Projektowanie Aplikacji Internetowych
Wykład 6

Nowoczesny JavaScript, narzędzia i programowanie asynchroniczne

Mateusz Pawełkiewicz

1. Nowe funkcje wprowadzone w ES6+

ECMAScript 6 (ES6), znany również jako ECMAScript 2015, wprowadził znaczące ulepszenia do języka JavaScript, czyniąc go bardziej efektywnym i elastycznym. W kolejnych latach wprowadzono kolejne wersje (ES7, ES8, itd.), które kontynuują rozwój języka.

1.1 let i const

1.1.1 let

- **Zakres blokowy**: Zmienna zadeklarowana za pomocą let jest ograniczona do bloku { }, w którym została zadeklarowana.
- **Brak hoistingu**: let nie jest hoistowane w ten sam sposób co var, co pomaga uniknąć błędów.

Przykład:

```
{
  let x = 10;
  console.log(x); // 10
}
console.log(x); // Błąd: x is not defined
```

1.1.2 const

- **Stała**: Wartość przypisana do const nie może być ponownie przypisana.
- **Obiekty i tablice**: Można modyfikować ich zawartość, ale nie można przypisać nowej referencji.

Przykład:

```
const PI = 3.1415;
PI = 3; // Błąd: nie można przypisać nowej wartości

const osoba = { imie: "Jan" };
osoba.imie = "Adam"; // Poprawne
osoba = {}; // Błąd: nie można przypisać nowej referencji
```

1.2 Funkcje strzałkowe (Arrow Functions)

Funkcje strzałkowe wprowadzają krótszą składnię deklaracji funkcji i mają lepsze zachowanie w kontekście this.

1.2.1 Składnia

```
// Funkcja tradycyjna
function suma(a, b) {
  return a + b;
}

// Funkcja strzałkowa
const suma = (a, b) => a + b;
```

- **Bez parametrów**:

  ```
  const powitanie = () => console.log("Witaj!");
  ```

- **Jeden parametr**:

  ```
  const kwadrat = x => x * x;
  ```

- **Więcej instrukcji**:

  ```
  const mnozenie = (a, b) => {
    let wynik = a * b;
    return wynik;
  };
  ```

1.2.2 this w funkcjach strzałkowych

Funkcje strzałkowe nie mają własnego this; dziedziczą this z kontekstu, w którym zostały zdefiniowane.

Przykład:

```
function Osoba() {
  this.wiek = 0;

  setInterval(() => {
    this.wiek++;
    console.log(this.wiek);
  }, 1000);
}

let jan = new Osoba();
```

W powyższym przykładzie this odnosi się do obiektu Osoba, dzięki czemu można poprawnie aktualizować wartość wiek.

1.3 Szablony stringów (Template Strings)

Szablony stringów pozwalają na tworzenie łańcuchów znaków z interpolacją zmiennych oraz wieloliniowe ciągi tekstowe.

1.3.1 Interpolacja zmiennych

- Używamy znaku backtick ` zamiast cudzysłowu.
- Wstawiamy zmienne za pomocą ${zmienna}.

Przykład:

```
let imie = "Jan";
let powitanie = `Cześć, ${imie}!`;
console.log(powitanie); // Cześć, Jan!
```

1.3.2 Wieloliniowe ciągi tekstowe

Przykład:

```
let tekst = `To jest
wieloliniowy
ciąg tekstowy.`;
console.log(tekst);
```

1.4 Destrukturyzacja

Destrukturyzacja pozwala na wyodrębnianie wartości z tablic lub obiektów i przypisywanie ich do zmiennych.

1.4.1 Destrukturyzacja tablic

Przykład:

```
let liczby = [1, 2, 3];
let [a, b, c] = liczby;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- **Pomijanie elementów**:

  ```
  let [pierwszy, , trzeci] = liczby;
  console.log(pierwszy); // 1
  console.log(trzeci); // 3
  ```

- **Operator rest w destrukturyzacji**:

  ```
  let [pierwszy, ...reszta] = liczby;
  console.log(pierwszy); // 1
  console.log(reszta); // [2, 3]
  ```

1.4.2 Destrukturyzacja obiektów

Przykład:

```
let osoba = { imie: "Jan", wiek: 30 };
let { imie, wiek } = osoba;
console.log(imie); // Jan
console.log(wiek); // 30
```

- **Zmiana nazw zmiennych**:

  ```
  let { imie: nazwa, wiek: lata } = osoba;
  console.log(nazwa); // Jan
  console.log(lata); // 30
  ```

- **Wartości domyślne**:

```
let { imie, nazwisko = "Nowak" } = osoba;
console.log(nazwisko); // Nowak
```

1.5 Parametry domyślne funkcji

Pozwalają na ustawienie domyślnych wartości dla parametrów funkcji.

Przykład:

```
function powitanie(imie = "Gość") {
  console.log(`Witaj, ${imie}!`);
}

powitanie(); // Witaj, Gość!
powitanie("Anna"); // Witaj, Anna!
```

1.6 Spread Operator

Operator spread (…) pozwala na rozwijanie iterowalnych obiektów (np. tablic) w miejscach, gdzie oczekiwane są zero lub więcej argumentów.

Przykład z tablicami:

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

Przykład z funkcjami:

```
function suma(a, b, c) {
  return a + b + c;
}

let liczby = [1, 2, 3];
console.log(suma(...liczby)); // 6
```

---

2. Programowanie asynchroniczne w JavaScript

JavaScript jest językiem jednoprocesowym (single-threaded), co oznacza, że wykonuje jedną operację na raz. Aby radzić sobie z operacjami asynchronicznymi (np. zapytania sieciowe, operacje na plikach), JavaScript wykorzystuje mechanizmy takie jak promisy i async/await.

2.1 Promisy (Promises)

Promisy to obiekty reprezentujące wartość, która może być dostępna teraz, w przyszłości lub nigdy. Ułatwiają zarządzanie asynchronicznością i unikają tzw. "callback hell".

2.1.1 Tworzenie promisa

```
let obietnica = new Promise((resolve, reject) => {
  // Operacja asynchroniczna
  if (/* sukces */) {
    resolve(wynik);
  } else {
    reject(błąd);
  }
});
```

2.1.2 Używanie promisa

```
obietnica
  .then(wynik => {
    // Obsługa wyniku w przypadku sukcesu
  })
  .catch(błąd => {
    // Obsługa błędu
  });
```

2.1.3 Przykład praktyczny

Symulacja operacji asynchronicznej:

```
function pobierzDane() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let sukces = true;
      if (sukces) {
        resolve("Dane zostały pobrane.");
      } else {
        reject("Błąd podczas pobierania danych.");
      }
    }, 2000);
  });
}

pobierzDane()
  .then(wiadomosc => {
    console.log(wiadomosc);
  })
  .catch(błąd => {
    console.error(błąd);
  });
```

2.2 Async/Await

Async/await to składnia ułatwiająca pracę z promisy, wprowadzona w ES2017. Pozwala na pisanie kodu asynchronicznego w sposób przypominający kod synchroniczny.

2.2.1 Funkcje asynchroniczne

- Deklarowane za pomocą słowa kluczowego `async`.
- Mogą używać słowa kluczowego `await` wewnątrz swojego ciała.

Przykład:

```
async function przykladAsync() {
  let wynik = await obietnica;
  console.log(wynik);
}
```

2.2.2 Używanie `await`

- `await` zatrzymuje wykonywanie funkcji asynchronicznej do momentu spełnienia promisa.
- Może być używane tylko wewnątrz funkcji oznaczonej jako `async`.

Przykład z funkcją `pobierzDane` z wcześniejszego przykładu:

```
async function pobierzIWyswietlDane() {
  try {
    let wiadomosc = await pobierzDane();
    console.log(wiadomosc);
  } catch (błąd) {
    console.error(błąd);
  }
}

pobierzIWyswietlDane();
```

2.2.3 Obsługa błędów

- Błędy można obsługiwać za pomocą bloku `try...catch`.

Przykład:

```
async function przykladAsync() {
  try {
    let wynik = await obietnica;
    console.log(wynik);
  } catch (błąd) {
    console.error(błąd);
  }
}
```

2.3 Korzyści z użycia Async/Await

- **Czytelność**: Kod jest bardziej czytelny i łatwiejszy do zrozumienia.
- **Obsługa błędów**: Łatwiejsza obsługa błędów za pomocą `try...catch`.
- **Sekwencyjne wykonanie**: Ułatwia sekwencyjne wykonywanie operacji asynchronicznych.

3. Wprowadzenie do modułów JavaScript

Moduły pozwalają na podzielenie kodu na oddzielne pliki i importowanie ich w razie potrzeby. Ułatwiają organizację kodu i ponowne wykorzystanie funkcji, klas i zmiennych.

3.1 Eksportowanie i importowanie

3.1.1 Eksportowanie

- **Eksport pojedynczy (domyślny)**:

```
// plik moduł.js
export default function powitanie() {
  console.log("Witaj!");
}
```

- **Eksport nazwany**:

```
// plik moduł.js
export function suma(a, b) {
  return a + b;
}

export const PI = 3.1415;
```

3.1.2 Importowanie

- **Import eksportu domyślnego**:

```
// plik główny.js
import powitanie from './moduł.js';

powitanie(); // Witaj!
```

- **Import eksportu nazwanego**:

```
import { suma, PI } from './moduł.js';

console.log(suma(2, 3)); // 5
console.log(PI); // 3.1415
```

- **Import wszystkiego**:

```
import * as narzedzia from './moduł.js';

console.log(narzedzia.suma(2, 3)); // 5
console.log(narzedzia.PI); // 3.1415
```

3.2 Korzyści z użycia modułów

- **Organizacja kodu**: Ułatwiają zarządzanie dużymi projektami.
- **Ponowne wykorzystanie**: Funkcje i klasy mogą być łatwo ponownie używane w różnych częściach aplikacji.
- **Enkapsulacja**: Zapobiegają zanieczyszczeniu globalnego zakresu.

3.3 Użycie modułów w przeglądarce

- **Nowoczesne przeglądarki** obsługują moduły ES6 bezpośrednio za pomocą atrybutu type="module" w tagu <script>.

  ```
  <script type="module" src="główny.js"></script>
  ```

- **Ograniczenia**:
  - Pliki modułów muszą być ładowane z serwera (nie działają z protokołem file://).
  - Importy muszą używać pełnych ścieżek z rozszerzeniami.

3.4 Bundlery

- **Bundlery** (np. Webpack, Rollup) pozwalają na użycie modułów w starszych przeglądarkach oraz łączenie wielu plików w jeden.

---

4. Podstawy TypeScript

TypeScript to nadzbiór JavaScript opracowany przez Microsoft, który wprowadza statyczne typowanie i nowoczesne funkcje języka. Kod TypeScript jest kompilowany do JavaScript, co pozwala na jego użycie w dowolnej przeglądarce lub środowisku.

4.1 Czym jest TypeScript

- **Statyczne typowanie**: Możliwość definiowania typów zmiennych, parametrów i zwracanych wartości.
- **Nowoczesne funkcje**: Wprowadza dodatkowe funkcje nieobecne w czystym JavaScript.
- **Poprawa jakości kodu**: Dzięki typom, błędy mogą być wykrywane w czasie kompilacji.

4.2 Instalacja TypeScript

- **Instalacja przez npm**:

  ```
  npm install -g typescript
  ```

- **Sprawdzenie wersji**:

  ```
  tsc -v
  ```

4.3 Kompilacja plików TypeScript

- **Kompilacja pojedynczego pliku**:

  tsc plik.ts

- **Kompilacja z plikiem konfiguracyjnym tsconfig.json**:

  tsc

4.4 Podstawowe funkcje TypeScript

4.4.1 Typy podstawowe

- **Typowanie zmiennych**:

  ```
  let imie: string = "Jan";
  let wiek: number = 30;
  let aktywny: boolean = true;
  ```

- **Tablice**:

  ```
  let liczby: number[] = [1, 2, 3];
  ```

4.4.2 Interfejsy

Pozwalają na definiowanie struktury obiektów.

```
interface Osoba {
  imie: string;
  wiek: number;
}

let osoba: Osoba = {
  imie: "Anna",
  wiek: 25
};
```

4.4.3 Klasy

TypeScript rozszerza składnię klas znaną z ES6.

```
class Zwierze {
  private gatunek: string;

  constructor(gatunek: string) {
    this.gatunek = gatunek;
  }

  public przedstawSie(): void {
    console.log(`Jestem ${this.gatunek}.`);
  }
```

```
}

let pies = new Zwierze("pies");
pies.przedstawSie(); // Jestem pies.
```

4.4.4 Funkcje z typami

```
function suma(a: number, b: number): number {
  return a + b;
}

let wynik = suma(5, 3);
```

4.5 Korzyści z użycia TypeScript

- **Bezpieczeństwo typów**: Pomaga wykryć błędy przed uruchomieniem kodu.
- **Czytelność i dokumentacja**: Typy służą jako dodatkowa dokumentacja kodu.
- **Nowoczesne funkcje**: Dostęp do funkcji języka przed ich implementacją w przeglądarkach.

---

5. Narzędzia do kontroli jakości kodu

Utrzymanie wysokiej jakości kodu jest kluczowe w każdym projekcie. Narzędzia takie jak ESLint i Prettier pomagają w automatycznym wykrywaniu błędów oraz formatowaniu kodu zgodnie z ustalonymi standardami.

5.1 ESLint

5.1.1 Czym jest ESLint

- **ESLint** to narzędzie do analizy statycznej kodu JavaScript/TypeScript.
- Pomaga wykrywać błędy, potencjalne problemy oraz niezgodności ze standardami kodowania.

5.1.2 Instalacja ESLint

- **Instalacja lokalna w projekcie**:

  ```
  npm install eslint --save-dev
  ```

5.1.3 Inicjalizacja konfiguracji

- **Tworzenie pliku konfiguracyjnego**:

  ```
  npx eslint --init
  ```

5.1.4 Użycie ESLint

- **Analiza plików**:

  npx eslint nazwa_pliku.js

- **Automatyczne naprawianie błędów**:

  npx eslint nazwa_pliku.js --fix

5.2 Prettier

5.2.1 Czym jest Prettier

- **Prettier** to narzędzie do formatowania kodu.
- Zapewnia jednolity styl kodowania, automatycznie formatując kod zgodnie z ustalonymi regułami.

5.2.2 Instalacja Prettier

- **Instalacja lokalna w projekcie**:

  npm install prettier --save-dev

5.2.3 Użycie Prettier

- **Formatowanie plików**:

  npx prettier --write nazwa_pliku.js

5.2.4 Integracja z ESLint

- **Instalacja wtyczki**:

  npm install eslint-config-prettier eslint-plugin-prettier --save-dev

- **Konfiguracja ESLint**:

  Dodaj w pliku .eslintrc:

```
{
  "extends": ["plugin:prettier/recommended"]
}
```

5.3 Integracja z edytorami kodu

Większość nowoczesnych edytorów (np. Visual Studio Code) posiada rozszerzenia dla ESLint i Prettier, umożliwiając automatyczną analizę i formatowanie kodu podczas pisania.