Tworzenie własnych funkcji, zdarzenia

W programowaniu w JS dążymy do tego, aby nasze skrypty były jak najbardziej elastyczne, czytelne i łatwe w utrzymaniu. Czy zdarzyło się nam kiedyś kopiować ten sam kod kilka razy w różnych miejscach? Jeśli tak, to znak, że czas zaprzyjaźnić się z funkcjami, czyli fragmentami kodu, które realizują konkretne zadanie – dzięki nim unikniemy powtarzania się (redundancji).

Co to jest funkcja i po co jej używać?

Funkcja to nic innego jak fragment kodu, który raz zdefiniujemy (i ewentualnie nazwiemy), a potem możemy go uruchamiać wielokrotnie. To jak przepis na ciasto – zamiast za każdym razem obmyślać na nowo jakie kroki pieczenia kolejno wykonać – mamy gotowy schemat postępowania i tylko go uruchamiamy (w programowaniu mówimy: wywołujemy). Inna popularna metafora to porównanie własnej funkcji do pracownika – wie on jak dane zadanie skutecznie zrealizować, stąd możemy wielokrotnie zatrudnić go (jak wspominaliśmy: wywołać) do pracy.

Załóżmy, że użytkownik podaje 3 różne teksty i za każdym razem chcemy sprawdzić ile zawiera samogłosek.

Bez funkcji kod będzie bardzo powtarzalny i męczący do utrzymania:

```
let tekst1 = "Ala ma kota";
let tekst1Lower = tekst1.toLowerCase();
let samogloski1 = 0;
for (let i = 0; i < tekst1Lower.length; i++) {
    if ("aeiouyąęó".includes(tekst1Lower[i])) {
        samogloski1++;
    }
}
console.log("Tekst 1 ma " + samogloski1 + " samogłosek.");
let tekst2 = "JavaScript jest fajny";
let tekst2Lower = tekst2.toLowerCase();
let samogloski2 = 0;
for (let i = 0; i < tekst2Lower.length; i++) {
    if ("aeiouyąęó".includes(tekst2Lower[i])) {
        samogloski2++;
    }
}
console.log("Tekst 2 ma " + samogloski2 + " samogłosek.");
let tekst3 = "Technikum informatyczne";
let tekst3Lower = tekst3.toLowerCase();
let samogloski3 = 0;
for (let i = 0; i < tekst3Lower.length; i++) {
    if ("aeiouyąęó".includes(tekst3Lower[i])) {
        samogloski3++;
    }
}
console.log("Tekst 3 ma " + samogloski3 + " samogłosek.");
```

Trzy razy powtarzamy to samo! Jeśli więc stworzyliśmy algorytm zliczania ile samogłosek znajduje się w tekście (łańcuchu), to wystarczy go zapisać jako funkcję, nazwać ją i potem używać (wywoływać) tyle razy ile chcemy!

Ten sam kod JS z własną funkcją:

```
function policzSamogloski(tekst) {
    let tekstLower = tekst.toLowerCase();
```

```
    let licznik = 0;
    for (let i = 0; i < tekstLower.length; i++) {
        if ("aeiouyąęó".includes(tekstLower[i])) {
            licznik++;
        }
    }
    return licznik;
}
// Trzykrotne jest tylko wywołanie funkcji (po jej nazwie) - logika działania zawsze
ta sama
console.log("Tekst 1 ma " + policzSamogloski("Ala ma kota") + " samogłosek.");
console.log("Tekst 2 ma " + policzSamogloski("JavaScript jest fajny") + "
samogłosek.");
console.log("Tekst 3 ma " + policzSamogloski("Technikum informatyczne") + "
samogłosek.");
```

Mamy w kodzie jedno “miejsce” odpowiedzialne za logikę – łatwiej testować i modyfikować oraz poprawić ewentualne błędy tylko w jednym miejscu. Zyskujemy krótszy i bardziej czytelny kod – raz zdefiniowaną funkcję możemy wykorzystać w dowolnym miejscu – często w JS “przypinamy” funkcję np. do zdarzenia kliknięciu przycisku.

Funkcje w skryptach nie “startują” same

Kiedy zdefiniujemy funkcję w JavaScript – na przykład bardzo prostą:

```
function przywitaj_kursanta() {
    alert("Witaj w kursie JavaScript!");
}
```

To musimy sobie uświadomić jedną kluczową sprawę: kod w środku (tutaj alert) nie uruchomi się sam podczas wczytywania skryptu przez przeglądarkę. Definiując funkcję, mówimy przeglądarce coś w stylu: *Hej, to jest zapis tego, co ma się wydarzyć, jeśli ktoś w przyszłości poprosi o wykonanie tego fragmentu*. Ale samo zapisanie funkcji nie znaczy jeszcze, że przeglądarka w ogóle ją wykona! Żeby faktycznie uruchomić funkcję, musimy ją – jak wspominaliśmy – wywołać. Wystarczy zapisać jej nazwę wraz z nawiasami okrągłymi:

```
przywitaj_kursanta();
```

Możemy to porównać do np. czajnika elektrycznego w kuchni. Nawet jeśli stoi na blacie, podłączony do gniazdka i pełny wody, to przecież sam się nie włączy. Musimy nacisnąć przycisk, żeby rozpocząć gotowanie wody. Tak samo jest z funkcją – dopóki jej nie “naciśniemy”, czyli nie *wywołamy* – nic się nie zadzieje. Wielu początkujących programistów zapisuje funkcję w skrypcie i dziwi się, że “*nic się w skrypcie nie dzieje*“. No i właśnie – nie ma się dziać, jeśli zapomnieliśmy o wywołaniu. Sama definicja funkcji to tylko “przepis” co zrobić (algorytm, lista kroków, sposób) – uruchomienie kodu to już co innego:

- Zapisanie funkcji (za pomocą `function`) to tylko definicja – nic się jeszcze nie uruchomi w skrypcie.
- Aby funkcja zadziałała, musimy ją wywołać albo przypisać do obsługi zdarzenia np. kliknięcia przycisku.
- To tak jak z przepisem na ciasto – samo posiadanie przepisu nie sprawi, że można zacząć jeść szarlotkę.

Funkcja obsługująca np. zdarzenie kliknięcia przycisku

W aplikacjach webowych bardzo często chcemy, aby coś zadziało się dopiero wtedy, gdy użytkownik kliknie jakiś przycisk – np. chcemy wyświetlić komunikat, zmienić kolor tekstu, wykonać jakieś obliczenia. Właśnie do takich sytuacji tworzymy w JavaScript funkcje, które będą wywoływane po kliknięciu. Spójrzmy na prosty przykład – w dokumencie HTML istnieje przycisk z atrybutem

`onclick="pokazKomunikat()"` :

```
<button onclick="pokazKomunikat()">Kliknij mnie</button>
```

I to odpowiadać musi nazwie konkretnej, istniejącej w JS funkcji:

```
function pokazKomunikat() {
    alert("Nastąpiło kliknięcie!");
}
```

Funkcja nie wykonuje się sama z siebie! Zostanie uruchomiona dopiero wtedy, gdy użytkownik kliknie przycisk. Czy atrybut

`onclick` to jedyny sposób wywołania funkcji w momencie kliknięcia? Nie! W nowoczesnym JavaScript istnieje także inny sposób, który jest bardziej profesjonalny – mówimy o specjalnej metodzie `addEventListener()`. Pozwala ona dodać obsługę kliknięcia bez umieszczania w HTML atrybutu. Ale nie martwmy się tym na razie – o różnicach pomiędzy `onclick` oraz `addEventListener()` opowiemy sobie dokładnie w lekcji: Atrybut onclick vs. addEventListener.

Zdarzenia w JavaScript oraz atrybuty HTML dla nich

Opowiedzieliśmy o zdarzeniu

`click`, czyli zwyczajnym kliknięciu na przycisk. Natomiast oczywiście świat zdarzeń (*ang. events*) w JS na tym się nie kończy. W rzeczywistości język ten umożliwia reagowanie na dziesiątki różnych zdarzeń – wszystko po to, by nasza strona była interaktywna, dynamiczna i potrafiła odpowiednio reagować na działania użytkownika.

Co więcej – do każdego z tych zdarzeń możemy przypisać funkcję (czyli napisać co ma się wtedy wydarzyć). W najprostszych projektach robimy to poprzez odpowiednie atrybuty HTML, np.

`onclick`, `onmouseover`, `onchange` itd. Oto lista najbardziej popularnych na egzaminie INF.03 zdarzeń oraz odpowiadających im atrybutów HTML, które możemy wykorzystać:

Zdarzenie	Atrybut HTML	Opis
click	`onclick`	Kliknięcie myszką na elemencie
mouseover	`onmouseover`	Najechanie kursorem na element
mouseout	`onmouseout`	Odsunięcie kursora z elementu
change	`onchange`	Zmiana wartości w polu – np. `<input>`, `<select>`
focus	`onfocus`	Wejście kursorem w pole edycyjne

Zdarzenie	Atrybut HTML	Opis
blur	`onblur`	Wyjście z pola edycyjnego (utrata “skupienia”)
keydown	`onkeydown`	Wciśnięcie klawisza na klawiaturze
keyup	`onkeyup`	Puszczenie klawisza na klawiaturze
load	`onload`	Załadowanie całej strony, np. atrybut w `<body>`
submit	`onsubmit`	Wysłanie formularza

Wyobraźmy sobie, że tworzymy prostą stronę, na której użytkownik może zmienić kolor tła za pomocą rozwijanej listy `<select>`. Kiedy użytkownik wybierze inny kolor z listy – chcemy, aby tło zmieniło się automatycznie. Kod HTML z atrybutem `onchange`:

```
<label for="kolor">Wybierz kolor tła:</label>
<select id="kolor" onchange="zmienTlo()">
    <option value="white">Biały</option>
    <option value="lightblue">Niebieski</option>
    <option value="lightgreen">Zielony</option>
    <option value="lightgray">Szary</option>
</select>
```

Skrypt JS z własną funkcją do zmiany koloru:

```
function zmienTlo() {
    let wybranyKolor = document.getElementById("kolor").value;
    document.body.style.backgroundColor = wybranyKolor;
}
```

Gdy tylko użytkownik zmieni wybór w polu `<select>`, zostanie wywołana funkcja `zmienTlo()` – w końcu jest przypisana do obsługi zdarzenia `change` naszej listy. Przy wywołaniu funkcja pobiera wybrany kolor z listy – właściwość `value` – a następnie ustawia go jako kolor tła całej strony.

Jak mądrze nazywać funkcje?

W zadaniach egzaminacyjnych INF.03 sposób nazywania funkcji w JavaScript ma naprawdę duże znaczenie, ponieważ:

- Zadania są oceniane na podstawie polecenia (klucz odpowiedzi), a nie tylko tego, czy coś poprawnie działa.

- Jeśli funkcja nazywa się inaczej niż wymaga polecenie – egzaminator może nie przyznać punktu, nawet jeśli wszystko działa!

Kilka porad praktycznych do egzaminu:

1. Zawsze nazywamy funkcję dokładnie tak, jak w treści zadania. Na przykład jeśli polecenie brzmi: "*Napisz funkcję o nazwie obliczSume, która po kliknięciu przycisku doda dwie liczby z formularza*" to nasza funkcja **musi** nazywać się `obliczSume()`. Nie używaj innej nazwy typu: `suma()`, `dodajLiczby()` czy `policz()` – szkoda tracić łatwe punkty!

2. Trzymamy się notacji wielbłądziej, czyli tzw. *camelCase*. W JavaScript funkcje nazywamy z małej litery (często dla funkcji rozpoczynamy od czasownika), a kolejne słowa w nazwie rozpoczynamy wielką literą: `obliczSume()`, `sprawdzDaneWejsciowe()`, `obliczPoleKwadratu`. Nie stosuj polskich znaków diakrytycznych (ą, ć, ł, ń…), spacji ani znaków specjalnych (#, @,).

3. Nie skracamy nazw, jeśli polecenie tego nie robi! Jeśli w arkuszu przeczytamy polecenie: "*Utwórz funkcję pokazNazwisko*" to nie twórzmy skrótu `pokNaz()` czy `pokNazwisko()`

4. Jeśli w poleceniu nie ma nazwy funkcji – wybieramy nazwę logiczną, najlepiej zawierającą czasownik – kilka przykładów `sprawdzFormularz()`, `obliczRabat()`

5. Funkcja nie może mieć nazwy zarezerwowanej w JS – nie twórzmy własnych funkcji mających w nazwach: alert, document, onclick, String, parseInt etc. Spowoduje to co najmniej zamieszanie, a czasami nawet konflikty w kodzie, uniemożliwiające poprawne wykonanie skryptu.

Funkcje zwracające wartości – słowo return

Nauczyliśmy się już, jak tworzyć zwykłe funkcje które coś robią – np. wyświetlają wynik w przeglądarce, zmieniają kolor etc. Natomiast teraz omówmy szczegółowo funkcje, które coś *zwracają*. Czyli dają nam konkretny wynik, który możemy dalej wykorzystać w programie. Słówko

`return` oznacza: "*zwróć wartość z funkcji na zewnątrz*". Czyli kiedy funkcja zakończy działanie, to nie tylko kończy wykonanie kodu, ale ponadto oddaje wartość, którą potem możemy zapisać w zmiennej, wypisać na ekranie, użyć w obliczeniach itd.

To tak, jakby funkcja była kalkulatorem – podajemy jej dane, a ona "wypluwa" wynik. Przykład: funkcja zwracająca pole prostokąta. Najpierw zrealizujmy to bez zwracania wartości – tylko wypisujemy wynik:

```
function obliczPole(a, b) {
    let pole = a * b;
    alert("Pole wynosi: " + pole);
}
```

Działa, lecz… nie da się użyć tego pola gdzie indziej – np. nie da się dodać go do innego pola we wzorze matematycznym albo zapisać do zmiennej. A teraz ta sama funkcja, lecz już zwracająca dzięki

`return` wartość:

```
function obliczPole(a, b) {
    return a * b;
}
```

I teraz możemy już funkcji użyć nie tylko w wypisaniu wartości ale także zapisać wynik do zmiennej albo wykorzystać funkcję w obliczeniach matematycznych:

```
// Zapis do zmiennej:
```

```
let wynik = obliczPole(5, 10);
// Pokazanie w oknie alert:
alert("Pole: " + wynik);
// Użycie w obliczeniach matematycznych:
let sumaPol = obliczPole(5, 10) + obliczPole(3, 4);
```

To właśnie jest siła funkcji zwracających wartości – są wielokrotnego użytku, niezależne i elastyczne. Możemy je łączyć, porównywać, wyświetlać albo przetwarzać. Funkcje zwracające wartości przydają się, gdy:

- Potrzebujemy wynik działania matematycznego (np. pole, obwód, średnia, suma).
- Chcemy pobrać dane i zapisać je do zmiennej (np. liczba samogłosek w tekście czy licznik powtórzeń konkretnego znaku).
- Chcemy wykonać logikę funkcji w warunku instrukcji `if` czyli zwrócić “tak” lub “nie” (np. *Czy liczba jest parzysta? Czy istnieją w tekście przynajmniej 3 samogłoski?*).
- Tworzymy funkcję pomocniczą dla innego kodu (np. funkcję do walidacji konkretnych danych).

Funkcje anonimowe, czyli nie mające nazwy

W JavaScript istnieje także inny rodzaj funkcji, której możemy używać – **funkcja anonimowa**. Jest to taka funkcja, która nie ma “imienia” (czyli nazwy). Taką funkcję najczęściej przekazujemy do zmiennej lub od razu mechanizmu obsługi zdarzenia. Zobaczmy najprostszy przykład – funkcja przypisana do zmiennej:

```
let powiedzCzesc = function() {
    alert("Cześć!");
};
powiedzCzesc(); // to jest wywołanie
```

Funkcja nie ma zdefiniowanej po słowie kluczowym

`function` nazwy, ale przypisaliśmy ją do zmiennej o nazwie `powiedzCzesc` i dzięki temu możemy ją wywołać. Można więc powiedzieć, że po prostu sposób zapisu jest inny, ale tak czy siak nazwa została pośrednio nadana. Natomiast zobaczmy funkcję anonimową przypisaną do obsługi zdarzenia kliknięcia:

```
document.getElementById("przycisk").onclick = function() {
    alert("Kliknięto przycisk!");
};
```

Lub to samo bardziej profesjonalnie, czyli zamiast podmiany atrybutu HTML

`onclick` użyjemy metody `addEventListener()`:

```
document.getElementById("przycisk").addEventListener("click", function() {
    alert("Kliknięto przycisk!");
});
```

W obu tych sytuacjach nie tworzymy żadnej osobno nazwanej funkcji – wszystko zapisujemy od razu jako anonimową funkcję przypisaną bezpośrednio do zdarzenia.

Zobaczmy jeszcze funkcję anonimową w metodzie

`forEach()`, która świetnie nadaje się do iterowania po elementach tablicy. Załóżmy, że mamy tablicę liczb i chcemy je po prostu z użyciem funkcji anonimowej wypisać:

```
let liczby = [1, 2, 3, 4, 5];
liczby.forEach(function(liczba) {
    console.log("Liczba: " + liczba);
});
```

Zamiast pisać osobną funkcję nazwaną

`wypiszLiczbe()`, podajemy anonimową funkcję bezpośrednio do metody `forEach()`.

Reasumując: funkcje anonimowe najczęściej zastosujemy wtedy, gdy:

- Nie potrzebujemy funkcji na stałe, tylko "na raz" – np. do obsługi kliknięcia czy wykonania konkretnego zadania.
- Chcemy przekazać funkcję jako argument do innej funkcji – np. do `forEach()`, `setTimeout()`, `addEventListener()` etc.

Funkcja anonimowa jest trochę jak tymczasowy pracownik, który ma wykonać jedno zadanie i znika. Nie potrzebujemy go znać po imieniu ani zaprzyjaźniać się – to najemnik, który robi co trzeba i znika.

Funkcje strzałkowe – jeszcze krótszy zapis!

JavaScript w wersji ECMAScript 6 (czyli tzw. ES6), wprowadził jeszcze krótszą, zgrabniejszą formę zapisu funkcji – tzw. funkcje strzałkowe (*ang. arrow functions*). I choć nie są one obowiązkowe na egzaminie INF.03, to ich znajomość może pomóc nam napisać bardziej nowoczesny i zwięzły kod. Zobaczmy więc, czym one są, jak je zapisać oraz kiedy warto z nich korzystać. Spójrzmy najpierw na dobrze nam znany, klasyczny sposób definiowania funkcji:

```
function przywitaj(imie) {
    return "Witaj, " + imie + "!";
}
console.log(przywitaj("Michał")); // wywołanie
```

Jest to klasyczna, nawet nie anonimowa funkcja w JS. Jak wyglądałaby ta sama funkcja jako funkcja strzałkowa? Otóż zamieniamy słowo

`function` na symbol `=>` (czytamy to jako właśnie "strzałkę"). Efekt działania będzie dokładnie taki sam! Ale zapis jest nieco krótszy, a co najważniejsze – bardziej nowoczesny:

```
const przywitaj = (imie) => { return "Witaj, " + imie + "!"; }
```

Lecz można jeszcze krócej! Jeśli funkcja zawiera tylko jedno polecenie

`return`, to możemy zapisać ją nawet bez nawiasów klamrowych i bez słowa `return`:

```
const przywitaj = (imie) => "Witaj, " + imie + "!";
```

I zadziała dokładnie tak samo. Piękna sprawa, prawda? Co jeszcze warto wiedzieć o funkcjach strzałkowych?

- Jeśli funkcja nie przyjmuje żadnych argumentów, to i tak musimy dać puste nawiasy: `() => { ... }`
- Jeśli jest tylko jeden argument, to nawiasy wokół niego są opcjonalne: `imie => "Witaj, " + imie`
- Jeśli argumentów jest więcej niż jeden, nawiasy muszą być: `(a, b) => a + b`

Oczywiście przydałby się przykład. Użyjmy zatem funkcji strzałkowej do obsługi zdarzenia kliknięcia – tak jak moglibyśmy to zrealizować na egzaminie INF.03:

```
document.getElementById("przycisk").addEventListener("click", () => {
document.body.style.backgroundColor = "lightblue"; });
```

To krótszy i najbardziej nowoczesny zapis funkcji przypisanej do obsługi zdarzenia

`click`. Czy warto zatem korzystać z funkcji strzałkowych? Zdecydowanie tak – szczególnie jeśli zależy nam na krótkim, nowoczesnym kodzie. Na egzaminie INF.03 oczywiście wystarczy znać klasyczny zapis funkcji, ale funkcji strzałkowych warto się nauczyć.

Jednak uwaga! Funkcje strzałkowe, czyli zapisane za pomocą

`=>` nie mają swojego własnego wskaźnika `this`. O co tutaj chodzi? Uprzedzam – jest to temat trudniejszy, więc jeśli funkcji strzałkowych nie zamierzasz używać, to możesz pominąć te wyjaśnienia. Natomiast przechodząc już do tłumaczeń – w języku JavaScript słowo kluczowe `this` odnosi się do **kontekstu**, w którym funkcja została **wywołana** – czyli do tego "*kto*" ją wywołał, albo inaczej "*z jakiego obiektu*" ją uruchomiliśmy.

Funkcje strzałkowe dziedziczą

`this` z miejsca, w którym zostały zadeklarowane, a nie z miejsca w którym są wywoływane! Czyli – mówiąc bardziej po ludzku – patrzą oczami otoczenia, w którym powstały, a nie "swoimi oczami". Co to oznacza w praktyce? Zobaczmy przykład z funkcją anonimową – taka implementacja zadziała poprawnie:

```
document.getElementById("przycisk").addEventListener("click", function() {
    this.style.backgroundColor = "lightblue";
});
```

Słowo kluczowe

`this` wskazuje na element, który wywołał zdarzenie, czyli w naszym przypadku na przycisk, który kliknęliśmy. Dzięki temu możemy odwołać się do jego właściwości `style`, np. zmienić jego kolor. Teraz zobaczmy ten sam przykład z funkcją strzałkową:

```
document.getElementById("przycisk").addEventListener("click", () => {
    this.style.backgroundColor = "lightblue"; // Błąd! this nie wskaże na przycisk!
});
```

W tym przypadku

`this` nie wskaże na przycisk. Funkcja strzałkowa nie widzi właściwości `style` przycisku, bo patrzy na zewnętrzny kontekst (obiekt window w przeglądarce, a nie przycisk). Efekt? Kod nie zadziała tak jak chcieliśmy. Podsumujmy zatem teraz tzw. *arrow functions*:

- Funkcje strzałkowe to najkrótszy zapis funkcji w JavaScript – raczej nie są wymagane na egzaminie INF.03, lecz osoby ambitne mogą ich z powodzeniem chcieć używać.
- Używamy w nich symbolu `=>` zamiast słowa kluczowego `function`.
- Są idealne do przypisywania działania do zdarzeń i funkcji zwracających wartości.
- Nie mają własnego `this` – więc przy klasach i obiektach trzeba uważać!

Podsumowanie – co zapamiętać?

- Funkcja w JS to fragment kodu, który wykonuje określone zadanie i może być wielokrotnie używany – pozwala to unikać powtórzeń i zwiększa czytelność kodu.
- Funkcje zapisujemy klasycznie za pomocą słowa kluczowego `function`, a wywołujemy je przez podanie nazwy i nawiasów: `nazwaFunkcji()`.
- Funkcja sama się nie uruchamia – musi zostać wywołana albo przypisana do obsługi zdarzenia (np. kliknięcia przycisku).
- Aby przypisać funkcję do zdarzenia, możemy użyć atrybutu HTML (np. `onclick`) lub bardziej profesjonalnie metody `addEventListener()`.
- Popularne zdarzenia w JS to m.in. `onclick`, `onchange`, `onmouseover`, `onkeydown` itd.
- Warto nazwać funkcję dokładnie tak, jak wymagają tego polecenia egzaminacyjne – np. `obliczSume()`, a nie `policz()` czy `suma()`.
- Funkcja może zwracać wartość za pomocą `return` – pozwala to np. zapisać wynik do zmiennej lub użyć go w obliczeniach matematycznych.
- Funkcje anonimowe to takie, które nie mają nazwy – przypisujemy je np. do zmiennych lub zdarzeń bezpośrednio: `function() {...}`.
- Funkcje strzałkowe (`() => {}`) są najkrótszą formą zapisu funkcji, przydatną do prostych operacji, ale nie mają własnego `this` – co może powodować błędy przy pracy z obiektami DOM.
- Funkcje strzałkowe dobrze sprawdzają się w iteracjach (np. `forEach()`) i przypisaniach obsługi zdarzeń, ale należy uważać z ich użyciem wewnątrz obiektów.
- Na egzaminie INF.03 umiejętność tworzenia, wywoływania i przypisywania funkcji do zdarzeń to absolutna podstawa, której opanowanie zapewnia łatwe punkty i poprawne działanie interfejsu strony.