {
    return new Promise(resolve => {
        setTimeout(() => resolve(`${breed} - medium size`), 1000);
    });
}

getDogBreed()
    .then(breed => getBreedSize(breed))
    .then(sizeDescription => console.log(sizeDescription));
// Output after 2 seconds: "Poodle - medium size"
```

Asynchroniczność

try…catch

```
async function getDogBreed() {
    throw new Error("Failed to fetch breed.");
}

async function showBreed() {
    try {
        const breed = await getDogBreed();
        console.log(`Breed: ${breed}`);
    } catch (error) {
        console.error(`Error: ${error.message}`);
    }
}

showBreed();
// Output: "Error: Failed to fetch breed."
```

Asynchroniczność

Promise.all()

```
const breedFetch1 = new Promise(resolve => setTimeout(() => resolve("Bulldog"), 1000));
const breedFetch2 = new Promise(resolve => setTimeout(() => resolve("Dachshund"), 2000));

Promise.all([breedFetch1, breedFetch2])
    .then(breeds => {
        console.log(`Fetched breeds: ${breeds.join(", ")}`);
    });
// Output after 2 seconds: "Fetched breeds: Bulldog, Dachshund"
```

Asynchroniczność

Promise.allSettled()

```
const breedFetch3 = new Promise((resolve) => setTimeout(() => resolve("Bulldog"), 1000));
const breedFetch4 = new Promise((resolve, reject) => setTimeout(() => reject("Beagle fetch failed"), 1500));
const breedFetch5 = new Promise((resolve) => setTimeout(() => resolve("Poodle"), 500));

Promise.allSettled([breedFetch3, breedFetch4, breedFetch5])
    .then((results) => {
        results.forEach((result, index) => {
            if (result.status === "fulfilled") {
                console.log(`Promise ${index + 1} fulfilled with value: ${result.value}`);
            } else {
                console.log(`Promise ${index + 1} rejected with reason: ${result.reason}`);
            }
        });
    });

// Output after 1.5 seconds:
// "Promise 1 fulfilled with value: Bulldog"
// "Promise 2 rejected with reason: Beagle fetch failed"
// "Promise 3 fulfilled with value: Poodle"
```

Asynchroniczność

Promise.race()

```
const breedFetch1 = new Promise(resolve => setTimeout(() => resolve("Pug"), 1000));
const breedFetch2 = new Promise(resolve => setTimeout(() => resolve("Chihuahua"), 2000));

Promise.race([breedFetch1, breedFetch2])
    .then(breed => {
        console.log(`Fastest fetched breed: ${breed}`);
    });
// Output after 1 second: "Fastest fetched breed: Pug"
```

Literatura

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
- Przykłady z wykładu - https://github.com/JakubGogola-IDENTT/dsw-frontend-lecture-2024/tree/main/lecture-2

Dziękuję za uwagę!