Skrypty dokonujące obliczeń, parsowanie liczb

Zadania związane z przetwarzaniem liczb w skryptach JS to absolutny klasyk na egzaminach INF.03. Na pierwszy rzut oka mogą wydawać się one stosunkowo łatwe do zrealizowania, ale czy na pewno tak jest? Jeśli kiedykolwiek wpisaliśmy liczbę w pole formularza i zauważyliśmy później dziwne wyniki działań matematycznych w przeglądarce, to już wiemy iż przeprowadzenie poprawnych obliczeń to nie zawsze w JS tak trywialna sprawa.

Problem polega na tym, że dane wczytane z pól edycyjnych, np.

`<input type="number">` są przechowywane w pamięci jako łańcuchy znaków (stringi), a nie liczby. Jeśli nie zadbamy o odpowiednie przekonwertowanie (parsowanie) tych wartości, to nasz skrypt może zwrócić nieoczekiwane (niepoprawne) rezultaty różnorodnych obliczeń.

Problem poprawnego typu danych

Zobaczmy klasyczny przykładowy skrypt, który pozornie powinien zadziałać poprawnie, lecz tak się niestety nie dzieje. W dokumencie HTML zdefiniowano dwa pola numeryczne oraz przycisk:

```
<input type="number" id="a">
<input type="number" id="b">
<input type="button" value="Dodawanie" onclick="suma()">
```

Naszym celem jest zwyczajnie dodać do siebie dwie wprowadzone w polach liczby i pokazać wynik dodawania w wyskakującym okienku typu alert. Skrypt realizujący dodawanie – poprawny składniowo! – zwróci niestety niepoprawny wynik dodawania:

```
function suma() {
    var a = document.getElementById("a").value;
    var b = document.getElementById("b").value;
    alert(a + b);
}
```

Załóżmy, że ktoś wpisze do obu pól wartości kolejno:

`5` oraz `7`. Oczekiwanym wynikiem dodawania będzie oczywiście suma równa `12`, jednak rzeczywisty wynik, który ujrzymy w przeglądarce to `57`. Dlaczego tak się stało?! Osoba, która po raz pierwszy doświadcza tego typu problemu, zazwyczaj odnajdzie siebie w stanie totalnego zdumienia.

Natomiast operator

`+` w JavaScript po prostu został – jak to mówimy w programowaniu – przeciążony, czyli w zależności od typu danych może realizować dwa różne działania – albo dodawanie liczb albo tzw. konkatenację (łączenie czy “klejenie”) tekstów. I otrzymana przez nas wartość `57` to faktycznie “sklejenie” `5` oraz `7` traktowanych jako napisy, a nie liczby! Warto zrozumieć tę różnicę, aby uniknąć na egzaminie przykrych rozczarowań. Reasumując:

- Jeśli chociaż jedna wartość jest napisem, to operator `+` zrealizuje łączenie (konkatenację) napisów.
- Jeśli obie wartości są liczbami, to operator `+` wykonuje dodawanie liczb.

I teraz sprawa najważniejsza – aby upewnić się, że JavaScript potraktuje nasze dane z pól formularza jako liczby, to musimy je odpowiednio przekonwertować. Mówimy wtedy o tzw. parsowaniu – czyli przekształceniu łańcucha tekstowego na typ liczbowy.

W JS mamy do dyspozycji dwie funkcje:

- `parseInt()` – konwertuje na liczbę całkowitą

- `parseFloat()` – konwertuje na liczbę zmiennoprzecinkową

```
function suma() {
    var a = document.getElementById("a").value;
    var b = document.getElementById("b").value;
    a = parseFloat(a);
    b = parseFloat(b);
    alert(a + b);
}
```

Co ważne – zabiegu parsowania oczywiście dokonujemy w odpowiednim momencie! A zatem dopiero wtedy, kiedy wartości zostały już w skrypcie wczytane z pól numerycznych, a zanim dokonało się dodawanie.

parseInt() czy parseFloat() – którą funkcję wybrać?

W zależności od kontekstu, wybieramy jedną z dwóch podstawowych funkcji konwertujących – czy interesują nas tylko poprawne liczby całkowite czy też dozwolone są wartości zmiennoprzecinkowe? Zobaczmy teraz kilka przykładów rezultatów parsowania!

Funkcja

`parseInt()` konwertuje tekst na liczbę całkowitą, ignorując wszystko po pierwszym znaku, który nie przypomina cyfry (oprócz znaku minus).

```
console.log(parseInt("123"));    // 123
console.log(parseInt(" 45 "));   // 45 (pomija białe znaki na początku i końcu)
console.log(parseInt("20px"));   // 20 (przerywa przy znaku niebędącym cyfrą)
console.log(parseInt("34gg6"));  // 34 (odczytuje tylko początkowe cyfry)
console.log(parseInt("gg34"));   // NaN (nie zaczyna się od cyfry)
console.log(parseInt("0045"));   // 45 (ignoruje zera wiodące)
console.log(parseInt("-78.99")); // -78 (część dziesiętna jest pomijana)
```

Funkcja

`parseFloat()` odczytuje liczby zmiennoprzecinkowe (z przecinkiem dziesiętnym) i działa bardzo podobnie, ale obsługuje również części ułamkowe.

```
console.log(parseFloat("123.45"));    // 123.45
console.log(parseFloat(" 3.14abc"));  // 3.14
console.log(parseFloat("20.5px"));    // 20.5
console.log(parseFloat("34.2gg6"));   // 34.2
console.log(parseFloat("abc123.45")); // NaN (nie zaczyna się od cyfry)
console.log(parseFloat("0.0001"));    // 0.0001
console.log(parseFloat("-98.76abc")); // -98.76
```

Podsumujmy nieco działanie parsowania:

- Jeśli tekst zaczyna się od litery, znaku specjalnego (poza znakiem minusa, który może rozpoczynać poprawną liczbę) lub spacji bez cyfry po nich – zwracana jest wartość `NaN` (*ang. Not a Number*). Taka wartość pozwala następnie użyć funkcji `isNaN()`, która zwróci prawdę jeśli nie udało się poprawnie skonwertować wartości do poprawnej liczby – przykład przedstawimy już za chwilę.

- Funkcja `parseInt()` ignoruje część ułamkową (po kropce), natomiast `parseFloat()` oczywiście poprawnie ją odczyta.

Funkcja isNaN()

Gdy pracujemy z danymi wprowadzanymi przez użytkownika, chcemy mieć absolutną pewność, że dana wartość jest prawidłową liczbą. W tym celu JavaScript udostępnia funkcję

`isNaN()`, która zwraca:

- `true` – jeśli wartość to `NaN` (*ang. Not a Number*),
- `false` – jeśli wartość to prawidłowa liczba (lub udało się ją na nią przekształcić).

Załóżmy, że chcemy przebudować nasz prosty kalkulator, w którym dodajemy dwie liczby wprowadzone w polach formularza – upewnijmy się dodatkowo, że wpisane dane to poprawne liczby:

```
function suma() {
    var a = document.getElementById("a").value;
    var b = document.getElementById("b").value;
    a = parseFloat(a);
    b = parseFloat(b);
    if (isNaN(a) || isNaN(b)) {
       alert("Błąd: Proszę wprowadzić poprawne liczby!");
       return; // przerwanie dalszego działania funkcji
    }
    alert(a + b);
}
```

Bez tego dodatkowego sprawdzenia moglibyśmy próbować wykonywać operacje matematyczne na błędnych danych, co prowadzić może do dziwnych wyników, np.

`NaN` + 5 = `NaN`. Dzięki `isNaN()` zyskaliśmy prosty mechanizm walidacji (sprawdzania poprawności) danych jeszcze przed ich przetwarzaniem. Co prawda pola numeryczne utrudniają użytkownikowi wpisanie niepoprawnych liczb, lecz absolutnie nic nie gwarantują.

Na egzaminie zawodowym INF.03 może pojawić się zadanie, w którym mamy pobrać wartości liczb z pól formularza i coś obliczyć – np. pole figury geometrycznej, średnią ocen, konwersję jednostek, procent udzielonego rabatu itp. Jeśli nie zabezpieczymy się przed błędnym wpisem (np. łańcuch zamiast liczby), nasz kod może działać błędnie. Walidacja z użyciem

`isNaN()` to świetny sposób na profesjonalne podejście do tematu i pokazanie egzaminatorowi, że rozumiemy podstawy pracy z danymi wejściowymi oraz praktyczne znaczenie typów danych. Zrealizujmy takie sprawdzenie, nawet jeśli arkusz tego nie wymaga.

Alternatywna konwersja: Number()

Jeśli zależy nam na precyzyjnej i jednoznacznej konwersji danych, to możemy zamiast parsowania użyć funkcji

`Number()`. Jest ona bardziej rygorystyczna i nie akceptuje żadnych znaków niebędących liczbami. W przeciwieństwie do funkcji `parseInt()` oraz `parseFloat()`, które odczytują liczby “od lewej” dopóki napotykają cyfry – `Number()` próbuje przekonwertować wartość od razu i w całości. Jeśli tekst zawiera coś, co nie jest poprawną liczbą, to cała konwersja się nie powiedzie i zwrócony zostanie `NaN`:

```
console.log(Number("20px"));    // NaN
console.log(Number("34.6abc")); // NaN
console.log(Number("123"));     // 123
console.log(Number("3.14"));    // 3.14
console.log(Number(""));        // 0
```

Automatyczna konwersja z + (tzw. unary plus)

Istnieje także skrótowy sposób na przekształcenie stringa na liczbę dzięki tzw. operatorowi unary plus (*ang. plus jednoargumentowy*) – wystarczy dodać znak

`+` przed wartością. To szybka metoda, lecz wymaga pewności, że dane nie zawierają żadnych znaków innych niż cyfry – w naszym kalkulatorze wyglądałoby to tak:

```
var a = +document.getElementById("a").value;
var b = +document.getElementById("b").value;
alert(a + b);
```

Jak dokładnie działa unary plus? Otóż tak naprawdę wewnętrznie wykorzystuje funkcję

`Number()`. A więc oznacza to, iż konwertuje całą wartość (czyli nie "parsuje" od lewej strony, tylko analizuje cały łańcuch znaków) – i jeśli choćby jeden fragment nie pasuje do poprawnego zapisu liczby, zwraca `NaN:`

- `"5"` → `5`
- `"3.14"` → `3.14`
- `"10px"` → `NaN`

Podsumowanie – co warto zapamiętać?

- Wartości z pól typu `<input type="number">` są tekstami (stringami), a nie liczbami – to może prowadzić do błędów w obliczeniach matematycznych.
- Przeciążony operator `+` w JavaScript wykonuje różne działania w zależności od typu danych: dodawanie liczb lub łączenie tekstów (konkatenację).
- Aby zapewnić poprawne działania matematyczne, musimy wcześniej przekształcić tekst na liczbę – proces ten nazywamy parsowaniem (parsowanie danych wejściowych).
- Do parsowania w JavaScript służą funkcje `parseInt()` (dla liczb całkowitych) oraz `parseFloat()` (dla liczb zmiennoprzecinkowych).
- `parseInt()` odczytuje liczby do momentu napotkania pierwszego niedozwolonego znaku – pomija zera wiodące i część dziesiętną.
- `parseFloat()` działa podobnie, ale potrafi odczytywać także wartości zmiennoprzecinkowe.
- Jeśli ciąg znaków nie zaczyna się od cyfry (lub minusa), to obie funkcje zwrócą `NaN` – czyli *Not a Number.*
- Do sprawdzenia czy konwersja się powiodła, używamy funkcji `isNaN()`, która pozwala walidować dane wejściowe przed przystąpieniem do obliczeń.
- Dodanie walidacji z użyciem `isNaN()` to dobre podejście i pokazuje egzaminatorowi zrozumienie problematyki typów danych.

- Alternatywą dla `parseInt()` oraz `parseFloat()` jest funkcja `Number()`, która konwertuje tekst do liczby w sposób rygorystyczny – nie toleruje żadnych dodatkowych znaków.
- Najkrótszym sposobem konwersji jest tzw. operator unary plus, czyli dodanie `+` przed tekstem – wewnętrznie działa tak samo jak `Number()`.
- Warto stosować konwersję danych przed wykonywaniem działań matematycznych, aby uniknąć błędów i niepoprawnych wyników (np. `5` + `7` = `57`).
- Na egzaminie INF.03 bardzo często pojawiają się zadania dokonania obliczeń na podstawie wprowadzonych danych wejściowych – konwersja i walidacja to klucz do poprawnych wyników.