# Javascript od pierwszej linijki

*Naucz się jak pisać gry, strony WWW i aplikacje internetowe!*

***Peter Ringel***

## Spis treści

# Wprowadzenie do JavaScript

## Historia języka i jego znaczenie

JavaScript narodził się w bardzo ciekawym momencie rozwoju internetu. W 1995 roku programista Brendan Eich, pracujący dla firmy Netscape, otrzymał niezwykle ambitne zadanie - stworzenie języka programowania, który działałby bezpośrednio w przeglądarce internetowej. W ciągu zaledwie 10 dni stworzył pierwszą wersję języka, początkowo nazwanego LiveScript. Był to moment przełomowy, ponieważ strony internetowe przestały być tylko statycznymi dokumentami. Gdy firma Netscape nawiązała współpracę z firmą Sun Microsystems (twórcą popularnego wówczas języka Java), LiveScript zmienił nazwę na JavaScript - był to sprytny zabieg marketingowy, choć oba języki nie miały ze sobą wiele wspólnego. W 1997 roku, aby uporządkować rozwój języka, powstał standard ECMAScript. JavaScript stał się jego najbardziej znaną implementacją, a organizacja ECMA International zajęła się dalszym rozwojem specyfikacji.

Początkowo JavaScript był niedoceniany przez programistów. Służył głównie do prostych zadań, takich jak walidacja formularzy czy tworzenie efektów wizualnych na stronach. Jednak z biegiem lat jego możliwości znacząco wzrosły. Prawdziwa rewolucja nastąpiła w 2009 roku wraz z powstaniem Node.js - środowiska pozwalającego uruchamiać JavaScript poza przeglądarką. Nagle okazało się, że ten sam język może służyć do tworzenia zarówno interfejsu użytkownika, jak i logiki serwerowej. Rozwój technologii webowych sprawił, że pojawiły się potężne frameworki frontendowe, umożliwiające tworzenie zaawansowanych aplikacji działających w przeglądarce. JavaScript przestał być tylko dodatkiem do HTML-a, a stał się pełnoprawną platformą programistyczną.

Dziś JavaScript jest jednym z najpopularniejszych języków programowania na świecie. Według indeksu TIOBE regularnie znajduje się w pierwszej dziesiątce, a na GitHubie jest językiem z największą liczbą repozytoriów. Jego wszechstronność jest imponująca - służy do tworzenia interfejsów użytkownika, aplikacji serwerowych, aplikacji mobilnych (poprzez frameworki hybrydowe), a nawet aplikacji desktopowych. W zasadzie każda nowoczesna strona internetowa wykorzystuje JavaScript, a znajomość tego języka jest jednym z podstawowych wymagań na stanowiskach związanych z tworzeniem oprogramowania.

## Gdzie można używać JavaScriptu

JavaScript to wyjątkowy język, który może działać w różnych środowiskach wykonawczych. Najpopularniejszym z nich jest oczywiście przeglądarka internetowa - każda nowoczesna przeglądarka posiada wbudowany silnik JavaScript, który interpretuje i wykonuje kod. Jednym z najważniejszych silników jest V8, stworzony przez Google, który napędza nie tylko przeglądarkę Chrome, ale stał się również podstawą Node.js. Node.js to środowisko, które pozwala uruchamiać JavaScript poza przeglądarką, bezpośrednio na komputerze czy serwerze. To właśnie dzięki Node.js JavaScript wyrwał się z ram przeglądarki i może być używany praktycznie wszędzie.

Różnorodność środowisk wykonawczych przekłada się na szerokie spektrum aplikacji, jakie możemy tworzyć w JavaScript. W przeglądarce możemy budować zarówno proste strony internetowe wzbogacone o interaktywne elementy, jak i złożone aplikacje webowe działające jak programy desktopowe. Po stronie serwera, dzięki Node.js, możemy tworzyć wydajne API RESTowe obsługujące tysiące zapytań. JavaScript świetnie sprawdza się też w tworzeniu aplikacji desktopowych - technologia Electron pozwala pakować aplikacje webowe w samodzielne programy działające na komputerze. W świecie mobile, dzięki technologiom jak React Native, możemy tworzyć natywne aplikacje mobilne używając znajomej składni JavaScriptu.

Do tworzenia aplikacji w JavaScript mamy do dyspozycji bogaty zestaw narzędzi programistycznych. Podstawowym narzędziem jest edytor kodu - od prostych jak Visual Studio Code po zaawansowane środowiska IDE jak WebStorm. Nieocenioną pomocą są narzędzia deweloperskie wbudowane w przeglądarki internetowe - pozwalają one na debugowanie kodu, analizę wydajności, podgląd zmiennych i struktury DOM. Console w narzędziach deweloperskich to pierwsze miejsce, gdzie zwykle testujemy nasz kod i sprawdzamy jego działanie. Większość edytorów oferuje kolorowanie składni, autouzupełnianie i podpowiedzi, co znacząco ułatwia pisanie kodu JavaScript.

```
// Przykład kodu, który możemy przetestować w konsoli przeglądarki
console.log("Hello World!");
// Prosty przykład interakcji z DOM-em
document.querySelector('button').addEventListener('click', () => {
```

```
    alert('Przycisk został kliknięty!');
});
// Przykład kodu Node.js
const http = require('http');
const server = http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello World!');
});
server.listen(8080);
```

## Pierwsze "Hello World"

Zacznijmy od stworzenia najprostszej strony internetowej z JavaScript. Kod JavaScript możemy umieścić w pliku HTML na kilka sposobów. Najpopularniejszym jest użycie znacznika script w sekcji head lub na końcu body dokumentu. Możemy też napisać kod JavaScript bezpośrednio między znacznikami script lub wskazać zewnętrzny plik .js. Oto przykład prostej strony z osadzonym kodem:

```
<!DOCTYPE html>
<html>
<head>
    <title>Moja pierwsza strona z JavaScript</title>
    <!-- JavaScript w sekcji head -->
    <script>
        // Tu możemy pisać kod JS
    </script>
</head>
<body>
    <h1>Hello World!</h1>
    <!-- JavaScript na końcu body (zalecane) -->
    <script src="moj-skrypt.js"></script>
</body>
</html>
```

JavaScript oferuje kilka podstawowych sposobów wyświetlania danych. Najprostszym jest użycie funkcji console.log(), która wyświetla informacje w konsoli przeglądarki - jest to podstawowe narzędzie podczas nauki i debugowania kodu. Funkcja alert() wyświetla okienko z komunikatem, które użytkownik musi zaakceptować. Z kolei document.write() pozwala wpisać treść bezpośrednio do dokumentu HTML. Oto przykłady użycia każdej z tych metod:

```
// Wyświetlenie w konsoli
console.log("Hello World w konsoli!");
// Wyświetlenie alertu
alert("Hello World w alercie!");
// Wyświetlenie na stronie
document.write("Hello World na stronie!");
// Możemy też wyświetlać liczby i wyrażenia
console.log(2 + 2); // wyświetli 4
console.log("Wynik:", 2 + 2); // wyświetli "Wynik: 4"
```

Pisząc kod JavaScript, warto od początku przywyknąć do pewnych zasad. Instrukcje kończymy średnikiem (;) - choć JavaScript często wybaczy nam jego brak, to jest to dobra praktyka. Kod warto formatować używając wcięć (najczęściej 2 lub 4 spacje) dla lepszej czytelności. Komentarze są niezwykle ważne - pomagają nam i innym zrozumieć kod. Możemy używać komentarzy jednoliniowych rozpoczynających się od // lub wieloliniowych między /* a */. Przykład:

```
// To jest komentarz jednoliniowy
console.log("Hello"); // To też jest komentarz jednoliniowy
/* To jest komentarz wieloliniowy, który może zajmować wiele linii */
/* Przykład wcięć w kodzie */
if (true) {
    console.log("Pierwsza linia");
    console.log("Druga linia");
}
```

## Konsola przeglądarki i podstawy debugowania

Zacznijmy od tego, jak dotrzeć do konsoli przeglądarki. W większości nowoczesnych przeglądarek możemy ją otworzyć naciskając F12 lub kombinację Ctrl+Shift+I (Cmd+Option+I na MacOS). Możemy też kliknąć prawym przyciskiem myszy na stronie i wybrać "Zbadaj element" lub "Inspect", a następnie przejść do zakładki "Console". Konsola oferuje różne typy komunikatów, które różnią się wizualnie i ważnością: zwykłe logi (console.log), informacje (console.info), ostrzeżenia (console.warn) i błędy (console.error). Każdy z nich ma inne przeznaczenie i jest oznaczony inną ikoną, co pomaga w szybkiej identyfikacji typu komunikatu.

Konsola JavaScript oferuje wiele przydatnych metod, które ułatwiają debugowanie kodu. Podstawowa metoda console.log() wyświetla przekazane wartości w najprostszej formie. Metoda console.table() jest świetna do wyświetlania danych tabelarycznych - automatycznie formatuje tablice i obiekty w czytelną tabelę. Console.group() i console.groupEnd() pozwalają grupować powiązane komunikaty, co jest szczególnie przydatne przy debugowaniu większych fragmentów kodu. Metody console.time() i console.timeEnd() służą do mierzenia czasu wykonania kodu. Oto przykłady:

```
// Różne typy komunikatów
console.log("Zwykły komunikat");
console.info("Informacja");
console.warn("Ostrzeżenie!");
console.error("Błąd!");
// Wyświetlanie danych w tabeli
const users = [
  {name: "Jan", age: 25},
  {name: "Anna", age: 30}
];
console.table(users);
// Grupowanie komunikatów
console.group("Testowanie funkcji");
console.log("Test 1 passed");
console.log("Test 2 passed");
console.groupEnd();
```

```
// Mierzenie czasu
console.time("Pętla");
for (let i = 0; i < 1000000; i++) {}
console.timeEnd("Pętla");
```

Kiedy w naszym kodzie występuje błąd, konsola jest pierwszym miejscem, gdzie powinniśmy szukać informacji o problemie. Błędy w JavaScript zawierają nazwę błędu (np. SyntaxError, ReferenceError, TypeError) oraz opis problemu. Stack trace (stos wywołań) pokazuje dokładną ścieżkę, która doprowadziła do błędu - od miejsca jego wystąpienia, przez wszystkie funkcje, które były po drodze wywołane. Najczęstsze błędy składniowe to brakujące nawiasy, przecinki czy średniki, literówki w nazwach zmiennych lub próba użycia zmiennej, która nie została zadeklarowana. Przykładowo:

```
// Przykład błędu składniowego
function test() {
  console.log("Start");
  undefinedVariable; // ReferenceError: undefinedVariable is not
defined
  console.log("Koniec");
}
// Przykład błędu typu
const number = 42;
number.toUppercase(); // TypeError: number.toUppercase is not a
function
// Przykład błędu składni
if (true { // SyntaxError: missing ) after condition
  console.log("Błąd!");
}
```

## Dodawanie skryptów do strony HTML

Umieszczenie skryptu JavaScript w dokumencie HTML ma kluczowe znaczenie dla działania naszej strony. Możemy umieścić skrypt w sekcji head lub na końcu body - każde rozwiązanie ma swoje zalety. Skrypty w

head są ładowane wcześniej, ale mogą opóźnić renderowanie strony. Skrypty na końcu body pozwalają stronie załadować się szybciej, ale kod wykona się później. Atrybuty async i defer dają nam dodatkową kontrolę: async powoduje asynchroniczne ładowanie skryptu, a defer odkłada wykonanie do momentu załadowania całego HTML-a. Oto przykłady różnych sposobów dodawania skryptów:

```
<!DOCTYPE html>
<html>
<head>
  <!-- Standardowy skrypt w head -->
  <script src="pierwszy.js"></script>
  <!-- Skrypt asynchroniczny -->
  <script async src="drugi.js"></script>
  <!-- Skrypt z defer -->
  <script defer src="trzeci.js"></script>
</head>
<body>
  <!-- Zawartość strony -->
  <!-- Skrypt na końcu body -->
  <script src="czwarty.js"></script>
</body>
</html>
```

Przechowywanie kodu JavaScript w zewnętrznych plikach .js ma wiele zalet. Przede wszystkim poprawia to czytelność i utrzymanie kodu - łatwiej znaleźć i poprawić błędy w osobnych plikach. Dodatkowo, przeglądarka może cachować zewnętrzne pliki JS, co przyspieszy ładowanie strony przy kolejnych wizytach. Podczas linkowania plików możemy używać ścieżek względnych (relatywnych do aktualnego pliku HTML) lub bezwzględnych (pełny URL). Przykład struktury projektu i linkowania:

```
<!-- Ścieżka względna do pliku w tym samym folderze -->
<script src="skrypt.js"></script>
<!-- Ścieżka względna do pliku w podfolderze -->
<script src="js/skrypt.js"></script>
<!-- Ścieżka względna poziom wyżej -->
<script src="../skrypt.js"></script>
```

```
<!-- Ścieżka bezwzględna -->
<script src="https://mojadomena.pl/skrypt.js"></script>
```

Przy organizacji kodu JavaScript warto kierować się kilkoma zasadami. Dla małych projektów jeden plik może wystarczyć, ale w większych projektach lepiej podzielić kod na mniejsze, tematyczne pliki. Ważna jest kolejność ładowania - jeśli jeden skrypt zależy od drugiego (np. używa jego funkcji), musi być załadowany po nim. Dobrą praktyką jest umieszczanie bibliotek zewnętrznych przed własnym kodem. Przykładowa struktura:

```
<!-- Najpierw biblioteki zewnętrzne -->
<script src="vendor/jquery.js"></script>
<script src="vendor/bootstrap.js"></script>
<!-- Potem własne skrypty, w odpowiedniej kolejności -->
<script src="js/utilities.js"></script>
<script src="js/main.js"></script>
<script src="js/app.js"></script>
```

Gdy skrypty nie ładują się poprawnie, pierwszym krokiem powinno być sprawdzenie konsoli przeglądarki. Błąd 404 oznacza, że plik nie został znaleziony - warto wtedy sprawdzić czy ścieżka jest poprawna i czy plik faktycznie istnieje w podanej lokalizacji. Jeśli skrypt się ładuje, ale nie działa poprawnie, może to oznaczać problem z kolejnością ładowania - na przykład próbujemy użyć funkcji z biblioteki, która jeszcze się nie załadowała. Pomocne jest sprawdzenie zakładki Network w narzędziach deweloperskich, która pokazuje kolejność i status ładowania wszystkich plików:

```
<!-- Przykład częstego błędu - próba użycia jQuery przed jego
załadowaniem -->
<script>
  // To nie zadziała, bo jQuery jeszcze nie jest załadowane
  $(document).ready(function() {
    console.log("Strona gotowa!");
  });
</script>
<script src="jquery.js">
<!-- Poprawna kolejność -->
```

```
<script src="jquery.js"></script>
<script>
  // Teraz zadziała poprawnie
  $(document).ready(function() {
    console.log("Strona gotowa!");
  });
</script>
```

## Zmienne i typy danych

## let, const i var