Skrypty przetwarzające napisy

Na egzaminie INF.03 (w kontekście JS) zazwyczaj otrzymamy zadanie opracowania skryptu przetwarzającego liczby, zajmującego się napisami albo wpływającego na stylowanie elementów zdefiniowanych w HTML. Przyjrzymy się teraz drugiemu rodzajowi zadań, czyli przetwarzaniu łańcuchów znaków. Jak dobrze wiemy, w programowaniu łańcuch (*ang. string*) to po prostu tekst składający się z pojedynczych znaków. Nazwa tego typu danych wzięła się od prostej metafory – znaki w napisach przypominają ogniwa w łańcuchu. Warto więc poznać zarówno zasady pracy z napisami, jak i podstawowy zestaw gotowych właściwości i metod, który niewątpliwe przyda się na egzaminie (oszczędzi nam masę czasu).

Oprócz porównania z łańcuchem składającym się z ogniw, możemy też ideowo traktować łańcuch jako tablicę znakową, w której każda litera (lub inny znak) posiada swój numer porządkowy (nazywamy go w tablicach *indeksem*), który w wielu językach pozwala nam precyzyjnie uzyskać dostęp do poszczególnych szuflad tablicy. Spójrzmy na poniższy przykład:

```
let napis = "JavaScript";
```

J	a	v	a	S	c	r	i	p	t
0	1	2	3	4	5	6	7	8	9

Podczas pracy z tekstem w JavaScript bardzo często musimy pobierać, analizować, porównywać czy też łączyć ze sobą (mówimy ładnie: konkatenować) fragmenty różnych napisów. Wiele osób – zwłaszcza przyzwyczajonych do języków takich jak *C++* czy *Python* – może odruchowo sięgnąć w tym miejscu po nawiasy kwadratowe, aby “dostać się” do konkretnej litery w napisie. Przykładowo:

```
let napis = "JavaScript";
alert(napis[2]); // wypisze literę "v"
```

I faktycznie – taki zapis bez problemu w JavaScript zadziała. Użycie indeksu wraz z nawiasami kwadratowymi pozwala nam **odczytać** konkretny znak z danego miejsca w łańcuchu. Jednak inaczej w JS sprawa ma się z **zapisem** (modyfikacją) znaku.

Czy możemy zmienić literę w łańcuchu?

I tutaj dochodzimy do bardzo ważnej kwestii. W JavaScript **łańcuchy znaków są niemodyfikowalne** (*ang. immutable*). Oznacza to, że nie możemy zmieniać ich zawartości bezpośrednio – stąd taki przykładowy zapis, zamieniający “JavaScript” na “JavaSkrypt” będzie niepoprawny:

```
let napis = "JavaScript";
napis[5] = "k"; // To nie zadziała!
napis[7] = "y"; // To nie zadziała!
alert(napis);
```

Ten kod wykona się co prawda bez błędu składniowego, lecz modyfikacja zwyczajnie nie zajdzie – litery nie zostaną zmienione. Dlaczego? Bo modyfikacja konkretnego znaku w istniejącym łańcuchu nie jest możliwa – możemy jedynie stworzyć **nową kopię** łańcucha z żądaną zmianą.

Jak zatem poprawnie “zmienić” litery w napisie?

Skoro nie możemy modyfikować konkretnych znaków, tworzymy nowy łańcuch z interesującymi nas modyfikacjami:

```
let napis = "JavaScript";
let nowyNapis = "";
for (var i = 0; i < napis.length; i++) {
```

```
    if (i == 5) nowyNapis += "k";
    else if (i == 7) nowyNapis += "y";
    else nowyNapis += napis[i];
}
alert(nowyNapis); // wypisze: JavaSkrypt
```

Widzimy tutaj bardzo ważną mechanikę – zamiast modyfikacji znaków doklejamy zmienione wartości operatorem

`+=`. Zauważyliśmy też zapewne automatyczne pobranie długości łańcucha w nagłówku pętli: `napis.length` – tę właściwość omówimy już w następnym rozdziale tej lekcji.

Natomiast koniecznie pamiętajmy o tym istniejącym zakazie modyfikacji bezpośredniej łańcuchów w JS na egzaminie zawodowym, bo inaczej nie będziemy wiedzieli dlaczego nasz skrypt nie działa poprawnie! Operator

`+=` zawsze w razie potrzeby posłuży nam jako sposób na stworzenie nowego tekstu poprzez dołączenie czy modyfikację fragmentów oryginału.

Oczywiście oprócz stosowania nawiasów kwadratowych, istnieje wiele metod pracy z łańcuchami w JavaScript – przedstawmy teraz katalog użytecznych właściwości oraz funkcji, które po prostu warto do egzaminu znać!

Sprawdzenie długości łańcucha – właściwość length

W pracy z tekstami bardzo często musimy poznać **liczbę znaków** zawartych w jakimś napisie (długość tego łańcucha). Na przykład: ile liter ma imię wpisane przez użytkownika? Jak długa jest wprowadzona wiadomość? Czy wpisane hasło posiada przynajmniej 8 znaków? Do wszystkich takich sytuacji używamy niezwykle ważnej właściwości o nazwie

`length`.

Co ważne – nie jest to funkcja, lecz **właściwość** (czyli cecha obiektu), dostępna dla każdego łańcucha znaków (stringa). Wystarczy, że dopiszemy

`.length` do zdefiniowanego napisu, a JavaScript zwróci nam jego długość jako liczbę całkowitą:

```
let imie = "Damian";
alert(imie.length); // wypisze: 6
```

Oczywiście właściwości tej (jako, iż zwraca do skryptu liczbę całkowitą) możemy także używać w instrukcjach warunkowych – na przykład chcemy sprawdzić czy hasło ma przynajmniej osiem znaków:

```
let haslo = "qwerty";
if (haslo.length < 8) {
    alert("Hasło musi posiadać przynajmniej 8 znaków!");
} else {
    alert("Hasło jest wystarczająco długie!");
}
```

Dzięki

`length` możemy w bardzo prosty sposób także porównywać długości łańcuchów, co pozwala nam tworzyć intuicyjne warunki i komunikaty. Ponadto, jeśli chcemy “przejść się” po wszystkich znakach napisu w pętli – na przykład, by wypisać je jeden po drugim – właściwość `length` pozwoli nam dokładnie wiedzieć, ile wykonać iteracji.

Oczywiście warto też zdawać sobie sprawę, że w obliczaniu długości łańcucha liczy się każdy znak, a nie tylko litery – czyli również spacje, cyfry, znaki specjalne. Wydaje się to oczywiste, ale czasem niektórym osobom może się to wydawać nieintuicyjne.

Odczytywanie pojedynczego znaku – metoda charAt()

Jak wspominaliśmy, do pobrania konkretnego znaku z łańcucha możemy użyć nawiasów kwadratowych, ale jeśli chcemy pisać kod zgodny z klasycznymi zasadami i dobrymi praktykami, to powinniśmy do tego celu używać w JS dedykowanej metody o nazwie

`charAt()`. To właśnie ona została stworzona z myślą o czytaniu znaków z napisu – i była dostępna już we wczesnych wersjach języka JavaScript, zanim pojawiła się możliwość użycia składni z nawiasami!

Metoda

`charAt()` analogicznie jak zapis z nawiasami kwadratowymi – pozwala nam odczytać znak o określonym indeksie w łańcuchu znaków:

```
let napis = "JavaScript";
alert(napis.charAt(2)); // wypisze literę "v"
```

Warto wiedzieć, że metoda

`charAt()` to rozwiązanie bardziej przenośne i czytelne, szczególnie gdy chcemy zachować kompatybilność ze starszymi przeglądarkami oraz pisać kod zgodny z filozofią i dokumentacją języka JavaScript. Używając tej metody jasno sygnalizujemy, że pracujemy na tekście, a nie na tablicy. Wykorzystujmy zatem na egzaminie INF.03 właśnie tę metodę **odczytu** znaków (zapis nadal możliwy nie jest!), zamiast nawiasów kwadratowych.

Odczytanie kodu ASCII znaku – metoda charCodeAt()

Czasem zechcemy odczytać nie tyle sam znak w łańcuchu, ale dowiedzieć się jaki jest jego kod numeryczny w tablicy znaków ASCII. Taka operacja przyda się, kiedy na przykład spróbujemy porównać dane znaki po kolei z całym alfabetem (aby wyznaczyć liczbę wystąpień każdej litery w napisie), sprawdzić czy dany znak to duża czy mała literka albo zaszyfrować napis metodą Cezara (przesunięcie +/- kodów ASCII znaków).

Zobaczmy przykład użycia:

```
let napis = "ABC123";
for (let i = 0; i < napis.length; i++) {
    let znak = napis.charAt(i);
    let kod = napis.charCodeAt(i);
    alert(znak + "ma kod ASCII " + kod);
}
```

Taka pętla wypisze po kolei kod ASCII każdego znaku w łańcuchu przechowywanym w zmiennej

`napis`. Najczęściej stosowane w praktyce zakresy wartości znaków w tablicy ASCII są następujące:

- 65 – 90 to wielkie litery (A-Z)
- 97 – 122 to małe litery (a-z)
- 48 – 57 to kody cyfr (0–9)

Wyszukiwanie w tekście – indexOf(), lastIndexOf(), search()

W pracy z tekstami w JS bardzo często może pojawić się potrzeba sprawdzenia, czy dany fragment (słowo, fraza, litera) znajduje się w jakimś łańcuchu znaków. Bywa też, że potrzebujemy się dowiedzieć, gdzie dokładnie się on znajduje (od którego indeksu występuje). JavaScript daje nam do tego trzy klasyczne metody:

`indexOf()`, `lastIndexOf()`, `search()` – każda z nich działa trochę inaczej, lecz wszystkie potrafią nam odpowiedzieć na dwa kluczowe pytania: *Czy coś istnieje w napisie? A jeśli tak, to gdzie?*

Metoda

`indexOf()` służy do znalezienia indeksu (czyli w praktyce pozycji w łańcuchu), na której dany ciąg znaków pojawia się po raz pierwszy. A jeśli nie znajdziemy wystąpienia, to zwrócona zostanie wartość `-1`. Dlaczego właśnie taka? Ponieważ rzecz jasna taka wartość nie wystąpi nigdy naturalnie jako indeks w łańcuchu. Przykłady zastosowania:

```
let napis = "JavaScript";
let wynik1 = napis.indexOf("a");
alert("Indeks pierwszego a: " + wynik1); // 1
let wynik2 = napis.indexOf("Script");
alert("Indeks napisu Script: " + wynik2); // 4
let wynik3 = napis.indexOf("x");
alert("Indeks litery x: " + wynik3); // -1, bo x nie wystąpiło
```

Metoda

`lastIndexOf()` – jak sama nazwa wskazuje, ta metoda działa prawie identycznie jak `indexOf()`, ale rozpoczyna przeszukiwanie łańcucha od końca:

```
let napis = "JavaScript";
let wynik4 = napis.lastIndexOf("a");
alert("Ostatni indeks litery a: " + wynik4); // 3
let wynik5 = napis.lastIndexOf("J");
alert("Ostatni indeks litery J: " + wynik5); // 0
let wynik6 = napis.lastIndexOf("x");
alert("Ostatni indeks litery x: " + wynik6); // -1, bo x nie wystąpiło
```

Metoda

`search()` działa analogicznie jak `indexOf()`, lecz z jedną różnicą – obsługuje tzw. *wyrażenia regularne* (*regex*). Dzięki temu możemy np. zignorować wielkość liter podczas wyszukiwania:

```
let napis = "JavaScript";
let wynik7 = napis.search("Script");
alert("Indeks frazy Script: " + wynik7); // 4
let wynik8 = napis.search(/script/); // bez flagi i – rozróżnia wielkość liter
alert("Indeks frazy script: " + wynik8); // -1, bo jest Script, ale nie script
let wynik9 = napis.search(/script/i); // z flagą i – ignoruje wielkość liter
alert("Indeks frazy script bez uwzględniania wielkości liter : " + wynik9); // 4
```

Wycinanie fragmentów tekstu – slice(), substring(), substr()

Czasami potrzebujemy pobrać tylko wycinek tekstu – na przykład kilka pierwszych znaków, fragment od 3. do 6. znaku albo wszystko od połowy do końca napisu etc. W JavaScript do tego celu służą trzy podstawowe metody:

`slice(start, end)`, `substring(start, end)` oraz `substr(start, length)`.

Metoda

`slice(start, end)`, pozwala nam wyciąć fragment łańcucha znaków, od znaku `start` do znaku `end` (bez niego, czyli nie włącznie). Jeśli pominiemy drugi argument, metoda zwróci wszystko od startu do końca. Można też podawać liczby ujemne, aby liczyć znaki od końca łańcucha:

```
let napis = "JavaScript";
let wynik1 = napis.slice(4, 10); // "Script"
alert("Wynik slice(4, 10): " + wynik1);
let wynik2 = napis.slice(-3); // ostatnie 3 litery - "ipt"
alert("Wynik slice(-3): " + wynik2);
```

Metoda

`substring(start, end)` również wycina fragment napisu, ale nie pozwala nam używać liczb ujemnych. Gdy podamy liczby w złej kolejności (tzn. najpierw większy indeks), to metoda automatycznie je zamieni.

```
let napis = "JavaScript";
let wynik3 = napis.substring(4, 10); // "Script"
alert("Wynik substring(4, 10): " + wynik3);
let wynik4 = napis.substring(10, 4); // też "Script" (bo zamieni indeksy miejscami)
alert("Wynik substring(10, 4): " + wynik4);
let wynik5 = napis.substring(-3); // potraktuje -3 jako 0
alert("Wynik substring(-3): " + wynik5); // "JavaScript"
```

Metoda

`substr(start, length)` działa podobnie, ale jako drugi parametr przyjmuje długość pobieranego fragmentu (ile znaków wyjąć?), zamiast końcowego indeksu. Można używać wartości ujemnych (rozpoczynając wyjmowanie od końca), lecz ta metoda jest obecnie uznawana za zdeprecjonowaną (*ang. deprecated*).

```
let napis = "JavaScript";
let wynik6 = napis.substr(4, 6); // "Script"
alert("Wynik substr(4, 6): " + wynik6);
let wynik7 = napis.substr(-3); // "ipt"
alert("Wynik substr(-3): " + wynik7);
```

W codziennej pracy oraz na egzaminie INF.03 (o ile arkusz nie wymusza inaczej) najlepiej zastosować metodę

`slice(start, end)`, bo pozwala na najwięcej (stosowanie liczb ujemnych do wyjmowania ostatnich znaków w napisie).

“Podmiana” fragmentu tekstu – replace()

Czasami chcielibyśmy w skrypcie JS zamienić część jakiegoś łańcucha – np. zastąpić w zdaniu jedno słowo innym. I oczywiście stosując podejście z konkatenacją nowych łańcuchów możemy tego dokonać sami, przygotowując kopię oryginału dzięki odpowiedniemu sklejeniu napisów. Natomiast istnieje też w języku JavaScript bardzo prosta, lecz niesamowicie użyteczna metoda

`replace()` (*ang. replace = zastąp*):

```
let napis1 = "Anna ma kota";
let napis2 = napis1.replace("kota", "psa");
alert(napis2); // Wynik: "Anna ma psa"
```

Lecz skoro wiemy, że łańcuchy znaków (stringi) są niezmienne (*ang. immutable*) to dlaczego istnieje metoda "zastępująca" jedno słowo innym? Cóż, zwróćmy uwagę, iż wcale nie zmieniliśmy tutaj oryginalnego łańcucha

`napis1`, tylko utworzyliśmy nowy o nazwie `napis2`, w którym już wskazany fragment został "podmieniony". A zatem nie jest to w praktyce żadne złamanie zasady niemodyfikowalności napisów, tylko po prostu dokonanie "automatycznej" konkatenacji dla kopii – oryginał pozostaje nietknięty! Jedynie angielskie słówko *replace* może nam, ludziom sugerować inny sposób działania. Natomiast ta metoda po prostu oszczędza nam czas potrzebny do napisania customowego kodu podmieniającego wybrane fragmenty łańcuchów.

Pytanie jakie pojawi się teraz w głowach wielu osób brzmi: *Czy ta metoda zamieni wszystkie wystąpienia podanej frazy w łańcuchu, czy jednie pierwsze?* Otóż domyślnie

`replace()` zamienia tylko **pierwsze wystąpienie** szukanego fragmentu:

```
let tekst = "kot, kot, kot";
let wynik1 = tekst.replace("kot", "pies");
alert(wynik1); // Wynik: "pies, kot, kot"
```

Natomiast jeśli chcielibyśmy zamienić **wszystkie wystąpienia**, to musimy użyć wyrażenia regularnego z flagą g (*ang. global*):

```
let tekst = "kot, kot, kot";
let wynik2 = tekst.replace(/kot/g, "pies");
alert(wynik2); // Wynik: "pies, pies, pies"
```

Drugie popularne pytanie dotyczące tej metody to: Czy wielkość liter ma w tej metodzie znaczenie? I ponownie, metoda wywołana domyślnie, czyli bez regexa **rozróżnia wielkość liter**:

```
let tekst = "JavaScript to nie javascript";
let wynik1 = tekst.replace("javascript", "Python");
alert(wynik1); // "JavaScript to nie Python" - zamiana dla słowa z małym j, nie z
dużym J
```

Dopiero gdy użyjemy wersji z wyrażeniem regularnym i dodamy flagę i (*ang. ignore case*), to zadziała bez względu na duże/małe litery:

```
let tekst = "JavaScript to nie javascript";
let wynik2 = tekst.replace(/javascript/i, "Python");
alert(wynik2); // "Python to nie javascript - zamiana już dla pierwszego wystąpienia
mimo dużego J"
```

Zmiana wielkości liter – toLowerCase(), toUpperCase()

Tych klasycznych operacji warto dokonać gotową metodą, bo własnoręczna podmiana znak po znaku okaże się czasochłonna. Nazwy metod mówią same za siebie – zobaczmy więc od razu przykłady ich użycia:

```
let napis = "Ala ma kota";
let maly = napis.toLowerCase();
let duzy = napis.toUpperCase();
alert(maly); // "ala ma kota"
alert(duzy); // "ALA MA KOTA"
```

Oczywiście ponownie nie łamiemy zasady niezmienności stringów – to kopie oryginału posiadają zmienioną wielkość liter. Oczywiście obie metody bezproblemowo radzą sobie ze znakami nie będącymi literami (typu spacja, cyfra, znak specjalny), tzn.

pozostawiają je niezmienione i poprawnie przepisane do stworzonej kopii łańcucha.

Podział tekstu na części – metoda split()

Czasami mamy w skrypcie do czynienia z długim napisem – na przykład całym zdaniem, ciągiem liczb rozdzielonych przecinkami albo imionami oddzielonymi spacją. I właśnie w takich momentach z pomocą przychodzi nam metoda

`split()`, która pozwala "rozbić" napis na mniejsze części – najczęściej na tablicę słów, znaków lub elementów. Metoda działa tak, iż przekształca łańcuch znaków w tablicę wartości (*ang. array*) – dzieląc je według wybranego separatora. Separator to znak (lub ciąg znaków), który mówi, gdzie należy zawsze dokonać "cięcia" w tekście. Zobaczmy przykład:

```
let zdanie = "To jest egzamin INF.03";
let tablica = zdanie.split(" ");
alert(tablica[2]); // wypisze: egzamin - trzecia szuflada tablicy
```

Możemy też dodać opcjonalny drugi parametr – maksymalną liczbę szuflad, jakie chcemy uzyskać w tablicy:

```
let zdanie = "To jest bardzo ważny egzamin zawodowy INF.03";
let tablica_na_trzy_szuflady = zdanie.split(" ", 3);
alert(tablica_na_trzy_szuflady[2]); // wypisze: bardzo i to ostatnia szuflada!
```

Dzięki temu metoda zatrzyma się po trzecim "cięciu", a reszta napisu zostanie pominięta.

Podsumowanie

Opanowanie metod pracy z łańcuchami znaków jest niezbędne na egzaminie INF.03 i w codziennej pracy programisty – wiele tych zabiegów okaże się czasochłonnych w samodzielnej implementacji – warto więc znać na egzaminie "gotowce". Przedstawmy najważniejsze wnioski z tej lekcji:

- W JavaScript napisy (stringi) są niezmienne *(ang. immutable)* – nie modyfikujemy ich bezpośrednio, tylko tworzymy nowe kopie.
- Do tworzenia nowego napisu używamy operatora `+=` – pozwala on doklejać do łańcucha kolejne znaki lub wybrane fragmenty.
- Odczyt znaku z napisu (np. trzeciego z kolei) możliwy jest przez `napis[2]` lub `napis.charAt(2)` – ale zapisu nie dokonamy! Plus lepiej używać metody `charAt()` zamiast nawiasów kwadratowych.
- Właściwość `length` zwraca długość napisu (liczba wszystkich znaków łańcucha, w tym spacje i znaki specjalne).
- Metoda `charCodeAt()` pozwala poznać kod ASCII danego znaku.
- Wyszukiwanie w napisie wykonujemy za pomocą:
  - `indexOf()` – zwraca indeks pierwszego wystąpienia tekstu,
  - `lastIndexOf()` – szukamy od końca łańcucha,
  - `search()` – z obsługą wyrażeń regularnych (np. bez rozróżniania wielkości liter).
- Fragmenty tekstu wycinamy za pomocą:
  - `slice(start, end)` – obsługuje indeksy ujemne,
  - `substring(start, end)` – nie obsługuje indeksów ujemnych,

- `substr(start, length)` – zdeprecjonowana, choć nadal zadziała.

- Metoda `replace()` "podmienia" wskazany fragment napisu na inny (w praktyce i tak stworzy kopię, nie łamiąc zasady niezmienności łańcuchów!) – domyślnie "zastąpi" tylko pierwsze wystąpienie. Aby zamienić wszystkie wystąpienia, używamy `replace(/tekst/g, "nowy")`.
- Wielkość liter przy `replace()` ma znaczenie – chyba że użyjemy flagi `i` w wyrażeniu regularnym.
- Do zmiany wielkości liter służą:
  - `toLowerCase()` – zamienia cały napis na małe litery,
  - `toUpperCase()` – na duże litery.

- Metoda `split()` dzieli tekst na tablicę – według podanego separatora (np. spacji, przecinka, znaku specjalnego). Do `split()` możemy dodać opcjonalny drugi argument ograniczający liczbę elementów w tworzonej tablicy.